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

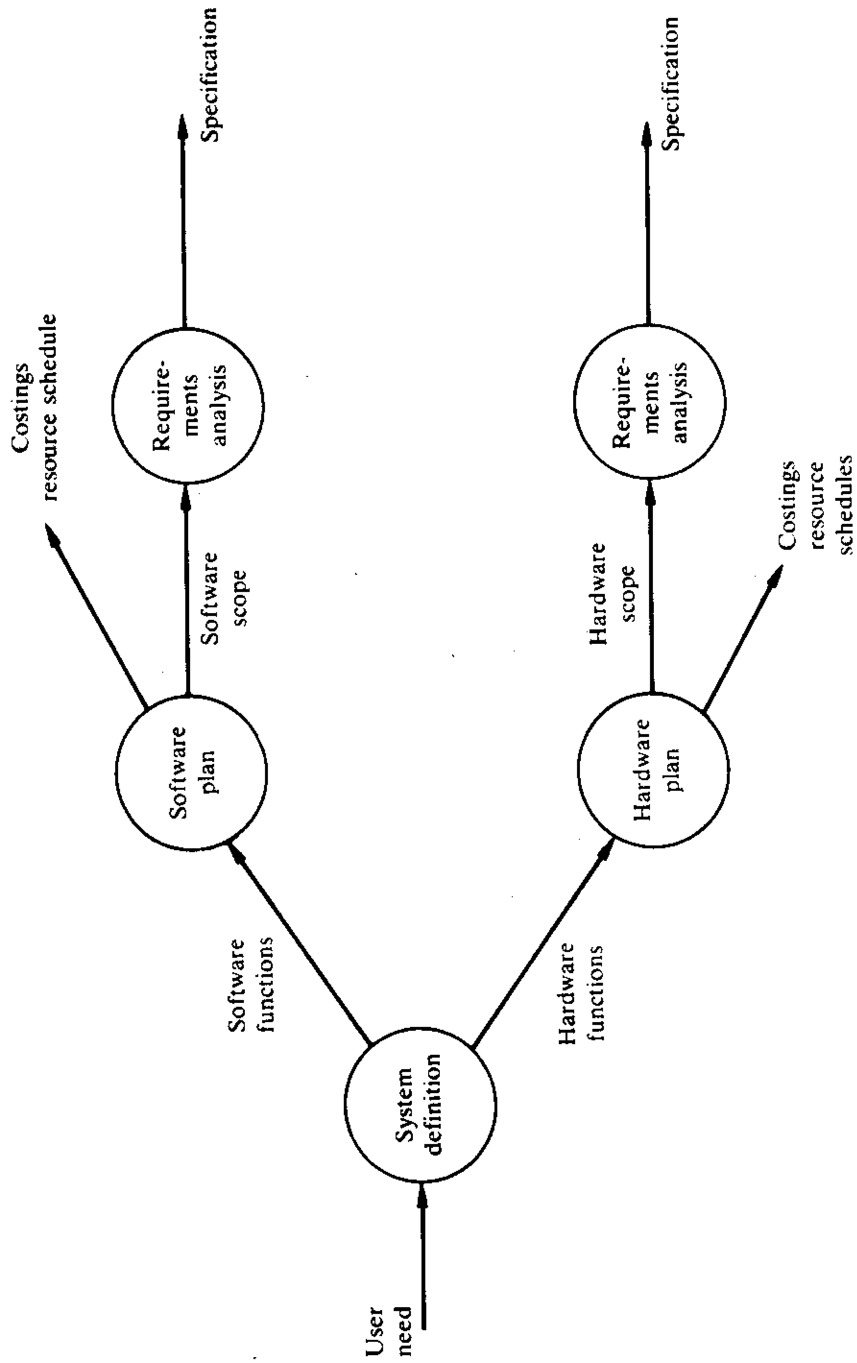

Figure 7.1 Planning phase.

requirements to produce a detailed specification of the system to be developed and an outline plan of the resources – people, time, equipment, costs – required to carry out the development. At this stage preliminary decisions regarding the division of functions between hardware and software will be made. A preliminary assessment of the type of computer structure – a single central computer, a hierarchical system, or a distributed system – will also be made. The outcome of this stage is a *specification* or *requirements* document. (The terminology used in books on software engineering can be confusing; some refer to a specification requirement document as well as to specification document and requirements document. It is clearer and simpler to consider that documents produced by the user or customer describe requirements, and documents produced by the supplier or designer give the specifications.)

It cannot be emphasised too strongly that the specification document for both the hardware and software which results from this phase must be complete, detailed and unambiguous. General experience has shown that a large proportion of *errors* which appear in the final system can be traced back to unclear, ambiguous or faulty specification documents. There is always a strong temptation to say 'It can be decided later'; deciding it later can result in the need to change parts of the system which have already been designed. Such changes are costly and frequently lead to the introduction of errors.

Some indication of the importance of this stage can be seen by examining Table 7.1 which shows the distribution of errors and cost of rectifying them (the figures are taken from DeMarco, 1978).

The stages of the development phase are shown in Figure 7.2. The aim of the preliminary design stage is to decompose the system into a set of specific sub-tasks which can be considered separately. The preliminary design stage is also referred to as the high-level design stage. The inputs to this stage are the high-level specifications; the outputs are the global data structures and the high-level software architecture. During this stage extensive liaison between the hardware and software designers is needed, particularly since, in the case of real-time systems, there will be a need to revise the decisions on the type of computer structure proposed and if, for example, a distributed system is to be used, to decide on the number of processors, communication systems (bandwidth, type), etc. The control strategy will

Table 7.1 Distribution of errors and of costs of correcting errors

Stage	*Distribution of errors %*	*Distribution of costs of rectifying errors %*
Requirements	56	82
Design	27	13
Code	7	1
Other	10	4

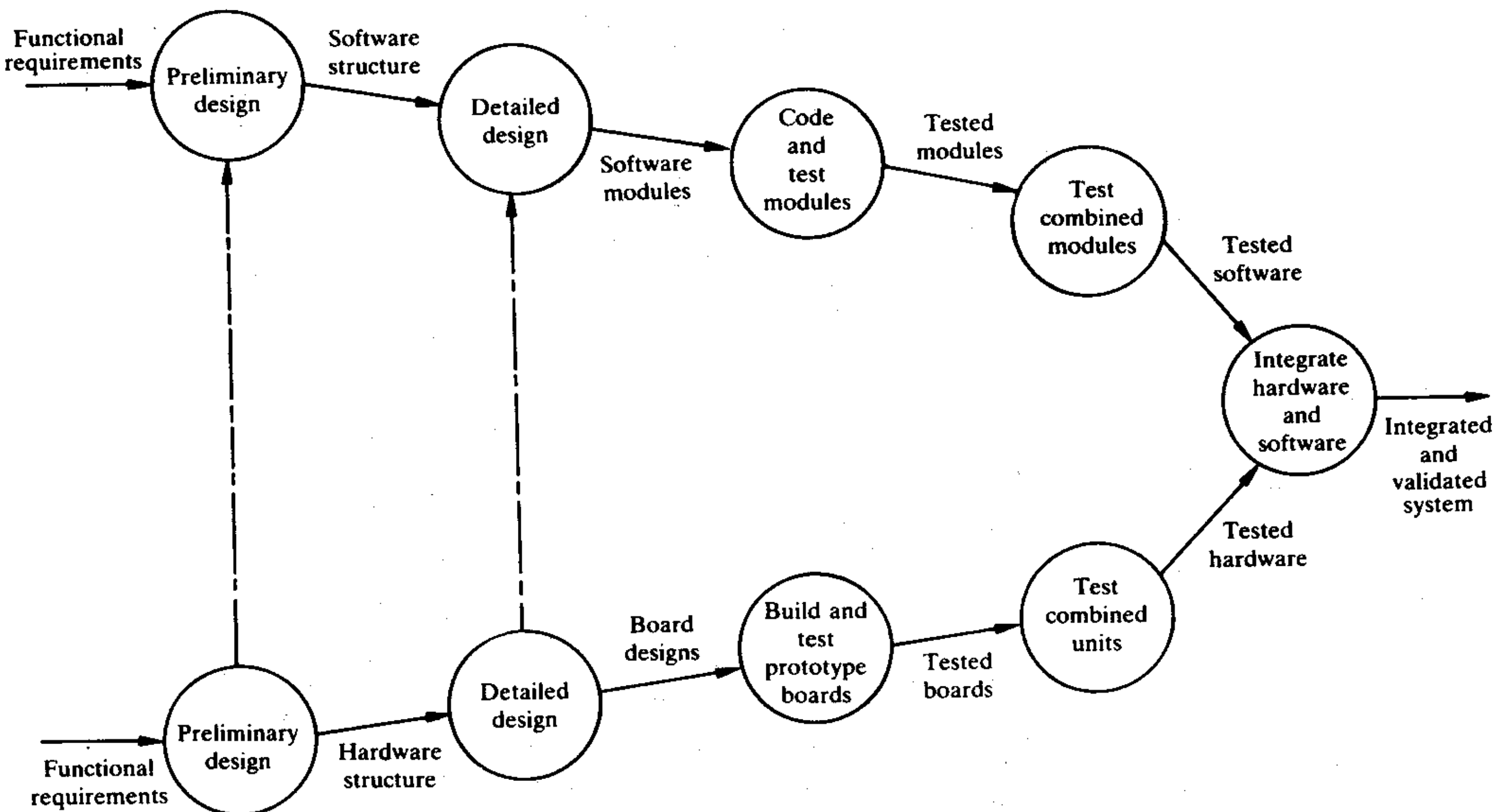

Figure 7.2 Development phase.

need reviewing including consideration of the control algorithms to be used. At the end of the preliminary design stage a review of both the hardware and software designs should be carried out.

The detailed design is usually broken down into two stages:

- decomposition into modules; and
- module internal design.

For hardware design, the first of these stages involves questions on the board structure of the system such as:

- Are separate boards going to be used for analog inputs and digital inputs or are all inputs going to be concentrated on one board?
- Can the processor and memory be located on one board?
- What type of bus structure should be used?

The second stage involves the design of the boards.

For the software engineer the first stage involves identifying activities which are related. Heuristic rules have been developed to aid the designer with decisions on division into modules (Pressman, 1992). The various heuristics are given differing emphasis in the different design methodologies. A brief description of some of the general methodologies is given below.

Functional decomposition: the top-down approach which has been advocated by Wirth and others leads to module subdivision based on a separation into functions, that is each module performs a specific function.

