5.18 SUMMARY

In this chapter we have reviewed some of the important language features that might influence our choice of a language for writing real-time software. Particular attention has been paid to elements of languages that contribute to the security of the resulting software. We have not attempted to compare real-time languages; if you are interested in such comparisons you will find a brief survey in Cooling (1991) and a more extensive survey in Tucker (1985). For a greater in-depth study of real-time languages see Young (1982) and Burns and Wellings (1990).

EXERCISES

5.1 Define the scope and visibility of the variables and parameters in the following code:

```
MODULE MyProgram;
VAR A,B:REAL;
    C,D:INTEGER;
PROCEDURE Pone ( A1:REAL;VAR A2:REAL);
  VAR M,N:INTEGER
  BEGIN (* Pone *)
   ..
  END Pone
PROCEDURE Ptwo;
  VAR P,D:INTEGER;
      Q,R:REAL;
  BEGIN (* TWO *)
   ...
    Pone (Q,R);
   ...
  END Ptwo;
  BEGIN (*MyProgram*)
   ...
  END MyProgram.
```

5.2 In the computer science literature you will find lots of arguments about 'global' and 'local' variables. What guidance would you give to somebody who asked for advice on how to decide on the use of global or local variables?

5.3 How does strong data typing contribute to the security of a programming language?

5.4 Why is it useful to have available a predefined data type BITSET in Modula-2? Give an example to illustrate how, and under what circumstances, BITSET would be used.

6

Operating Systems

This chapter is not a complete discussion of operating systems. In it we concentrate on the aspects of operating systems that are particularly relevant to real-time control applications. We first look at what they are, how they differ from non-real-time operating systems and why we use them. We will then examine in some detail how they handle the management of tasks. Finally, we will look briefly at some ways of implementing real-time operating systems.

The aims of the chapter are to:

- Explain why we use a real-time operating system (RTOS).
- Explain what an RTOS does.
- Explain how an RTOS works.
- Describe the benefits and drawbacks of an RTOS.
- List the minimum language primitives required for creating an RTOS.
- Describe the problem of sharing resources and explain several techniques for providing mutual exclusion.
- Explain what a binary semaphore does and write a program in Modula-2 to demonstrate its use.
- Describe and explain the basic task synchronisation mechanisms.

6.1 INTRODUCTION

Software design is simplified if details of the lower levels of implementation on a specific computer using a particular language can be hidden from the designer. An operating system for a given computer converts the hardware of the system into a virtual machine with characteristics defined by the operating system. Operating systems were developed, as their name implies, to assist the operator in running a batch processing computer; they then developed to support both real-time systems and multi-access on-line systems.

The traditional approach is to incorporate all the requirements inside a general purpose operating system as illustrated in Figure 6.1. Access to the hardware of the system and to the I/O devices is through the operating system. In many real-time

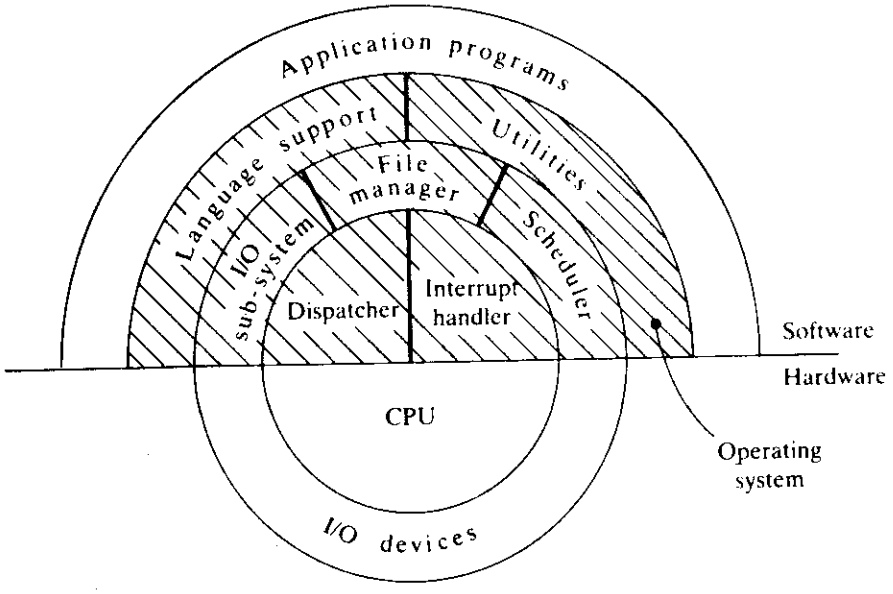


Figure 6.1 General purpose operating system.

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

A general purpose operating system will provide some facilities that are not required in a particular application, and to be forced to include them adds unnecessarily to the system overheads. Usually during the installation of an operating system certain features can be selected or omitted. A general purpose operating system can thus be 'tailored' to meet a specific application requirement.

Recently operating systems which provide only a minimum kernel or nucleus have become popular; additional features can be added by the applications programmer writing in a high-level language. This structure is shown in Figure 6.2. In this type of operating system the distinction between the operating system and the application software becomes blurred. The approach has many advantages for applications that involve small, embedded systems.

The relationship between the various sections of a simple operating system, the computer hardware and the user is illustrated in Figure 6.3. The command processor provides a means by which the user can communicate with the operating system from the computer console device. Through it the user issues commands to the

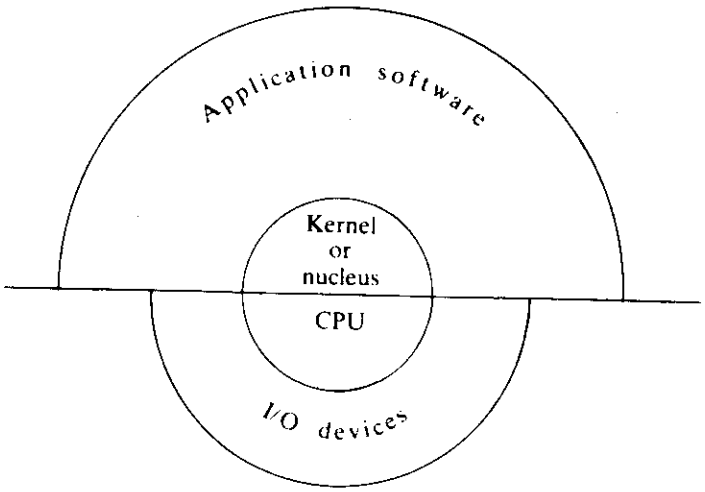


Figure 6.2 Minimal operating system.

operating system and it provides the user with information about the actions being performed by the operating system. The actual processing of the commands issued by the user is done by the BDOS (Basic Disk Operating System) which also handles the input and output and the file operations on the disks. The BDOS makes the actual management of the file and input/output operations transparent to the user. Application programs will normally communicate with the hardware of the system through *system calls* which are processed by the BDOS.

The BIOS (Basic Input Output System) contains the various device drivers which manipulate the physical devices and this section of the operating system may vary from implementation to implementation as it has to operate directly with the underlying hardware of the computer. For example, the physical addresses of the peripherals may vary according to the manufacturer; these differences will be accommodated in the coding of the BIOS.

Devices are treated as *logical* or *physical* units. Logical devices are software constructs used to simplify the user interface; user programs perform input and output to logical devices and the BDOS connects the logical device to the physical device. The actual operation of the physical device is performed by software in the BIOS.

Access to the operating system functions is by means of subroutine calls and information is passed in the CPU registers of the machine. Functions cannot be called directly from most high-level languages and this provides isolation between the operating system and a programmer using a high-level language. The isolation is deliberate; it is an example of *information hiding*. The connection between the high-level language and the operating system is made by the compiler writer through the provision of run-time support routines which convert the operating system into the virtual machine described by the high-level language.

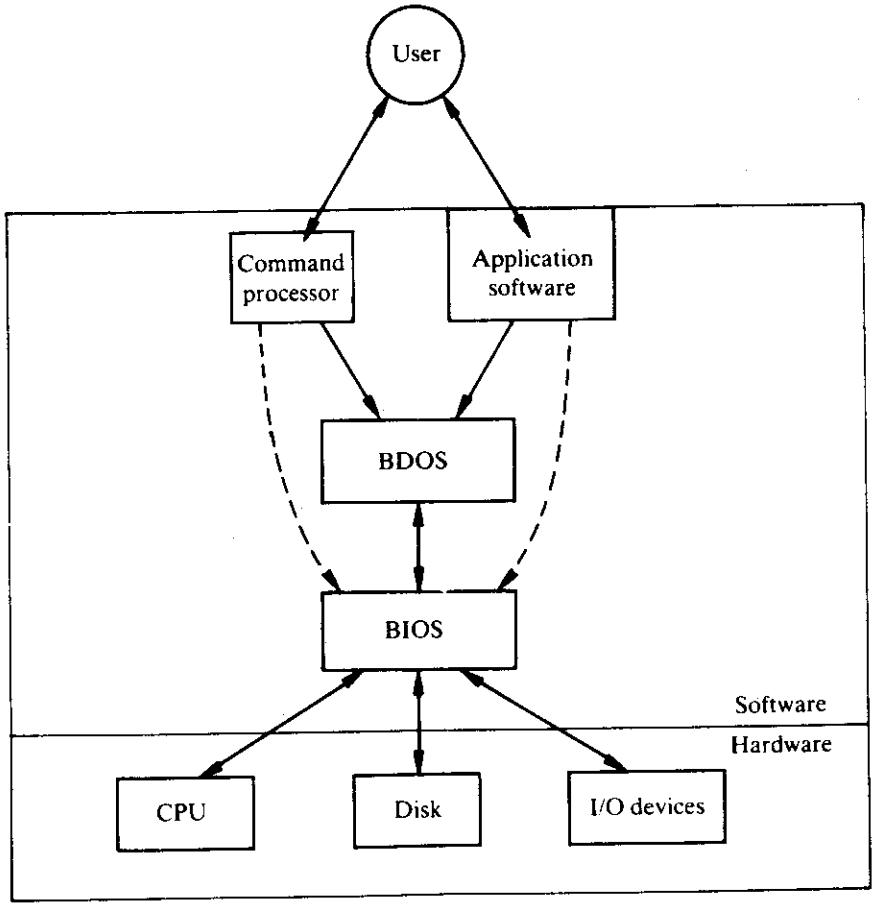


Figure 6.3 General structure of a simple operating system.

The isolation is not complete in that it is possible to call assembly-coded routines from high-level languages and to pass parameters between the high-level language code and the assembly code; this does, however, require detailed knowledge of the system. Again information hiding is used in that the details of the physical implementation on the CPU and of the I/O devices are hidden within the operating system and hence operations are performed on the operating system virtual machine.

6.2 REAL-TIME MULTI-TASKING OPERATING SYSTEMS

There are many different types of operating systems and until the early 1980s there was a clear distinction between operating systems designed for use in real-time

applications and other types of operating system. In recent years the dividing line has become blurred. For example, languages such as Modula-2 enable us to construct multi-tasking real-time applications that run on top of single-user, single-task operating systems. And operating systems such as UNIX and OS/2 support multi-user, multi-tasking applications.

Confusion can arise between multi-user or multi-programming operating systems and multi-tasking operating systems. The function of a multi-user operating system is illustrated in Figure 6.4: the operating system ensures that each user can run a single program as if they had the whole of the computer system for their program. Although at any given instance it is not possible to predict which user will have the use of the CPU, or even if the user's code is in the memory, the operating system ensures that one user program cannot interfere with the operation of another user program. Each user program runs in its own protected environment. A primary concern of the operating system is to prevent one program, either deliberately or through error, corrupting another. In a multi-tasking operating system it is assumed that there is a single user and that the various tasks co-operate to serve the requirements of the user. Co-operation requires that the tasks communicate with each other and share common data. This is illustrated in Figure 6.5. In a good multi-tasking operating system task communication and data sharing will be regulated so that the operating system is able to prevent inadvertent communication or data access (that is, arising through an error in the coding of one task) and hence protect

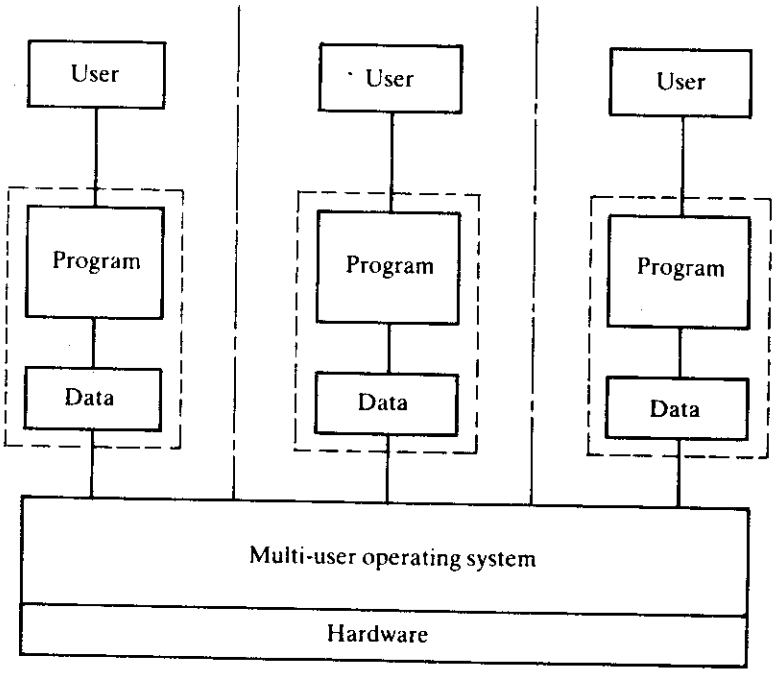


Figure 6.4 Multi-user operating system.

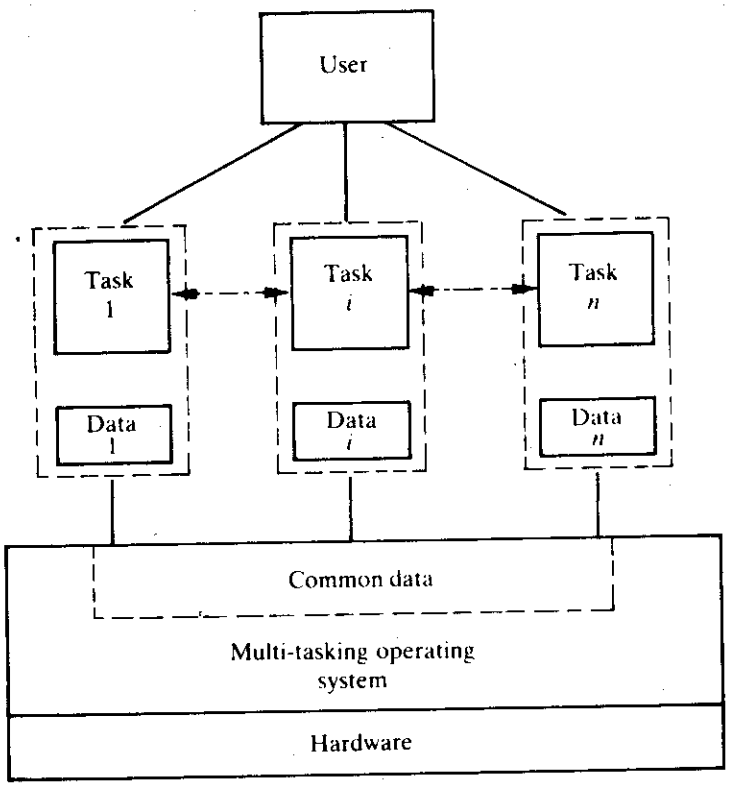


Figure 6.5 Multi-tasking operating system.

data which is private to a task (note that deliberate interference cannot be prevented – the tasks are assumed to be co-operating).

A fundamental requirement of an operating system is to allocate the resources of the computer to the various activities which have to be performed. In a real-time operating system this allocation procedure is complicated by the fact that some of the activities are time critical and hence have a higher priority than others. Therefore there must be some means of allocating priorities to tasks and of scheduling allocation of CPU time to the tasks according to some priority scheme.

A task may use another task, that is it may require certain activities which are contained in another task to be performed and it may itself be used by another task. Thus tasks may need to communicate with each other. The operating system must have some means of enabling tasks either to share memory for the exchange of data, or to provide a mechanism by which tasks can send messages to each other. Also tasks may need to be invoked by external events and hence the operating system must support the use of interrupts. Similarly tasks may need to share data and they may require access to various hardware and software components; hence there has to be a mechanism for preventing two tasks from attempting to use the same resource at the same time.

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

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

Memory management: control of memory allocation.

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

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

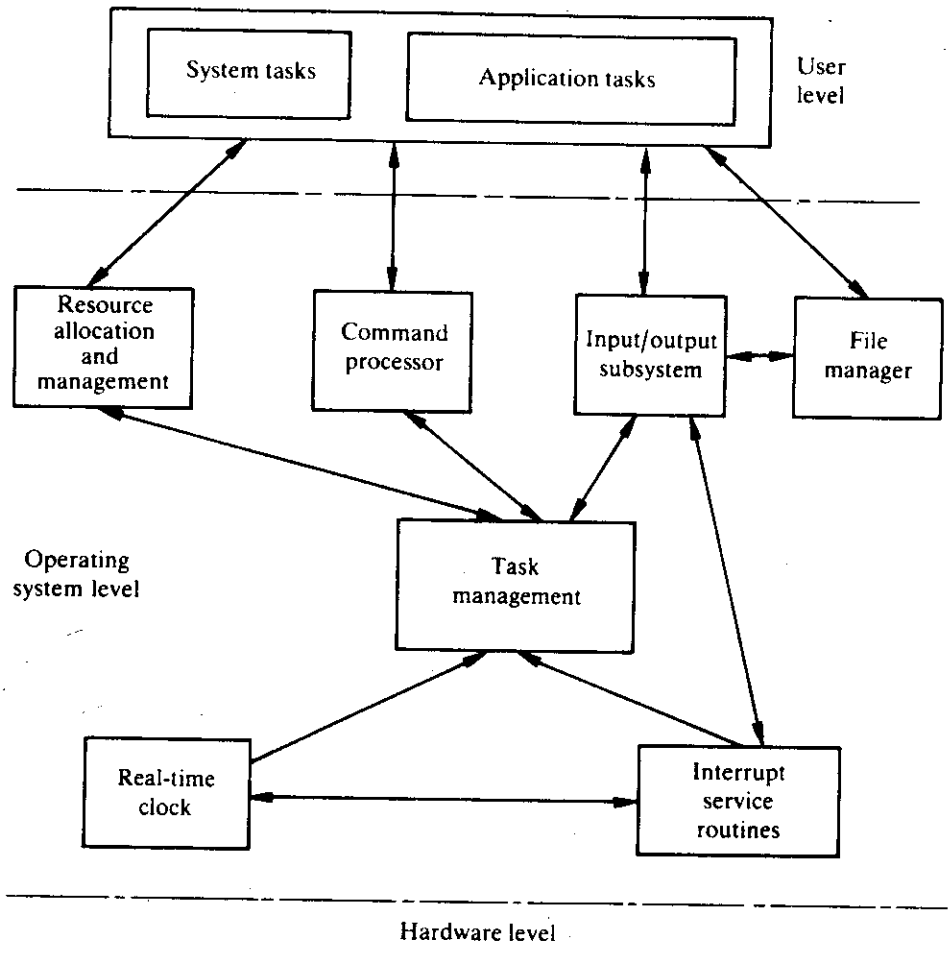


Figure 6.6 Typical structure of a real-time operating system.

