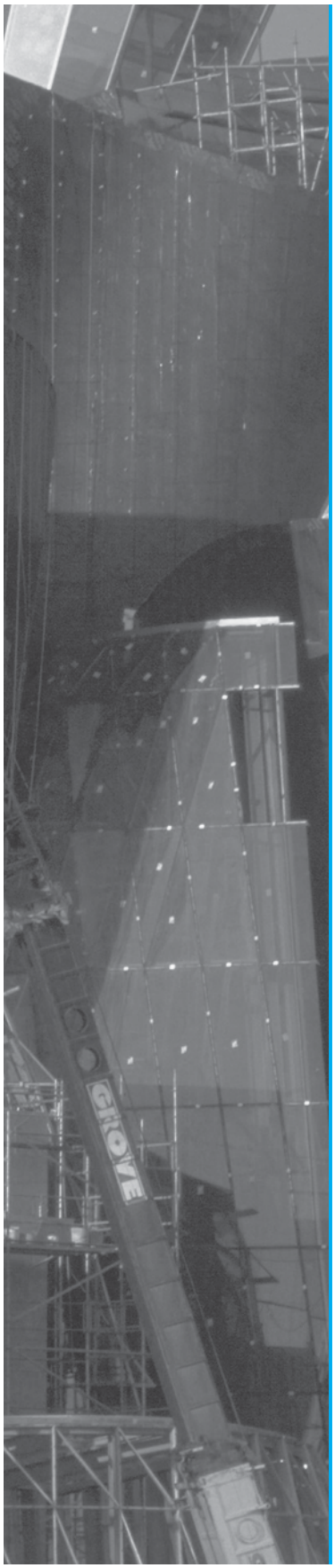


20

Embedded software

Objectives

The objective of this chapter is to introduce some of the characteristic features of embedded real-time systems and real-time software engineering. When you have read this chapter, you will:

- understand the concept of embedded software, which is used to control systems that must react to external events in their environment;
- have been introduced to a design process for real-time systems, where the software systems are organized as a set of cooperating processes;
- understand three architectural patterns that are commonly used in embedded real-time systems design;
- understand the organization of real-time operating systems and the role that they play in an embedded, real-time system.

Contents

Computers are used to control a wide range of systems from simple domestic machines, through games controllers, to entire manufacturing plants. These computers interact directly with hardware devices. Their software must react to events generated by the hardware and, often, issue control signals in response to these events. These signals result in an action, such as the initiation of a phone call, the movement of a character on the screen, the opening of a valve, or the display of the system status. The software in these systems is embedded in system hardware, often in read-only memory, and usually responds, in real time, to events from the system's environment. By real time, I mean that the software system has a deadline for responding to external events. If this deadline is missed, then the overall hardware–software system will not operate correctly.

Embedded software is very important economically because almost every electrical device now includes software. There are therefore many more embedded software systems than other types of software system. If you look around your house you may have three or four personal computers. But you probably have 20 or 30 embedded systems, such as systems in phones, cookers, microwaves, etc.

Responsiveness in real time is the critical difference between embedded systems and other software systems, such as information systems, web-based systems, or personal software systems, whose main purpose is data processing. For non-real-time systems, the correctness of a system can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system. In response to an input, a corresponding output should be generated by the system and, often, some data should be stored. For example, if you choose a create command in a patient information system, then the correct system response is to create a new patient record in a database, and to confirm that this has been done. Within reasonable limits, it does not matter how long this takes.

However, in a real-time system, the correctness depends both on the response to an input and the time taken to generate that response. If the system takes too long to respond, then the required response may be ineffective. For example, if embedded software controlling a car braking system is too slow, then an accident may occur because it is impossible to stop the car in time.

Therefore, time is inherent in the definition of a real-time software system:

> *A real-time software system is a system whose correct operation depends on both 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. If results are not produced according to the timing specification in a 'hard real-time system', this is considered to be a system failure.*

Timely response is an important factor in all embedded systems but not all embedded systems require a very fast response. For example, the insulin pump software that I have used as an example in several chapters of this book is an embedded system. However, although it needs to check the glucose level at periodic intervals, it does not need to respond very quickly to external events. The wilderness weather

station software is also an embedded system but, again, it does not require a fast response to external events.

As well as the need for real-time response, there are other important differences between embedded systems and other types of software system:

1. Embedded systems generally run continuously and do not terminate. They start when the hardware is switched on and must execute until the hardware is switched off. This means that techniques for reliable software engineering, as discussed in Chapter 13, may have to be used to ensure continuous operation. The real-time system may include update mechanisms that support dynamic reconfiguration so that the system can be updated while it is in service.
2. Interactions with the system's environment are uncontrollable and unpredictable. In interactive systems, the pace of the interaction is controlled by the system and, by limiting user options, the events to be processed are known in advance. By contrast, real-time embedded systems must be able to respond to unexpected events at any time. This leads to a design for real-time systems based on concurrency, with several processes executing in parallel.
3. There may be physical limitations that affect the design of a system. Examples of these include limitations on the power available to the system and on the physical space taken up by the hardware. These limitations may generate requirements for the embedded software, such as the need to conserve power and so prolong battery life. Size and weight limitations may mean that the software has to take over some hardware functions because of the need to limit the number of chips used in the system.
4. Direct hardware interaction may be necessary. In interactive systems and information systems, there is a layer of software (the device drivers) that hides the hardware from the operating system. This is possible because you can only connect a few types of device to these systems, such as keyboards, mice, displays, etc. By contrast, embedded systems may have to interact with a wide range of hardware devices that do not have separate device drivers.
5. Issues of safety and reliability may dominate the system design. Many embedded systems control devices whose failure may have high human or economic costs. Therefore, dependability is critical and the system design has to ensure safety-critical behavior at all times. This often leads to a conservative approach to design where tried and tested techniques are used instead of newer techniques that may introduce new failure modes.

Embedded systems can be thought of as reactive systems; that is, they must react to events in their environment at the speed of that environment (Berry, 1989; Lee, 2002). Response times are often governed by the laws of physics rather than chosen for human convenience. This is in contrast to other types of software where the system controls the speed of the interaction. For example, the word processor that I am

using to write this book can check spelling and grammar and there are no practical limits on the time taken to do this.

20.1 Embedded systems design

The design process for embedded systems is a systems engineering process in which the software designers have to consider in detail the design and performance of the system hardware. Part of the system design process may involve deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems embedded in consumer products, such as the systems in cell phones, the costs, and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used and, for some systems, special-purpose hardware may have to be designed and built.

This means that a top-down software design process, in which the design starts with an abstract model that is decomposed and developed in a series of stages, is impractical for most real-time systems. Low-level decisions on hardware, support software, and system timing must be considered early in the process. These limit the flexibility of system designers and may mean that additional software functionality, such as battery and power management, has to be included in the system.

Given that embedded systems are reactive systems that react to events in their environment, the most general approach to embedded, real-time software design is based on a stimulus-response model. A stimulus is an event occurring in the software system's environment that causes the system to react in some way; a response is a signal or message that is sent by the software to its environment.

You can define the behavior of a real-time system by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. For example, Figure 20.1 shows possible stimuli and system responses for a burglar alarm system. I give more information about this system in Section 20.2.1.