Information hiding: Parnas (1972a, b) has argued strongly against the functional decomposition approach and for module division based on hiding as much as possible of the information used by a module within the module.

Object oriented: this is the division of the system into entities which contain both data and the functions that operate on the data. It is a way of achieving the aims set out by Parnas in his information hiding proposals.

The main properties of objects are as follows:

- They encapsulate data and the operations that can be performed on that data.
- Operations on the object are performed by sending a message to the object requesting the operation.
- An object is an instance of a *class*.
- A class defines a common structure that describes all members of the class.
- Classes are organised hierarchically and sub-classes inherit all the features of the superordinate class.

As an example of the differences between the functional and information hiding (object-oriented) approaches consider a program which has to read a block of text from a device. The text has to be sorted into words and the words sorted into alphabetical order with duplicate words eliminated. The sorted list has to be printed. The simple *functional decomposition* approach would be to divide the system initially into three modules: input, sort and print. All the modules would need access to a shared data structure in which the text was held. Each module would thus know what form of data structure (array, linked list, record, or file) was being used to hold the data.

The Parnas approach would be to subdivide so that one module, say `StoreManager`, deals with the storage of the data. The other modules can access the data only through functions provided by the `StoreManager`. For example, it may provide two functions:

```
Put (word);
```

and

```
Get (word);
```

to enable information to be put into store and retrieved from store. The other modules need not know how the storage is organised. The advantage claimed for this approach is that design changes to one module do not affect another module.

The above approaches do not give any guidance on how to decide on boundaries between function modules or object modules. The following are commonly used heuristics for guiding module subdivision:

Coupling and cohesion: the maximising of module cohesion and the minimising of coupling between modules are the heuristics underlying data-flow design methodologies. The heuristic and methodologies have been developed by Constantine and Yourdon (1979), Myers (1978) and Stevens *et al.* (1974).

Partition to minimise interfaces: this heuristic was proposed by DeMarco (1978) and can be combined with data-flow methods. It suggests that the transformations indicated on the data-flow diagram should be grouped so as to minimise the number of interconnections between the modules.

For real-time systems additional heuristics are required, one of which is to divide modules into the following categories:

- real-time, hard constraint;
- real-time, soft constraint; and
- interactive.

The arguments given in Chapter 1 regarding the verification and validation of different types of program suggest a rule that aims to minimise the amount of software that falls into the hard constraint category since this type is the most difficult to design and test.

The major differences in software design between real-time and standard systems occur in preliminary design and decomposition into modules, and in this chapter we concentrate on these areas. Module internal design, coding, and testing are similar in both types of system.

From the description given above the whole process of specification, design and construction appears sequential; this is a simplistic view. In practice top-level design decisions frequently cannot be made until lower-level design decisions have been made. For example, we cannot decide what type of computer is required (processing power, memory requirements) until either we have coded the software or we have made detailed estimates of the amount of code and the type of computation required. A control algorithm which requires extensive arithmetic operations on real numbers results in a different computational load from one involving simple logic operations.

Modern software development methodologies address this problem and we shall examine some of them in detail in the later chapters. For the purposes of this chapter we will assume that we can proceed step-by-step through specification, preliminary design and detailed design.

7.2 SPECIFICATION DOCUMENT

To provide an example for the design procedures being described we shall consider a system comprising several of the hot-air blowers described in Chapter 1. It is assumed that the planning phase has been completed and a specification document has been prepared. A shortened version of such a document is given in Example 7.1.

EXAMPLE 7.1

Hot-Air Blower Specification

Version 3.1
Date 10 January 1992

1.0 *Introduction*
The system comprises a set of hot-air blowers arranged along a conveyor belt. Several different configurations may be used with a minimum of 6 blowers and a maximum of 12.

2.0 *Plant interface*

2.1 *Input from plant*
Outlet temperature: analog signal, range 0–10 V, corresponding to 20°C to 64°C, linear relationship.

2.2 *Output to plant*
Heater control: analog signal 0 V to −10 V, corresponding to full heat (0 V) to no heat (−10 V), linear relationship.

3.0 *Control*
A PID controller with a sampling interval of 40 ms is to be used. The sampling interval may be changed, but will not be less than 40 ms. The controller parameters are to be expressed to the user in standard analog form, that is proportional gain, integral action time and derivative action time. The set point is to be entered from the keyboard. The controller parameters are to be variable and are to be entered from the keyboard.

4.0 *Operator communication*

4.1 *Display*
The operator display is as shown below:

Set temperature	:nn.n°C	Date	:dd/mm/yyyy
Actual temperature	:nn.n°C	Time	:hh.mm
Error	:nn.n°C		
Heater output	:nn% FS	Sampling Interval	:nn ms
Controller settings			
Proportional gain	:nn.n		
Integral action	:nn.nn s		
Derivative action	:nn.nn s		

The values on the display will be updated every 5 seconds.

4.2 *Operator input*
The operator can at any time enter a new set point or new values for the control parameters. This is done by pressing the 'ESC' key. In response to 'ESC' a menu is shown on the bottom of the display screen:

1. Set temperature = nn.n	2. Proportional gain = nn.n
3. Integral action = nn.nn	4. Derivative action = nn.nn
5. Sampling interval = nn	6. Management information
7. Accept entries	
Select menu number to change	

In response to the number entered, the present value of the item selected will be deleted from the display and the cursor positioned ready for the input of a new value. The process will be repeated until item 7 – *Accept entries* – is selected at which time the bottom part of the display will be cleared and the new values shown in the top part of the display.

5.0 *Management information*
On selection of item 6 of the operator menu a management summary of the performance of the plant over the previous 24 hours will be given. The summary

provides the following information:

(a) Average error in °C in 24 hour period.
(b) Average heat demand %FS in 24 hour period.
(c) For each 15 minute period:
  (i) average demanded temperature;
  (ii) average error; and
  (iii) average heat demand.
(d) Date and time of output.

6.0 *General information*
There will be a requirement for a maximum of 12 control units. A single display and entry keyboard which can be switched between the units is adequate.

7.3 PRELIMINARY DESIGN

7.3.1 Hardware Design

There are many different possibilities for the hardware structure. Obvious arrangements are:

1. Single computer with multi-channel ADC and DAC boards.
2. Separate general purpose computers on each unit.
3. Separate computer-based microcontrollers on each unit linked to a single general purpose computer.

Each of these configurations needs to be analysed and evaluated. Some points to consider are:

Option 1: given that the specification calls for the system to be able to run with a sample interval for the control loop of 40 ms, can this be met with 12 units sharing a single processor?

Option 2: is putting a processor that includes a display and keyboard on each unit an expensive solution? Will communication between processors be required? (Almost certainly the answer to this is yes; operators and managers will not want to have to use separate displays and keyboards.)

Option 3: what sort of communication linkage should be used? A shared high-speed bus? A local-area network? Where should the microcontrollers be located? At each blower unit or together in a central location?

Each option needs careful analysis and evaluation in terms of cost and performance. The analysis must include consideration of development costs, performance operating and maintenance costs. It should also include consideration of reliability and safety.

To provide a basis for consideration of the widest range of approaches to software design we will assume that option 1 above is chosen.

7.3.2 Software Design

Examining the specification shows that the software has to perform several different functions:

- DDC for temperature control;
- operator display;
- operator input;
- provision of management information;
- system start-up and shut-down; and
- clock/calendar function.

The various functions and type of time constraint are shown in Figure 7.3. The control module has a hard constraint in that it must run every 40 ms. In practice this constraint may be relaxed a little to, say, 40 ms ± 1 ms with an average value over 1 minute of, say, 40 ms ± 0.5 ms. In general the sampling time can be specified

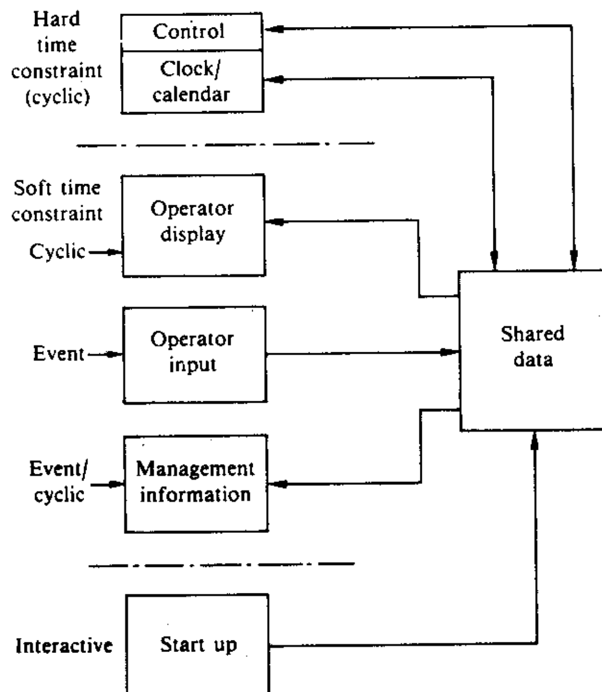

Figure 7.3 Basic software modules.

as $T_s \pm e_s$ with an average value, over time T, of $T_s \pm e_a$. The requirement may also be relaxed to allow, for example, one sample in 100 to be missed. These constraints will form part of the test specification.

The clock/calendar module must run every 20 ms in order not to miss a clock pulse. This constraint can be changed into a soft constraint if some additional hardware is provided in the form of a counter which can be read and reset by the clock/calendar module. The constraint could now be, say, an average response time of 1 second with a maximum interval between reading the counter of 5 seconds. (For these values what size of counter would be required?)

The operator display, as specified, has a hard constraint in that an update interval of 5 seconds is given. Common sense suggests that this is unnecessary and an average time of 5 seconds should be adequate; however, a maximum time would also have to be specified, say 10 seconds.

Similarly soft constraints are adequate for operator input and for the management information logs. These would have to be decided upon and agreed with the customer. They should form part of the specification in the requirements document. The start-up module does not have to operate in real time and hence can be considered as a standard interactive module.

There are obviously several different activities which can be divided into sub-problems. The sub-problems will have to share a certain amount of information and how this is done and how the next stages of the design proceed will depend upon the general approach to the implementation. There are three possibilities:

- single program;
- foreground/background system; and
- multi-tasking.

Each of these approaches is discussed in the following sections.