In addition to the above the system has to provide the standard features such as support for disk files, basic input/output device drivers and utility programs. The typical structure is illustrated in Figure 6.6. The overall control of the system is provided by the *task management* module which is responsible for allocating the use of the CPU. This module is often referred to as the *monitor* or as the *executive control program* (or more simply the executive). At the user level, in addition to application tasks, a box labelled 'system tasks' is also shown since in many operating systems some operations performed by the operating system and the utility programs run in the memory space allocated to the user or applications – this space is sometimes called 'working memory'.

From the user's viewpoint the two most important features of task management are how to create a task, that is make its existence known to the RTOS, and what scheduling strategy or strategies the RTOS supports. Task creation is largely a function of the interface between the operating system and high-level programming language and we gave some examples of the mechanisms involved in the previous chapter.

6.3 SCHEDULING STRATEGIES

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

1. Cyclic.
2. Pre-emptive.

6.3.1 Cyclic

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

6.3.2 Pre-emptive

There are many pre-emptive strategies. All involve the possibility that a task will be interrupted – hence the term pre-emptive – before it has completed a particular

invocation. A consequence of this is that the executive has to make provision to save the volatile environment for each task, since at some later time it will be allocated CPU time and will want to continue from the exact point at which it was interrupted. This process is called *context switching* and a mechanism for supporting it is described below.

The simplest form of pre-emptive scheduling is to use a time slicing approach (sometimes called a round-robin method). Using this strategy each task is allocated a fixed amount of CPU time – a specified number of *ticks* of the clock – and at the end of this time it is stopped and the next task in the list is run. Thus each task in turn is allocated an equal share of the CPU time. If a task completes before the end of its time slice the next task in the list is run immediately.

The majority of existing RTOSs use a priority scheduling mechanism. Tasks are allocated a priority level and at the end of a predetermined time slice the task with the highest priority of those ready to run is chosen and is given control of the CPU. Note that this may mean that the task which is currently running continues to run.

Task priorities may be fixed – a *static* priority system – or may be changed during system execution – a *dynamic* priority system. Dynamic priority schemes can increase the flexibility of the system, for example they can be used to increase the priority of particular tasks under alarm conditions. Changing priorities is, however, risky as it makes it much harder to predict the behaviour of the system and to test it. There is the risk of locking out certain tasks for long periods of time. If the software is well designed and there is adequate computing power there should be no need to change priorities – all the necessary constraints will be met. If it is badly designed and/or there are inadequate computing resources then dynamic allocation of priorities will not produce a viable, reliable system.

Whatever scheduling strategy is adopted the task management system has to deal with the handling of interrupts. These may be hardware interrupts caused by external events, or software interrupts generated by a running task. An interrupt forces a context switch. The running task is suspended and an interrupt handler is run. The interrupt handler should only contain a small amount of code and should execute very quickly. When the handler terminates either the task that was interrupted is restored or the scheduler is entered and it determines which task should run. The RTOS designer has to decide which approach to adopt.

EXAMPLE 6.1

Interrupt Handling and Scheduling

A system receives an alarm signal interrupt from a plant and in response to the alarm it is to run an alarm alert task which is a high-priority, base level task. The interrupt service routine for the alarm signal will, by some mechanism, cause the alarm alert task to be placed in the runnable queue, and there are then two actions which it can take: (a) return to the interrupted task, (b) jump to the scheduler. If a return to the interrupted task is made then the alarm alert task will not be run until the system

reschedules either at the system rescheduling interval or because the running task terminates or becomes suspended waiting for a system resource. However, if a jump is made directly to the dispatcher from the interrupt service routine, then if the alarm alert task is of higher priority than the interrupted task it will be run immediately and the interrupted task will have been pre-empted. The argument for entering the scheduler and rescheduling is that the occurrence of an interrupt is likely to have changed the state of a task and hence the task that was running may no longer be the highest-priority task. The argument for returning to the running task is that it involves less time loss since performing only the context switch requires less CPU time than running the scheduler as well. A rational decision depends on the executive having some knowledge of the application: if it is known that the majority of interrupts are generated by events that have high priority – alarms, or important changes in plant conditions – then entering the scheduler after an interrupt is the best choice; on the other hand, if a large number of interrupts are from serial-based input and output and communications devices that simply involve placing a character in a buffer then to enter the scheduler every time would be a waste of CPU time.

6.4 PRIORITY STRUCTURES

In a real-time system the designer has to assign priorities to the tasks in the system. The priority will depend on how quickly a task will have to respond to a particular event. An event may be some activity of the process or may be the elapsing of a specified amount of time. Most RTOSs provide facilities such that tasks can be divided into three broad levels of priority as shown in Figure 6.7.

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

6.4.1 Interrupt Level

As we have already seen an interrupt forces a rescheduling of the work of the CPU and the system has no control over the timing of the rescheduling. Because an interrupt-generated rescheduling is outside the control of the system it is necessary

to keep the amount of processing to be done by the interrupt handling routine to a minimum. Usually the interrupt handling routine does sufficient processing to preserve the necessary information and to pass this information to a further handling routine which operates at a lower-priority level, either clock level or base level. Interrupt handling routines have to provide a mechanism for task swapping, that is they have to save the volatile environment. On completion the routine either

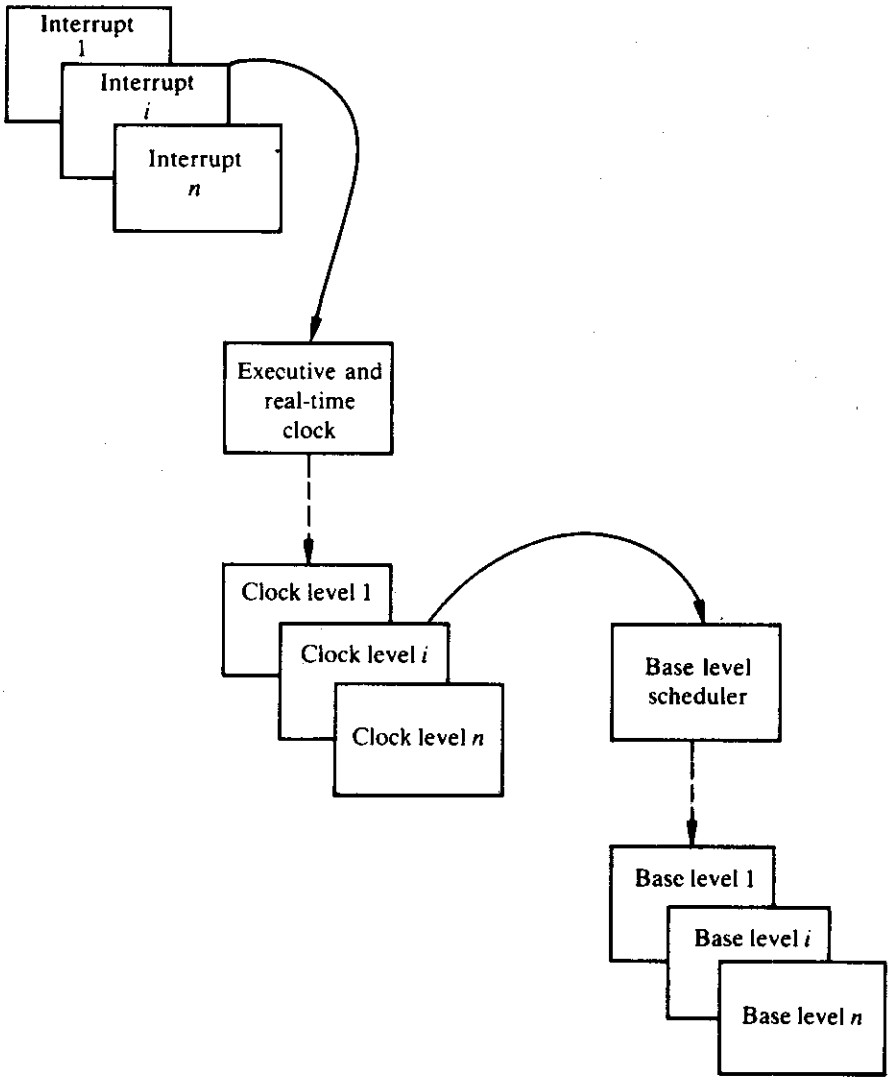


Figure 6.7 Priority levels in an RTOS.

will simply restore the volatile environment and hence will return to the interrupted task, or it may exit to the scheduler.

Within the interrupt level of tasks there will be different priorities and there will have to be provision for preventing interrupts of lower priority interrupting higher-priority interrupt tasks. On most modern computer systems there will be hardware to assist in this operation (see Chapter 3).

6.4.2 Clock Level

One interrupt level task will be the real-time clock handling routine which will be entered at some interval, usually determined by the required activation rate for the most frequently required task. Typical values are 1 to 200 ms. Each clock interrupt is known as a *tick* and represents the smallest time interval known to the system. The function of the clock interrupt handling routine is to update the time-of-day clock in the system and to transfer control to the dispatcher. The scheduler selects which task is to run at a particular clock tick.

Clock level tasks divide into two categories:

1. *CYCLIC*: these are tasks which require accurate synchronisation with the outside world.
2. *DELAY*: these tasks simply wish to have a fixed delay between successive repetitions or to delay their activities for a given period of time.

6.4.3 Cyclic Tasks

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

EXAMPLE 6.2

Cyclic Tasks

Three tasks *A*, *B* and *C* are required to run at 20 ms, 40 ms and 80 ms intervals (corresponding to 1 tick, 2 ticks and 4 ticks, if the clock interrupt rate is set at 20 ms). If the task priority order is set as *A*, *B* and *C* with *A* as the highest priority then the processing will proceed as shown in Figure 6.8a with the result that the tasks will be run at constant intervals. It should be noted that using a single CPU it is not possible to have all the tasks starting in synchronism with the clock tick. All but one of the tasks will be delayed relative to the clock tick; however, the interval between

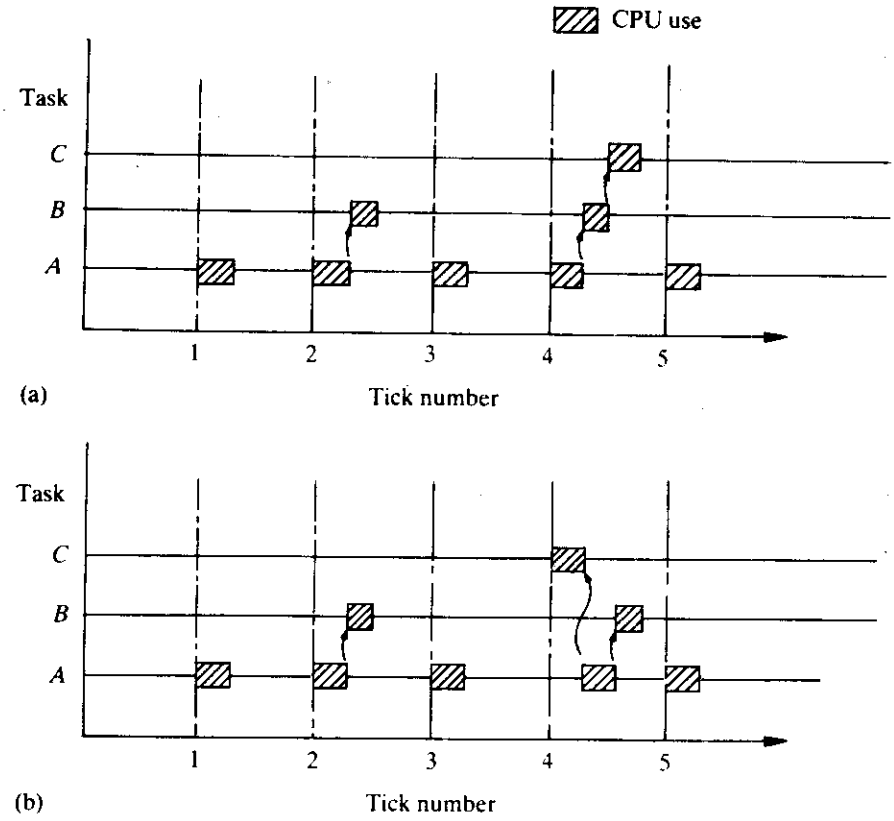


Figure 6.8 Task activation diagram for Example 6.2: (a) task priorities A,B,C; (b) task priorities C,A,B.

successive invocations of the task will be constant (if the execution time for each task is a constant value). If the priority order is now rearranged so that it is *C*, *A* and *B* then the activation diagram is as shown in Figure 6.8b and every fourth tick of the clock there will be a delay in the timing of tasks *A* and *B*. In practice there is unlikely to be any justification for choosing a priority order *C*, *A* and *B* rather than *A*, *B* and *C*. Usually the task with the highest repetition rate will have the most stringent timing requirements and hence will be assigned the highest priority.

A further problem which can arise is that a clock level task may require a longer time than the interval between clock interrupts to complete its processing (note that for overall satisfactory operation of the system such a task cannot run at a high repetition rate).

EXAMPLE 6.3

Timing of Cyclic Tasks

Assume that in Example 6.2 task *C* takes 25 ms to complete, task *A* takes 1 ms and task *B* takes 6 ms. If task *C* is allowed to run to completion then the activity diagram will be as shown in Figure 6.9 and task *A* will be delayed by 11 ms at every fourth invocation. It is normal therefore to divide the *cyclic* tasks into high-priority tasks which are guaranteed to complete within the clock interval and lower-priority tasks which can be interrupted by the next clock tick.

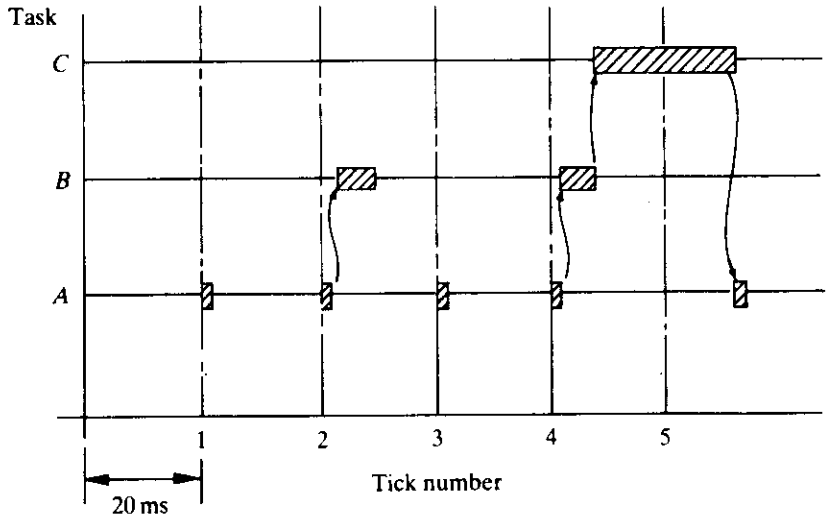


Figure 6.9 Task activation diagram for Example 6.3.

6.4.4 Delay Tasks

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

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

which calculates the time when the task is next due to run and inserts the task descriptor in the appropriate place in the queue.

A task running at the clock level checks the first task in the DELAYED queue to see if it is time for that task to run. If the task is due to run it is removed from the DELAYED queue and placed in the runnable queue. The task which checks the DELAYED queue may be either the dispatcher which is entered every time the real-time clock interrupts or another clock level task which runs at less frequent intervals, say every 10 ticks, in which case it is then frequently part of the base level scheduler. Many real-time operating systems do not support the cycle operation and the user has to create an accurate repetitive timing for the task by using the delay function.

6.4.5 Base Level

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

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

Alternatively priorities can be adjusted in response to particular events or under the control of the operator. For example, alarm tasks will usually have a high priority and during an alarm condition tasks such as the log of plant data may be delayed with the consequence that the output of the log lags behind real time (note that the data will be stored in buffer areas inside the computer). So that the log can catch up with real time quickly it may be advisable to increase, temporarily, the priority of the printer output task.