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 and unpredictably and are usually signaled 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.

Stimuli come from sensors in the system's environment and responses are sent to actuators, as shown in Figure 20.2. A general design guideline for real-time systems

Stimulus	Response
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police; run power supply test.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.

Figure 20.1 Stimuli and responses for a burglar alarm system

is to have separate processes for each type of sensor and actuator (Figure 20.3). These actuators control equipment, such as a pump, which then makes changes to the system's environment. The actuators themselves may also generate stimuli. The stimuli from actuators often indicate that some problem has occurred, which must be handled by the system.

For each type of sensor, there may be a sensor management process that handles data collection from the sensors. Data processing processes compute the required responses for the stimuli received by the system. Actuator control processes are associated with

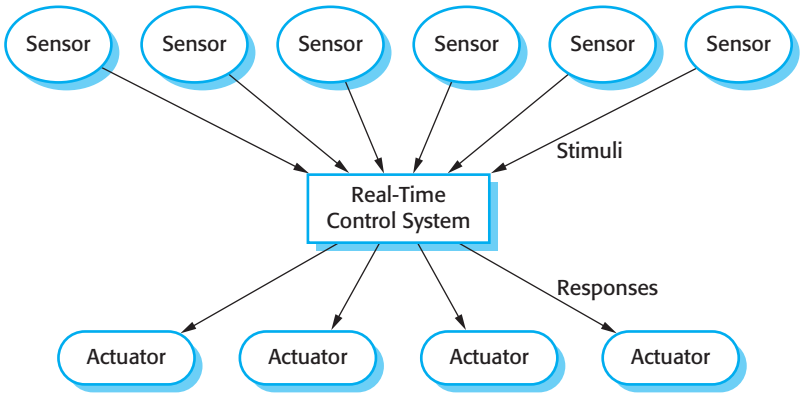


Figure 20.2 A general model of an embedded real-time system

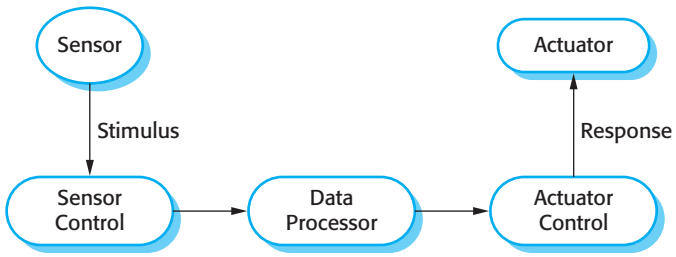


Figure 20.3 Sensor and actuator processes

each actuator and manage the operation of that actuator. This model allows data to be collected quickly from the sensor (before it is overwritten by the next input) and allows processing and the associated actuator response to be carried out later.

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize the system architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time software systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform on which the real-time system executes may include a real-time operating system (discussed in Section 20.4). The functions provided by this operating system are accessed through the run-time support system for the real-time programming language that is used.

There is no standard embedded system design process. Rather, different processes are used that depend on the type of system, available hardware, and the organization that is developing the system. The following activities may be included in a real-time software design process:

1. *Platform selection* In this activity, you choose an execution platform for the system (i.e., the hardware and the real-time operating system to be used). Factors that influence these choices include the timing constraints on the system, limitations on power available, the experience of the development team, and the price target for the delivered system.
2. *Stimuli/response identification* This involves identifying the stimuli that the system must process and the associated response or responses for each stimulus.
3. *Timing analysis* For each stimulus and associated response, you identify the timing constraints that apply to both stimulus and response processing. These are used to establish the deadlines for the processes in the system.
4. *Process design* At this stage, you aggregate the stimulus and response processing into a number of concurrent processes. A good starting point for designing the process architecture is the architectural patterns that I describe in Section 20.2. You then optimize the process architecture to reflect the specific requirements that you have to implement.
5. *Algorithm design* For each stimulus and response, you design algorithms to carry out the required computations. Algorithm designs may have to be

developed relatively early in the design process to give an indication of the amount of processing required and the time needed to complete that processing. This is especially important for computationally intensive tasks, such as signal processing.

6. *Data design* You specify the information that is exchanged by processes and the events that coordinate information exchange, and design data structures to manage this information exchange. Several concurrent processes may share these data structures.

7. *Process scheduling* You design a scheduling system that will ensure that processes are started in time to meet their deadlines.

The order of these activities in the real-time software design process depends on the type of system being developed, as well as its process and platform requirements. In some cases, you may be able to follow a fairly abstract approach where you start with the stimuli and associated processing, and decide on the hardware and execution platforms late in the process. In other cases, the choice of hardware and operating system is made before the software design starts. In such a situation, you have to design the software to take account of the constraints imposed by the hardware capabilities.

Processes in a real-time system have to be coordinated and share information. 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, 1968), monitors (Hoare, 1974), and critical regions (Brinch-Hansen, 1973). These process synchronization mechanisms are described in most operating system texts (Silberschatz et al., 2008; Tanenbaum, 2007).

When designing the information exchange between processes, you have to take into account the fact that these processes may be running at different speeds. One process is producing information; the other process is consuming that information. If the producer is running faster than the consumer, new information could overwrite a previously read information item before the consumer process has read the original information. If the consumer process is running faster than the producer process, the same item could be read twice.

To get around this problem, you should implement information exchange using a shared buffer and use mutual exclusion mechanisms to control access to that buffer. This means that information can't be overwritten before it has been read and that information cannot be read twice. Figure 20.4 illustrates the notion of a shared buffer. This is usually implemented as a circular queue, so that mismatches in speed between the producer and consumer processes can be accommodated without having to delay process execution.

The producer process always enters data in the buffer location at the tail of the queue (represented as v10 in Figure 20.4). The consumer process always retrieves information from the head of the queue (represented as v1 in Figure 20.4). After the consumer process has retrieved the information, the head of the list is adjusted to point at the next item (v2). After the producer process has added information, the tail of the list is adjusted to point at the next free slot in the list.

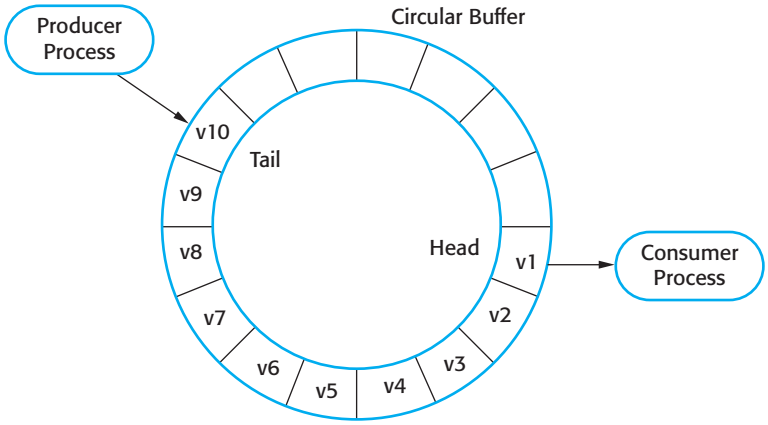


Figure 20.4 Producer/consumer processes sharing a circular buffer

Obviously, it is important to ensure that the producer and consumer process do not attempt to access the same item at the same time (i.e., when Head = Tail). You also have to ensure that the producer process does not add items to a full buffer and that the consumer process does not take items from an empty buffer. To do this, you implement the circular buffer as a process with Get and Put operations to access the buffer. The Put operation is called by the producer process and the Get operation by the consumer process. Synchronization primitives, such as semaphores or critical regions, are used to ensure that the operation of Get and Put are synchronized, so that they don't access the same location at the same time. If the buffer is full, the Put process has to wait until a slot is free; if the buffer is empty, the Get process has to wait until an entry has been made.