7.4 SINGLE-PROGRAM APPROACH

Using the standard programming approach the modules shown in Figure 7.3 are treated as procedures or subroutines of a single main program. The flowchart of such a program is illustrated in Figure 7.4. This structure is easy to program; however, it imposes the most severe of the time constraints – the requirement that the clock/calendar module must run every 20 ms – on all of the modules. For the system to work the clock/calendar module and any one of the other modules must complete their operations within 20 ms. If t_1, t_2, t_3, t_4 and t_5 are the *maximum* computation times for the module's clock/calendar, control, operator display, operator input and management output respectively, then a requirement for the system to work can be expressed as

$$t_1 + \max(t_2, t_3, t_4, t_5) < 20 \text{ ms}$$

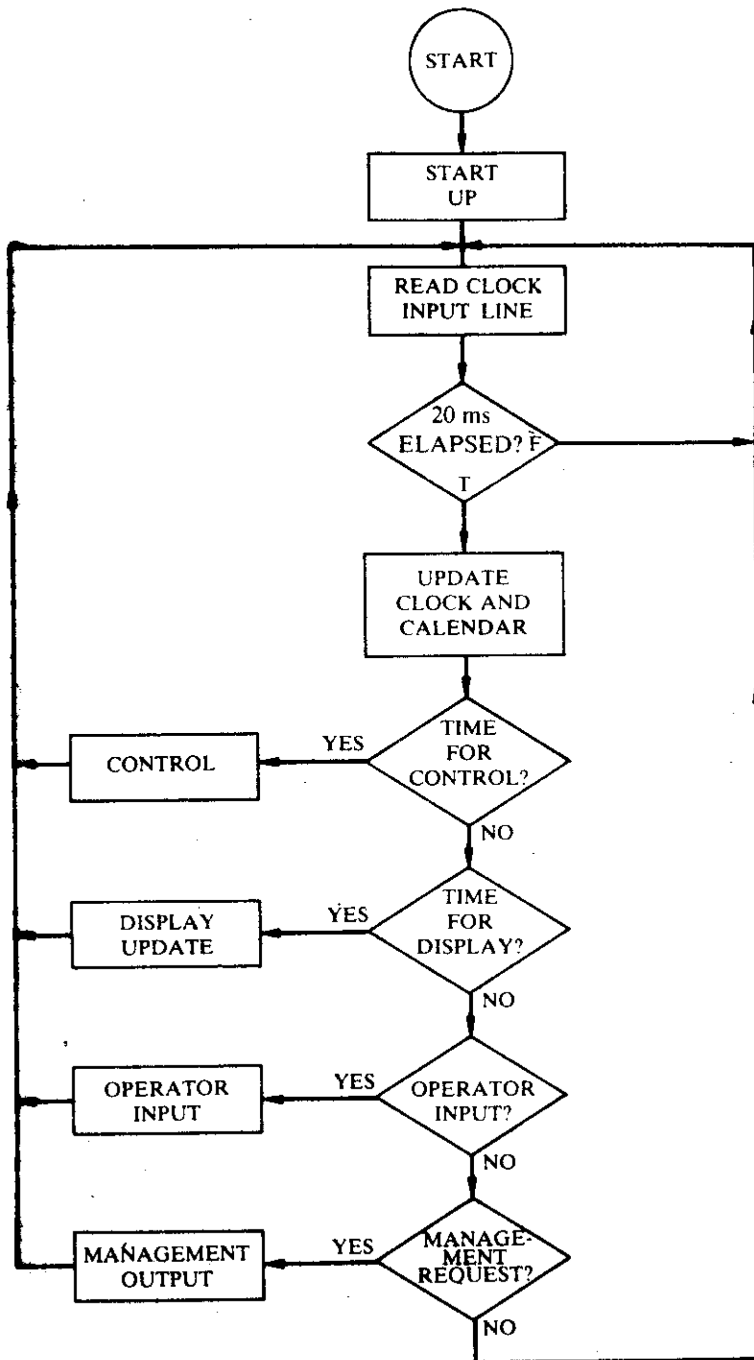

Figure 7.4 Single-program approach.

(Note: (a) The *control* module provides the control computations for each of the 12 units; (b) the values of t_1, t_2, t_3, t_4, and t_5 must include the time taken to carry out the tests required and t_1 must also include the time taken to read the clock input line.)

The single-program approach can be used for simple, small systems and it leads to a clear and easily understandable design, with a minimum of both hardware and software. Such systems are usually easy to test. As the size of the problem increases, there is a tendency at the detail design stage to split modules not because they are functionally different but simply to enable them to complete within the required time interval. In the above example the management output requirement makes it unsuitable for the single-program approach; if that requirement is removed the approach could be used. It may, however, require the division of the display update module into three modules: display date and time; display process values; and display controller parameters.

7.5 FOREGROUND/BACKGROUND SYSTEM

There are obvious advantages – less module interaction, less tight time constraints – if the modules with hard time constraints can be separated from, and handled independently of, the modules with soft time constraints or no time constraints. The modules with hard time constraints are run in the so-called 'foreground', and the modules with soft constraints (or no constraints) are run in the 'background'. The foreground modules, or 'tasks' as they are usually termed, have a higher priority than the background tasks and a foreground task must be able to interrupt a background task.

The partitioning into foreground and background usually requires the support of a real-time operating system, for example the Digital Equipment Corporation's RT/11 system. It is possible, however, to adapt many standard operating systems, for example MS-DOS, to give simple foreground/background operation if the hardware supports interrupts. The foreground task is written as an interrupt routine and the background task as a standard program.

If you use a PC you are in practice using a foreground/background system. The application program that you are using (a word processor, a spreadsheet, graphics package or some program which you have written yourself in a high-level language) is, if we use the terminology given above, running in the background. In the foreground are several interrupt-driven routines – the clock, the keyboard input, the disk controller – and possibly some memory-resident programs which you have installed – a disk cacheing program or an extended memory manager. The terminology foreground and background can be confusing; literature concerned with non-real-time software uses foreground to refer to the application software and background to refer to interrupt routines that are hidden from the user.

Using the foreground/background approach the structure shown in Figure 7.4

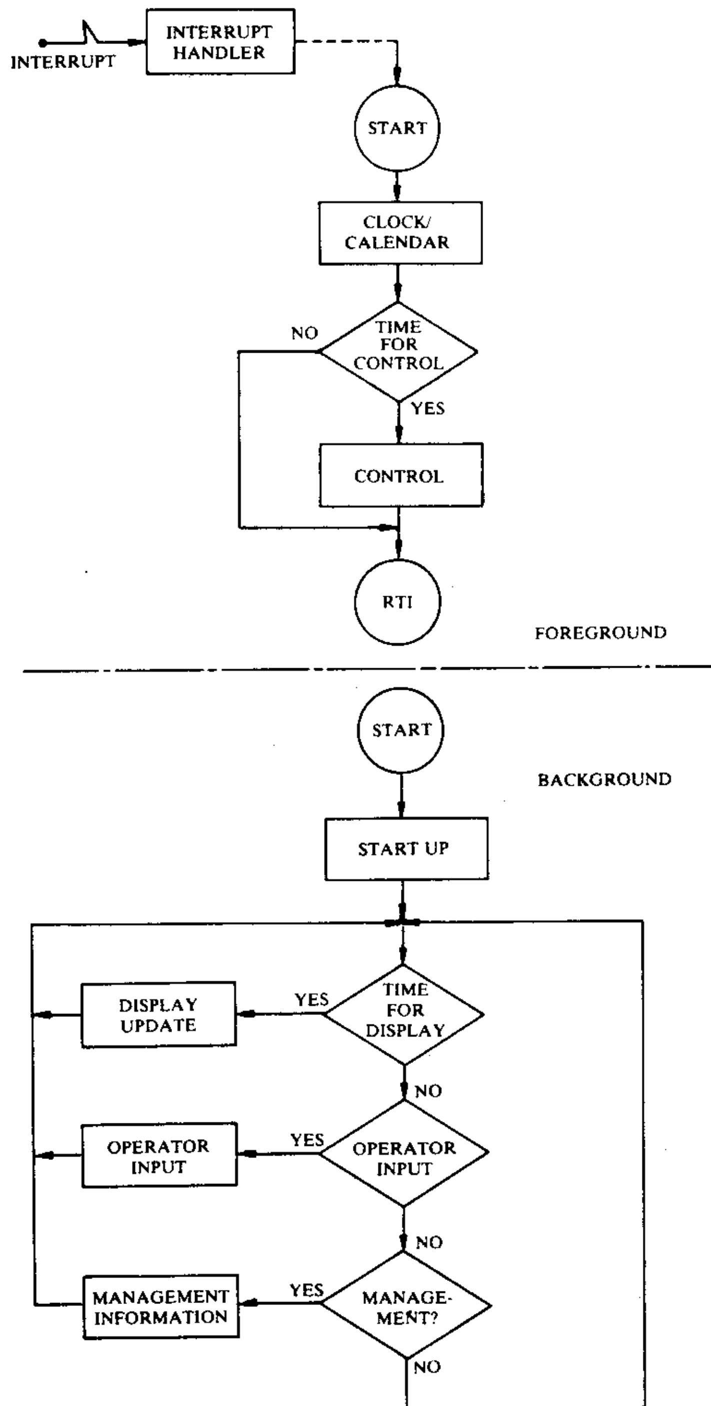

Figure 7.5 Foreground/background approach.

can be modified to that shown in Figure 7.5. There is now a very clear separation between the two parts of the system. A requirement for the foreground part to work is that

$$t_1 + t_2 < 20 \text{ ms}$$

where t_1 = maximum execution time for clock/calendar module and t_2 = maximum execution time for the control module. A requirement for the background part to work is that:

1. max (t_3, t_4, t_5) < 10 s;
2. display module runs on average every 5 s; and
3. operator input responds in < 10 s.

Although the time constraints have been relaxed the measurements to be made in order to check the performance are more complicated than in the single-program case and hence the evaluation of the performance of the system has been made more difficult.

