13 Real-time software design

Objectives

The objectives of this chapter are to introduce techniques that are used in the design of real-time systems and to describe some generic real-time system architectures. When you have read this chapter, you will:

- understand the concept of a real-time system and why real-time systems are usually implemented as a set of concurrent processes;
- have been introduced to a design process for real-time systems;
- understand the role of a real-time executive;
- understand common process architectures for monitoring and control systems and data acquisition systems.

Contents

Computers are used to control a wide range of systems ranging from simple domestic machines to entire manufacturing plants. These computers interact directly with hardware devices. The software in these systems is an embedded real-time system that must react to events generated by the hardware and issue control signals in response to these events. It is *embedded* in some larger hardware system and must respond, in *real time*, to events from the system's environment.

Real-time systems are different from other types of software system. Their correct functioning is dependent on the system responding to events within a given (usually short) time interval. I define a real-time system as follows:

> A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. A 'soft' real-time system is a system whose operation is *degraded* if results are not produced according to the specified timing requirements. A 'hard' real-time system is a system whose operation is *incorrect* if results are not produced according to the timing specification.

One way of looking at a real-time system is as a stimulus/response system. Given a particular input stimulus, the system must produce some corresponding response. The behaviour of a real-time system can therefore be defined by listing stimuli that are received by the system, the associated responses and the time at which the response must be produced.

Stimuli fall into two classes:

1. *Periodic stimuli* These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).
2. *Aperiodic stimuli* These occur irregularly. They are usually signalled using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

Periodic stimuli in a real-time system are usually generated by sensors associated with the system. These provide information about the state of the system's environment. The responses are directed to a set of actuators that control some hardware unit that then influences the system's environment. Aperiodic stimuli may be generated either by the actuators or by sensors. They often indicate some exceptional condition, such as a hardware failure, which must be handled by the system. This sensor–system–actuator model of an embedded real-time system is illustrated in Figure 13.1.

A real-time system has to respond to stimuli that occur at different times. Its architecture must therefore be organised so that control is transferred to the appropriate handler for that stimulus as soon as it is received. This is impractical in sequential programs. Real-time systems are, therefore, normally designed as a set

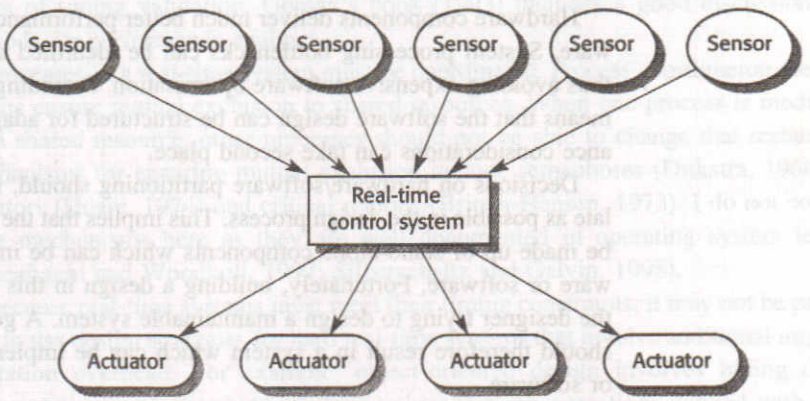


Figure 13.1 General model of a real-time system

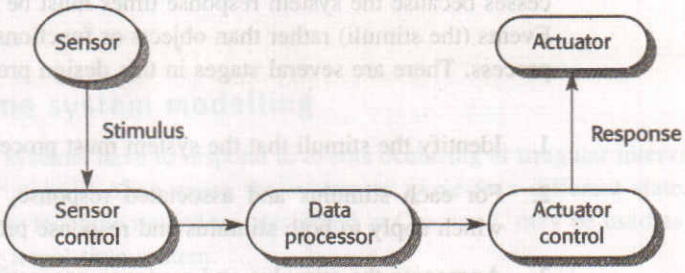


Figure 13.2 Sensor/actuator control processes

of concurrent, cooperating processes. Part of the real-time system (the real-time executive) is dedicated to managing these processes.

The generality of this stimulus/response model of a real-time system leads to the generic architectural model where there are three types of process (see Figure 13.2). For each type of sensor, there is a sensor management process; computational processes compute the required response for the stimuli received by the system; actuator control processes manage actuator operation. This model allows data to be collected quickly from the sensor (before the next input becomes available) and allows processing and the associated actuator response to be carried out later.