Once you have chosen the execution platform for the system, designed a process architecture, and decided on a scheduling policy, you may need to check that the system will meet its timing requirements. You can do this through static analysis of the system using knowledge of the timing behavior of components, or through simulation. This analysis may reveal that the system will not perform adequately. The process architecture, the scheduling policy, the execution platform, or all of these may then have to be redesigned to improve the performance of the system.

Timing constraints or other requirements may sometimes mean that it is best to implement some system functions, such as signal processing, in hardware. Modern hardware components, such as FPGAs, are flexible and can be adapted to different functions. Hardware components deliver much better performance than the equivalent software. System processing bottlenecks can be identified and replaced by hardware, thus avoiding expensive software optimization.

20.1.1 Real-time system modeling

The events that a real-time system must react to often cause the system to move from one state to another. For this reason, state models, which I introduced in Chapter 5, are often used to describe real-time systems. A state model of a system assumes that,

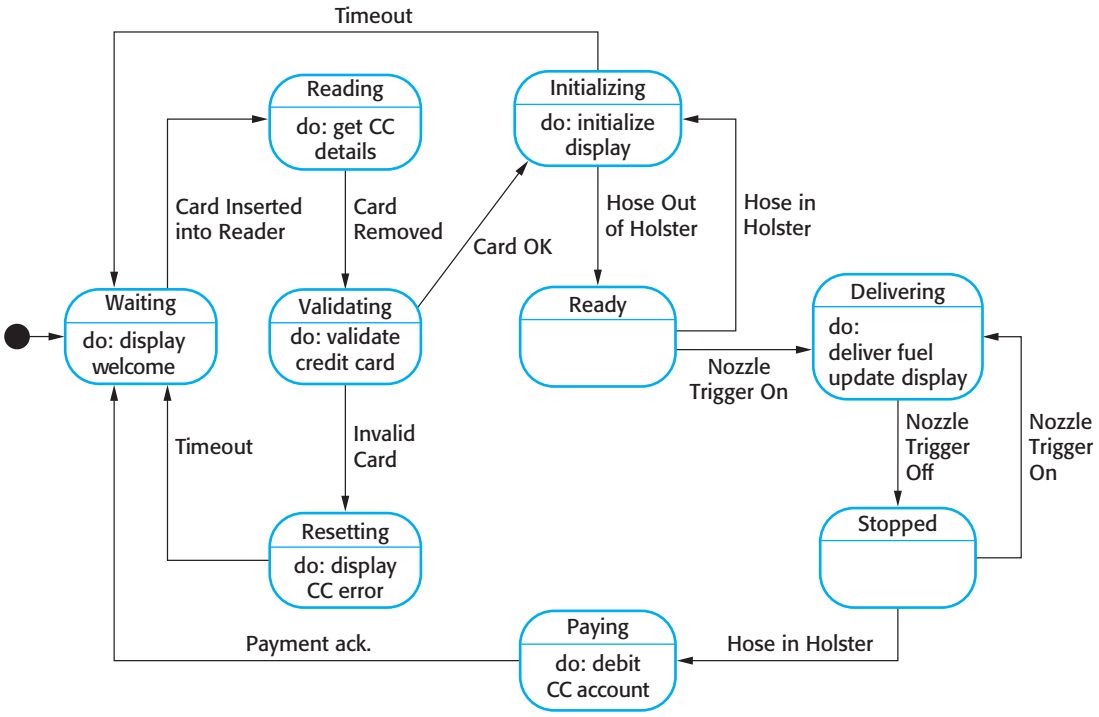


Figure 20.5 State machine model of a petrol (gas) pump

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.

State models are a language-independent way of representing the design of a real-time system and are therefore an integral part of real-time system design methods (Gomaa, 1993). The UML supports the development of state models based on Statecharts (Harel, 1987; Harel, 1988). Statecharts are formal state machine models that support hierarchical states, so that groups of states can be considered as a single entity. Douglass discusses the use of the UML in real-time systems development (Douglass, 1999). State models are used in model-driven engineering, which I discussed in Chapter 5, to define the operation of a system. They can be transformed automatically to an executable program.

I have already illustrated this approach to system modeling in Chapter 5 where I used an example of a model of a simple microwave oven. Figure 20.5 is another example of a state machine model that shows the operation of a fuel delivery software system embedded in a petrol (gas) pump. The rounded rectangles represent system states and the arrows represent stimuli that force a transition from one state to another. The names chosen in the state machine diagram are descriptive. The associated information indicates actions taken by the system actuators or information that is displayed. Notice that this system never terminates but idles in a waiting state when the pump is not operating.

The fuel delivery system is designed to allow unattended operation. The buyer inserts a credit card into a card reader built into the pump. This causes a transition to a Reading state where the card details are read and the buyer is then asked to remove the card. Removal of the card triggers a transition to a Validating state where the card is validated. If the card is valid, the system initializes the pump and, when the fuel hose is removed from its holster, transitions to the Delivering state, where it is ready to deliver fuel. Activating the trigger on the nozzle causes fuel to be pumped; this stops when the trigger is released (for simplicity, I have ignored the pressure switch that is designed to stop fuel spillage). After the fuel delivery is complete and the buyer has replaced the hose in its holster, the system moves to a Paying state where the user's account is debited. After payment, the pump software returns to the Waiting state.

20.1.2 Real-time programming

Programming languages for real-time systems development have to include facilities to access system hardware, and it should be possible to predict the timing of particular operations in these languages. Hard real-time systems are still sometimes programmed in assembly language so that tight deadlines can be met. Systems-level languages, such as C, which allow efficient code to be generated are also widely used.

The advantage of using a systems programming language like C is that it allows the development of very efficient programs. However, these languages do not include constructs to support concurrency or the management of shared resources. Concurrency and resource management are implemented through calls to primitives provided by the real-time operating system, such as semaphores for mutual exclusion. These calls cannot be checked by the compiler, so programming errors are more likely. Programs are also often more difficult to understand because the language does not include real-time features. As well as understanding the program, the reader also has to know how real-time support is provided using system calls.

Because real-time systems must meet their timing constraints, you may not be able to use object-oriented development for hard real-time systems. Object-oriented development involves hiding data representations and accessing attribute values through operations defined with the object. This means that there is a significant performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations. The consequent loss of performance may make it impossible to meet real-time deadlines.