EXAMPLE 7.2

Foreground/Background System Using Modula-2

Using the facilities provided by SYSTEM a simple foreground/background structure can easily be created to handle real-time control applications.

```
MODULE Main;
FROM SYSTEM IMPORT ADR, SIZE, WORD, PROCESS, NEWPROCESS,
TRANSFER, IOTRANSFER;
VAR
   main, operator, control : PROCESS;

PROCEDURE Control;
BEGIN
   LOOP
      IOTRANSFER(control, operator, clockVector);
      ...
   (* control actions go here
      routine should keep track of time as well *)
      ...
   END;
END Control;
```

```
PROCEDURE Display;
***...
(* insert the display update code here *)
***...
END Display;
PROCEDURE Keyboard;
***...
(* insert keyboard code here *)
***..
END Keyboard;

PROCEDURE Operator;
BEGIN
   LOOP
      IF time = displayTime THEN Display;
      Keyboard;
   END (* LOOP *);
END Operator;

BEGIN
   NEWPROCESS (Control, ADR(controlWksp),
   SIZE(controlWksp),
            control);
   NEWPROCESS (Operator, ADR(operatorWksp),
   SIZE(operatorWksp),
            operator);
   TRANSFER (main, control);
END Main.
```

Note that because we have used the low-level facilities of the language directly and simply none of the problems of data sharing and mutual exclusion discussed in the previous sections have been solved. All the variables required by the controller are assumed to be stored in common storage and hence are accessible at any time to either the operator task or the control task. Also in the above example the control task, which is entered on an interrupt, can return only to a specific named task and hence there can be only one background task. However, as the next section shows the low-level facilities provided can be used to create a much more powerful set of real-time multi-tasking support routines.

Although the foreground/background approach separates the *control* structure of the foreground and background modules, the modules are still linked through the data structure as is shown in Figure 7.6. The linkage occurs because they share data variables; for example, in the hot-air blower system, the control task, the display task and the operator input task all require access to the controller parameters. In the single program (sometimes called single tasking) there was no difficulty in controlling access to the shared variable since only one module (task) was active at

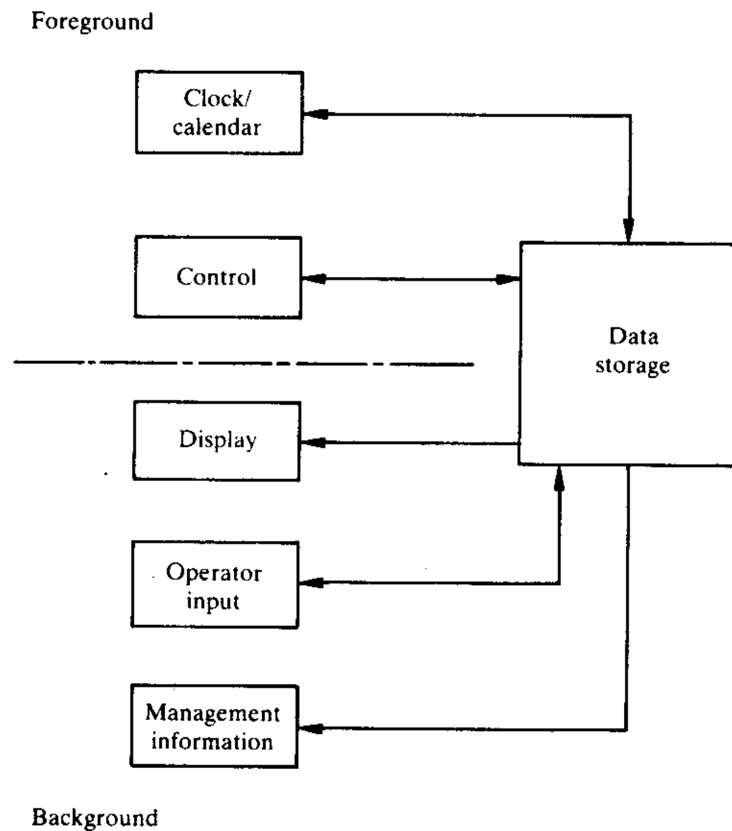

Figure 7.6 Software modules for foreground/background system showing data storage.

any one time, whereas in the foreground/background system tasks may operate in parallel, that is one foreground module and one background module may be active at the same time. (Note: active does not mean 'running' since if one CPU is being used only one task can be using it at any instant; however, both the foreground and background tasks may have the potential to run.)

In this particular example the variables can be shared between the control, display and operator input modules without any difficulty since only one module writes to any given variable. The operator input module writes the controller parameters and set point variables, the clock/calendar module writes to the date and time variables and the control module writes to the plant data variables (error and output temperature). However, the input from the operator must be buffered and only transferred to the shared storage when it has been verified. Example 7.3 shows a method of doing this.

EXAMPLE 7.3

Buffering of Parameter Input Data

```
MODULE HotAirBlower;
VAR
    p1, p2, p3 : REAL; (* Controller parameters declared as
global variables *)
PROCEDURE GetParameters (VAR x, y, z : REAL);
BEGIN
    ...
    (* get new parameters from terminal and store in x, y, z *)
    ...
END GetParameters;

PROCEDURE OperatorInput;
VAR
    x, y, z : REAL;
BEGIN
    GetParameters (x, y, z);
    (* insert code to verify here *)
    p1: = x; (* transfer parameters to global variables *)
    p2 : = y;
    p3 : = z;
END OperatorInput;
BEGIN
    (* main program *)
END HotAirBlower.
```

To understand the reasons for buffering, let us consider what would happen if, when a new value was entered, it was stored directly in the shared data areas. Suppose the controller was operating with $p_1 = 10$, $p_2 = 5$ and $p_3 = 6$ and it was decided that the new values of the control parameters should be $p_1 = 20$, $p_2 = 3$ and $p_3 = 0.5$. As soon as the new value of p_1 is entered the controller begins to operate with $p_1 = 20$, $p_2 = 5$, $p_3 = 6$, that is neither the old nor the new values. This may not matter if the operator enters the values quickly. But what happens if, after entering p_1, the telephone rings or the operator is interrupted in some other way and consequently forgets to complete the entry? The plant could be left running with a completely incorrect (and possibly unstable) controller.

The method used in Example 7.3 is not strictly correct and safe since an interrupt could occur between transferring x to p_1 and y to p_2, in which case an incorrect controller would be used. For a simple feedback controller this would have little effect since it would be corrected on the next sample. It may be more serious if the change were to a sequence of operations. The potential for serious and

possibly dangerous consequences is not great in small, simple systems (a good reason for keeping systems small and simple whenever possible); it is much greater in large systems.

The transfer of data between the foreground and background tasks, that is the statements

```
p1:=x;
p2:=y;
p3:=z;
```

form what is known as a *critical section* of the program and should be an indivisible action. The simple way of ensuring this is to inhibit all interrupts during the transfer:

```
InhibitInterrupts;
p1:=x;
p2:=y;
p3:=z;
EnableInterrupts;
```

However, it is undesirable for several separate modules each to have access to the basic hardware of the machine and each to be able to change the status of the interrupts. From experience we know that modules concerned with the details of the computer hardware are difficult to design, code and test, and have a higher error rate than the average module. It is good practice to limit the number of such modules. Ideally transfers should take place at a time suitable for the controller module, which implies that the operator module and the controller module should be synchronised or should rendezvous.

7.6 MULTI-TASKING APPROACH

The design and programming of large real-time systems is eased if the foreground/background partitioning can be extended into multiple partitions to allow the concept of many active tasks. At the preliminary design stage each activity is considered to be a separate task. (Computer scientists use the word *process* rather than task but this usage has not been adopted because of the possible confusion which could arise between internal computer processes and the external processes on the plant.) The implications of this approach are that each task may be carried out in parallel and there is no assumption made at the preliminary design stage as to how many processors will be used in the system.

The implementation of a multi-tasking system requires the ability to:

- create separate tasks;
- schedule running of the tasks, usually on a priority basis;
- share data between tasks;

- synchronise tasks with each other and with external events;
- prevent tasks corrupting each other; and
- control the starting and stopping of tasks.

The facilities to perform the above actions are typically provided by a real-time operating system (RTOS) or a combination of RTOS and a real-time programming language. We dealt with these in detail in the previous two chapters.

We now examine some examples to illustrate the problems that arise with multi-tasking and why real-time systems require special language and operating system facilities. For simplicity we will assume that we are using only one CPU and that the use of this CPU is time shared between the tasks. We also assume that a number of so-called primitive instructions exist. These are instructions which are part of a programming language or the operating system and their implementation and correctness is guaranteed by the system. All that is of concern to the user is that an accurate description of the syntax and semantics is made available. In practice, with some understanding of the computer system, it should not be difficult to implement the primitive instructions. Underlying the implementation of primitive instructions will be an eventual reliance on the system hardware. For example, in a common memory system some form of arbiter will exist to provide for mutual exclusion in accessing an individual memory location.

7.7 MUTUAL EXCLUSION

EXAMPLE 7.4

Mutual Exclusion

Consider the transfer of information from an input task to a control task as shown in Figure 7.7. The input task gets the values for the proportional gain, the integral action time and the derivative action time. From these it computes the controller parameters `KP`, `KI` and `KD` and these are transferred to the `CONTROL` task. A simple method is to hold the parameter values in an area of memory which has been declared as being `COMMON` and hence is accessible to both tasks. Unless the input task is given exclusive rights to this `COMMON` data area while it writes the parameter values there is a danger that the control task will read one new value, say `KP`, and two old values, `KD` and `KI`. Giving exclusive rights to the input task is not a satisfactory solution in this case as will be seen later.

As another example of the need for mutual exclusion consider the problem in Example 7.5.

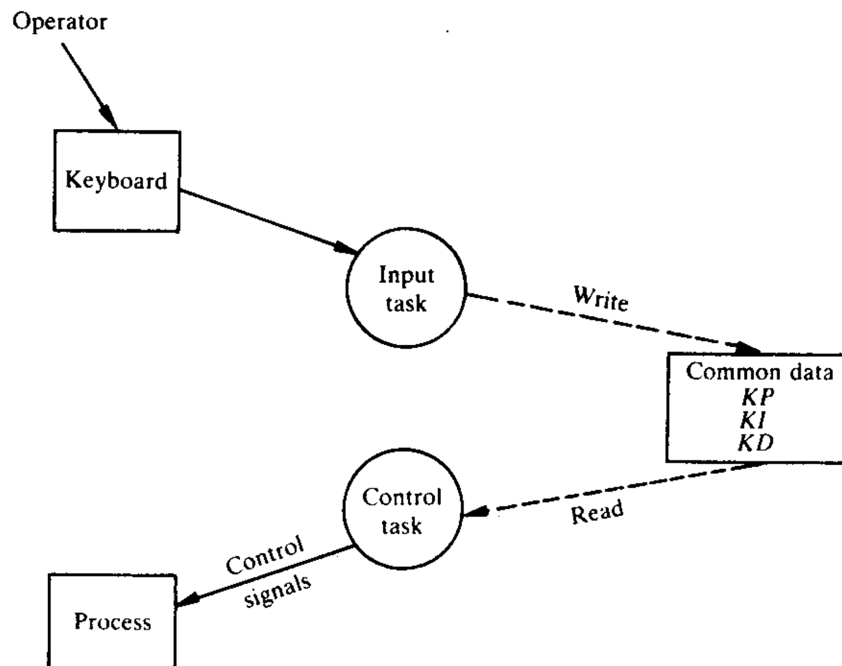

Figure 7.7 Data sharing using common memory.