13.1 System design

As discussed in Chapter 2, part of the system design process involves deciding which system capabilities are to be implemented in software and which in hardware. Timing constraints or other requirements may mean that some system functions, such as signal processing, have to be implemented using specially designed hardware. The system design process may therefore involve the design of special-purpose hardware as well as real-time software design.

Hardware components deliver much better performance than the equivalent software. System processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimisation. Providing performance in hardware means that the software design can be structured for adaptability and that performance considerations can take second place.

Decisions on hardware/software partitioning should, if possible, be left until as late as possible in the design process. This implies that the system architecture should be made up of stand-alone components which can be implemented in either hardware or software. Fortunately, building a design in this way is exactly the aim of the designer trying to design a maintainable system. A good system design process should therefore result in a system which can be implemented in either hardware or software.

The design process for real-time systems differs from other software design processes because the system response times must be considered early in the process. Events (the stimuli) rather than objects or functions should be central to the design process. There are several stages in this design process:

1. Identify the stimuli that the system must process and the associated responses.
2. For each stimulus and associated response, identify the timing constraints which apply to both stimulus and response processing.
3. Aggregate the stimulus and response processing into a number of concurrent processes. A good general model for the system architecture is to associate a process with each class of stimulus and response as shown in Figure 13.2.
4. For each stimulus and response, design algorithms to carry out the required computations. Algorithm designs often have to be developed relatively early in the design process to give an indication of the amount of processing required and the time required to complete that processing.
5. Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
6. Integrate the system under the control of a real-time executive.

Naturally, this is an iterative process. Once a process architecture has been established and a scheduling policy decided, extensive assessments and simulations are needed to check that the system will meet its timing constraints. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the executive or all of these may then have to be redesigned to improve the performance of the system.

Analysing the timing of a real-time system is difficult. Because of unpredictable nature of aperiodic stimuli, the designers must make some assumptions as to the probability of these stimuli occurring (and therefore requiring service) at any particular time. These assumptions may be incorrect and system performance after delivery may not be adequate. Dasarthy (1985) and Burns (1991) discuss general

issues of timing validation. Gomaa's book (1993) includes a good discussion of methods for performance analysis.

Processes in a real-time system must be coordinated. Process coordination mechanisms ensure mutual exclusion to shared resources. When one process is modifying a shared resource, other processes should not be able to change that resource. Mechanisms for ensuring mutual exclusion include semaphores (Dijkstra, 1968b), monitors (Hoare, 1974) and critical regions (Brinch-Hansen, 1973). I do not cover these mechanisms here as they are well documented in operating system texts (Tanenbaum and Woodhull, 1997; Silberschaltz and Galvin, 1998).

Because real-time systems must meet their timing constraints, it may not be practical to use design strategies for hard real-time systems that involve additional implementation overhead. For example, object-oriented design involves hiding data representations and accessing attribute values through operations defined with the object. This involves an inevitable overhead and consequent loss of performance.

13.1.1 Real-time system modelling

Real-time systems have to respond to events occurring at irregular intervals. These events (or stimuli) often cause the system to move to a different state. For this reason, state machine modelling, described in Chapter 7, may be used as a way of describing a real-time system.

A state model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may cause a transition to a different state. For example, a system controlling a valve may move from a state 'Valve open' to a state 'Valve closed' when an operator command (the stimulus) is received.

I illustrate this approach to system modelling using the model of a simple microwave oven that I introduced in Chapter 7. Figure 13.3 shows a state machine model of a simple microwave oven equipped with buttons to set the power and the timer and to start the system. The rounded rectangles represent system states and the arrowed labels represent stimuli which force a transition from one state to another. The names chosen in the state machine diagram are descriptive and the associated information indicates actions taken by the system actuators or information that is displayed.

You can trace the operation of the oven by reading the model from left to right. In the initial state (Waiting), the user may select either full-power or half-power. The next state is entered when the user presses the timer button and sets the time. Operation may then be enabled when the oven door is closed and the food is cooked in the Operation state. Finally, when cooking is complete, the oven returns to the Waiting state.