A version of Java has been designed for embedded systems development (Dibble, 2008), with implementations from different companies such as IBM and Sun. This language includes a modified thread mechanism, which allows threads to be specified that will not be interrupted by the language garbage collection mechanism. Asynchronous event handling and timing specification has also been included. However, at the time of writing, this has mostly been used on platforms that have significant processor and memory capacity (e.g., a cell phone) rather than simpler embedded systems, with more limited resources. These systems are still usually implemented in C.

Real-time Java

The Java programming language has been modified in a number of ways to make it suitable for real-time systems development. These modifications include asynchronous communications; the addition of time, including absolute and relative time; a new thread model where threads cannot be interrupted by garbage collection; and a new memory management model that avoids the unpredictable delays that can result from garbage collection.

http://www.SoftwareEngineering-9.com/Web/RTS/Java.html

20.2 Architectural patterns

Architectural patterns, which I introduced in Chapter 6, are abstract, stylized descriptions of good design practice. They encapsulate knowledge about the organization of system architectures, when these architectures should be used and their advantages and disadvantages. You should not, however, think of an architectural pattern as a generic design to be instantiated. Rather, you use the pattern to understand an architecture and as starting point for creating your own specific architectural design.

As you might expect, the differences between embedded and interactive software means that different architectural patterns are used for embedded systems, rather than the architectural patterns discussed in Chapter 6. Embedded systems' patterns are process-oriented rather than object- or component-oriented. In this section, I discuss three real-time architectural patterns that are commonly used:

1. *Observe and React* This pattern is used when a set of sensors are routinely monitored and displayed. When the sensors show that some event has occurred (e.g., an incoming call on a cell phone), the system reacts by initiating a process to handle that event.
2. *Environmental Control* This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment. In response to environmental changes detected by the sensor, control signals are sent to the system actuators.
3. *Process Pipeline* This pattern is used when data has to be transformed from one representation to another before it can be processed. The transformation is implemented as a sequence of processing steps, which may be carried out concurrently. This allows for very fast data processing, because a separate core or processor can execute each transformation.

These patterns can of course be combined and you will often see more than one of them in a single system. For example, when the Environmental Control pattern is used, it is very common for the actuators to be monitored using the Observe and React pattern. In the event of an actuator failure, the system may

Name	Observe and React
Description	The input values of a set of sensors of the same types are collected and analyzed. These values are displayed in some way. If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, in certain cases, to take actions in response to the exceptional value.
Stimuli	Values from sensors attached to the system.
Responses	Outputs to display, alarm triggers, signals to reacting systems.
Processes	Observer, Analysis, Display, Alarm, Reactor.
Used in	Monitoring systems, alarm systems.

Figure 20.6 The Observe and React pattern

react by displaying a warning message, shutting down the actuator, switching in a backup system, etc.

The patterns that I discuss here are architectural patterns that describe the overall structure of an embedded system. Douglass (2002) describes lower-level, real-time design patterns that are used to help you make more detailed design decisions. These patterns include design patterns for execution control, communications, resource allocation, and safety and reliability.

These architectural patterns should be the starting point for an embedded systems design; however they are not design templates. If you use them as such, you will probably end up with an inefficient process architecture. You therefore have to optimize the process structure to ensure that you do not have too many processes. You also should ensure that there is a clear correspondence between the processes and the sensors and actuators in the system.

20.2.1 Observe and React

Monitoring systems are an important class of embedded real-time systems. A monitoring system examines its environment through a set of sensors and, usually, displays the state of the environment in some way. This could be on a built-in screen, on special-purpose instrument displays or on a remote display. If some exceptional event or sensor state is detected by the system, the monitoring system takes some action. Often, this involves raising an alarm to draw an operator's attention to the event. Sometimes the system may initiate some other preventative action, such as shutting down the system to preserve it from damage.

The Observe and React pattern (Figure 20.6 and Figure 20.7) is a pattern that is commonly used in monitoring systems. The values of sensors are observed and when

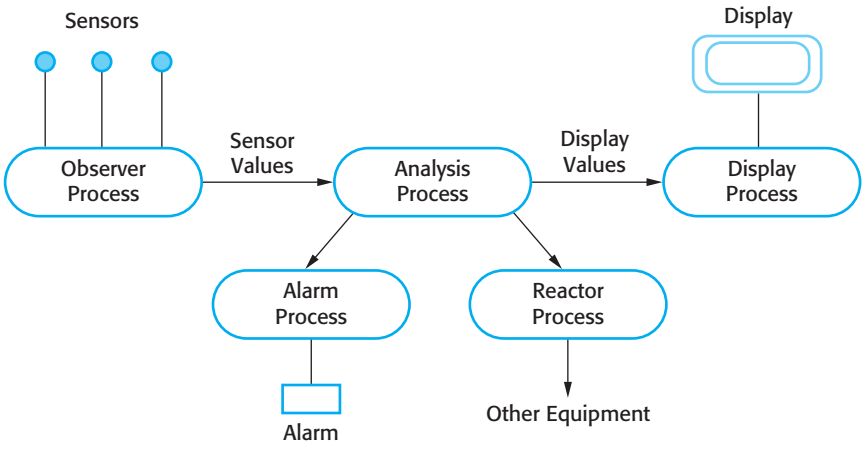


Figure 20.7 Observe and React process structure

particular values are detected, the system reacts in some way. Monitoring systems may be composed of several instantiations of the Observe and React pattern, one for each type of sensor in the system. Depending on the system requirements, you may then optimize the design by combining processes (e.g., you may use a single display process to display the information from all of the different types of sensors).

As an example of the use of this pattern, consider the design of a burglar alarm system that might be installed in an office building:

> *A software system is to be implemented as part of a burglar alarm system for commercial buildings. This uses several different types of sensors. These include movement detectors in individual rooms, door sensors that detect corridor doors opening, and window sensors on ground-floor windows that can detect when a window has been opened.*
>
> *When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The sensor 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. If a voltage drop is detected, the system assumes that intruders have interrupted the power supply so an alarm is raised.*

A possible process architecture for the alarm system is shown in Figure 20.8. In this diagram, the arrows represent signals sent from one process to another. This system is a 'soft' real-time system that does not have stringent timing requirements. The sensors do not need to detect high-speed events, so they need only be polled relatively infrequently. The timing requirements for this system are covered in Section 20.3.

I have already introduced the stimuli and responses in this alarm system in Figure 20.1. These are used as a starting point for the system design. The Observe

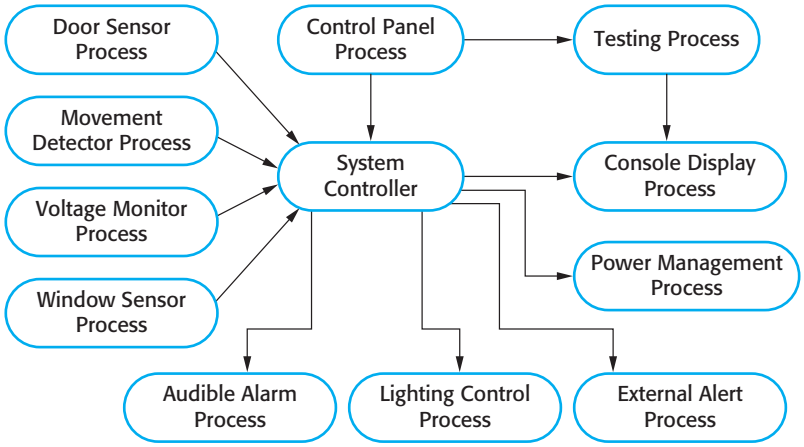


Figure 20.8 Process structure for a burglar alarm system

and React pattern is used in this design. There are observer processes associated with each type of sensor and reactor processes for each type of reaction. There is a single analysis process that checks the data from all of the sensors. The display processes in the pattern are combined into a single display process.