EXAMPLE 7.5

As part of the maintenance procedures a record is kept of accesses to a particular device (after a specified number of accesses some preventative maintenance has to be carried out). The system designer arranges that each task in the system which uses the device will increment a common variable, `deviceUse`, by using the code

```
deviceUse := deviceUse+1
```

The hardware will resolve the problem of simultaneous access to the memory location in which `deviceUse` is stored, but this is not sufficient to guarantee the correct functioning of the counter.

Consider the following scenario. The compiler generates machine code in the following form:

```
load deviceUse
add 1
store deviceUse
```

Suppose the value of `deviceUse` is 38 and task `A` executes the load `deviceUse`; then the tasks reschedule and task `B` executes load `deviceUse`: both tasks now have in their own environments a register containing the current value of

deviceUse, that is 38. Task B now executes the add 1 instruction and the store deviceUse instruction giving a value of 39 in deviceUse. Control is now returned to task A which executes the add 1 and the store deviceUse instructions which again give a value of 39. The final value of deviceUse is thus 39 even though it started as 38 and has been incremented twice.

In abstract terms, as we saw in the previous chapter, mutual exclusion can be expressed in the form

remainder 1
pre-protocol
critical section
post-protocol
remainder 2

where *remainder 1* and *remainder 2* represent sequential code that does not require access to a particular resource or to a common area of memory. The critical section is the part of the code which must be protected from interference from another task. The protocols called before and after the critical sections are code 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 short such 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.

7.7.1 Condition Flags

A simple method of indicating if a resource is being used or not is to have associated with that resource a flag variable which can be set to TRUE or FALSE (or to 0 or 1, or SET or RESET). A task wishing to access the resource has to test the flag before using the resource. If the flag is FALSE (0 or RESET) then the resource is available and the task sets the flag TRUE (1 or SET) and uses the resource. The procedure is illustrated in Example 7.6.

EXAMPLE 7.6

Mutual Exclusion Using a Condition Flag

```
MODULE MutualExclusion1;
(* Mutual exclusion problem Condition Flag solution 1*)
VAR
   deviceInUse: BOOLEAN;
```

```
PROCEDURE Task; (* task assumed to be running in
parallel with other tasks *)
BEGIN
   (*      remainder1  *)
   WHILE deviceInUse DO
      (* test and wait until available *)
   END (* while *)
   deviceInUse:= TRUE;  (*claim resource*)
   (*......
   use the resource - critical section
   ......*)
   deviceInUse := FALSE;
   (*      remainder2 *)
END Task;
   (* main program *)
END MutualExclusion1.
```

In this solution there are two problems:

1. The `WHILE` statement forms a *busy wait* operation which relies on a pre-emptive interrupt to escape from the loop. If the task which has already claimed the resource cannot interrupt the *busy wait* then the task will continue to use the CPU and will exclude all other tasks.
2. The testing and setting of the flag are separate operations and hence the task could be suspended and replaced by another task between checking the flag and setting the resource unavailable. A consequence could be that, as is shown in Figure 7.8, two tasks could both claim the same resource.

The two tasks A and B shown in Figure 7.8 both share a printer. It is assumed that the flag variable `printerInUse`, which is set to 1 when the printer is in use and to 0 when it is available, controls access to the printer. Task A checks the `printerInUse` flag and finds that the printer is available, but before it can execute the next instruction, which would be to set the `printerInUse` flag to 1 and hence claim the printer, the dispatcher forces a task status change and task B runs. Task B also wishes to use the printer and checks the flag: it finds that the printer is available, sets the `printerInUse` flag to 1 and begins to use the printer. At some time later it requires some other resource and the dispatcher suspends it and makes task A the active task. Task A now claims the printer and begins to use it. Thus both tasks think that they have the exclusive use of the printer, whereas they are both using it and the output from the two tasks will be mixed up. After some time task A is again suspended and task B continues; it now finishes with the printer and releases it by setting `printerInUse` to 0, making the printer available to any other task even though task A still thinks that it has exclusive use of the printer. At task change 4, task A again uses the printer and eventually releases it although it has in fact already been released by task B.

For a condition flag to work securely it is therefore vital that the operations of test condition/set condition are indivisible. If a primitive instruction at the machine

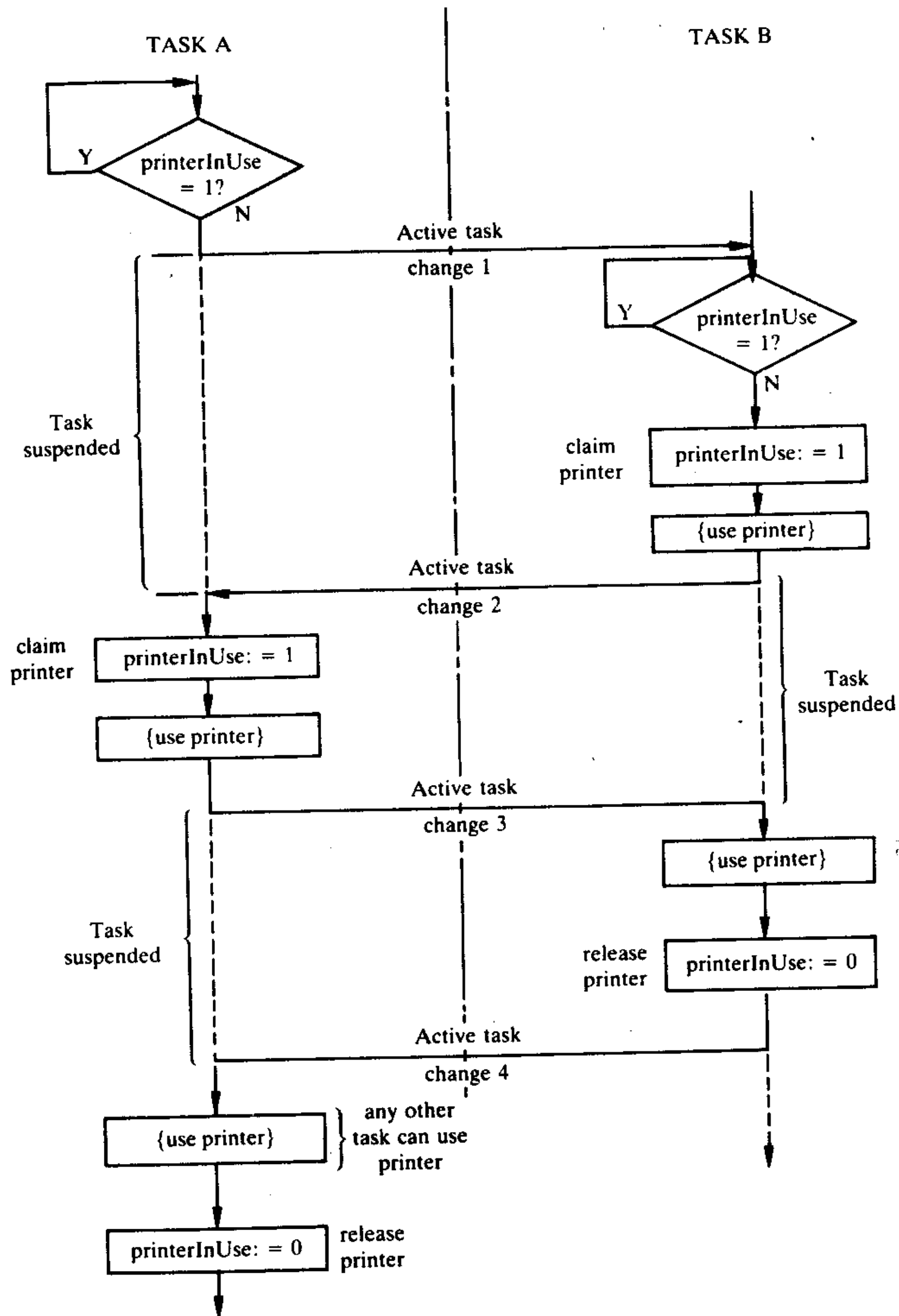

Figure 7.8 Attempt at mutual exclusion using condition flags.

code level which provides a combined test and set operation does not exist then the test/set operation must be made indivisible by the use of the enable/disable interrupt instructions.

EXAMPLE 7.7

Mutual Exclusion Condition Flags – Solution 2

```
(* Mutual exclusion problem Condition flag solution 2*)
VAR
   deviceInUse:BOOLEAN;
   deviceClaimed: BOOLEAN;
PROCEDURE Task;
BEGIN
   (* remainder1 *)
   REPEAT
      DisableInterrupts; (* procedure *)
      IF deviceInUse THEN
         deviceClaimed:= FALSE
      ELSE
         deviceInUse:= TRUE;
         deviceClaimed:= TRUE
      END (* IF *)
      EnableInterrupts;
   UNTIL deviceClaimed;
   (*......
   use resource (*critical section*)
   .....*)
   DisableInterrupts;
   deviceInUse:=FALSE;
   deviceClaimed:= FALSE;
   Enable interrupts;
   (* remainder2 *)
END Task;
```

Solution 2 is an improvement in that it will prevent two tasks gaining access to the same resource. There is still the problem of being in an endless loop waiting for the resource to become available and thus relying on some form of pre-emption to allow other tasks to run. If this approach is used in practice it would be sensible to incorporate a request for a short delay between each testing of the condition flag. The insertion of a call delay statement between lines 13 and 14, that is

```
enable interrupts;
delay(delayTime);
UNTIL deviceClaimed;
```

would be appropriate.

Because errors in the interrupt enable/disable status are potentially dangerous (a failure to enable interrupts at the end of a critical section will cause the whole system to fail), manipulation of the interrupt status flag should be restricted as much as possible and preferably should not occur in application level programs; therefore the solution in Example 7.7 is not recommended.

7.7.2 Semaphores

It is possible to devise a safe and reliable flag-based mutual exclusion system. One such system is the *turn flag* technique in which the flag, instead of showing if the resource is free or in use, indicates which task can *next* use the resource. The problem with this technique is that the tasks must run in strict sequence. The most reliable solution is to use Dekker's algorithm but the method becomes unwieldy as the number of tasks increases (see Cooling (1991, pp. 306–9) for a discussion of the methods). The most commonly used approach is the use of some form of semaphore (Example 7.8).