6.5 TASK MANAGEMENT

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

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

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

6.5.1 Task States

With one processor only one task can be running at any given time and hence the other tasks must be in some other state. The number of other states, the names

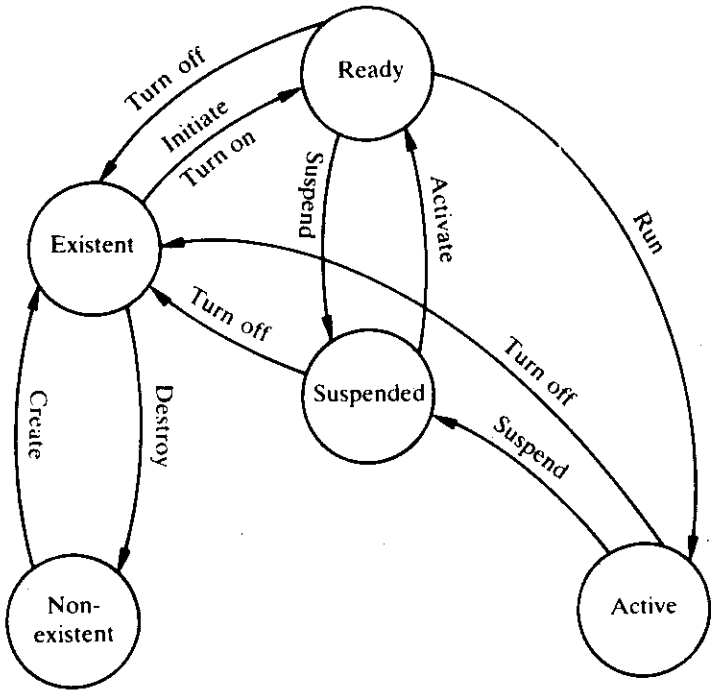


Figure 6.10 Example of a typical task state diagram.

given to the states, and the transition paths between the different states vary from operating system to operating system. A typical state diagram is given in Figure 6.10 and the various states are as follows (names in parentheses are commonly used alternatives):

- *Active (running)*: this is the task which has control of the CPU. It will normally be the task with the highest priority of the tasks which are ready to run.
- *Ready (runnable, on)*: there may be several tasks in this state. The attributes of the task and the resources required to run the task must be available for the task to be placed in the *Ready* state.
- *Suspended (waiting, locked out, delayed)*: the execution of tasks placed in this state has been suspended because the task requires some resource which is not available or because the task is waiting for some signal from the plant, for example input from the analog-to-digital converter, or because the task is waiting for the elapse of time.
- *Existent (dormant, off)*: the operating system is aware of the existence of this task, but the task has not been allocated a priority and has not been made runnable.
- *Non-existent (terminated)*: the operating system has not as yet been made aware of the existence of this task, although it may be resident in the memory of the computer.

The status of the various tasks may be changed by actions within the operating system – a resource becoming available or unavailable – or by commands from the application tasks. A typical command is:

TURN ON (ID) – transfer a task from *existent* to *ready* state,

where ID is the name by which the task is known to the operating system. A typical set of commands is given in Table 6.1. It should be noted that the transition from *ready* to *active* can only be made at the behest of the dispatcher.

6.5.2 Task Descriptor

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

- task identification (ID);
- task priority (P);
- current state of task;

Table 6.1 RTOS task state transition commands

OFFC01	Turn off the task leaving the memory marked as occupied
OFFC02	Turn off the task leaving the memory marked as unoccupied
DELC01	Delay the task leaving the memory marked as occupied; delay is calculated using current value of time
DELC02	Delay the task leaving the memory marked as unoccupied; delay is calculated as for DELC01
DELC03	Delay the task leaving the memory marked as occupied; delay is calculated by adding the delay to the value of time stored in the task descriptor
TPNC01	Turn the task on; will be accepted if the task is ON, OFF or DELAYED; either the ON constant can be placed in the task descriptor or a specified turn-on time
TPNC02	Turn on the task; will only be accepted if the task is in the OFF state
TPNC03	Run the task immediately regardless of priority; will be accepted if the task is ON, OFF or DELAYED

- area to store volatile environment (or a pointer to an area for storing the volatile environment); and
- pointer to next task in a list.

The reason for including the last item in the list above is that the task descriptors are usually held in a linked list structure. The executive keeps a set of lists, one for each task state as shown in Figure 6.11. There is one active task (task ID = 10) and three tasks that are ready to run (IDs = 20, 9 and 6). The entry held in the executive for the ready queue head points to task 20, which in turn points to task 9 and so on.

The advantage of the list structure is that the actual task descriptor can be located anywhere in the memory and hence the operating system is not restricted to a fixed number of tasks as was often the case in the older operating systems which used fixed length tables to hold task state information. With the list structure moving tasks between lists, reordering the lists, creating and deleting tasks can all be achieved simply by changing pointers. There is no need to copy or move the task descriptors themselves.

The information that has to be stored in order to continue running a task that has been suspended by the scheduler for some reason or other comprises:

Housekeeping information:	CPU register contents;
	stack pointer;
	program counter.
Task data:	task stack;
	task general work area (heap).

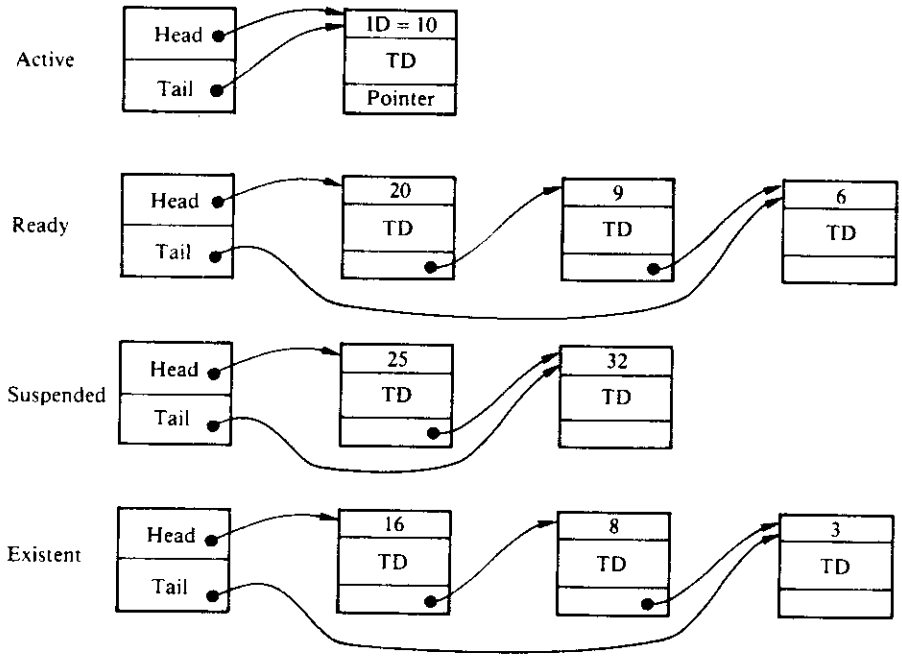


Figure 6.11 List structure for holding task state information.

6.6 SCHEDULER AND REAL-TIME CLOCK INTERRUPT HANDLER

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

6.6.1 System Commands Which Change Task Status

The range of system commands affecting task status varies with the operating system. Typical states and commands are shown in Figure 6.12 and fuller details of the commands are given in Table 6.1. Note that this system distinguishes between tasks which are suspended awaiting the passage of time – these tasks are marked as delayed – and those tasks which are waiting for an event or a system resource – these are marked as locked out.

The system does not explicitly support base level tasks; however, the lowest four priority levels of the clock level tasks can be used to create a base level system. A

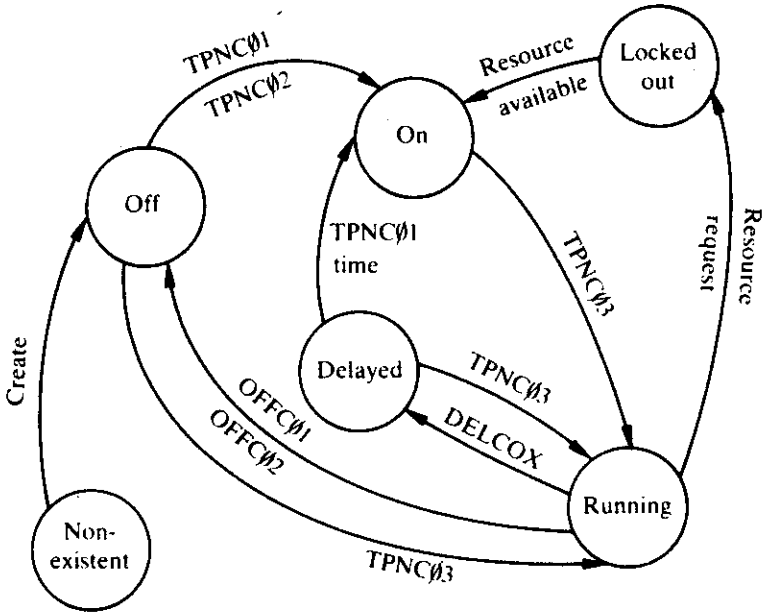


Figure 6.12 RTOS task state diagram.

so-called free time executive (FTX) is provided which if used runs at priority level $n-3$ (see Figure 6.13) where n is the lowest-priority task number. The FTX is used to run tasks at priority levels $n-2$, $n-1$ and n; it also provides support for the chaining of tasks. The dispatcher is unaware of the fact that tasks at these three priority levels are being changed; it simply treats whichever tasks are in the lowest three priority levels as low-priority tasks. Tasks run under the FTX do not have access to the system commands (except OFFC01, that is turn task off).

6.6.2 Dispatcher – Search for Work

The dispatcher/scheduler has two entry conditions:

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

In response to the first condition the scheduler searches for work starting with the highest-priority task and checking each task in priority order (see Figure 6.14). Thus if tasks with a high repetition rate are given a high priority they will be treated as if they were clock level tasks, that is they will be run first during each system clock period. In response to the second condition a search for work is started at the task

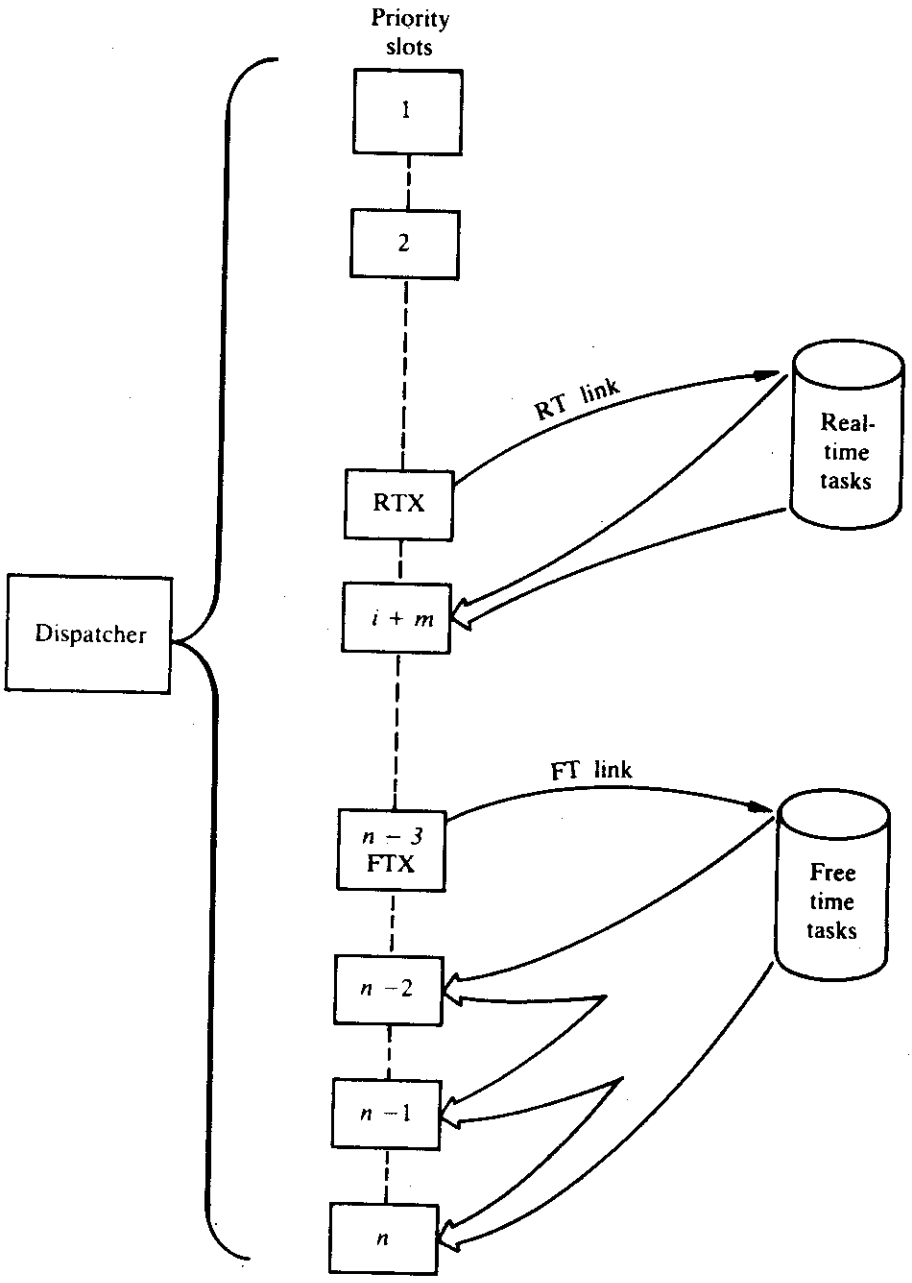


Figure 6.13 RTOS task structure diagram.

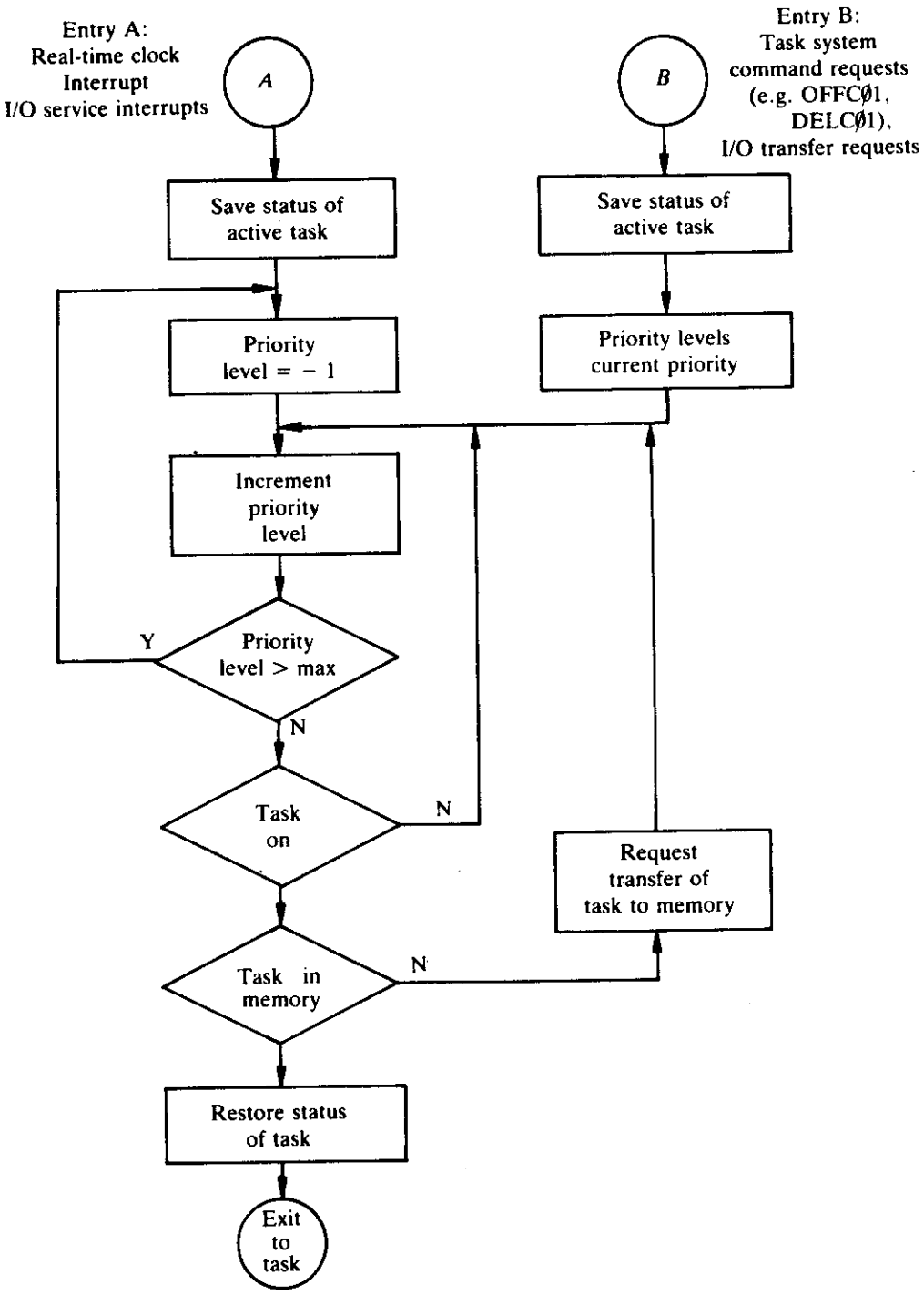


Figure 6.14 RTOS search for work by the dispatcher.

with the next lowest priority to the task which has just been running. There cannot be another higher-priority task ready to run since a higher-priority task becoming ready always pre-empts a lower-priority-running task.

The system commands for task management are issued as calls from the assembly level language and the parameters are passed either in the CPU registers or as a control word immediately following the call statement.