20.2.2 Environmental Control

Perhaps the most widespread use of embedded software is in control systems. In these systems, the software controls the operation of equipment, based on stimuli from the equipment's environment. For example, an anti-skid braking system in a car monitors the car's wheels and brake system (the system's environment). It looks for signs that the wheels are skidding when brake pressure is applied. If this is the case, the system adjusts the brake pressure to stop the wheels locking and reduce the likelihood of a skid.

Control systems may make use of the Environmental Control pattern, which is a general control pattern that includes sensor and actuator processes. This pattern is described in Figure 20.9, with the process architecture shown in Figure 20.10. A variant of this pattern leaves out the display process. This variant is used in situations where there is no requirement for user intervention or where the rate of control is so high that a display would not be meaningful.

This pattern can be the basis for a control system design with an instantiation of the Environmental Control pattern for each actuator (or actuator type) that is being controlled. You then optimize the design to reduce the number of processes. For example, you may combine actuator monitoring and actuator control processes, or may have a single monitoring and control process for several actuators. The optimizations that you choose depend on the timing requirements. You may need to monitor sensors more frequently than you send control signals, in which case it may be impractical to combine control and monitoring processes. There may also be

Name	Environmental Control
Description	The system analyzes information from a set of sensors that collect data from the system's environment. Further information may also be collected on the state of the actuators that are connected to the system. Based on the data from the sensors and actuators, control signals are sent to the actuators that then cause changes to the system's environment. Information about the sensor values and the state of the actuators may be displayed.
Stimuli	Values from sensors attached to the system and the state of the system actuators.
Responses	Control signals to actuators, display information.
Processes	Monitor, Control, Display, Actuator Driver, Actuator monitor.
Used in	Control systems.

Figure 20.9 The Environmental Control pattern

direct feedback between the actuator control and the actuator monitoring process. This allows fine-grain control decisions to be made by the actuator control process.

You can see how this pattern is used in Figure 20.11, which shows an example of a controller for a car braking system. The starting point for the design is associating an instance of the pattern with each actuator type in the system. In this case, there are four actuators, with each controlling the brake on one wheel. The individual sensor processes are combined into a single wheel-monitoring process that monitors the sensors on all wheels. This monitors the state of each wheel to check if the wheel is

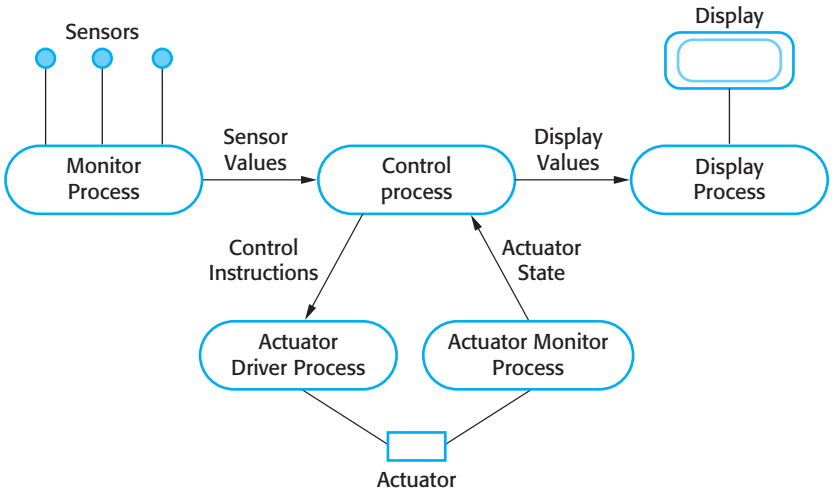


Figure 20.10 Environmental Control process structure

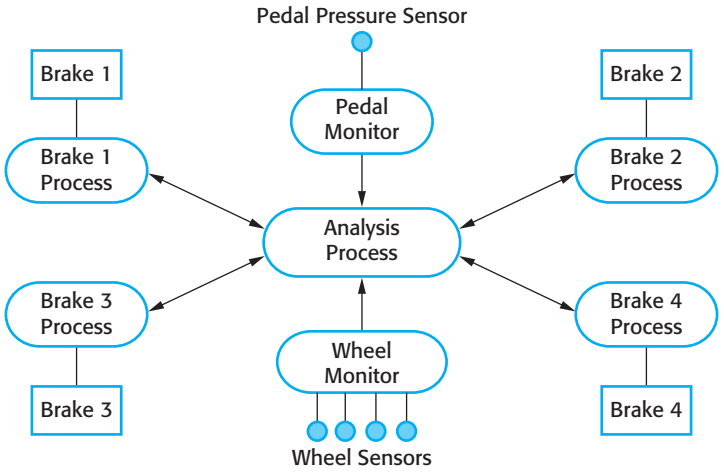


Figure 20.11 Control system architecture for an anti-skid braking system

turning or locked. A separate process monitors the pressure on the brake pedal exerted by the car driver.

The system includes an anti-skid feature, which is triggered if the sensors indicate that a wheel is locked when the brake has been applied. This means that there is insufficient friction between the road and the tyre; in other words, the car is skidding. If the wheel is locked, the driver cannot steer that wheel. To counteract this, the system sends a rapid sequence of on/off signals to the brake on that wheel, which allows the wheel to turn and control to be regained.

20.2.3 Process Pipeline

Many real-time systems are concerned with collecting data from the system's environment, then transforming that data from its original representation into some other digital representation that can be more readily analyzed and processed by the system. The system may also convert digital data to analog data, which it then sends to its environment. For example, a software radio accepts incoming packets of digital data representing the radio transmission and transforms these into a sound signal that people can listen to.

The data processing that is involved in many of these systems has to be carried out very quickly. Otherwise, incoming data may be lost and outgoing signals may be broken up because essential information is missing. The Process Pipeline pattern makes this rapid processing possible by breaking down the required data processing into a sequence of separate transformations, with each transformation carried out by an independent process. This is a very efficient architecture for systems that use multiple processors or multicore processors. Each process in the pipeline can be associated with a separate processor or core, so that the processing steps can be carried out in parallel.