EXAMPLE 7.8

Use of Semaphores to Solve Transfer of Controller Parameters Problem

```
MODULE Controller;
(*Mutual exclusion-transfer of controller parameters
solution 1*)
TYPE
   AParameterRecord = RECORD
      kp  : REAL;
      kd  : REAL;
      ki  : REAL
      END;
VAR
   mutex : SEMAPHORE;
   controlParameters : AParameterRecord;
   inputBlock : AParameterRecord;

TASK DataTransfer;
(* transfers input data to the controller *)

BEGIN
   Secure (mutex);
   controlParameters := inputBlock;
   Release (mutex);
END DataTransfer;
```

```
TASK Control;

BEGIN
    Secure (mutex);
    DoControl (*actual routines to perform control would be
    placed here*)
    Release (mutex);
END Control;

BEGIN (*main body of program *)
    Initialise (mutex, 1);
    StartTask (DataTransfer,5);
    StartTask (Control,1);
(*
DataTransfer is allocated priority level 5 which is a lower
priority than Control at priority level 1
*)
END Main.
```

This is an acceptable solution for co-operating tasks in a non-real-time environment, but for real-time work there are several problems. The first is in `TASK DataTransfer`. The use of a semaphore does not prevent the task being suspended and another task being run during the critical section: it prevents any other task accessing the `ControlParameters` record (providing that the task checks the semaphore `mutex` before proceeding). If, for example, it is time to run the `Control` task, which has a higher priority than the `DataTransfer`, the `Control` task will check the `mutex` semaphore and will then be suspended awaiting completion of the data transfer by the `DataTransfer` task. The consequences of the delay could be unpredictable: if the `DataTransfer` task is the only other task waiting to run, or has the highest priority of any of the waiting tasks, then the solution could be acceptable in that the transfer will be completed and the `control` task will run immediately the operation `Release (mutex)` is performed. However, if we introduce a third task, `DisplayUpdate`, with a priority level of, say, 3 then the sequence of events could be as illustrated in Figure 7.9.

In this figure we assume that the `DataTransfer` task has just secured the semaphore `mutex`, at which time the clock interrupt forces a rescheduling of the tasks and the `Control` task runs. The `Control` task will be suspended when it attempts to execute `Secure(mutex)` and there will again be a rescheduling of the tasks. Assume now that the `DisplayUpdate` task is ready to run; because it is of higher priority than `DataTransfer`, it will be run. The task `DataTransfer` cannot run until `DisplayUpdate` suspends or finishes running, and the `Control` task cannot run until `DisplayUpdate` has run and released `mutex`. The consequence of this delay may be that the `Control` task is not run within the specified sampling period.

At first sight it would seem that the use of a semaphore in this manner for mutual exclusion is not appropriate for real-time control system applications.

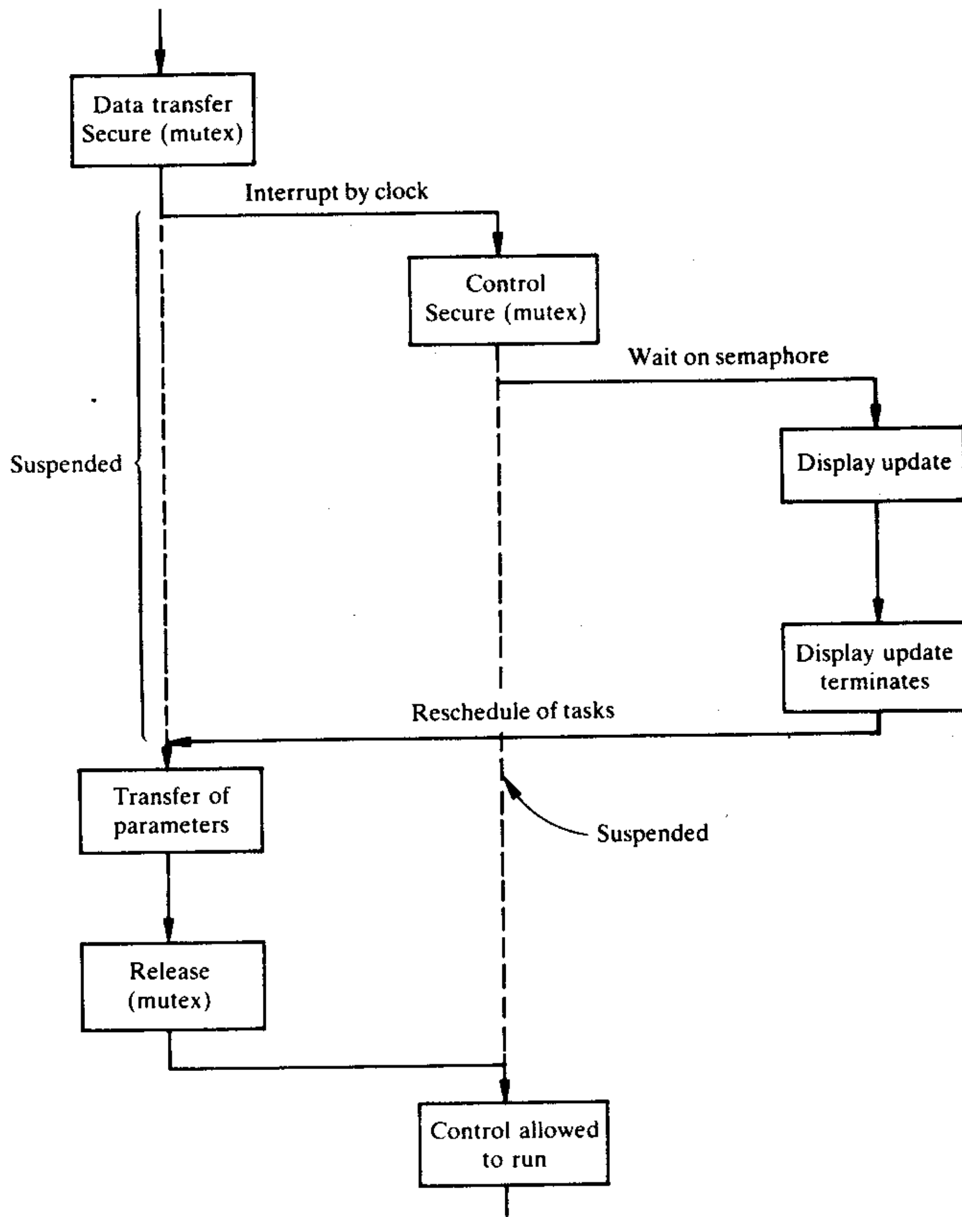

Figure 7.9 Transfer of controller parameters – use of semaphore.

However, you should be able to see that by changing the priorities of the task an adequate solution can be obtained. Try running through the above scenario if the task priorities are

```
DataTransfer = 1;
Control = 2;
DisplayUpdate =5.
```

Although the allocation of a high priority to the `DataTransfer` task will provide a solution it is not a safe solution for general use. Reliance on the priorities of tasks is unsafe as once the number of tasks becomes larger than five to ten it becomes very difficult to construct the scenarios to prove that the system will work correctly.

A semaphore-based solution can be used if there is a means of escape from the commitment to wait for a resource to become free. A suitable mechanism is the time out which forces a return from the task suspension if the resource does not become free within a specified period of time.

An alternative solution is to exploit the fact that the system we are designing is using feedback control and for a feedback control algorithm it is usually preferable to continue using the old values of the parameters for a short time rather than delaying the calculation of the next value of the manipulated variable. A possible way of doing this is shown in Example 7.9 below. Double buffering is provided by holding a local copy of `controlParameters` in the `Control` task. We ensure that the `Control` task does not wait if the reference copy of `controlParameters` held by `DataTransfer` is in use but continues using its local copy by giving it control of the timing of the transfer. A flag variable `messagePresent` is used to indicate that new values of the parameters are available.

EXAMPLE 7.9

Solution of Controller Parameters Problem Using Double Buffering

```
MODULE Controller;
(*
Transfer of controller parameter using double
buffering.
Delay(delayTime)is assumed provided by another
module and enables a task to suspend itself for a
given period of time. The variables kp, kd, ki, and
r are assumed to be obtained from the operator by
another task and held in some data area - a pool -
which is accessible to DataTransfer.
*)
```

```
TYPE
   AParameterRecord = RECORD
      kp : REAL;
      kd : REAL;
      ki : REAL
      r : REAL (* set point *)
      END;

VAR
   messagePresent : BOOLEAN;
   controlMessage : AParameterRecord;
   inputBlock : AParameterRecord;

TASK DataTransfer;

CONST
   delayTime = 20;
BEGIN
   (* wait if previous message has not been taken *)
   WHILE messagePresent DO
      Delay(delayTime)
   END (* while *);
   controlMessage.kp := kp;
   controlMessage.ki := ki;
   controlMessage.r := r;
   messagePresent := TRUE;
END DataTransfer;

PROCEDURE Control;

VAR
   controlParameter : AParameterRecord;
BEGIN
   DoControl; (*actual control statements would go
here *)
   IF messagePresent THEN
      controlParameters := controlMessage;
      messagePresent := FALSE
   END (* if *);
END Control;
BEGIN (*main body*)
   messagePresent := FALSE;
   StartTask(DataTransfer,5);
   StartTask(DisplayUpdate,2);
   StartTask(Control,1);
END Main.
```

The flag variable `messagePresent` is used to signal that a data transfer should

take place. The timing of the transfer is left to the task `Control` and hence there should be no danger of the `Control` task being interrupted while it is transferring the data since it is running at the highest priority in the system. The penalties involved in using this solution are (a) that the task `Control` has to execute an additional instruction

```
IF messagePresent THEN
```

every time it runs (and because tasks such as `Control` are typically the most frequently run tasks in the system it is usually desirable to keep their execution time as short as possible); and (b) the use of a small amount of extra memory.

In this particular example – passing of the controller parameters – you may be wondering why we are not just updating `kp`, `kd`, `ki` and `r` without any mutual exclusion provision since the probability of the task `Control` interrupting the transfer is low. If we assume that the transfer of the parameter takes 20 microseconds, that `Control` runs every 100 ms with an execution time of 1 ms and if a transfer of parameters takes place during every run of `Control` then assuming that they are not synchronised in any way the chance of interference is approximately 1 in 5000. Given that the controller parameters are not going to be changed at this rate the chance of task `Control` interrupting a transfer is very low and since for this type of feedback control the disturbance to the controller from using a wrong set of parameters for one sample is likely to be small we can conclude that mutual exclusion for the examples shown above is unnecessary.