EXAMPLE 6.4

Use of RTOS System Calls

As an example consider the system whose outline design is given in Figure 6.15. It is assumed that the `Control`, `Display` and `Operator` input programs are to be run as separate tasks with priorities 1, 10, 20, respectively. The `Control` task has to run at 40 ms intervals and the `Display` update task at 5 s intervals. The system

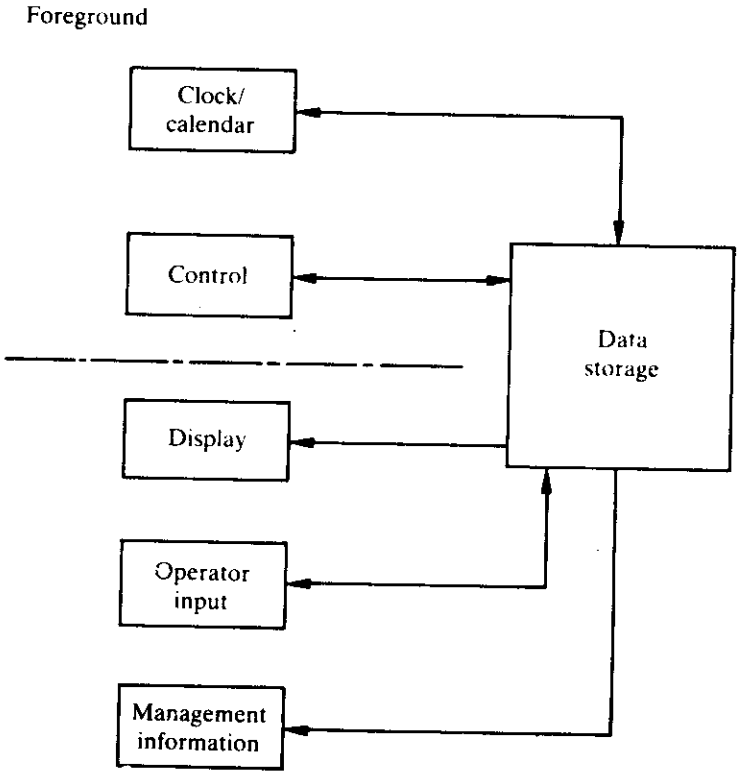


Figure 6.15 Software modules for foreground/background system showing data storage.

clock is set at 20 ms and hence the `Control` task has to run every 2 system ticks. The outline structure of the system is given below:

```
    TASK MAIN
;
; Starts up the system by creating the various
; tasks and setting them to the ON condition
;
    CREATE(CONTROL,1,STCTRL)
    CREATE(DISPLAY,10,STDISP)
    CREATE(OPERATOR,20,STOPR)
;
; STCTRL, STDISP, STOPR are common symbols which
; define the starting locations for each task,
; the values will be inserted by the linker/loader.
;
    LDA TIME ; TIME is system variable which
        ; gives current time
    CALL TPNC01
    FCB 1 ; turn on control task
;
    LDA 0
    CALL TPNC02
    FCB 10    ; turn on display task
;
    LDA 0
    CALL TPNC02
    FCB 20 ; turn on operator input task
;
    CALL OFFC02 ; terminate main task
;
    END
```

In the above code by using `TPNC01` to turn on the control task the current value of time is placed in the task descriptor and hence the task can be synchronised to the clock.

```
    TASK CONTROL
;
    .....
    main body of task
    .....
;
    CALL DELC03
    FCB 0,2 ; set next time for running
            ; to previous time plus two ticks
;
    END
```

```
  TASK DISPLAY
;
  ....
  main body of task
  ....
;
  CALL DELC02
  FCB 5,0 ; delay task by 5 seconds
;
  END
```

The difference between using DELC03 and DELC02 in the above task segments is that DELC03 adds the delay increment to the value of time stored in the task descriptor; this time is the time at which the task was last due to run. The use of DELC03 therefore provides a means of running tasks in a cyclic mode at clock level. The DELC02 command adds the delay value to the current time and stores the result in the task descriptor; hence the delay is calculated not from when the task was last due to run, but from the time at which the delay command is issued.

In designing a real-time system it is important to know how the scheduler searches for work. In the system described in Example 6.4, the scheduler searches in strict priority order and hence the overheads in terms of the time spent searching for work will be increased if some of the high-priority tasks rarely run. A careful assessment of task priority is required and particular attention will have to be paid to alarm action tasks. Such tasks are normally accorded high priority; however, it is hoped that they will rarely be required. One solution with the above system which avoids having a group of high-priority but rarely run alarm tasks is to make use of the TPNC03 command. The alarm action tasks are given low priority, but can be made to run immediately if the alarm scanning routine uses the TPNC03 call to invoke the appropriate task.

6.7 MEMORY MANAGEMENT

Since the majority of control application software is static – the software is not dynamically created or eliminated at run-time – the problem of memory management is simpler than for multi-programming, on-line systems. Indeed with the cost of computer hardware, both processors and memory, reducing many control applications use programs which are permanently resident in fast access memory.

With permanently resident software the memory can be divided as shown in Figure 6.16. The user space is treated as one unit and the software is linked and loaded as a single program into the user area. The information about the various

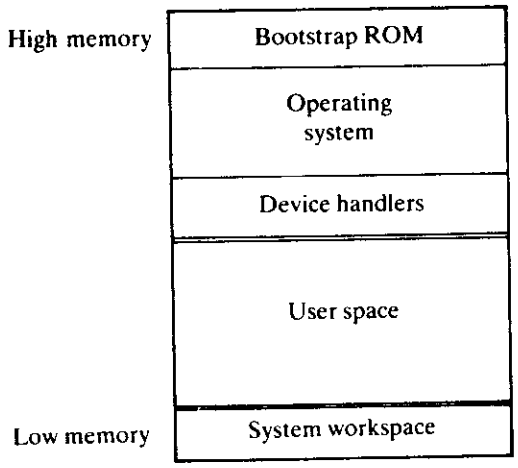


Figure 6.16 Non-partitioned memory.

tasks is conveyed to the operating system by means of a create task statement. Such a statement may be of the form

```
Create(TaskID, Priority, StartAddress, WorkSpace)
```

The exact form of the statement will depend on the interface between the high-level language and the operating system.

An alternative arrangement is shown in Figure 6.17. The available memory is divided into predetermined segments and the tasks are loaded individually into the

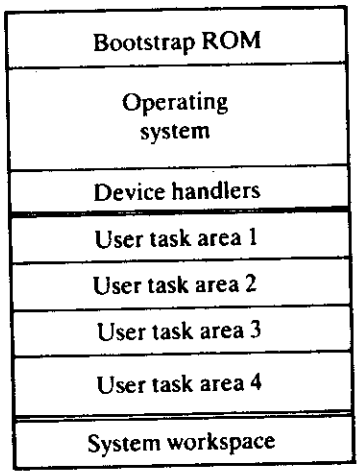


Figure 6.17 Partitioned memory.

various segments. The load operation would normally be carried out using the command processor. With this type of system the entries in the TD (or the operating system tables) have to be made from the console using a memory examine and change facility.

Divided (partitioned) memory was widely used in many early real-time operating systems and it was frequently extended to allow several tasks to share one partition; the tasks were kept on the backing store and loaded into the appropriate partition when required. There was of course a need to keep any tasks in which timing was crucial (hard time constraint tasks) in fast access memory permanently. Other tasks could be swapped between fast memory and backing store. The difficulty with this method is, of course, in choosing the best mix of partition sizes. The partition size and boundaries have to be determined at system generation.

A number of methods have been used to overcome the problem of fixed partitions. One method, referred to as floating memory, divides the available memory into small blocks, for example 64 words. The tasks are installed on the backing store and when a task is required to run the operating system examines a map of memory and finds a contiguous area of memory which will hold the task. The task is loaded into the memory and the memory blocks occupied by the tasks are marked as occupied in the memory map. The area occupied by a task is closely related to the actual size of the task and not to some predetermined fixed partition size. A task which for some reason becomes suspended or delayed will have the memory area it occupies marked in the memory map as occupied but available; hence if another task becomes ready then the suspended task can be returned to backing store and the ready task loaded into its area. In this type of system information must be held in the task descriptor to indicate if the task can be swapped, since, for example, a control task which has to run every 40 ms would have to be held permanently in memory in order to guarantee the sampling rate. A problem which is generated by this system is fragmentation of the available memory. Small areas of free memory become spread about the memory address space; none of the individual areas are large enough to take a task but the combined areas could if they could be brought together. Some form of *garbage* collection is necessary to bring dispersed areas into contiguous blocks.

Other systems which permit dynamic allocation of memory allow the tasks themselves to initiate program segment transfers, either by chaining or by overlaying. In chaining the task is divided into several segments which run sequentially. On completion of one segment the next segment is loaded from memory into the area occupied by the previous segment; any data required to be passed is held either on the disk or in a common area of memory.

Task swapping involves one task invoking another task: the first task is transferred to backing store and the second task brought into memory and made available to run. The procedure is shown in Figure 6.18. Task 1 invokes task 5 by swapping it into priority level 41 and in turn task 5 chains task 6 into level 41. Task 6 swaps task 7 into level 42. When task 7 terminates the operating system returns control to task 6 and when it terminates control is returned to task 1. It should be

noted that task 1 remains suspended until task 6 terminates and similarly task 6 is suspended until task 7 terminates.

The difference between chaining and overlaying is that in overlaying a part of the task, the root task, remains in memory and the various segments are brought into an overlay area of memory. In a multi-tasking system there may be several

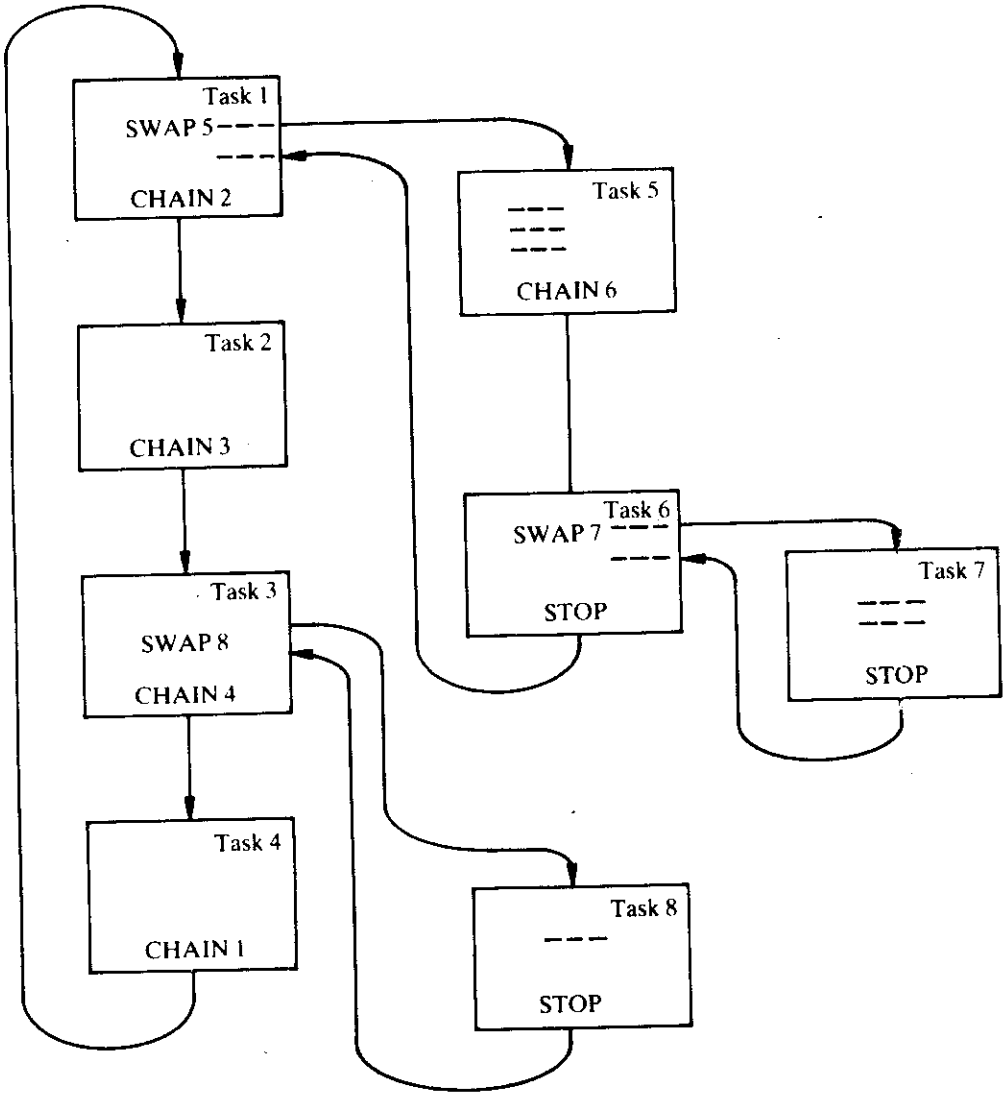


Figure 6.18 Task chaining and swapping.

different overlay areas each of which may be shared by several tasks. A typical arrangement is shown in Figure 6.19 in which it is assumed that two tasks (1 and 15) have overlay segments; each task maintains a root segment and overlay area and the various overlays are loaded into and out of the overlay areas.

Many of the real-time operating systems which provide facilities to swap tasks between fast access memory and backing store were designed for computer systems with small amounts of fast access memory, typically 32K words, and limited memory address space (this is limited by the address lines available on the bus and the CPU architecture). Some of these computer systems have been extended to operate with larger memories by increasing the number of address lines on the bus system and providing memory management units. The way in which a memory management unit operates is to map areas of physical memory onto the actual memory address space. As a consequence the operating systems have been modified to support a larger memory and many do this by using the extended memory to hold the task overlays with a requirement that the root task still be located in the first segment of the memory.

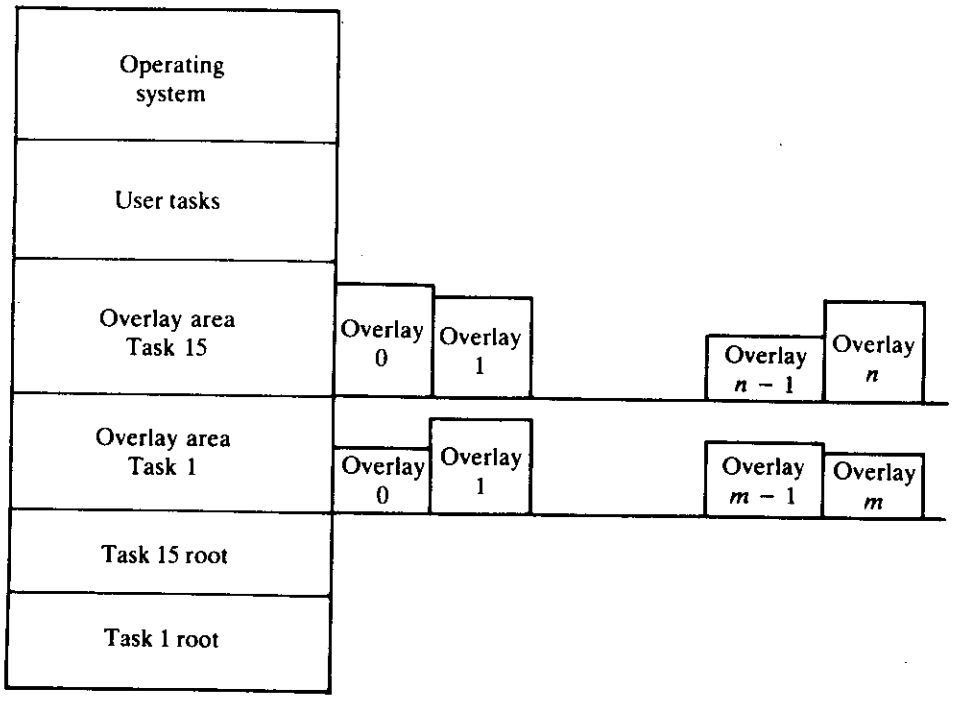


Figure 6.19 Task overlaying.

> Dynamic memory allocation is complex to handle and should be avoided wherever possible in embedded real-time systems. RAM is now so cheap that the cost of adding extra memory is usually much less than the cost of programming to provide dynamic memory allocation. It should **never** be used in safety-critical applications.

6.8 CODE SHARING

In many applications the same actions have to be carried out in several different tasks. In a conventional program the actions would be coded as a subroutine and one copy of the subroutine would be included in the program. In a multi-tasking system each task must have its own copy of the subroutine or some mechanism must be provided to prevent one task interfering with the use of the code by another task. The problems which can arise are illustrated in Figure 6.20. Two tasks share the

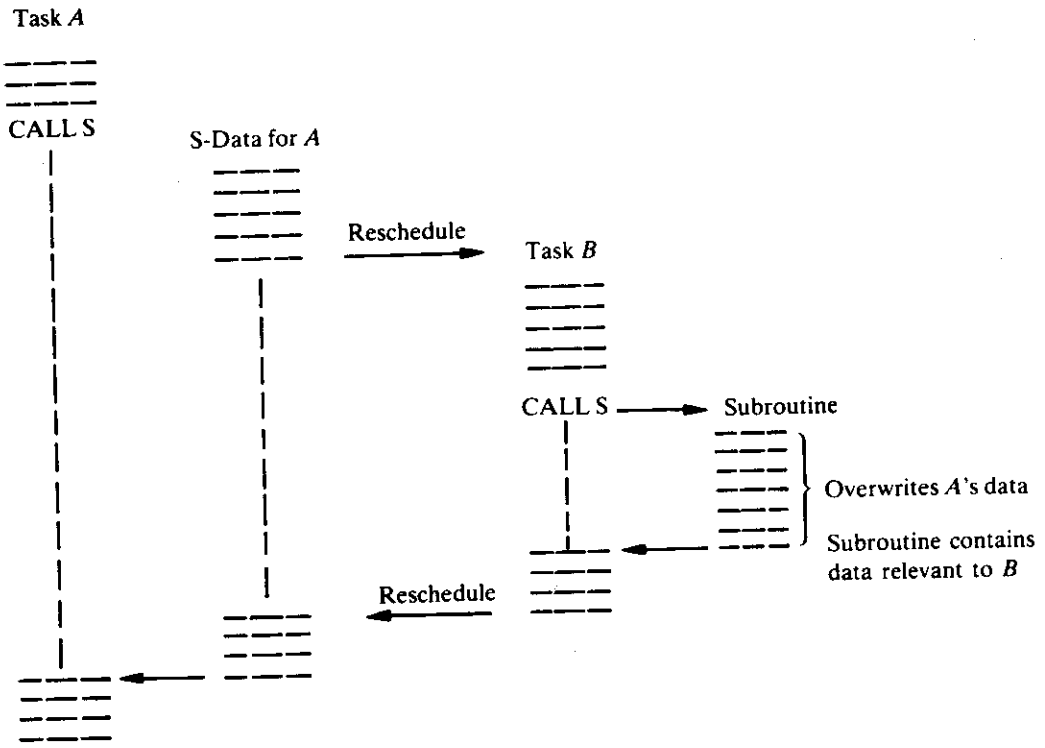


Figure 6.20 Sharing a subroutine in a multi-tasking system.

subroutine S. If task *A* is using the subroutine but before it finishes some event occurs which causes a rescheduling of the tasks and task *B* runs and uses the subroutine, then when a return is made to task *A*, although it will begin to use subroutine *S* again at the correct place, the values of locally held data will have been changed and will reflect the information processed within the subroutine by task *B*.