State machine models are a good, language-independent way of representing the design of a real-time system. For this reason, they are an integral part of real-time design methods (Ward and Mellor, 1985; Harel, 1987, 1988). Harel's method, which is based on a notation called *Statecharts*, has addressed the problem of the inherent

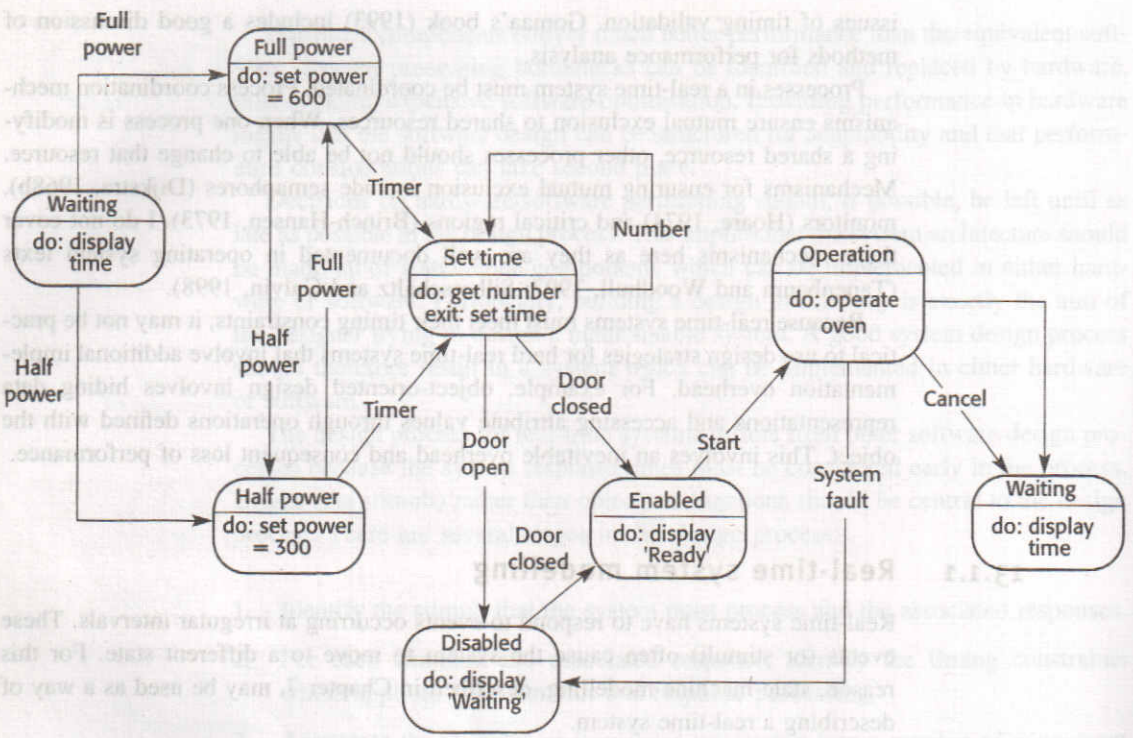


Figure 13.3 State machine model of a microwave oven

complexity of state machine models. Statecharts structure state models so that groups of state can be considered as a single entity. The notation also allows concurrent systems to be represented as state models. State models are also supported in the Unified Modeling Language (Rumbaugh *et al.*, 1999a) and I use the UML notation here.

13.1.2 Real-time programming

The programming language used for implementing a real-time system may also influence the design. Hard real-time systems are still sometimes programmed in assembly language so that tight deadlines can be met. Systems-level languages, such as C, that allow efficient code to be generated may also be used.

The advantage of using a low-level systems programming language like C is that it allows the development of very efficient programs. However, the language does not include any constructs to support concurrency or the management of shared resources. It relies on operating system or executive facilities and, hence, there is increased scope for programming error. Programs are also often more difficult to understand.

Ada was originally designed for embedded systems implementation and has features such as tasking, exceptions and representation clauses. Its *rendezvous* capability is a good general-purpose mechanism for task synchronisation (Burns and Wellings, 1990; Barnes, 1994). Unfortunately, the original version of Ada (Ada 83) was unsuitable for hard real-time systems implementation. It was impossible to specify task deadlines, there is no inbuilt exception if a deadline is not met and a strict first-in, first-out policy for servicing a queue of task entries is imposed. A revision of the 1983 Ada standard (Barnes, 1998) addressed some of these issues. The revised version of the language provided protected types that allowed for the easier implementation of protected, shared data structures and more control over task scheduling and timing. These have improved Ada as a real-time programming language but they still do not provide sufficient control for hard real-time systems.

The initial versions of Java (then called Oak) were designed for writing small-scale embedded systems such as those in domestic appliance controllers. The Java designers have therefore included some support for concurrent processes in the form of concurrent objects (threads) and synchronised methods. However, as these systems do not have strict timing constraints, Java does not include facilities to control the scheduling of threads or to specify that threads should run at particular times.