However, even in this simple system we must ensure that the operator does not directly change the values of the variables `kp`, `ki`, `kd` and `r` used by the `Control` task. The reason for this restriction should be obvious. The variables form a related data set, that is the individual values are not independent (see Chapter 4 if you want to know why this is so), and so we must protect against the possibility that an inconsistent set is being used. For example, consider the scenario where the operator enters `kp` and `ki` and is then interrupted by a colleague or by a telephone call; it could be several minutes before the entry of the new values is completed. We must therefore always ensure that there is buffering between the operator input and the transfer of input data to the set of variables available to the `Control` task and for security the variables used by the `Control` task should be hidden from the input task. Example 7.10 shows in outline one method of doing this by using the features of Modula-2.

EXAMPLE 7.10

Buffering of Operator Input

```
MODULE Operator;
IMPORT FROM Controller
   PutData (* Procedure *)
   ...
```

```
TASK Operator
(* get, assemble and check parameters *)
   PutData(controlParameters)
   ...
END (* TASK Operator *)
BEGIN
   ...
END Operator.

DEFINITION MODULE Controller;
EXPORT
TYPE
   AParameterRecord = RECORD
      kp : REAL;
      kd : REAL;
      ki : REAL
      r : REAL (* set point *)
   END (* RECORD*);

PROCEDURE PutData(VAR ControlParameters:
AParameterRecord);
END Controller.
```

A final word of warning: real-time control systems can rapidly become complex and it then becomes difficult to work out all possible what-if scenarios. For example, assume that the designer/programmer of the `Controller` module decided not to use any form of mutual exclusion on the parameter transfer on the grounds that the chance of an interruption to the transfer was very small, but the system also includes several high-priority alarm condition tasks. There is now the possibility that the alarm occurs at the very instant when a partial transfer of values has occurred. The control loop continues running but completion of the transfer is held up because the alarm tasks take up all the available processor time. We are thus attempting to correct some problem on the plant and may well be making the problem worse because our control loop is running with an inconsistent set of controller parameters. A remote possibility perhaps, but if we are to produce safe systems we must expect that such remote possibilities will occur and plan for them. In general it is safest to assume that if something can possibly occur it will.

7.7.3 Notes on Using Semaphores

A further word of caution: in using the semaphore construct for mutual exclusion it is perhaps natural to assume that suspended tasks gain access to the resource in the order in which they performed the `secure(s)` operation, that is the task which has been waiting longest is served first. However, the order in which waiting tasks are selected when a resource becomes available is a matter for the designer of

the operating system. Possible schemes are:

First in, first out: the task that has been waiting longest is chosen.
Priority order: the highest-priority waiting task is chosen.
Non-deterministic: any waiting task may be chosen arbitrarily.

The software designer must know what selection mechanism is being used. This is an example of just one of the many practical difficulties of separating out design from implementation details. Ideally the designer should be able to choose which mechanism she/he wants to use but often the choice will be restricted because of a decision to use a particular operating system or family of operating systems.

The semaphore provides an elegant mechanism for mutual exclusion but it is a low-level primitive and like the simple use of enable and disable interrupts it is error prone. One omitted or one misplaced semaphore instruction will cause the whole mutual exclusion protection to fail and the collapse of the whole system. Semaphores are historically important but for real-time systems a safer approach, for example the use of monitors as discussed below, is required.

7.8 MONITORS

The basic idea of a monitor was explained in Chapter 6. In Example 7.11 the implementation of a monitor in Modula-2 to protect access to a buffer area is shown. Monitors themselves do not provide a mechanism for synchronising tasks and hence for this purpose the monitor construct has to be supplemented by allowing, for example, signals to be used within it.

EXAMPLE 7.11

Monitor Using Signals

```
MODULE Buffer[monitorPriority];
(*solution to producer-consumer problem, also
implementation of a simple CHANNEL *)
   FROM Signals IMPORT
      Signal, InitSignal, AwaitSignal, SendSignal;
   EXPORT
      Put,Get;
   CONST
      nMax = 32;
   VAR
      nFree, nTaken: [0..nMax];
      in, out: [1..nMax];
      b: ARRAY [1..nMax] OF INTEGER;
      notFull, notEmpty: Signal;
```

```
    PROCEDURE Put(i: INTEGER);
    BEGIN
      IF nFree = 0 THEN
      AwaitSignal(nonFull)
      (* another task can call Put during the wait - it will
      also find nFree=0 and will wait *)
      ELSE
        DEC(nFree)
      END;
      b[in] := i;
      in := in MOD nMax + 1;
      IF Awaited(nonEmpty) THEN
        SendSignal(nonEmpty)
        (* a higher priority task waiting for this signal
        will run now and may lead to another call of Put *)
      ELSE
        INC(nTaken)
      END
    END Put;

  PROCEDURE Get(VAR i: INTEGER);
  BEGIN
    IF nTaken = 0 THEN
      AwaitSignal(nonEmpty)
      (* another task can call Get during the wait - it will
      also find nTaken = 0 *)
    ELSE
      DEC(nTaken)
    END;
    i := b[out];
    out := out MOD nMax + 1;
    IF Awaited(nonFull) THEN
      SendSignal(nonFull);
      (* a higher priority task waiting for this signal will
      run now and may lead to another call of Get *)
    ELSE
      INC(nFree)
    END
  END Get;
  BEGIN
    nFree := nMax; nTaken := 0;
    in := 1; out := 1;
    InitSignal(nonFull); InitSignal(nonEmpty);
  END Buffer;
```

To prevent deadlock when using signals within a monitor for task synchronisation, a task that gains access to a monitor procedure but then executes a `wait(signal)` operation must be suspended and placed outside the monitor. This procedure is

necessary to allow another task to enter. Referring to the producer–consumer problem above, suppose the producer task enters the monitor with a call to `Put` but is forced to wait because the buffer is full; then the buffer can become non-full only if another task, the consumer, is able to enter the monitor and remove an item from the buffer using the `Get` procedure. The consumer will then issue a `send(signal)` to awaken the producer task and unless the `send(signal)` operation is the last executable statement in the `Get` procedure then two tasks could be active within the monitor thus breaching the mutual exclusivity rule. Hence the use of signals within a monitor requires the rule that the `Send` operation must be the last executable statement of a monitor procedure.

The standard monitor construction outlined above, like the semaphore, does not reflect the priority of the task trying to use a resource; the first task to gain entry can lock out other tasks until it completes. Hence a lower-priority task could hold up a higher-priority task in the manner described in Example 7.8. The lack of priority causes difficulties for real-time systems. Traditional operating systems built as monolithic monitors avoided the problem by ensuring that once an operating system call was made (in other words, when a monitor function was invoked) then the call would be completed without interruption from other tasks. The monitor function is treated as a critical section. This does not mean that the whole operation requested was necessarily completed without interruption. For example, a request for access to a printer for output would be accepted and the request queued; once this had been done another task could enter the monitor to request output and either be queued, or receive information from the monitor as to the status of the resource. The return of information is particularly important as it allows the application program to make a decision as to whether to wait for the resource or take some other action.

Preventing lower-priority tasks locking out higher-priority tasks through the monitor access mechanism can be tackled in a number of ways. One solution adopted in some implementations of Modula-2 is to run a monitor with all interrupts locked out; hence a monitor function once invoked runs to completion. In many applications, however, this is too restrictive and some implementations allow the programmer to set a priority level on a monitor such that all lower-priority tasks are locked out – note that this is an *interrupt* priority level, not a task priority.

The monitor has proved to be a popular idea and in practice it provides a good solution to many of the problems of concurrent programming. The benefits and popularity of the monitor constructs stem from its modularity which means that it can be built and tested separately from other parts of the system, in particular from the tasks which will use it. Once a fully tested monitor is introduced into the system the integrity of the data or resource which it protects is guaranteed and a fault in a task using the monitor cannot corrupt the monitor or the resource which it protects. Although it does rely on the use of signals for intertask synchronisation it does have the benefit that the signal operations are hidden within the monitor.

The monitor is an ideal vehicle for creating abstract mechanisms and thus fits in well with the idea of top-down design. However, the nested monitor call problem – calling procedures in one monitor from within another monitor – can lead to deadlock. Providing that nested monitor calls are prohibited the use of the monitor concept provides a satisfactory solution to many of the problems for a single-processor machine or for a multi-processor machine with shared memory. It can also be used on distributed systems.

The monitor's usefulness in some real-time applications is restricted because a task leaving a monitor can only signal and awaken one other task – to do otherwise would breach the requirement that only one task be active within a monitor. This means that a single controlling synchroniser task, for example a clock level scheduler, cannot be built as a monitor. The problem can be avoided by allowing signals to be used outside a monitor but then all the problems associated with signals and semaphores re-emerge.

7.9 RENDEZVOUS

The rendezvous, developed by Hoare (1978) and Brinch Hansen (1973), provides an alternative to the use of monitors and signals to ensure mutual exclusion and synchronisation in intertask communication. In the rendezvous the actions of synchronisation and data transmission are seen as inseparable activities. The fundamental idea is that if two tasks *A* and *B* wish to exchange data, for example if *A* wishes to transmit data to *B*, then *A* must issue a transmit request and *B* a receive request. If task *A* issues the transmit request before *B* has issued the receive, then *A* must wait until *B* issues its request and vice versa. When both tasks have synchronised the data is transferred and the tasks can then proceed independently.

A problem with the original formulation is that both tasks must name each other and hence general library tasks cannot be created. The solution adopted in the language Ada is to use an asymmetric rendezvous in which only one task, known as the caller, names the other task, known as the server. In the descriptions that follow it is assumed that the language which supports the rendezvous concept does so by means of a construct of the form

```
ACCEPT name(parameter list)
    statements
END
```

The statements within the `ACCEPT...END` are assumed to be a critical section and are executed in a mutually exclusive manner. They would normally be executed by the server task. The `ACCEPT` statement represents an entry point and the calling task specifies the name of the entry point when it wishes to synchronise with the server task.