Two methods can be used to overcome this problem:

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

6.8.1 Serially Reusable Code

As shown in Figure 6.21, some form of lock mechanism is placed at the beginning of the routine such that if any task is already using the routine the calling task will not be allowed entry until the task which is using the routine unlocks it. The use of a lock mechanism to protect a subroutine is an example of the need for mechanisms to support mutual exclusion when constructing an operating system.

6.8.2 Re-entrant Code

If the subroutine can be coded such that it does not hold within it any data, that is it is purely code – any intermediate results are stored in the calling task or in a

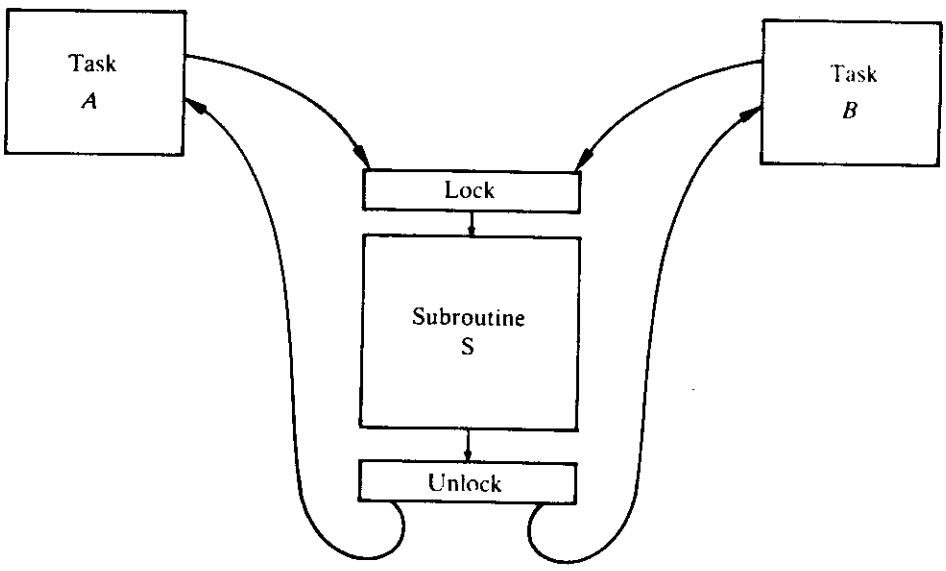


Figure 6.21 Serially reusable code.

stack associated with the task – then the subroutine is said to be re-entrant. Figure 6.22 shows an arrangement which can be used: the task descriptor for each task contains a pointer to a data area – usually a stack area – which is used for the storage of all information relevant to that task when using the subroutine. Swapping between tasks while they are using the subroutine will not now cause any problems since the contents of the stack pointer will be saved with the volatile environment of the task and will be restored when the task resumes. All accesses to data by the subroutine will be through the stack and hence it will automatically manipulate the correct data.

Re-entrant routines can be shared between several tasks since they contain no data relevant to a particular task and hence can be stopped and restarted at a different point in the routine without any loss of information. The data held in the working registers of the CPU is stored in the relevant task descriptor when task swapping takes place.

Device drivers in conventional operating systems are frequently implemented using re-entrant code. Another application could be for the actual three-term (PID)

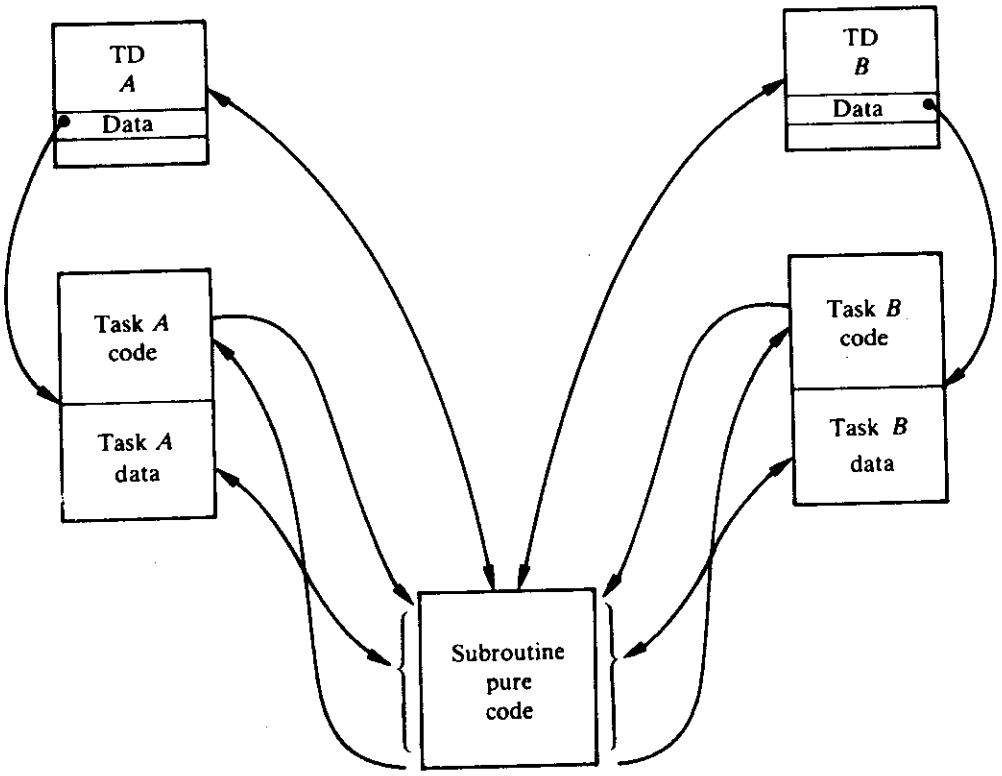


Figure 6.22 Use of re-entrant code for code sharing.

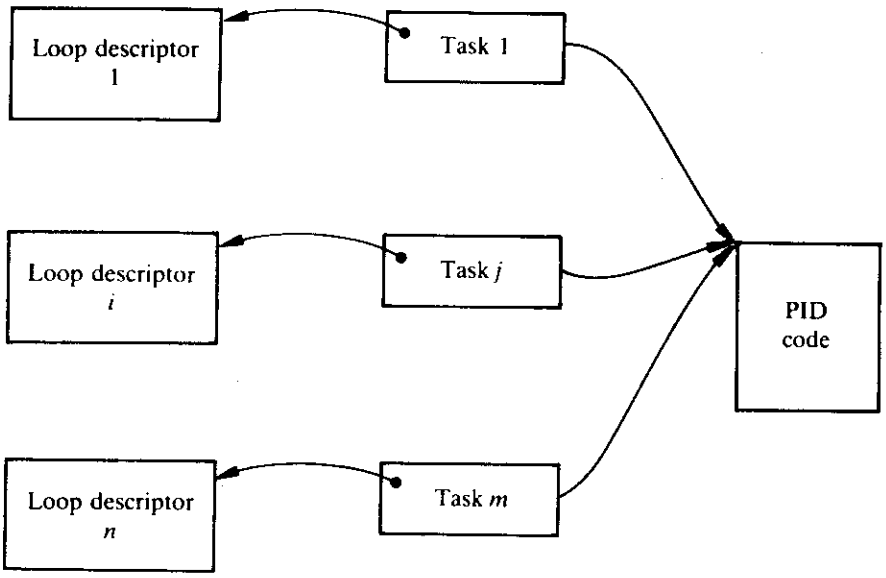


Figure 6.23 Use of re-entrant code in process control.

control algorithm in a process control system with a large number of control loops. The mechanism is illustrated in Figure 6.23; associated with each control loop is a LOOP descriptor as well as a TASK descriptor. The LOOP descriptor contains information about the measuring and actuation devices for the particular loop, for example the scaling of the measuring instrument, the actuator limits, the physical addresses of the input and output devices, and the parameters for the PID controller. The PID controller code segment uses the information in the LOOP descriptor and the TASK to calculate the control value and to send it to the controller. The actual task is made up of the LOOP descriptor, the TASK segment and the PID control code segment. The addition of another loop to the system requires the provision of new loop descriptors; the actual PID control code remains unchanged.

6.9 RESOURCE CONTROL: AN EXAMPLE OF AN INPUT/OUTPUT SUBSYSTEM (IOSS)

One of the most difficult areas of programming is the transfer of information to and from external devices. The availability of a well-designed and implemented input/output subsystem (IOSS) in an operating system is essential for efficient programming. The presence of such a system enables the application programmer to perform input or output by means of system calls either from a high-level

language or from the assembler. The IOSS handles all the details of the devices. In a multi-tasking system the IOSS should also deal with all the problems of several tasks attempting to access the same device.

A typical IOSS will be divided into two levels as shown in Figure 6.24. The I/O manager accepts the system calls from the user tasks and transfers the information contained in the calls to the *device control block* (DCB) for the particular device. The information supplied in the call by the user task will be, for example, the

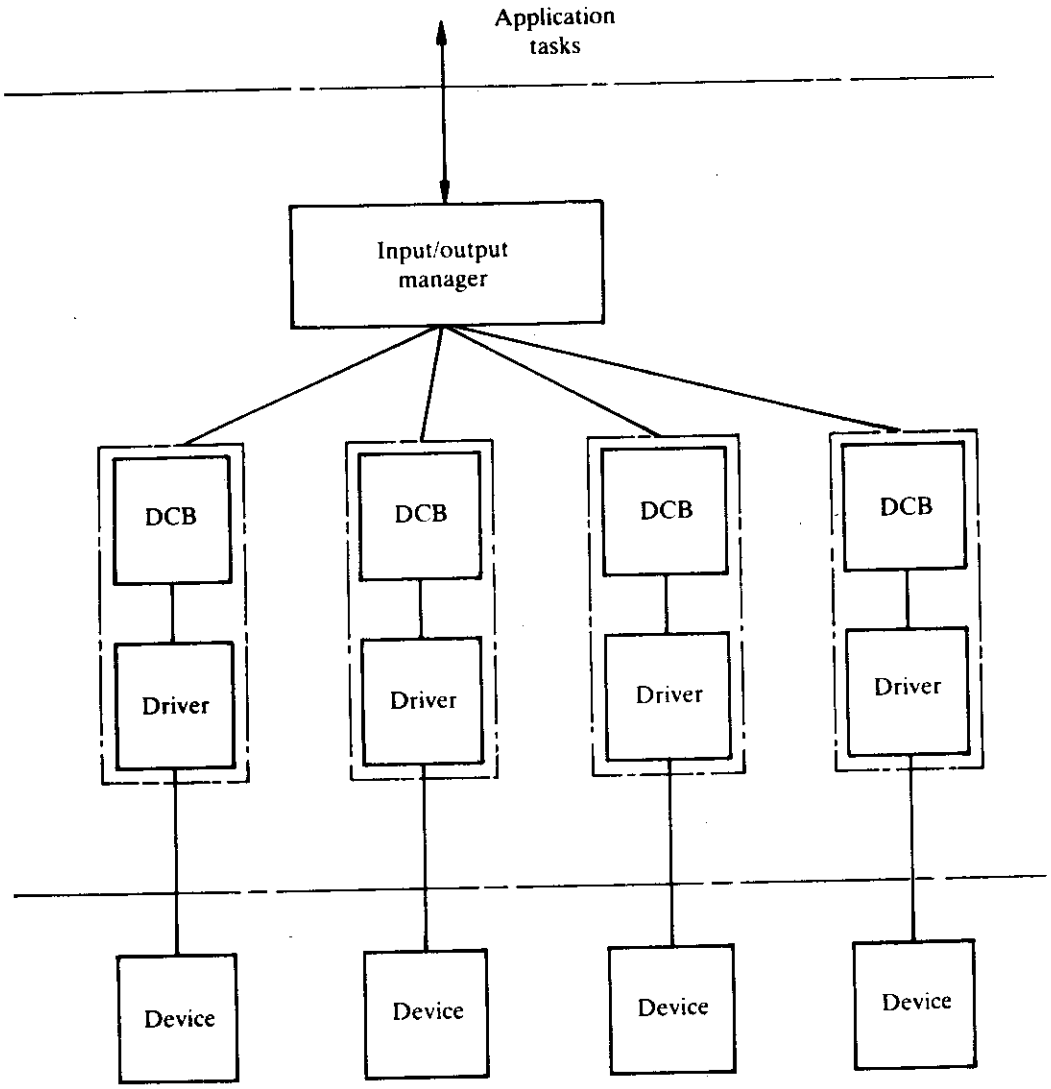


Figure 6.24 General structure of IOSS.

location of a buffer area in which the data to be transferred is stored (output) or is to be stored (input); the amount of data to be transferred; type of data, for example binary or ASCII; the direction of transfer; and the device to be used.

The actual transfer of the data between the user task and the device will be carried out by the device driver and this segment of code will make use of other information stored in the DCB. A separate device driver may be provided for each device or, as is shown in Figure 6.25, a single driver may be shared between several devices; however, each device will require its own DCB. The actual data transfer will usually be carried out under interrupt control.

Typically a DCB will contain the information shown in Table 6.2. The physical device name is the name by which the operating system recognises the device and the type of device is usually given in the form of a code recognised by the operating system. The operating system will normally be supplied with DCBs for the more

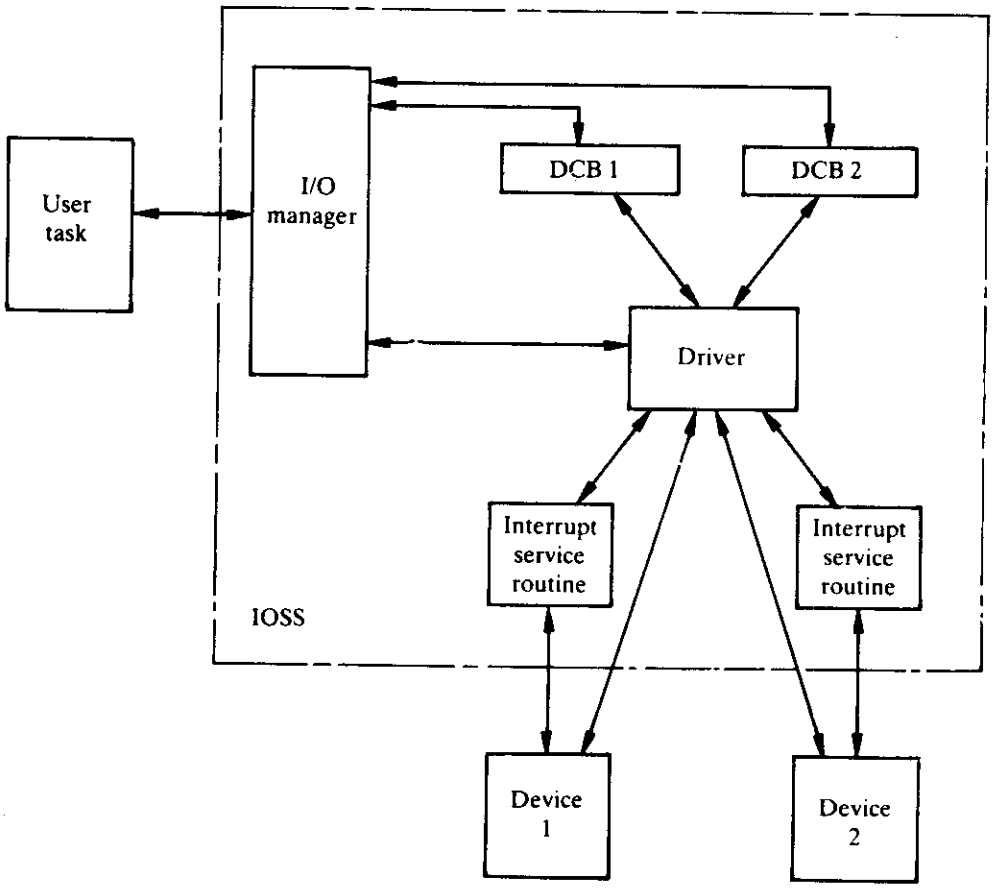


Figure 6.25 Detailed arrangement of IOSS.

Table 6.2 Device control block

Physical device name Type of device Device address Interrupt address Interrupt service routine address Device status	Device-related information
Data area address Bytes to be transferred Current byte count Binary/ASCII	Data-related information

common devices. The DCBs may require modifying to reflect the addresses used in a particular system, although many suppliers adopt the policy of using standard addresses both for the physical address of the device on the bus and for the interrupt locations and interrupt handling routines. The addition of non-standard devices will require the user to provide appropriate DCBs. This task is usually made reasonably simple by providing source code for sample DCBs which can be modified to meet particular needs.

In a multi-tasking system provision has to be made to deal with overlapping requests for a particular device, for example several tasks may wish to send information to the log device – typically a printer. The normal way of handling output to a printer in a single-user environment is to send a record, that is one line, at a time, a return being made to the user program between each line. If this is done in a multi-programming system and the printer is not allocated to the specific program then there is a danger of the output from the different programs becoming intermingled. The solution usually adopted in a multi-user environment is to spool the output, that is it is intercepted by the operating system and stored in a file on the disk. When the program terminates or the user signs off, the contents of the spool file associated with that program or that user are printed out.

A similar solution can be used in a multi-tasking environment providing the user task can force the printing out of the spool file for that task. This addition is needed because in a real-time multi-tasking system tasks may not terminate. Although spooling provides user tasks with the ability to control the interleaving of output, there is still the problem of what action to take if the device is in use when a user task makes a request to use it. There are several possible solutions:

1. Suspend the task until the device becomes available.
2. Return immediately to the task with information that the device is busy and leave it to the task to decide what action to take – normally to call a delay and try again later.
3. Add the request to a device request queue and return to the calling task; the

calling task must check at some later time to see if the request has been completed.

There are advantages and disadvantages to each method and a good operating system will provide the programmer with a choice of actions, although not all options will be available for every device.

Option 1 is referred to as a non-buffered request, in that the user task and the device have to rendezvous. In some ways it can be thought of as the equivalent of hardware handshaking – the user task asks the device 'are you ready?' and waits for a reply from the device before proceeding.

Option 2 is the equivalent of polling and is rarely used.