Java, therefore, is not suitable for programming hard-real time systems or systems where processes have strict deadlines. The fundamental problems with Java as a real-time programming language are:

1. It is not possible to specify the time at which threads should execute.
2. Garbage collection is uncontrollable – it may be started at any time. Therefore, the timing behaviour of threads is unpredictable.
3. It is not possible to discover the sizes of queue associated with shared resources.
4. The implementation of the Java Virtual Machine varies from one computer to another, so the same program can have different timing behaviours.
5. The language does not allow for detailed run-time space or processor analysis.

At the time of writing, work is under way to address some of these problems and to define a real-time version of Java (Nilsen, 1998). However, it is not clear how this can be separated from the implementation of the underlying Java Virtual Machine. There is therefore an inevitable conflict between language portability and real-time characteristics.

13.2 Real-time executives

A real-time executive is analogous to an operating system in a general-purpose computer. It manages processes and resource allocation in a real-time system.

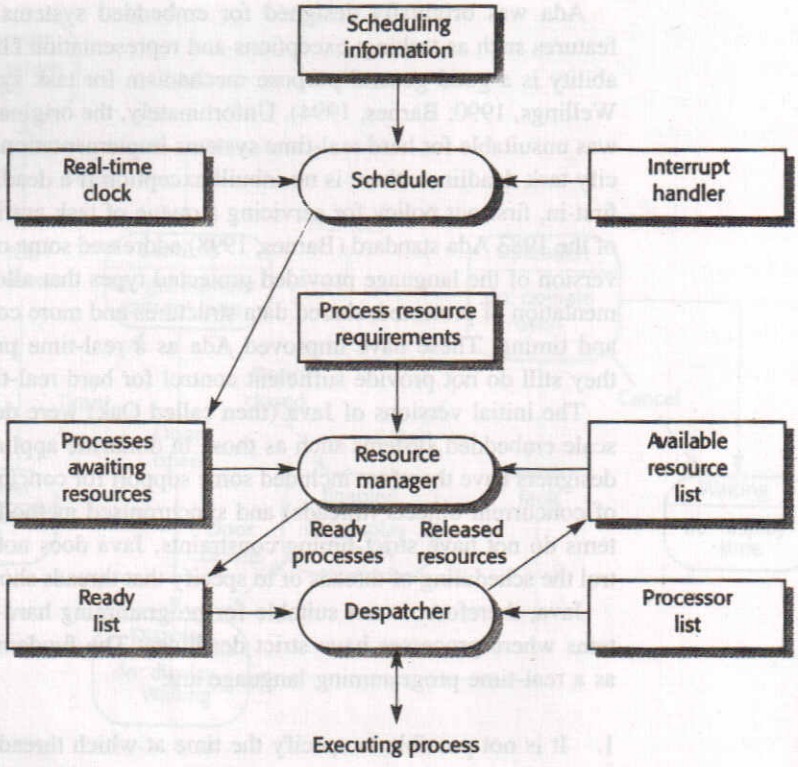


Figure 13.4
Components of a real-time executive

It starts and stops appropriate processes so that stimuli can be handled and allocates memory and processor resources. It does not, however, usually include more complex operating system facilities such as file management.

Baker and Scallon (1986) present a good discussion of the facilities required in real-time executives. Cooling (1991) also covers this topic and briefly discusses commercial real-time executive products. Although there are several real-time executive products available, the specialised requirements of many real-time systems often require that the executive has to be designed as part of the system.

The components of an executive (Figure 13.4) depend on the size and complexity of the real-time system being developed. Normally, for all except the simplest systems, they will include:

1. *A real-time clock* This provides information to schedule processes periodically.
2. *An interrupt handler* This manages aperiodic requests for service.
3. *A scheduler* This component is responsible for examining the processes which can be executed and choosing one of these for execution.
4. *A resource manager* Given a process which is scheduled for execution, the resource manager allocates appropriate memory and processor resources.
5. *A despatcher* This component is responsible for starting the execution of a process.

Systems that provide a continuous service, such as telecommunication and monitoring systems with high reliability requirements, may also include further executive capabilities:

- *A configuration manager* This is responsible for the dynamic reconfiguration (Kramer and Magee, 1985) of the system's hardware. Hardware modules may be taken out of service and the system upgraded by adding new hardware without shutting down the system.
- *A fault manager* This component is responsible for detecting hardware and software faults and taking appropriate action to recover from these faults. Principles of fault tolerance and recovery are discussed in Chapter 18.

Stimuli processed by a real-time system usually have different levels of priority. For some stimuli, such as those associated with certain exceptional events, it is essential that their processing should be completed within the specified time limits. Other processes may be safely delayed if a more critical process requires service. Consequently, the executive for a real-time system has to be able to manage at least two priority levels for system processes:

1. *Interrupt level* This is the highest priority level. It is allocated to processes which need a very fast response. One of these processes will be the real-time clock process.
2. *Clock level* This level of priority is allocated to periodic processes.