EXAMPLE 7.12

Simple Rendezvous

```
TASK A;
    VAR x:ADataItem;
    BEGIN
        ...
        B.Transfer(x);
        ...
    END;

TASK B;
    VAR y:ADataItem;
    BEGIN
        ...
        ACCEPT Transfer(IN item:ADataItem);
        y:=item;
    END;
END;
```

`TASK A` wishes to pass information held in variable `x` to a variable `y` in `TASK B`. The actual data transfer takes place using the normal parameter passing mechanisms – the actual parameters supplied in the call (in this case the variable `x`) are bound to the formal parameters of the `ACCEPT` statement (in this case item). The synchronisation of the two tasks is obtained by the requirement that the entry procedure call – `B.Transfer(x)` – cannot be completed until the corresponding `ACCEPT` statement – `ACCEPT Transfer` – is executed and conversely the execution of the `ACCEPT` statement cannot be completed until the entry call is executed. The actual transfer is completed within the body of the `ACCEPT` statement; in this case the data supplied by the entry call is transferred to a variable which is local to `TASK B`.

Note that in the `ACCEPT` statement the direction of the transfer is specified, in this case `IN`. Variables can be declared as being for input (`IN`) or output (`OUT`) or as bidirectional (`IN OUT`).

When using the rendezvous the two tasks have to synchronise in order to transfer information. The task which is producing the information cannot leave it in a buffer and continue but must wait for the consumer task to arrive before the transfer can take place. The position is equivalent to that which the motorist would face if there were no filling stations but only roving petrol tankers. The motorist and the petrol tanker driver would have to arrange for a rendezvous; when both arrived at the designated place the motorist would fill up with petrol from the tanker, and then both would continue on their respective ways. The requirement of the rendezvous that tasks synchronise in order to exchange data is too severe a constraint for many applications.

One solution to strict synchronisation is to introduce a buffer task between the two tasks which wish to exchange data. A simple buffer requires that the tasks have to call the buffer task in strict rotation. Continuing the filling station analogy this is the equivalent of demanding that the tanker and motorist alternate visits to the same filling station – clearly impractical as the tanker will deliver a much larger quantity of fuel than a single motorist will receive. The problem can be solved by introducing indeterminacy into the task by means of a SELECT statement (Example 7.13).

EXAMPLE 7.13

Illustrating Use of a SELECT Statement

```
TASK AFillingStation;
   VAR y:AFuel;
   BEGIN
      DO
         SELECT
            ACCEPT deliver(IN fuel:AFuel);
            y:=fuel;
            END;
      OR
         ACCEPT receive(OUT fuel:AFuel);
         fuel:=y;
         END;
         END SELECT;
      END DO;
   END;
```

The key to the operation of the above task is the action of the SELECT statement: each time the SELECT statement is executed there are four possible states in which the entry points to the task can be:

1. a call to deliver is pending;
2. a call to receive is pending;
3. calls to deliver and to receive are pending; and
4. no calls are pending.

For cases 1 and 2 then the appropriate ACCEPT statement is immediately executed. In case 3 one of the ACCEPT statements is selected at random and executed. In case 4 the task is suspended until a call is made to either of the ACCEPT statements at which time the task is resumed and the appropriate statement is executed. The SELECT statement ensures that only one ACCEPT statement will be executed at any one time but the order is not predetermined and there can be successive calls to the same ACCEPT.

As was seen in Example 7.8, involving the transfer of control parameters from an input task to the control task itself, there is a need to be able to test if another task is waiting, or if another task has left data to be collected in order to avoid committing a high-priority repetitive task to wait for an event. There is also frequently the requirement in real-time systems to have some form of time out such that a task only commits itself to wait for a predetermined length of time. Two extensions to the rendezvous primitive provide facilities to support these actions.

The time-out facility is provided in a simple and natural way by extending the SELECT statement to allow a delay option in the possible choices within the SELECT construct. This is illustrated in Example 7.14 for the control parameter problem. It is assumed that when an input task has gathered the new parameters it makes a call to a Put entry point in the control task. The control task includes the following code.

EXAMPLE 7.14

Use of Time Out

```
TASK Control;
    ...
BEGIN
    ...
    (*control action*)
    ...
(*start of section to check if update of parameters is
required*)
    SELECT
        ACCEPT Put(IN parameters:AControlParRecord);
            kp := parameter.kp;
            kd := parameter.kd;
            ki := parameter.ki;
        END
    OR
        DELAY 1 (*delay in milliseconds*)
            ...
        END
    END
END.
```

In the above code fragment if the control task reaches the SELECT statement when a call to the entry point Put is pending, then the ACCEPT part of the SELECT statement is executed and the parameter values are transferred to the control task. However, if no call is pending then the DELAY part of the SELECT statement is executed. The action of the delay is to cause the control task to wait for the length of time specified in the delay; during this period of suspension any call to the ACCEPT statement will be recognised and the ACCEPT statement executed. If no

calls are received, then at the end of the delay period the statements following the DELAY statement are executed.

An alternative to the delay part within a SELECT statement is an else part (this can be thought of as a delay 0). The above problem could be coded using the ELSE statement as in Example 7.15.

EXAMPLE 7.15

Use of ELSE with SELECT

```
TASK Control;
   ...
BEGIN
   ....
   (*control action*)
   ...
   (*start of section to check if update of parameters is
   required*)
   SELECT
      ACCEPT Put(IN parameters:AControlParRecord);
         kp := parameter.kp;
         kd := parameter.kd;
         ki := parameter.ki;
      END
   OR
      ELSE
         ...
      END
   END
END.
```

In this example, if there is no call pending for the ACCEPT statement when the SELECT statement is reached, then the ELSE part of the SELECT statement is executed immediately. The use of the SELECT...OR...ELSE construct is the most appropriate for the control parameters problem.

The DELAY statement is useful in many applications; for example, on detection of an alarm condition the operator may be alerted and expected to acknowledge the alarm and take appropriate action within a predetermined time. If the operator does not respond, then the computer system has to take further action, possibly by sounding an audible alarm or by beginning to close down the plant. The SELECT...DELAY construct provides a natural and simple way of expressing

the requirement:

```
SELECT
    ACCEPT OperatorAcknowledge;
OR
    DELAY 30 (*delay 30 seconds*)
    ...
    AlternativeAction;
    ...
END
```

Another example of the use of the DELAY statement is to provide time out in communications with peripherals or other computers.

The rendezvous concept provides the most flexible and easily understood mechanism for handling multi-tasking problems and in the SELECT mechanism provides facilities which none of the other concepts have. It has been implemented as part of the Ada language.

7.10 SUMMARY

In this chapter we have dealt informally with the basic approaches to the design of real-time systems. We have emphasised the division of the system into subsystems – modules – and briefly considered the heuristics commonly used to guide this process. An important aspect of subdivision is that the modules should be used to hide information.

There are three models on which the implementation of real-time software can be based. These are:

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

For small, simple systems the first two models should be used. They result in a simple implementation that can be easily understood and tested. However, only the single-task model, without interrupts, can be formally proved correct: once interrupts are permitted the system immediately becomes non-deterministic and its correctness cannot be formally proved.

As systems become larger and more complex they can be most easily implemented if a multi-tasking model is adopted. Multi-tasking introduces problems of mutual exclusion, intertask communication and intertask synchronisation. These problems are now well understood. Modern real-time languages and operating systems provide primitive instructions and various mechanisms that support multi-tasking. Some of the standard problems and their solution were described.

Detailed knowledge of the application and judicious use of the application

characteristics can simplify some multi-tasking problems as we illustrated when considering the transfer of the controller parameters. Simplifications of this sort are part of the art of engineering; however, they must be used with care and must be *documented* – in particular the conditions for which the simplification is valid must be clearly stated.

EXERCISES

7.1 The standard input routines in languages such as FORTRAN, Pascal and BASIC cannot be used within a timed loop to obtain information from the keyboard. This is also true of Modula-2. Why can't we use the standard Modula-2 routines?

7.2 A plant operating in a remote location is controlled by an embedded computer control system. The plant operates in two modes referred to as Amode and Bmode. The control algorithm for Amode is of the form

$$m(n) = Ae(n) + Be(n-1) + Ce(n-2) + Dm(n-1) + Em(n-2)$$

and for Bmode

$$m(n) = K_1e(n) + K_2e(n-1) + K_3e(n-2)$$

where $e(n) = R - c(n)$
R = set point
$c(n)$ = the measured output of the plant at interval n.

The change-over from Amode to Bmode is to be made when $c(n) > ChangeA$ for five successive readings. The change-over from Bmode to Amode is to be made when $c(n) < ChangeB$ for five successive readings. The parameters *ChangeA* and *ChangeB* and the set point R can all be changed from a central station.

A change to the value of R requires a change in the values of *ChangeA* and *ChangeB*. The controller parameters A, B, C, D, E and K_1, K_2 and K_3 also need changing. They must be changed as the set $\{A, B, C, D, E, K_1, K_2, K_3\}$ and not as individual elements. The data transmission link to the remote station has a slow transmission speed and is subject to frequent bursts of interference. You can assume that the data transmission system support software contains error checking software and organises retransmission of erroneous data.

Discuss the problems of designing the software for the embedded computer system and discuss possible ways of dealing with the slow and unreliable data transmission system.

7.3 What is the principal difference between a pool and a channel? Explain why you would use (a) a pool and (b) a channel.

8

Real-time System Development Methodologies – 1

This chapter begins with an overview of the general approach now being adopted in the specification, design and construction of complex real-time systems, followed by a brief description of some of the standard methodologies. The Yourdon methodologies are then described in detail. The aims of the chapter are to:

- Show how specification, design and implementation can be considered as a process of *modelling*.
- Describe the major methodologies.
- Provide a more detailed understanding of one methodology, the Yourdon methodology.

8.1 INTRODUCTION

The production of robust, reliable software of high quality for real-time computer control applications is a difficult task which requires the application of engineering methods. During the last ten years increasing emphasis has been placed on formalising the specification, design and construction of such software, and several methodologies are now extant. The major ones are shown in Table 8.1. All of the methodologies address the problem in three distinct phases. The production of a *logical* or *abstract* model – the process of *specification*; the development of an *implementation* model for a *virtual machine* from the logical model – the process of *design*; and the construction of software for the virtual machine together with the implementation of the virtual machine on a physical system – the process of *implementation*. These phases, although differently named, correspond to the phases of development generally recognised in software engineering texts. Their relationship to each other is shown in Figure 8.1.

Abstract model: the equivalent of a requirements specification, it is the result of the requirements capture and analysis phase.

Implementation model: this is the equivalent of the system design; it is the product of the design stages – architectural design and the detail design.