Option 3 is referred to as a buffered request. It is a form of message passing: the user task passes to the IOSS the equivalent of a letter – this consists of both the message and instructions about the destination of the message – and then the user task continues on the assumption that eventually the message will be delivered, that is sent to the output device. Usually some mechanism is provided which enables the user task to check if the message has been received, that is a form of recorded delivery in which the IOSS records that the message has been delivered and allows the user task to check. Buffered input is slightly different in that the user task invites an external device to send it a message – this can be considered as the equivalent of providing your address to a person or to a group of people. The IOSS will collect the message and deliver it but it is up to the user task to check its 'mail box' to see if a message has been delivered.

6.9.1 Example of an IOSS

The description which follows is of a particular IOSS of an RTOS which supports both computer peripherals – VDUs, printers, disk drives, etc. – and process-related peripherals – analog and digital input and output devices. The system commands used to access the IOSS functions are listed in Table 6.3. In addition to the commands listed in the table there are commands for analog output, for pulse output devices and for incremental output devices.

The IOSS system manager maintains a device request queue for each device and is responsible for interpreting the user task request and placing the appropriate information in the device request queue. If the request is a buffered request, then a return is made immediately to the calling task. If the request is non-buffered, then the IOSS manager changes the status of the calling task to LOCKED OUT and jumps to the dispatcher to begin the search for other work. The IOSS manager, in addition to dealing with requests, has to take action on the completion of a transfer. The driver associated with a given device signals the IOSS manager on completion

Table 6.3 IOSS system commands for RTOS

DTRC01	Disk transfer request – buffered
DTRC02	Disk transfer request – non-buffered
DTRC03	Call to check for completion of buffered request
INRC01	Input request from keyboard device – buffered
INRC02	Call to check for completion of input request
OUCC01	Call to (a) request system data area, i.e. spool area (b) request user data area (c) check status of device (d) check if user data area is free
OURC01	Request output of message to printer or terminal – buffered
FMRC01	Find and reserve area of memory external to the calling task
RMRC01	Release area of memory found using an FMRC01 call
SCRC10	Check if the previously requested buffered scans have been completed
SCRC11	Request non-buffered, non-priority analog scan
SCRC12	Request non-buffered, priority scan
SCRC13	Request buffered, non-priority scan
SCRC14	Request buffered, priority scan
DORC01	Request for a normal digital output, non-priority
DORC02	Request for a normal digital output, priority
DORC03	Request for a timed digital output, non-priority
DORC04	Request for a timed digital output, priority (Note: all the DORC requests can be buffered or non-buffered – the selection is made by setting a parameter for the call)

of a transfer. The IOSS manager, for non-buffered requests, sets the status of the user task which made the request to ON. For buffered requests there are two possible actions: if the calling task has checked to see if the action has been completed before it was completed it will have been placed in the LOCKED OUT state and hence the IOSS treats it as a non-buffered request. If completion occurs prior to a check for completion by the user task, then the IOSS records that the transfer has been completed. When a check is made a return to the calling task will be made with an indication that the transfer is complete. The actual detail of the actions on completion varies for the different types of device.

In addition to dealing with the above, the IOSS manager following completion of a transfer by a device has to check if further requests are waiting in the device request queue; if they are it transfers information to the DCB and initiates the start of transfer before returning to the dispatcher.

6.9.2 Output to Printing Devices

The RTOS provides the programmer with a choice of spooling mechanisms:

System data areas: a number of fixed sized areas on the disk are provided and are identified by a tag number;

User data areas: the user may define data areas of any size which can be in memory or on the disk and are again identified by a tag number.

The system data areas are made available to any user task which requests a data area; the request can be only for a system data area, not one with a specified tag number. A user data area can be assigned to a particular user task. (Note that in this system the assignment is by implication only; all user tasks have to agree that a given tag number applies to a given data area and have to agree that use will be restricted to a given task.) The advantages of a user data area are:

1. the area can be in memory or on disk;
2. the area can be of any size;
3. if use of the area is restricted to one task then it can contain a mixture of permanent and variable data and the user task only needs to transfer the information which has been changed since the last output.

The sequence of operations to be carried out in order to output a message via a user data area is as follows in Example 6.5.

EXAMPLE 6.5

Input From Keyboard

```
; request for user data area
;
   LDA label
   SPB OUCC01 ;system call
   ;return here if area in use
   ;normal return, request accepted
;
label DW parameters ; type of request
         ; tag number
         ; device check yes/no
         ; device number
```

```
;
; transfer of data to data area can take place
  .........
;
; request for output to device
;
  SPB OURC01 ;system call
  LDK labela
;return is made to this location
;
labela DW parameters ;device number
       ;data area type
       ;address of start of data area
;
;further processing can be done during data output
;
  ........
;
;check for completion of transfer
;
  LDA label
  SPB OUCC01
  ;return here if not complete
  ;return here if complete
```

For input the system expects the user task to provide a buffer area in memory to contain the input. The size is limited to a maximum of 256 words and hence an input record from the keyboard, including control characters used to edit the input line, is restricted to 256 characters. The input buffer area can be part of the user task or a separate area of memory found using the `FMRC01` call. In this particular RTOS it is preferable to use a system area provided by the `FMRC01` call, since this allows the user task to be swapped out of memory during the input. The steps involved in the input request are shown on the flowchart in Figure 6. 6 and this also shows the different layers of operation of the operating system. In outline the steps are:

1. Use `FMRC01` call to obtain input buffer area.
2. Request input using `INRC01`.
3. Do other processing if required.
4. Check if input completed using the `INRC02` call – if input is not complete the task will be suspended until it is.
5. Transfer input from buffer area to program area.
6. Release buffer area using `RMRC01` call.

Note that the operating system treats the user task differently when it checks for completion of input compared with the check for completion of output. The reason is that it assumes that a check for completion of input will be made only when there

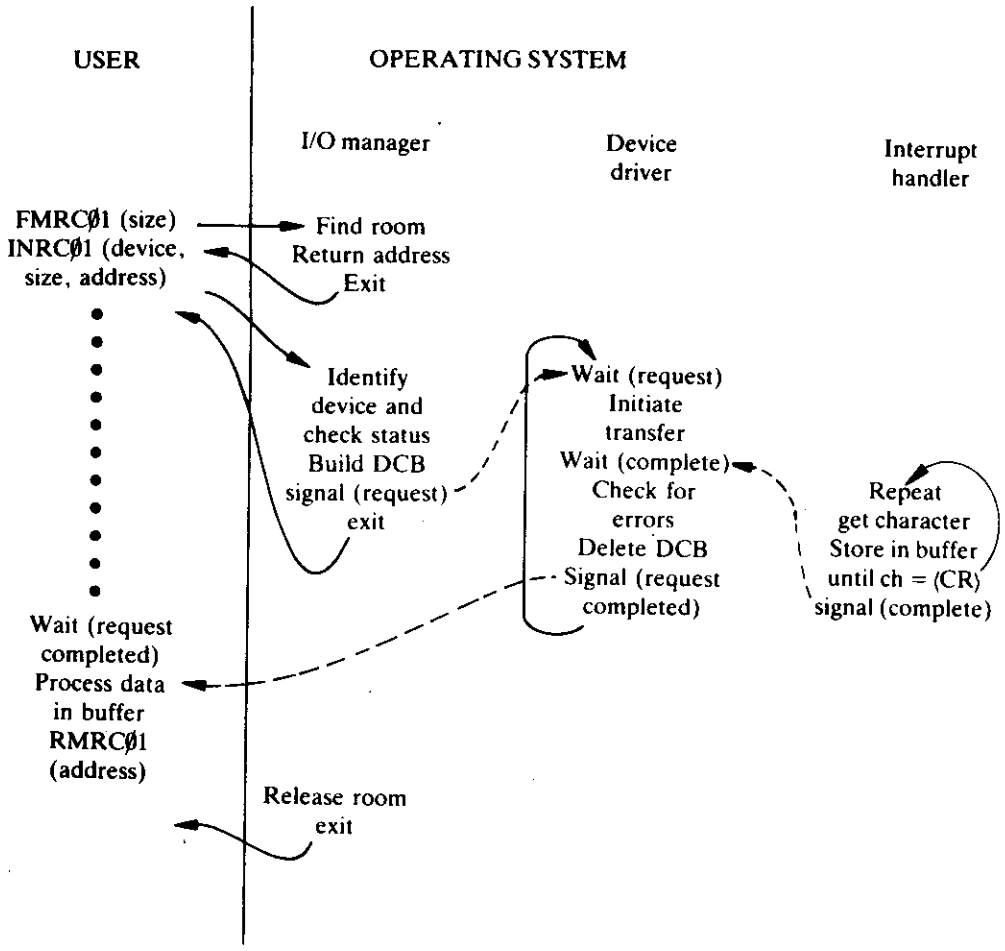


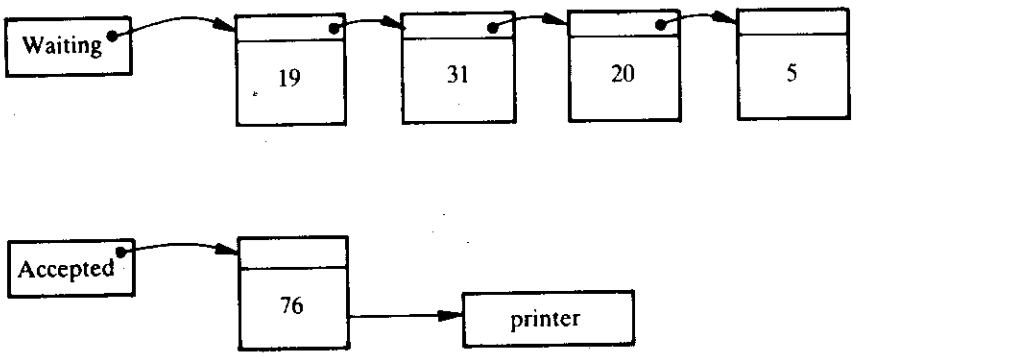
Figure 6.26 IOSS operations for input of data.

is no other work for the task to do; hence if a return to the task on non-completion was made then all the task can do is delay for a short period and then check again – an inefficient procedure. However, on output a task which finds that the previous output is not complete may be able to take some other action, for example set up another data area with a further request and continue.

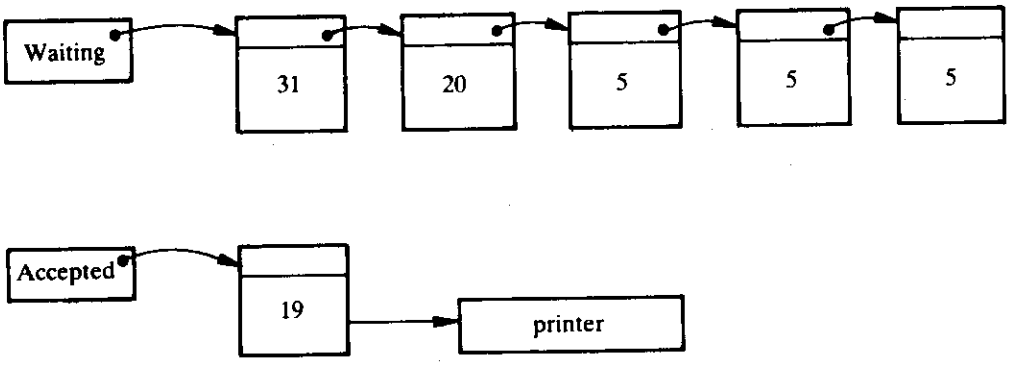
6.9.3 Device Queues and Priorities

In a real-time system a simple device queue based on a first-in-first-out organisation can cause problems in that the task requesting a device effectively loses its priority.

Figures 6.27a and 6.27b illustrate this. In Figure 6.27a a number of tasks are queued waiting for the printer and one task (76) is already using the device. The higher-priority tasks including the very high-priority task 5 will not gain access to the printer until task 76 releases it. If the tasks have made a non-buffered request they will be locked out until they reach the head of the printer queue. However, if task 5 has made a buffered request it will be able to continue and, if it runs frequently, then after a short period of time the printer queue will contain several requests from task 5 as is shown in Figure 6.27b. If the system is not overloaded the printer will eventually catch up with the output from task 5. The delay between the requests



(a)



(b)

Figure 6.27 Printer queue: (a) buffered request; (b) non-buffered request.

from task 5 and the eventual output on the printer can be reduced if the printer queue is organised on a priority basis.

The position regarding priorities becomes even more complicated when the IOSS deals with many devices and hence has several device queues. Decisions on the order in which device queues are serviced are complicated and difficult.

6.10 TASK CO-OPERATION AND COMMUNICATION

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

- mutual exclusion;
- synchronisation; and
- data transfer.

6.11 MUTUAL EXCLUSION

A multi-tasking operating system allows the sharing of resources between several concurrently active tasks. This does not imply that the resources can be used simultaneously. The use of some resources is restricted to only one task at a time. For others, for example a re-entrant code module, several tasks can be using them at the same time. The restriction to one task at a time has to be made for resources such as input and output devices, otherwise there is a danger that input intended for one task could get corrupted by input for another task. Similarly problems can arise if two tasks share a data area and both tasks can write to the data area. This is illustrated by Example 6.6.

EXAMPLE 6.6

Two software modules, `bottle_in_count` and `bottle_out_count`, are used to count pulses issued from detectors which observe bottles entering and leaving a processing area. The two modules run as independent tasks. The two tasks operate on the same variable `bottle_count`. Module `bottle_in_count` increments the variable and `bottle_out_count` decrements it. The modules are programmed in a high-level language and the relevant program language

statements are:

```
bottle_count := bottle_count +1; (bottle_in_count)
bottle_count := bottle_count - 1; (bottle_out_count)
```

At assembler code level the high-level instructions become:

```
{bottle_in_count}      {bottle_out_count}
LD A, (bottle_count)   LD A, (bottle_count)
ADD 1                  SUB 1
LD (bottle_count), A   LD (bottle_count), A
```

Now if variable `bottle_count` contains the value 10, `bottle_in_count` is running and executes the statement `LD A, (bottle_count)` then as Figure 6.28 shows, the `A` register is loaded with the value 10. If the operating system now reschedules and `bottle_out_count` runs it will also pick up the value 10, subtract one from it and store 9 in `bottle_count`. When execution of `bottle_in_count` resumes its environment will be restored and the `A` register will contain the value 10, one will be added and the value 11 stored in `bottle_count`. Thus the final value of `bottle_count` after adding one to it and subtracting one from it will be 11 instead of the correct value 10.

A reg	*bottle_in_count*	*Count*	*bottle_out_count*	*A reg*
?	LD A, (bottle_count)	10		
10	context	10	LD A, (bottle_count)	10
10	change forced	10	SUB 1	9
10	by operating system	10	LD (bottle_count), A	9
10	ADD 1	9		9
11	LD (bottle_count), A	9		9
11		11		9

Figure 6.28 Problem of shared memory (see Example 6.6).

In abstract terms mutual exclusion can be expressed in the form

remainder 1
pre-protocol (necessary overhead)
critical section
post-protocol (necessary overhead)
remainder 2

Remainder 1 and *remainder 2* represent sequential code that does not require access to a particular resource or to a common area of memory.

Critical section is the part of the code which must be protected from interference from another task.

Pre-protocol and post-protocol called before and after the critical sections are code segments that will ensure that the critical section is executed so as to exclude all other tasks.

To benefit from concurrency both the critical section and the protocols must be much shorter than the remainders, so that the remainders represent a significant body of code that can be overlapped with other tasks. The protocols represent an overhead which has to be paid in order to obtain concurrency.

It is implicit in concurrent programming that there is 'loose connection' or 'low coupling' between tasks (see for example Pressman (1992) for a definition of 'loose connection'). Low coupling increases reliability since an error in one task which causes an abnormal termination of that task should not, if there is low coupling, cause other tasks in the system to fail. In abstract terms this can be expressed as the requirement that an abnormal termination in the code forming the remainder should not affect any other task. It would be unreasonable to demand that a failure of the protocol or the critical sections did not affect another task, since the critical section represents the code by which communication or sharing of a resource with another task is taking place.

In considering solutions to mutual exclusion problems it is normal to assume that a number of so-called primitive instructions exist. The correctness of the primitives is assumed to be guaranteed by the language or operating system supporting them. A basic assumption is that a primitive forms an indivisible instruction and hence the task invoking a primitive is guaranteed not to be pre-empted during the execution of the primitive. At the operating system level the basic primitive instruction is the machine code instruction and there is reliance on the CPU hardware to support the indivisibility of an instruction. Furthermore there is also reliance on the hardware implementation of mutual exclusion on the basic access to memory. For example, in a common memory system there will be some form of arbiter which will provide for mutual exclusion in accessing an individual memory location. The arbitration mechanism for a common bus structure was discussed in Chapter 3 in which the CPU controls access to the bus. If, for example, direct memory access is used then the hardware associated with the DMA unit has to inform the CPU when it wants to take control of the bus. Some systems have been designed with memory shared between processors, each of which has its own bus; in these cases dual ported memory devices have been used and the problem of mutual exclusion is thereby transferred to the memory itself. (Note: true dual ported memory allows concurrent access to both processors. Often the memory does not allow true concurrent access; it delays one device for a short period of time. However, to the processor the memory appears to permit concurrent access.)

6.11.1 Semaphore

The most widely used form of primitive for the purposes of mutual exclusion is the binary semaphore. The semaphore mechanism was first proposed by E. W. Dijkstra

in 1968. A binary semaphore is a condition flag which records whether or not a resource is available. If, for a binary semaphore `s`, `s=1`, then the resource is available and the task may proceed; if `s=0` then the resource is unavailable and the task must wait. To avoid the processor wasting time while a task is waiting for a resource to become available there has to be a mechanism for suspending the running of a task when it is waiting and for recording that the task is waiting for a particular semaphore. A typical mechanism for doing this is to associate with each semaphore a queue (often referred to as a condition queue) of tasks that are waiting for a particular semaphore. The use which the operating system makes of this queue is explained more fully in Chapter 7. For the present we will assume that such a queue is created when we declare a semaphore and tasks can be removed from and added to the queue.