There may be a further priority level allocated to background processes (such as a self-checking process) which do not need to meet real-time deadlines. These processes are scheduled for execution when processor capacity is available.

Within each of these priority levels, different classes of process may be allocated different priorities. For example, there may be several interrupt lines. An interrupt from a very fast device may have to pre-empt processing of an interrupt from a slower device to avoid information loss. The allocation of process priorities so that all processes are serviced in time usually requires extensive analysis and simulation.

13.2.1 Process management

Process management in a real-time executive is concerned with managing the set of concurrent processes that are part of the real-time system. The process manager has to choose a process for execution, allocate memory and processor resources to that process and start its execution on a processor.

Periodic processes are processes which must be executed at pre-specified time intervals for data acquisition and actuator control. The executive uses its real-time clock to determine when a process is to be executed. In most real-time systems,

When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesiser, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The alarm system is normally powered by mains power but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the mains voltage. It interrupts the alarm system when a voltage drop is detected.

This system is a 'soft' real-time system which does not have stringent timing requirements. The sensors do not need to detect high-speed events so they need only be polled twice per second.

The design process starts by identifying the aperiodic stimuli which the system receives and the associated responses. I have simplified the design by ignoring stimuli generated by system self-checking procedures and external stimuli generated to test the system or to switch it off in the event of a false alarm. This means that there are only two classes of stimulus which must be processed:

1. *Power failure* This is generated by the circuit monitor. The required response is to switch the circuit to backup power by signalling an electronic power switching device.
2. *Intruder alarm* This is a stimulus generated by one of the system sensors. The response to this stimulus is to compute the room number of the active sensor, set up a call to the police, initiate the voice synthesiser to manage the call, and switch on the audible intruder alarm and the building lights in the area.

Figure 13.6 Stimulus/response timing requirements

Stimulus/Response	Timing requirements
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Door alarm	Each door alarm should be polled twice per second.
Window alarm	Each window alarm should be polled twice per second.
Movement detector	Each movement detector should be polled twice per second.
Audible alarm	The audible alarm should be switched on within $\frac{1}{2}$ second of an alarm being raised by a sensor.
Lights switch	The lights should be switched on within $\frac{1}{2}$ second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Voice synthesiser	A synthesised message should be available within 4 seconds of an alarm being raised by a sensor.

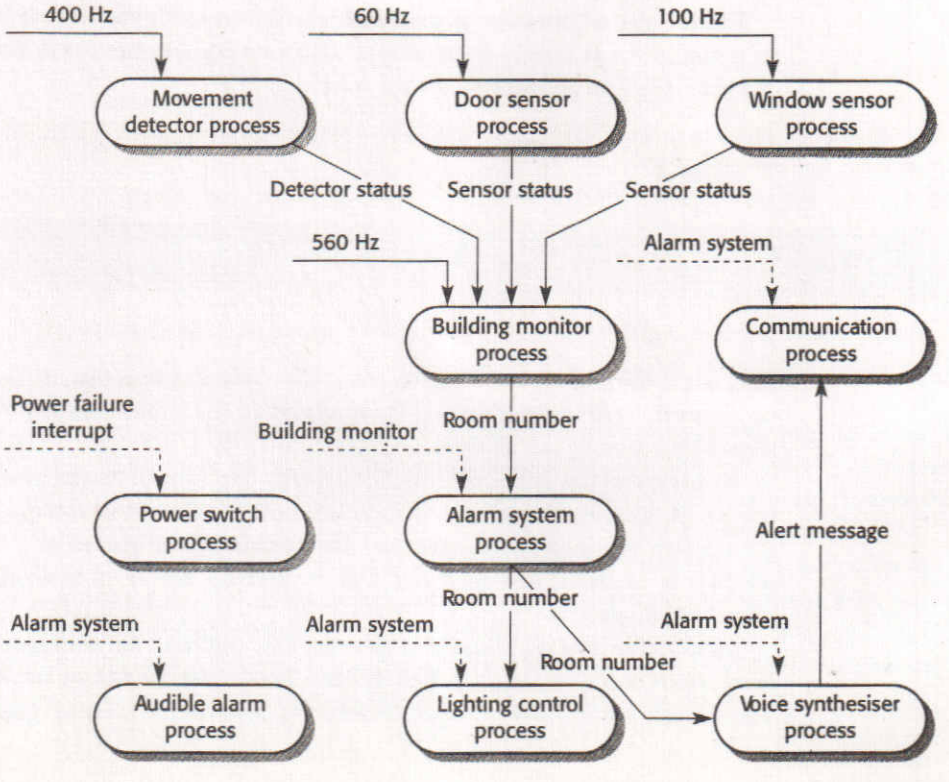
The next step in the design process is to consider the timing constraints associated with each stimulus and associated response. These timing constraints are shown in Figure 13.6. In this diagram, the different classes of sensor that can generate an alarm stimulus have been listed separately as these have different timing requirements.