Name	Process Pipeline
Description	A pipeline of processes is set up with data moving in sequence from one end of the pipeline to another. The processes are often linked by synchronized buffers to allow the producer and consumer processes to run at different speeds. The culmination of a pipeline may be display or data storage or the pipeline may terminate in an actuator.
Stimuli	Input values from the environment or some other process
Responses	Output values to the environment or a shared buffer
Processes	Producer, Buffer, Consumer
Used in	Data acquisition systems, multimedia systems

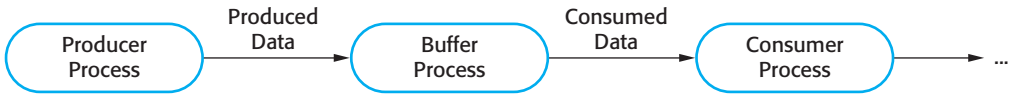
Figure 20.12 The Process Pipeline pattern

Figure 20.12 is a brief description of the data pipeline pattern, and Figure 20.13 shows the process architecture for this pattern. Notice that the processes involved may produce and consume information. They are linked by synchronized buffers, as discussed in Section 20.1. This allows producer and consumer processes to operate at different speeds without data losses.

An example of a system that may use a process pipeline is a high-speed data acquisition system. Data acquisition systems collect data from sensors for subsequent processing and analysis. These systems are used in situations where the sensors are collecting a lot of data from the system's environment and it isn't possible or necessary to process that data in real time. Rather, it is collected and stored for later analysis. Data acquisition systems are often used in scientific experiments and process control systems where physical processes, such as chemical reactions, are very rapid. In these systems, the sensors may be generating data very quickly and the data acquisition system has to ensure that a sensor reading is collected before the sensor value changes.

Figure 20.14 is a simplified model of a data acquisition system than might be part of the control software in a nuclear reactor. This is a system that collects data from sensors monitoring the neutron flux (the density of neutrons) in the reactor. The sensor data is placed in a buffer from which it is extracted and processed. The average flux level is displayed on an operator's display and stored for future processing.

Figure 20.13 Process Pipeline process structure



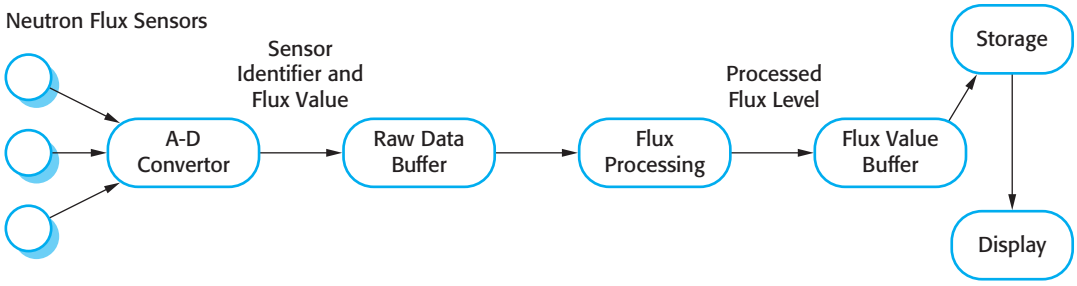


Figure 20.14 Neutron flux data acquisition

20.3 Timing analysis

As I discussed in the introduction, the correctness of a real-time system depends not just on the correctness of its outputs but also on the time at which these outputs were produced. This means that an important activity in the embedded, real-time software development process is timing analysis. In such an analysis, you calculate how often each process in the system must be executed to ensure that all inputs are processed and all system responses are produced in a timely way. The results of the timing analysis are used to decide how frequently each process should execute and how these processes should be scheduled by the real-time operating system.

Timing analysis for real-time systems is particularly difficult when the systems must deal with a mixture of periodic and aperiodic stimuli and responses. Because aperiodic stimuli are unpredictable, you have to make assumptions about 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. Cooling's book (2003) discusses techniques for real-time system performance analysis that takes aperiodic events into account.

However, as computers have become faster it has become possible, in many systems, to design using only periodic stimuli. When processors were slow, aperiodic stimuli had to be used to ensure that critical events were processed before their deadline, as delays in processing usually involved some loss to the system. For example, the failure of a power supply in an embedded system may mean that the system has to shut down attached equipment in a controlled way, within a very short time (say 50 milliseconds). This could be implemented as a 'power fail' interrupt. However, it can also be implemented using a periodic process that runs very frequently and checks the power. So long as the time between process invocations is short, there is still time to perform a controlled shutdown of the system before the lack of power causes damage. For this reason, I focus on timing issues for periodic processes.

When you are analyzing the timing requirements of embedded real-time systems and designing systems to meet these requirements, there are three key factors that you have to consider:

1. *Deadlines* The times by which stimuli must be processed and some response produced by the system. If the system does not meet a deadline then, if it is a

hard-real time system, this is a system failure; in a soft real-time system, it results in degraded system service.

2. *Frequency* The number of times per second that a process must execute so that you are confident that it can always meet its deadlines.

3. *Execution time* The time required to process a stimulus and produce a response. Often, you have to take two execution times into account—the average execution time of a process and the worst-case execution time for that process. Execution time is not always the same because of the conditional execution of code, delays waiting for other processes, etc. In a hard real-time system, you may have to make assumptions based on the worst-case execution time to ensure that deadlines are not missed. In soft real-time systems, you may be able to base your calculations on the average execution time.

To continue the example of a power supply failure, let's assume that, after a failure event, it takes 50 ms for the supplied voltage to drop to a level where the equipment may be damaged. Therefore, the equipment shutdown process must begin within 50 ms of a power failure event. In such cases, it would be prudent to set a shorter deadline of 40 ms, because of physical variations in the equipment. This means that shutdown instructions for all attached equipment that is at risk must be issued and processed within 40 ms, assuming that the equipment is also dependent on the failing power supply.

If you detect power failure by monitoring a voltage level, you have to make more than one observation to detect that the voltage is dropping. If you run the process 250 times per second, this means that it runs every 4 ms and you may require up to two periods to detect the voltage drop. Therefore, it takes up to 8 ms to detect the problem. Consequently, the worst-case execution time of the shutdown process should not exceed 16 ms, to ensure that the deadline of 40 ms is met. This figure is calculated by subtracting the process periods (8 ms) from the deadline (40 ms) and dividing the result by two, as two process executions are necessary.

In reality, you would normally aim for something considerably less than 16 ms to give you a safety margin in case your calculations were wrong. In fact, the time required to examine a sensor and check that there has been no significant voltage loss should be much less than 16 ms. It only involves a simple comparison of two values. The average execution time of the power monitor process should be less than 1 ms.

The starting point for timing analysis in a real-time system is the timing requirements, which should set out the deadlines for each required response in the system. Figure 20.15 shows possible timing requirements for the office building burglar alarm system discussed in Section 20.2.1. To simplify this example, let us ignore stimuli generated by system testing procedures and external signals to reset the system in the event of a false alarm. This means there are only two types of stimulus to be processed by the system:

1. *Power failure* This is detected by observing a voltage drop of more than 20%. The required response is to switch the circuit to backup power by signaling an electronic power-switching device, which switches the mains power to battery backup.

Stimulus/Response	Timing Requirements
Power failure	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 half a second of an alarm being raised by a sensor.
Lights switch	The lights should be switched on within half a 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 synthesizer	A synthesized message should be available within 2 seconds of an alarm being raised by a sensor.

Figure 20.15 Timing requirements for the burglar alarm system

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 synthesizer to manage the call, and switch on the audible intruder alarm and building lights in the area.