There are only three permissible operations on a semaphore, `Initialise`, `Secure`, and `Release`, and the operating system must provide the following procedures:

`Initialise`	`(s:ABinarySemaphore, v:INTEGER)`: set semaphore `s` to value of `v(v=0 or 1)`.
`Secure(s)`:	if `s=1` then set `s:=0` and allow the task to proceed, otherwise suspend the calling task.
`Release(s)`:	if there is no task waiting for semaphore `s` then set `s:=1`, otherwise resume any task that is waiting on semaphore `s`.

The operations `Secure(s)` and `Release(s)` are system primitives which are carried out as indivisible operations and hence the testing and setting of the condition flag are performed effectively as one operation.

Example 6.7 considers a task which wishes to access a printer.

EXAMPLE 6.7

Mutual Exclusion

```
(* Mutual exclusion problem - use of binary semaphore*)
VAR
   printerAccess: SEMAPHORE;
PROCEDURE Task;
BEGIN
   (* remainder1 *)
   Secure(printerAccess)
(*
      if printer is not available task will
      be suspended at this point
*)
(*
      printer available - critical section
*)
(*
      do output
*)
   Release(printerAccess)
   (* remainder2 *)
END Task ;
```

In Figure 6.29 the underlying operations which take place as several tasks attempt to access the same resource (assumed to be a printer) are shown. The binary semaphore, `printerAccess`, is initialised to the value 1 in step 1. As part of this process a three-item record with the semaphore value set to 1 and the pointers to the head and tail of the semaphore queue set to null is created. In step 2, `Task A` performs a `Secure(printerAccess)` operation and the semaphore value is set to 0. Since there was no other task waiting, `Task A` is allowed to use the resource. Sometime later `Task A` suspends and `Task B` performs a `Secure(printerAccess)`, step 3. Since `printerAccess=0` it cannot continue and is added to the semaphore queue by inserting pointers to the task descriptor for `Task B`, in the semaphore control block. If `Task C` now performs a `Secure(printerAccess)`, step 4, then the pointer in the task descriptor for `Task B` is filled in with the address of the TD for `Task C` and the tail pointer entry in the semaphore control block is filled in to point to `Task C`. When `Task A` performs the `Release(printerAccess)` operation, step 5, `Task B` is removed from the semaphore queue and then obtains access to the resource. At step 6, `Task B` performs the `Release(printerAccess)` operation and hence `Task C` is allowed to run and at step 7 when `Task C` performs `Release(printerAccess)` the value of the semaphore is set to 1.

1. Initialise (printerAccess, 1)

printerAccess

Value = 1
Head
Tail

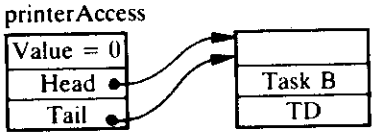
2. Task A active Secure (printerAccess)

printerAccess

Value = 0
Head
Tail

3. Task A suspends, Task B active Secure (printerAccess)

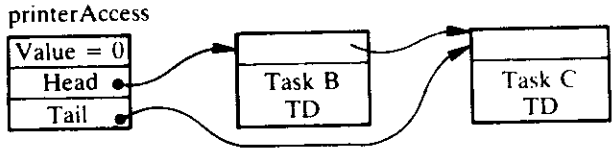
printerAccess



Task B is suspended and placed in printerAccess queue

4. Task C runs attempts Secure (printerAccess)

printerAccess



Task C is suspended and added to printerAccess queue

5. Task A runs, Release (printerAccess)

printerAccess

Task B is removed from printerAccess queue and placed in Ready queue

6. Task B runs, Release (printerAccess)

printerAccess

Value = 0
Head
Tail

Task C is transferred from printerAccess queue to Ready queue

7. Task C runs, Release (printerAccess)

printerAccess

Value = 1
Head
Tail

Printer is available to any task

Figure 6.29 Mutual exclusion using binary semaphore.

6.11.2 The Monitor

Although the binary and general semaphore provide a simple and effective means of enforcing mutual access they have one weakness: in use they are scattered around the code. Each task that requires access to a particular resource has to know the details of the semaphore used to protect that resource and to use it. The onus on correct use is placed on the designer and implementer of each task. In a small system this causes few problems but in larger, more complex systems where the tasks involved may be divided between several people use of the semaphore becomes more difficult and the probability of introducing errors increases.

An alternative solution which associates the control of mutual exclusion with the resource rather than with the user task is the monitor, introduced by Brinch Hansen (1973, 1975) and by Hoare (1974). A monitor is a set of procedures that provide access to data or to a device. The procedures are encapsulated inside a module that has the special property that only one task at a time can be actively executing a monitor procedure. It can be thought of as providing a fence around critical data. The operations which can be performed on the data are moved inside the fence as well as the data itself. The user task thus communicates with the monitor rather than directly with the resource.

Figure 6.30 shows an example of a simple monitor. Two procedures, `WriteData` and `ReadData`, provide access to the data. These procedures represent gates through which access to the monitor is obtained. The monitor prevents any other form of access to the critical data. A task wishing to write data calls the procedure `WriteData` and as long as no other task is already accessing the monitor it will be allowed to enter and write new data. If any other task was already using either the `WriteData` or `ReadData` operations then the task would be halted at the gate and suspended, since only one task at a time is allowed to be within the monitor fence.

Figure 6.31 shows a more complicated monitor with three entry points, `Entry 1`, `Entry 2` and `Entry 3`, and two conditions on which tasks which have gained entry may have to wait. In the figure, one task, `T15`, is in the monitor and three tasks are waiting to enter, two at entry point 1 – `T16` and `T2` – and one at entry

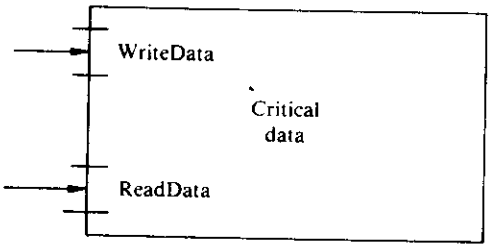


Figure 6.30 A simple monitor.

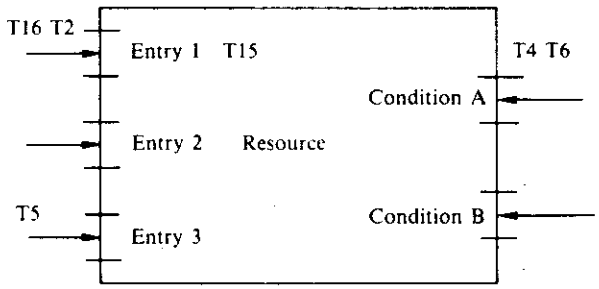


Figure 6.31 A general monitor.

point 3 – T5. Two tasks have previously entered and have been suspended waiting for condition A. There are no tasks waiting at entry point 2 or for condition B.

The advantage of a monitor over the use of semaphores or other mechanisms to enforce mutual exclusion is that the exclusion is implicit; the only action required by the programmer of the task requiring to use the resource is to invoke the entry to the monitor. If the monitor is correctly coded then an applications program cannot use a resource protected by a monitor incorrectly.

6.11.3 Intertask Communication

We can divide the issues of synchronisation and communication into three areas:

- synchronisation without data transfer;
- data transfer without synchronisation; and
- synchronisation with data transfer.

6.11.4 Task Synchronisation Without Data Transfer

Frequently one wishes to be able to inform another task that an event has occurred, or to set a task to wait for an event to occur. No data needs to be exchanged by the tasks. A mechanism that enables this to be done is the so-called *signal*:

> A `signal s` is defined as a binary variable such that if `s=1` then a signal has been sent but has not yet been received.

Associated with a signal is a queue and the permissible operations on a signal are:

`Initialise (s:Signal: v:INTEGER)`	set `s` to the value of `v` (0 or 1).
`Wait(s)`	if `s=1` then `s:=0` else suspend the calling task and place it in the condition queue `s`.
`Send(s)`	if the condition queue `s` is empty then `s:=1` else transfer the first task in the condition queue to the ready queue.

Clearly a signal is similar to a semaphore; in fact the difference between the two is not in the way in which they are implemented but in the way in which they are used. A semaphore is used to secure and release a resource and as such the calls will both be made by one task; a signal is used to synchronise the activities of two tasks and one task will issue the send and the other task the wait. (Note that a signal is sometimes implemented such that if a task is not waiting it has no effect, that is the receipt of a signal is not remembered.)

In practice two important additions to the basic *signal* mechanism are required: one is the facility to check to see if a task is waiting to send a signal, and the other is to be able to restrict the length of time which a task waits for a signal to occur. In a real-time application it is rarely correct for a task to be committed to wait indefinitely for an event to occur. The standard `Wait(s)` commits a task to an indefinite wait.

6.12 DATA TRANSFER (THE PRODUCER–CONSUMER PROBLEM)

6.12.1 Data Transfer Without Synchronisation

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

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

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

It is normal to create a large number of pools so as to limit the use of global common data areas. To avoid the problem of two or more tasks accessing a pool simultaneously mutual exclusion on pools is required. The most reliable form of mutual exclusion for a pool is to embed the pool inside a monitor. Given that the read operation does not change the data in a pool there is no need to restrict read access to a pool to one task at a time.

Channels provide a direct communication link between tasks, normally on a one-to-one basis. The communication is like a pipe down which successive collections of items of data – messages – can pass. Normally they are implemented so that they can contain several messages and so they act as a buffer between the tasks. One task is seen as the *producer* of information and the other as the *consumer*. Because of the buffer function of the channel the producer and consumer tasks can run asynchronously.

There are two basic implementation mechanisms for a channel:

- queue (linked list); and
- circular buffer.

The advantage of the queue is that the number of successive messages held in the channel is not fixed. The length of the queue can grow, the only limit being the amount of available memory. The disadvantage of the queue is that as the length of the queue increases the access time, that is the time to add and remove items from the queue, increases. For this reason and because it is not good practice to have undefined limits on functions in real-time systems queues are rarely used.

The circular buffer uses a fixed amount of memory, the size being defined by the designer of the application. If the producer and consumer tasks run normally they would typically add and remove items from the buffer alternately. If for some reason one or the other is suspended for any length of time the buffer will either fill up or empty. The tasks using the buffer have to check, as appropriate, for buffer full and buffer empty conditions and suspend their operations until the empty or full condition changes.

As an example let us consider an alarm scanning task which for a period of time produces data at a rate much greater than that at which the logging task can print it out. A buffer is needed to store the data until the consuming task is ready to take it. The system is shown diagrammatically in Figure 6.32. We assume that the buffer is bounded, that is of finite size, and that the operation of storing an item of data in it is performed by the call

```
Put(x)
```

and an item of data is removed by the call

```
Get(x)
```

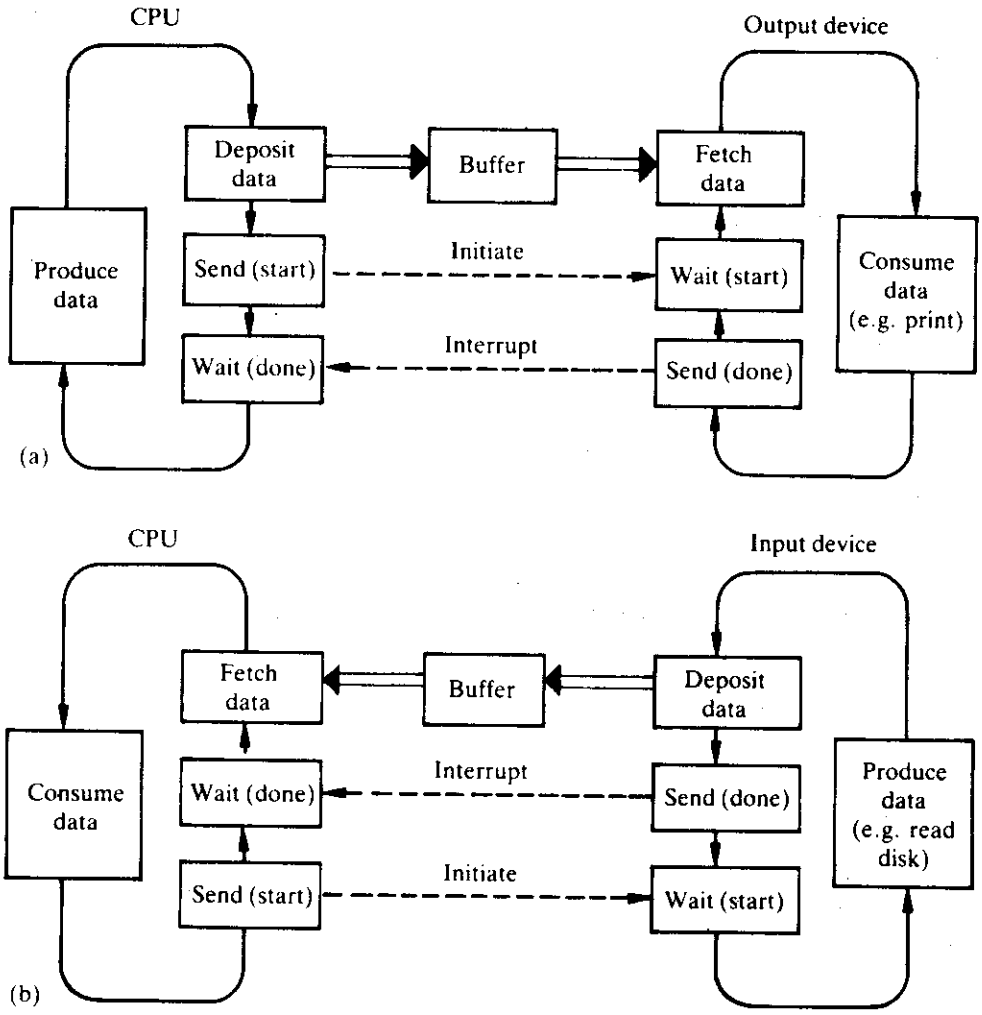


Figure 6.32 Input (a) and output (b) device model (from Young, *Real-time Languages*, Ellis Horwood (1982)).

Since the buffer is of finite size it is necessary to know when it is full and when it is empty. The following function calls are used:

`Full` – which returns the value true if the buffer is full;
`Empty` – which returns the value true if the buffer is empty.

Let us assume that the producer and consumer are formed by separate tasks which share a common buffer area.

EXAMPLE 6.8

Producer–Consumer Problem

```
(* Producer-consumer problem - solution 1*)
VAR commonBuffer : buffer;
TASK Producer;
VAR x:data;
BEGIN
   LOOP
      Produce(x);
      WHILE Full DO
      Wait
      END (* while *);
      Put(x);
   END (* loop *);
END Producer;
TASK Consumer;
VAR x:data;
BEGIN
   LOOP
      WHILE Empty DO
      Wait
      END (* while *);
      Get(x);
      Consume(x);
   END (* loop *);
END Consumer;
```

The producer operates in an endless cycle producing some item x and waiting until the buffer is not full to place x in the buffer; the consumer also operates in an endless cycle waiting until the buffer is not empty and removing item x from the buffer.

The solution in Example 6.8 is not satisfactory for two reasons:

1. the `Put(x)` and `Get(x)` are both operating on the same buffer and for security of the data simultaneous access to the buffer cannot be allowed – the mutual exclusion problem;
2. both the producer and the consumer use a 'busy wait' in order to deal with the buffer full and buffer empty problem.

The first problem can be solved using the semaphore, with the operations secure and release. The second problem can be solved by using the `signal` mechanism described above.

Because of the need to test for empty and full and to suspend the task if one or the other condition appertains then although transfer of a data item is not

synchronised there is task synchronisation on the buffer full and buffer empty conditions.

This is illustrated in Example 6.9.

EXAMPLE 6.9

Producer–Consumer Problem – Solution 2

```
(*
Data transfer problem - solution 2 using semaphores and
signals
*)
VAR commonBuffer : Abuffer;
   bufferAccess : ABinarySemaphore;
   nonFull, nonEmpty : Signal;
TASK Producer;
VAR x:data;
BEGIN
   LOOP
      Produce(x);
      Secure(bufferAccess);
      IF Full THEN
         Release(bufferAccess);
         Wait(nonFull);
         Secure(bufferAccess);
      END (*if*);
      Put(x);
      Release(bufferAccess);
      Send(nonEmpty);
   END (*loop*);
END Producer;
TASK Consumer;
VAR x:data;
BEGIN
   LOOP
      Secure(bufferAccess);
      IF Empty THEN
         Release(bufferAccess);
         Wait(nonEmpty);
         Secure(bufferAccess);
      END (* if *);
      Get(x);
      Release(bufferAccess);
      Send(nonFull);
      Consume(x);
   END (*loop*);
END Consumer;
```

In this example the critical code is enclosed between secure and release operations but it is essential that the `bufferAccess` semaphore is released before executing the `Wait(nonFull)` or `Wait(nonEmpty)` primitives. If this is not done the system will deadlock. For example, if the `Producer` executes `Wait(nonFull)` while holding the access rights to the buffer then the buffer can never become non-full since the only way it can is for the `Consumer` to remove an item of data, but the `Consumer` cannot gain access to it until it is released by the `Producer`.

Both semaphores and signals can be generalised to allow a semaphore or a signal variable to have any non-negative integer value – in this form they are sometimes referred to as counting semaphores.

6.12.2 Synchronisation With Data Transfer

There are two main forms of synchronisation involving data transfer. The first involves the producer task simply signalling to say that a message has been produced and is waiting to be collected, and the second is to signal that a message is ready and to wait for the consumer task to reach a point where the two tasks can exchange the data.

The first method is simply an extension of the mechanism used in the example in the previous section to signal that a channel was empty or full. Instead of signalling these conditions a signal is sent each time a message is placed in the channel. Either a generalised semaphore or signal that counts the number of sends and waits, or a counter, has to be used.

Two examples of buffers written in Modula-2 are shown in Figures 6.33 and 6.34.

6.13 LIVENESS

An important property of a multi-tasking real-time system is *liveness*. A system (a set of tasks) is said to possess liveness if it is free from

livelock,
deadlock, and
indefinite postponement.

Livelock is the condition under which the tasks requiring mutually exclusive access to a set of resources both enter *busy wait* routines but neither can get out of the *busy wait* because they are waiting for each other. The CPU appears to be doing useful work and hence the term livelock.