Allocation of the system functions to concurrent processes is the next design stage. There are three different types of sensor which must be polled periodically, so each of these sensor types has an associated process. There is an interrupt-driven system to handle power failure and switching, a communications system, a voice synthesiser, an audible alarm system and a light switching system to switch on lights around the sensor. Each of these systems is controlled by an independent process. This suggests the system architecture shown in Figure 13.7.

In the notation used in Figure 13.7, annotated arrows joining processes indicate data flows between processes with the annotation indicating the type of data flow. The arrow associated with each process on the top right indicates control. The arrows on a periodic process use solid lines with the minimum number of times a process should be executed per second as an annotation.

Figure 13.7
Process architecture of the burglar alarm system

The rate of period scheduling is determined by the number of sensors and the timing requirements of the system. For example, there are 30 door sensors which



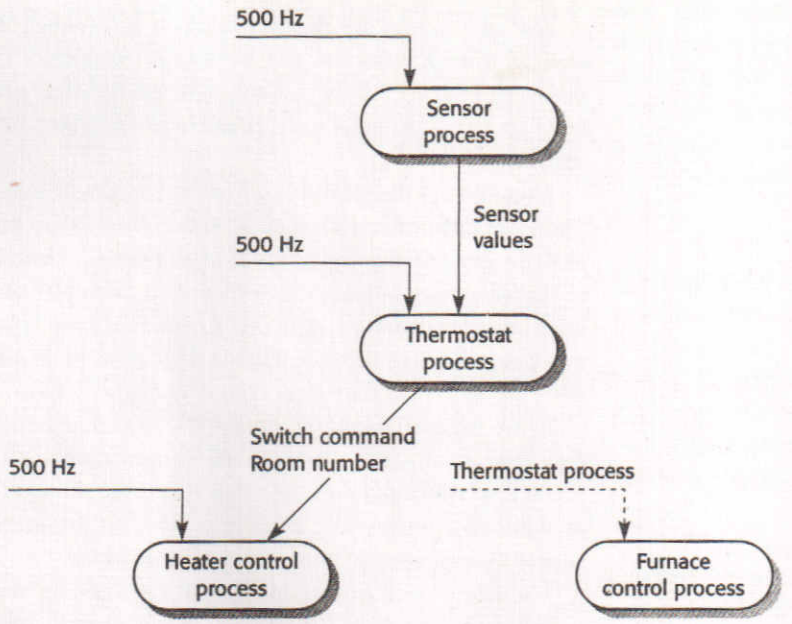


Figure 13.9
Process architecture of a temperature control system

The process architecture of this system is shown in Figure 13.9. It is clear that its general form is similar to the burglar alarm system. Further development of this example is left as an exercise for the reader.

13.4 Data acquisition systems

Data acquisition systems are another class of real-time system that are usually based on a generic architectural model. These systems collect data from sensors for subsequent processing and analysis.

To illustrate this class of system, consider the system model shown in Figure 13.10. This represents a system which collects data from sensors monitoring the neutron flux in a nuclear reactor. The sensor data is placed in a buffer from which it is extracted and processed and the average flux level is displayed on an operator's display.

Each sensor has an associated process that converts the analogue input flux level into a digital signal. It passes this flux level, with the sensor identifier, to the sensor data buffer. The process responsible for data processing takes the data