As shown in Figure 20.15, you should list the timing constraints for each class of sensor separately, even when (as in this case) they are the same. By considering them separately, you leave scope for future change and make it easier to compute the number of times the controlling process has to be executed each second.

Allocating the system functions to concurrent processes is the next design stage. There are four types of sensors that must be polled periodically, each with an associated process. These are the voltage sensor, door sensors, window sensors, and movement detectors. Normally, the processes associated with the sensor will execute very quickly as all they are doing is checking whether or not a sensor has changed its status (e.g., from off to on). It is reasonable to assume that the execution time to check and assess the state of one sensor is no more than 1 ms.

To ensure that you meet the deadlines defined by the timing requirements, you then have to decide how frequently the related processes have to run and how many sensors should be examined during each execution of the process. There are obvious trade-offs here between frequency and execution time:

1. If you examine one sensor during each process execution, then if there are N sensors of a particular type, you must schedule the process 4N times per second to ensure that you meet the deadline of detecting a change of state within 0.25 seconds.

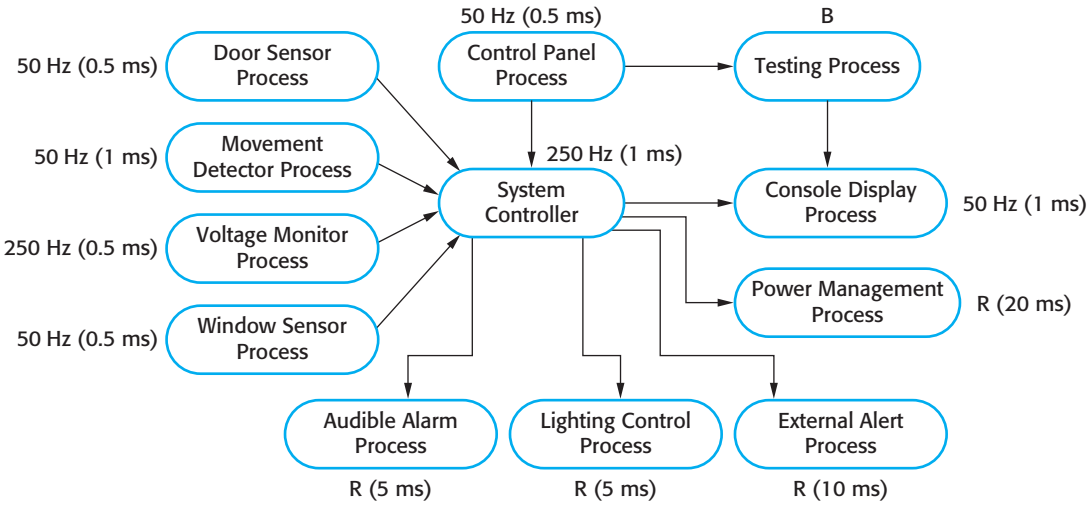


Figure 20.16 Alarm process timing

2. If you examine four sensors, say, during each process execution, then the execution time is increased to 4 ms, but you need only run the process N times/second to meet the timing requirement.

In this case, because the system requirements define actions when two or more sensors are positive, it may be sensible to examine sensors in groups, with groups based on the physical proximity of the sensors. If an intruder has entered the building then it will probably be adjacent sensors that are positive.

When you have completed the timing analysis, you may then annotate the process model with information about frequency of execution and their expected execution time (see Figure 20.16 as an example). Here, periodic processes are annotated with their frequency, processes that are started in response to a stimulus are annotated with R, and the testing process is a background process, annotated with B. This means that it only runs when processor time is available. In general, it is simpler to design a system so that there are a small number of process frequencies. The execution times represent the required worst-case execution times of the processes.

The final step in the design process is to design a scheduling system that will ensure that a process will always be scheduled to meet its deadlines. You can only do this if you know the scheduling approaches that are supported by the real-time operating system used (Burns and Wellings, 2009). The scheduler in the real-time OS allocates a process to a processor for a given amount of time. The time can be fixed, or may vary depending on the priority of the process.

In allocating process priorities, you have to consider the deadlines of each process so that processes with short deadlines receive processor time to meet these deadlines. For example, the voltage monitor process in the burglar alarm needs to be scheduled so that voltage drops can be detected and a switch made to backup power before the system fails. This should therefore have a higher priority than the processes that check sensor values, as these have fairly relaxed deadlines compared to their expected execution time.

20.4 Real-time operating systems

The execution platform for most application systems is an operating system that manages shared resources and provides features such as a file system, run-time process management, etc. However, the extensive functionality in a conventional operating system takes up a great deal of space and slows down the operation of programs. Furthermore, the process management features in the system may not be designed to allow fine-grain control over the scheduling of processes.

For these reasons, standard operating systems, such as Linux and Windows, are not normally used as the execution platform for real-time systems. Very simple embedded systems may be implemented as 'bare metal' systems. The systems themselves include system startup and shutdown, process and resource management, and process scheduling. More commonly, however, embedded applications are built on top of a real-time operating system (RTOS), which is an efficient operating system that offers the features needed by real-time systems. Examples of RTOS are Windows/CE, Vxworks, and RTLinux.

A real-time operating system manages processes and resource allocation for a real-time system. It starts and stops processes so that stimuli can be handled and allocates memory and processor resources. The components of an RTOS (Figure 20.17) depend on the size and complexity of the real-time system being developed. For all except the simplest systems, they usually include:

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

Real-time operating systems for large systems, such as process control or telecommunication systems, may have additional facilities, namely disk storage management, fault management facilities that detect and report system faults, and a configuration manager that supports the dynamic reconfiguration of real-time applications.

20.4.1 Process management

Real-time systems have to handle external events quickly and, in some cases, meet deadlines for processing these events. This means that the event-handling processes must be scheduled for execution in time to detect the event. They must also be allocated

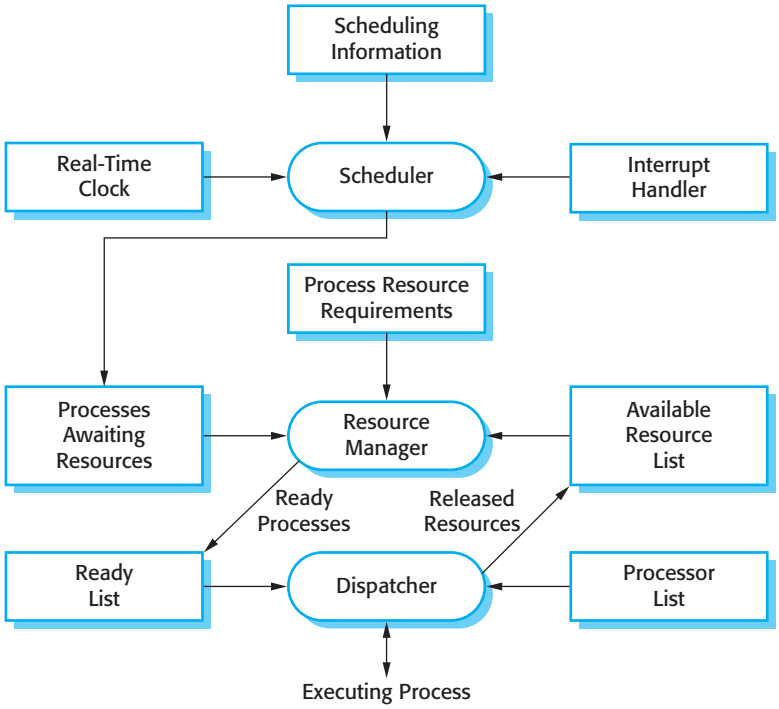


Figure 20.17 Components of a real-time operating system

sufficient processor resources to meet their deadline. The process manager in an RTOS is responsible for choosing processes for execution, allocating processor and memory resources, and starting and stopping process execution on a processor.