Deadlock is the condition in which a set of tasks are in a state such that it is

```
IMPLEMENTATION MODULE Buffer;
(*
Title   : Implementation of a buffer using a monitor
File    : buffert1.mod
*)
FROM Monitor IMPORT
  monitorPriority;
 FROM Semaphores IMPORT
  Claim, InitSemaphore, Release, Semaphore;
 (* following is required for display of contents *)
  FROM Ansi IMPORT
   WriteCh;
 (* end of display *)
CONST moduleName='bufferT1';

MODULE BufferM [monitorPriority];
IMPORT
 Claim, InitSemaphore, Release, Semaphore, WriteCh;
EXPORT Put, Get;
 CONST
  nMax=10;
 VAR
  nFree, nTaken : Semaphore;
  in, out : [1..nMax];
  b: ARRAY [1..nMax] OF CHAR;
 (* following variables are required only for demonstration purposes *)
  row, col : CARDINAL;
  PROCEDURE Put(ch : CHAR);
  BEGIN
   Claim(nFree);
   b[in]:=ch;
   in:=in MOD nMax+1;
    WriteCh(ch, row, col+in) (* display purposes only *);
   Release(nTaken)
  END Put;
  PROCEDURE Get(VAR ch:CHAR);
  BEGIN
   Claim(nTaken);
   ch:=b[out];
   out:=out MOD nMax+1;
    WriteCh(' ', row, col+out) (* display purposes *);
   Release(nFree)
  END Get;
BEGIN
 row:=15; col:=20 (* initialise display part *);
 in:=1; out:=1;
 InitSemaphore(nFree, nMax);
 InitSemaphore(nTaken, 0);
END BufferM;
END Buffer.
```

Figure 6.33 IMPLEMENTATION MODULE of a buffer using a monitor.

```
IMPLEMENTATION MODULE Buffer;
(*
Title   : Implementation of a buffer using a semaphore
File    : buffert2.mod
*)
 FROM Semaphores IMPORT
  Claim, InitSemaphore, Release, Semaphore;
 (* following is required for display of contents *)
  FROM Ansi IMPORT
   WriteCh;
 (* end of display *)
CONST moduleName='bufferT2';
  CONST
   nMax=10;
 VAR
  nFree, nTaken, inPut, inGet : Semaphore;
  in, out : [1..nMax];
  b: ARRAY [1..nMax] OF CHAR;
 (* following variables are required only for demonstration purposes *)
  row, col : CARDINAL;
  PROCEDURE Put(ch : CHAR);
  BEGIN
   Claim(inPut);
   Claim(nFree);
   b[in]:=ch;
   in:=in MOD nMax+1;
    WriteCh(ch, row, col+in) (* display purposes only *);
   Release(nTaken);
   Release(inPut);
  END Put;
  PROCEDURE Get(VAR ch:CHAR);
  BEGIN
   Claim(inGet);
   Claim(nTaken);
   ch:=b[out];
   out:=out MOD nMax+1;
    WriteCh(' ', row, col+out) (* display purposes *);
   Release(nFree);
   Release(inGet);
  END Get;
BEGIN
 row:=15; col:=20 (* initialise display part *);
 in:=1; out:=1;
 InitSemaphore(nFree, nMax);
 InitSemaphore(nTaken, 0);
 InitSemaphore(inPut, 1);
 InitSemaphore(inGet, 1);
END Buffer.
```

Figure 6.34 IMPLEMENTATION MODULE of a buffer using a semaphore.

impossible for any of them to proceed. The CPU is free but there are no tasks that are ready to run. As an example of how deadlock can occur consider the following.

> Suppose task A has acquired exclusive use of resource X and now requests resource Y, but between A acquiring X and requesting Y, task B has obtained exclusive use of Y and has requested use of X. Neither task can proceed, since A is holding X and waiting for Y and B is holding Y and waiting for X.

The detection of deadlock or the provision of resource sharing commands in such a way as to avoid deadlock is the responsibility of the operating system (see Lister (1979, pp. 94–7) for a discussion of deadlock avoidance and detection mechanisms).

Indefinite postponement is the condition that occurs when a task is unable to gain access to a resource because some other task always gains access ahead of it.

6.14 MINIMUM OPERATING SYSTEM KERNEL

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

Functions:

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

Primitives:

`WAIT` for some condition (including release of exclusive access rights).
`SIGNAL` condition and thus release one (or all) tasks waiting on the condition.
`ACQUIRE` exclusive rights to a resource (option – specify a time-out condition).
`RELEASE` exclusive rights to a resource.
`DELAY` task for a specified time.
`CYCLE` task, that is suspend until the end of its specified cyclic period.

6.15 EXAMPLE OF CREATING AN RTOS BASED ON A MODULA-2 KERNEL

The standard module `Processes` suggested by Wirth (1986) and supplied by most systems is not all that versatile. Many alternative versions offering a wider range of

facilities have been developed. An example of one such system is described below. It was developed by Roger Henry, of Nottingham University.

The lowest-level module is `Processes` (a replacement for the Wirth module `Processes`) which provides the procedures and functions

`Cp`	– return current process identity
`Disable`	– disable a given process
`Enable`	– make a given task runnable
`MinWksp`	– return the minimum workspace size for task
`NewProcess`	– create a new task
`PriorityOf`	– return the priority of a task
`SuspendMe`	– suspend the calling task
`SuspendUntilInterrupt`	– suspend the calling task until a specific interrupt occurs.

The relationship between the procedures and the state of the tasks is shown in Figure 6.35 – `NewProcess` is used to inform `Processes` of the existence of a task, but before it can be run it must be made runnable by a call to `Enable`. In the call `NewProcess` a task is allocated a priority (an integer value – the range depending on the implementation) and the runnable task with the highest priority becomes the running task. A running task can call `Disable` which will cause a named runnable task to be changed to existent (non-runnable). If the named task is not runnable the call will be ignored; to make itself non-runnable the running task uses the call `SuspendMe`. A running task may also suspend itself to wait for a hardware

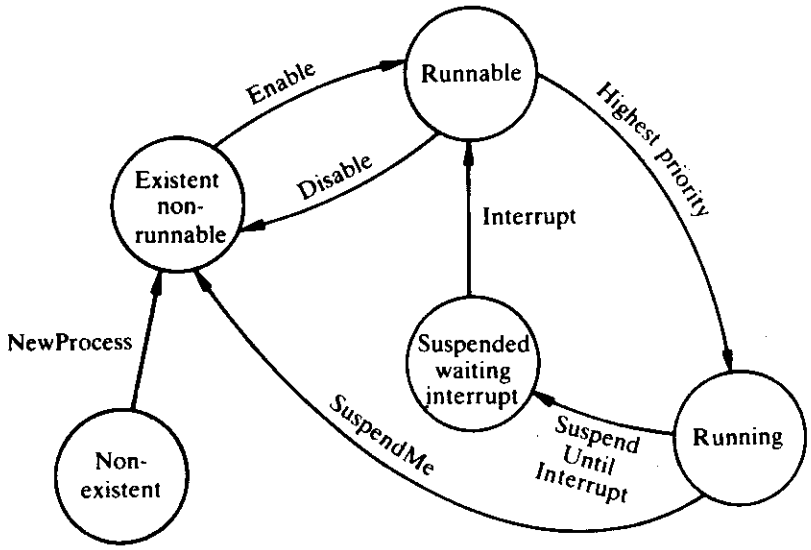


Figure 6.35 Task states and transitions for a Modula-2 kernel.

interrupt by using the call `SuspendUntilInterrupt` in which case it will not become runnable until the specified hardware interrupt occurs and is accepted.

For simple applications a high-level module, `Scheduler`, is provided which has two procedures:

`StartProcess` – equivalent of `NewProcess` and `Enable`, it makes a task known to `Processes` and makes it runnable;
`StopMe` – stops the current task.

The separate modules of `Signals`, `Semaphores` and `Timing` allow the tasks started by `Scheduler` to synchronise and to run at specified time intervals.

The module `Signals` provides the standard operations on signals (`Initialise`, `Wait` and `Send` – the names used are `InitSignal`, `SendSignal` and `AwaitSignal`) but in addition two further operations are supported:

`Awaited` – returns true value if at least one task is waiting for the `SendSignal` operation;
`SentWithin` – the caller will only wait for a specified length of time; the function returns a true value if the signal was sent within the specified time.

The module `Semaphore` supports the standard semaphore operations – the names used are `InitSemaphore`, `Claim` and `Release`.

The module `Timing` enables users to operate in absolute and relative time intervals. Absolute time begins when the `Timing` module is initialised and one value of absolute time can be said to be earlier or later than another. The difference between two values of absolute time – a time interval – is said to be relative time and one interval can be said to be longer or shorter than another.

Time – both absolute and relative – is measured in units of seconds and ticks. The number of ticks in a second is implementation defined (it depends upon the system clock used) and its value is returned by the function `TicksPerSecond`. The current time (absolute) can be found using the procedure `TellTime`. Two procedures are provided to enable tasks to wait for specified times:

`DelayFor` – the task waits for a specified time interval
`DelayUntil` – the task waits until a specified absolute time.

In both cases the calling task is suspended until either the time interval has elapsed or the absolute time is reached; the task is then made runnable – there is no guarantee that the task will run immediately on attaining the specified condition since `Processes` will choose the task with the highest priority.

In calculating absolute times or time intervals the value of time in seconds and ticks has to be manipulated. To support such operations the module `TimeOps` provides the following procedures:

`IncTime` – increase a time value by a given interval
`IncInterval` – increase an interval value by a given interval

`DecTime` – decrease a time value by a given interval
`DecInterval` – decrease an interval value by a given interval
`DiffTimes` – subtract second time from the first time
`DiffIntervals` – subtract second interval from first interval
`CompareTimes` – compare first time with second time
`CompareIntervals` – compare first interval with second interval.

A simple example of the use of the RTOS kernel facilities is given in Figure 6.36. Two tasks, the task forming the main program and the task formed by procedure

```
MODULE TwoTasks;
(*
Title   : Example of two tasks synchronising using signals
File    : sb1 twotasks.mod
*)
FROM Scheduler IMPORT
 ProcessId, Priority, StartProcess, StopMe;

FROM Signals IMPORT
 AwaitSignal, SendSignal, InitSignal, Signal;

FROM InOut IMPORT
 WriteString, WriteLn;

 CONST
  priority=2;
  worksp=600;
 VAR
  messageSent : Signal;
  count : CARDINAL;
  taskTwoId : ProcessId;

  PROCEDURE TaskTwo;
  BEGIN
   LOOP
    AwaitSignal(messageSent);
    WriteString(' message received by task two');
    WriteLn
   END (* loop *);

  END TaskTwo;

BEGIN (* body of program forms task 1 *)
 InitSignal(messageSent);
 StartProcess(TaskTwo, priority, worksp, taskTwoId);
 FOR count:=1 TO 10 DO
  WriteString('Sending message');
  SendSignal(messageSent)
 END (* for *);

END TwoTasks.
```

Figure 6.36 Two tasks synchronised by using signals.

TaskTwo, are run alternately, synchronised by the use of the signal messageSent. The output is ten lines of 'Sending message' – output by the main task – and 'message received by task two' – output by TaskTwo. TaskTwo is started with a priority of level 2 by the use of the procedure Scheduler.StartProcess. The main body of the program is automatically run at priority level 0.

Some of the features of the RTOS kernel can be explored by using the example program shown in Figure 6.37. Two tasks, TaskA and TaskB, are created. TaskA prints out on the screen a number of rows of the letter A; the number is specified in the constant numberOfLines. TaskB prints out a number of rows of the

```
MODULE RTS3;
(*
Title   : Demonstration of resource sharing
File    : sb1: RTS3.mod
*)
FROM InOut IMPORT
 Write, WriteLn, WriteString;

FROM Semaphores IMPORT
 InitSemaphore, Semaphore, Claim, Release;

FROM Scheduler IMPORT
 ProcessId, Priority, StartProcess, StopMe;

FROM Timing IMPORT
 DelayFor, DelayUntil, Interval, TellTime, Time, TicksPerSecond;

CONST moduleName='RTS3';
 numberOfLines=5;
VAR
 screen : Semaphore;
 endA, endB : BOOLEAN;

 PROCEDURE TaskA;
 CONST ch='A';
 VAR
  i,j : CARDINAL;
  delayA : Interval;
 BEGIN
  i:=0; j:=0; delayA.secs:=0; delayA.ticks:=1;
  LOOP
   Claim(screen); (* omit in versions 1 and 2 *)
   FOR i:=1 TO 79 DO
    Write(ch);
    DelayFor(delayA) (* omit in version 1 *)
   END (* for *);
   WriteLn;
   Release(screen); (* omit in versions 1 and 2 *)
   INC(j);
   IF j>numberOfLines THEN
    EXIT
   END (* if *);
```

```
 END (* loop *);
 endA:=TRUE;
 StopMe;
END TaskA;
PROCEDURE TaskB;
CONST ch='B';
VAR
 i,j : CARDINAL;
 delayB : Interval;
BEGIN
 i:=0; j:=0; delayB.secs:=0; delayB.ticks:=1;
 LOOP
   Claim(screen); (* omit in versions 1 and 2 *)
   FOR i:=1 TO 79 DO
    Write(ch);
    DelayFor(delayB); (* omit in version 1*)
   END (* for *);
   WriteLn;
   Release(screen); (* omit in versions 1 and 2*)
   INC(j);
   IF j>numberOfLines THEN
    EXIT
   END (* if *);

  END (* loop *);
  endB:=TRUE;
  StopMe;
 END TaskB;
CONST
 priorityA=1;
 priorityB=1;
 wkspSizeA=1000;
 wkspSizeB=1000;

VAR
 taskAId, taskBId : ProcessId;
BEGIN
 WriteString(moduleName);
 WriteLn;
 endA:=FALSE; endB:=FALSE;
 InitSemaphore(screen, 1);
 StartProcess(TaskA, priorityA, wkspSizeA, taskAId);
 StartProcess(TaskB, priorityB, wkspSizeB, taskBId);
 LOOP
  (* Idle process *)
  IF endA AND endB THEN
   EXIT
  END (* if *);
 END (* loop *);
 WriteLn;
 WriteString('Program end');
END RTS3.
```

Figure 6.37 Example showing resource sharing.

letter `B`. The display which is obtained on the screen depends on the way the tasks are scheduled and whether the tasks are given exclusive access to the screen. By compiling the module in version 1 form (not using the semaphore and leaving out the `DelayFor` calls), the scheduler treats the two tasks as coroutines. Hence `TaskA`, which is started first, gains control and runs to completion; only then does `TaskB` run. The result of this is that first several rows of `A`s are displayed on the screen followed by several rows of `B`s. The reason for this behaviour is that the scheduler continues to run a task until either a higher-priority task wishes to run or until the running task suspends or ends. Introducing the `DelayFor` statements to form version 2 of the program causes the two tasks to run alternately giving an alternating sequence of `A`s and `B`s. You are invited to work out what the output will be if the `Claim(screen)` and `Release(screen)` statements are included.

The creation of modules for specific purposes is in keeping with the Modula-2 philosophy. The aim is that the core language should remain fixed and standard and that any extensions required for special purposes should be provided in the form of library modules. It should be noted that all the input/output, file handling and other operations are not handled as part of the language, but by standard procedures imported from modules which are assumed to be provided as part of the system.

6.16 SUMMARY

In this chapter we have concentrated on describing the features to be found in traditional operating systems. Such operating systems are usually specific to a particular computer, or range of computers. Examples are the Digital Equipment Corporation's RT/11 and RSX/11 operating systems for the PDP-11 series; the Data General RTOS and RDOS for the Nova range; and more recently RMX-80 for the Intel 8080 range and OS-9 for the Motorola 68XXXX series.

The advantage of many of the traditional operating systems is their wide user base and the fact that they have seen extensive use in control applications. There has, however, been a tendency for the size of the operating systems to increase with each successive upgrade and it is often difficult to create small subsets for a particular application. Another disadvantage with many is that access to the system from high-level languages is very restricted and the addition of new devices normally requires hardware drivers to be written in assembler.

The development of the MASCOT (see Chapter 9) environment represents one way in which some of the problems of lack of standardisation and difficulty of accessing operating system functions have been addressed. A similar approach has been taken by Baker and Scallon (1986) with the Rex architecture. As with MASCOT the Rex system presents the user with a virtual machine which hides the details of the operating system and the hardware. The detailed procedures required for carrying out the various functions of the application program are written in a conventional language and compiled using a standard compiler. A separate language

is used to describe how the components of the system should be connected together to form a multi-tasking system and to describe how the data sets can be shared. At this stage decisions on the number of processors to be used are made.

The system has been designed for use in the aerospace industry and the problem of overheads involved in context switching has been carefully considered. Individual processes are short procedures which once started are not interrupted; they are considered to be the equivalent of a single assembler instruction. The allocation of storage for data and code for processes is static.

EXERCISES

6.1 Draw up a list of functions that you would expect to find in a real-time operating system. Identify the functions which are essential for a real-time system.

6.2 Why is it advantageous to treat a computer system as a virtual machine?

6.3 Discuss the advantages and disadvantages of using
(a) fixed table
(b) linked list
methods for holding task descriptors in a multi-tasking real-time operating system.

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

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

6.6 What is meant by context switching and why is it required?

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

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

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

7

Design of Real-time Systems – General Introduction

As we said at the end of Chapter 4, there is much more to designing and implementing computer control systems than simply programming the control algorithm. In this chapter we first give an outline of a general approach to the design of computer-based systems (it actually applies to all engineering systems). We will then consider, as an example, the hot-air blower system described in Chapter 1. In designing the software structure we illustrate three approaches:

- single task;
- foreground/background; and
- multi-tasking.

We end the chapter by considering in detail some of the problems that arise when using a multi-tasking approach. We deal with both multi-tasking on a single computer and the case in which the tasks are distributed across several computers.

The objectives are:

- To show how to approach the planning and design of a computer-based system.
- To illustrate the basic approaches for the top level design of real-time software.
- To illustrate some of the problems associated with real-time, multi-tasking software.

7.1 INTRODUCTION

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

- the planning phase; and
- the development phase.

The planning phase is illustrated in Figure 7.1. It is concerned with interpreting user