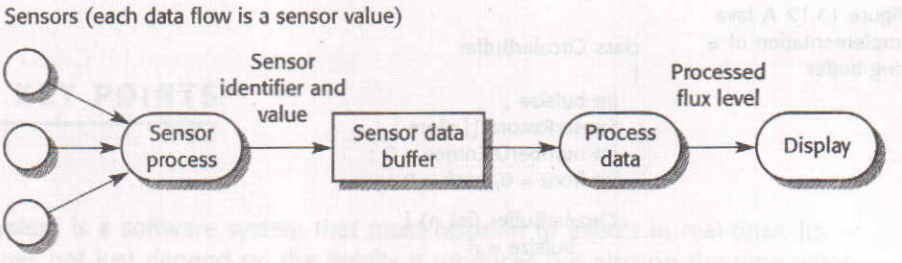


Figure 13.10 The architecture of a flux monitoring system

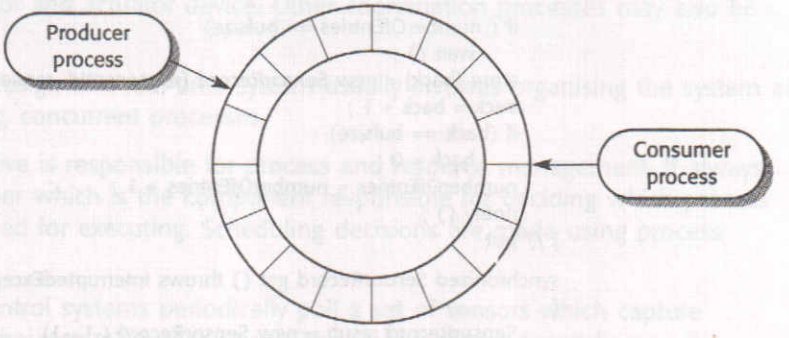


Figure 13.11 A ring buffer for data acquisition

from this buffer, processes it and passes it to a display process for output on an operator console.

In real-time systems that involve data acquisition and processing, the execution speeds and periods of the acquisition process and the processing process may be out of step. When significant processing is required, the data acquisition may go faster than the data processing. If only simple computations need be carried out, the processing may be faster than the data acquisition.

To smooth out these speed differences, most data acquisition systems buffer input data using a circular or ring buffer. The process producing the data (the producer) adds information to this buffer and the process using the data (the consumer) takes information from the buffer (Figure 13.11).

Obviously, mutual exclusion must be implemented to prevent the producer and consumer processes accessing the same element in the buffer at the same time. The system must also ensure that the producer does not try to add information to a full buffer and the consumer does not take information from an empty buffer.

In Figure 13.12 I show a possible implementation of the data buffer as a Java object. The values in the buffer are of type SensorRecord and there are two operations that are defined, namely get and put. The get operation takes an item from the buffer and the put operation adds an item to the buffer. The constructor for the buffer sets the size when objects of type CircularBuffer are declared.

Figure 13.12 A Java implementation of a ring buffer

```
class CircularBuffer
{
  int bufsize ;
  SensorRecord [] store ;
  int numberOfEntries = 0 ;
  int front = 0, back = 0 ;

  CircularBuffer (int n) {
      bufsize = n ;
      store = new SensorRecord [bufsize] ;
  } // CircularBuffer

  synchronized void put (SensorRecord rec ) throws InterruptedException
  {
      if ( numberOfEntries == bufsize)
          wait () ;
      store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
      back = back + 1 ;
      if (back == bufsize)
          back = 0 ;
      numberOfEntries = numberOfEntries + 1 ;
      notify () ;
  } // put

  synchronized SensorRecord get () throws InterruptedException
  {
      SensorRecord result = new SensorRecord (-1, -1) ;
      if (numberOfEntries == 0)
              wait () ;
      result = store [front] ;
      front = front + 1 ;
      if (front == bufsize)
          front = 0 ;
      numberOfEntries = numberOfEntries - 1 ;
      notify () ;
      return result ;
  } // get
} // CircularBuffer
```

The synchronized modifier associated with the get and put methods indicates that these methods should not operate concurrently. When one of these methods is invoked, the run-time system obtains a lock on the object instance to ensure that the other method cannot be invoked at the same time and thus manipulate the same entry in the buffer. The wait and notify method invocations within these methods ensure that entries cannot be put into a full buffer or taken from an empty buffer. The wait method causes the invoking thread to suspend itself until another thread tells it to stop waiting by calling the notify method. When wait is called, the lock on the protected object is released. The notify method wakes up one of the threads that is waiting and causes it to restart execution.

KEY POINTS

- A real-time system is a software system that must respond to events in real time. Its correctness does not just depend on the results it produces but also on the time when these results are produced.
- A general model for real-time systems architecture involves associating a process with each class of sensor and actuator device. Other coordination processes may also be required.
- The architectural design of a real-time system usually involves organising the system as a set of interacting, concurrent processes.
- A real-time executive is responsible for process and resource management. It always includes a scheduler which is the component responsible for deciding which process should be scheduled for executing. Scheduling decisions are made using process priorities.
- Monitoring and control systems periodically poll a set of sensors which capture information from the system's environment. They take actions, depending on the sensor readings, by issuing commands to actuators.
- Data acquisition systems are usually organised according to a producer–consumer model. The producer process puts the data into a circular buffer where it is consumed by the consumer process. The buffer is also implemented as a process so that conflicts between the producer and consumer are eliminated.
- Although Java has facilities for supporting concurrency, it is not suitable for the development of time-critical real-time systems. It lacks facilities to control execution and it is impossible to analyse the timing behaviour of Java programs.