The process manager has to manage processes with different priorities. 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 RTOS 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 that 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) that 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.

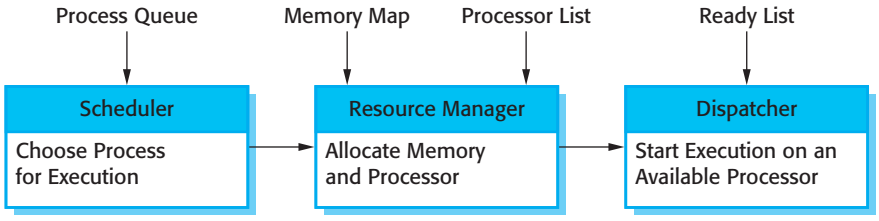


Figure 20.18 RTOS actions required to start a process

Periodic processes are processes that must be executed at specified time intervals for data acquisition and actuator control. In most real-time systems, there will be several types of periodic process. Using the timing requirements specified in the application program, the RTOS arranges the execution of periodic processes so that they can all meet their deadlines.

The actions taken by the operating system for periodic process management are shown in Figure 20.18. The scheduler examines the list of periodic processes and selects a process to be executed. The choice depends on the process priority, the process periods, the expected execution times, and the deadlines of the ready processes. Sometimes, two processes with different deadlines should be executed at the same clock tick. In such a situation, one process must be delayed. Normally, the system will choose to delay the process with the longest deadline.

Processes that have to respond quickly to asynchronous events may be interrupt-driven. The computer's interrupt mechanism causes control to transfer to a pre-determined memory location. This location contains an instruction to jump to a simple and fast interrupt service routine. The service routine disables further interrupts to avoid being interrupted itself. It then discovers the cause of the interrupt and initiates, with a high priority, a process to handle the stimulus causing the interrupt. In some high-speed data acquisition systems, the interrupt handler saves the data that the interrupt signaled was available in a buffer for later processing. Interrupts are then enabled again and control is returned to the operating system.

At any one time, there may be several processes, all with different priorities, that could be executed. The process scheduler implements system-scheduling policies that determine the order of process execution. There are two commonly used scheduling strategies:

1. *Non-pre-emptive scheduling* Once a process has been scheduled for execution it runs to completion or until it is blocked for some reason, such as waiting for input. This can cause problems, however, when there are processes with different priorities and a high-priority process has to wait for a low-priority process to finish.
2. *Pre-emptive scheduling* The execution of an executing process may be stopped if a higher-priority process requires service. The higher-priority process pre-empts the execution of the lower-priority process and is allocated to a processor.

Within these strategies, different scheduling algorithms have been developed. These include round-robin scheduling, where each process is executed in turn;

rate monotonic scheduling, where the process with the shortest period (highest frequency) is given priority; and shortest deadline first scheduling, where the process in the queue with the shortest deadline is scheduled (Burns and Wellings, 2009).

Information about the process to be executed is passed to the resource manager. The resource manager allocates memory and, in a multiprocessor system, also adds a processor to this process. The process is then placed on the 'ready list', a list of processes that are ready for execution. When a processor finishes executing a process and becomes available, the dispatcher is invoked. It scans the ready list to find a process that can be executed on the available processor and starts its execution.

KEY POINTS

- An embedded software system is part of a hardware/software system that reacts to events in its environment. The software is 'embedded' in the hardware. Embedded systems are normally real-time systems.
- A real-time system is a software system that must respond to events in real time. System correctness does not just depend on the results it produces, but also on the time when these results are produced.
- Real-time systems are usually implemented as a set of communicating processes that react to stimuli to produce responses.
- State models are an important design representation for embedded real-time systems. They are used to show how the system reacts to its environment as events trigger changes of state in the system.
- There are several standard patterns that can be observed in different types of embedded systems. These include a pattern for monitoring the system's environment for adverse events, a pattern for actuator control and a data-processing pattern.
- Designers of real-time systems have to do a timing analysis, which is driven by the deadlines for processing and responding to stimuli. They have to decide how often each process in the system should run and the expected and worst-case execution time for processes.
- A real-time operating system 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 execution.

FURTHER READING

Software Engineering for Real-Time Systems. Written from an engineering rather than a computer science perspective, this book is a good practical guide to real-time systems engineering. It has good coverage of hardware issues, so is an excellent complement to Burns and Wellings' book (see below). (J. Cooling, Addison-Wesley, 2003.)

Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th edition. An excellent and comprehensive text that provides broad coverage of all aspects of real-time systems. (A. Burns and A. Wellings, Addison-Wesley, 2009.)

'Trends in Embedded Software Engineering'. This article suggests that model-driven development (as discussed in Chapter 5 of this book), will become an important approach to embedded systems development. This is part of a special issue on embedded systems and you may find that other articles are also useful reading. (*IEEE Software*, **26** (3), May–June 2009.) http://dx.doi.org/10.1109/MS.2009.80.

EXERCISES

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

20.2. Identify possible stimuli and the expected responses for an embedded system that controls a home refrigerator or a domestic washing machine.

20.3. Using the state-based approach to modeling, as discussed in Section 20.1.1, model the operation of an embedded software system for a voice mail system included in a landline phone. This should display the number of recorded messages on an LED display and should allow the user to dial in and listen to the recorded messages.

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

20.5. Show how the Environmental Control pattern could be used as the basis of the design of a system to control the temperature in a greenhouse. The temperature should be between 10 and 30 degrees Celsius. If it falls below 10 degrees, the heating system should be switched on; if it goes above 30, the windows should be automatically opened.

20.6. Design a process architecture for an environmental monitoring system that collects data from a set of air quality sensors situated around a city. There are 5,000 sensors organized into 100 neighborhoods. Each sensor must be interrogated four times per second. When more than 30% of the sensors in a particular neighborhood 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.

Train protection system

- 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 ms.
- 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 ms) and the train brake status (updated every 100 ms).
- 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 ms of the time when the excessive train speed has been detected.
- If the train enters a track signaled that is signaled 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 ms of the time when the red light signal is received.
- The system continually updates a status display in the driver's cabin.

Figure 20.19
Requirements for a train protection system

20.7. 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 that is currently signaled with a red light (i.e., the segment should not be entered). Details are shown in Figure 20.19. Identify the stimuli that must be processed by the onboard train control system and the associated responses to these stimuli.

20.8. Suggest a possible process architecture for this system.

20.9. If a periodic process in the onboard 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.

20.10. Why are general-purpose operating systems, such as Linux or Windows, not suitable as real-time system platforms? Use your experience of using a general-purpose system to help answer this question.