FURTHER READING

Doing Hard Time: Developing Real-time Systems with UML, Objects Frameworks and Patterns. This book discusses how object-oriented techniques can be used in the design of real-time systems. As hardware speeds increase, this is becoming an increasingly viable approach to real-time systems design. (B. P. Douglass, 1999, Addison-Wesley.)

'Adding real-time capabilities to Java'. A good description of why Java is not suited to real-time systems development. (K. Nilsen, *Comm. ACM,* 41(6), June 1998.)

Real-time Systems and Programming Languages, 2nd edition. An excellent and comprehensive text that provides broad coverage of all aspects of real-time systems. (A. Burns and A. Wellings, 1997, Addison-Wesley.)

Advances in Real-time Systems. This is an excellent IEEE tutorial volume that covers various aspects of real-time systems design and implementation. Some of the chapters assume that the reader is familiar with the subject area but others are good introductions. (J. A. Stankovic and K. Ramamritham, 1993, IEEE Press.)

EXERCISES

13.1 Using examples, explain why real-time systems usually have to be implemented using concurrent processes.

13.2 Explain why an object-oriented approach to software development may not be suitable for real-time systems.

13.3 Draw state machine models of the control software for the following systems:

- An automatic washing machine which has different programs for different types of clothes.
- The software for a compact disc player.
- A telephone answering machine which records incoming messages and displays the number of accepted messages on an LED display. The system should allow the telephone owner to dial in, type a sequence of numbers (identified as tones) and have the recorded messages replayed over the phone.
- A drinks vending machine which can dispense coffee with and without milk and sugar. The user deposits a coin and makes his or her selection by pressing a button on the machine. This causes a cup with powdered coffee to be output. The user places this cup under a tap, presses another button and hot water is dispensed.

13.4 Using the real-time system design techniques discussed in this chapter, redesign the weather station data collection system covered in Chapter 12 as a stimulus/response system.

13.5 Design a process architecture for an environmental monitoring system which collects data from a set of air quality sensors situated around a city. There are 5000 sensors organised into 100 neighbourhoods. Each sensor must be interrogated 4 times per second. When more than 30 per cent of the sensors in a particular neighbourhood indicate that the air quality is below an acceptable level, local warning lights are activated. All sensors return the readings to a central computer which generates/reports every 15 minutes on the air quality in the city.

13.6 Discuss the strengths and weaknesses of Java as a programming language for real-time systems.

Figure 13.13 Train protection system description

- The system acquires information on the speed limit of a segment from a trackside transmitter which continually broadcasts the segment identifier and its speed limit. The same transmitter also broadcasts information on the status of the signal controlling that track segment. The time required to broadcast track segment and signal information is 50 milliseconds.
- The train can receive information from the trackside transmitter when it is within 10 m of a transmitter.
- The maximum train speed is 180 kph.
- Sensors on the train provide information about the current train speed (updated every 250 milliseconds) and the train brake status (updated every 100 milliseconds).
- If the train speed exceeds the current segment speed limit by more than 5 kph, a warning is sounded in the driver's cabin. If the train speed exceeds the current segment speed limit by more than 10 kph, the train's brakes are automatically applied until the speed falls to the segment speed limit. Train brakes should be applied within 100 milliseconds of the time when the excessive train speed has been detected.
- If the train enters a track segment which is signalled with a red light, the train protection system applies the train brakes and reduces the speed to zero. Train brakes should be applied within 100 milliseconds of the time when the red light signal is received.
- The system continually updates a status display in the driver's cabin.

13.7 A train protection system automatically applies the brakes of a train if the speed limit for a segment of track is exceeded or if the train enters a track segment which is currently signalled with a red light (i.e. the segment should not be entered). Details are shown in Figure 13.13. Identify the stimuli that must be processed by the on-board train control system and the associated responses to these stimuli.

13.8 Suggest a possible process architecture for this system.

13.9 If a periodic process in the on-board train protection system is used to collect data from the trackside transmitter, how often must it be scheduled to ensure that the system is guaranteed to collect information from the transmitter? Explain how you arrived at your answer.

13.10 You are asked to work on a real-time development project for a military application but have no previous experience of projects in that domain. Discuss what you, as a professional software engineer, should do before starting work on the project.