5

Languages for Real-time Applications

Languages are an important implementation tool for all systems that include embedded computers. To understand fully methods for designing software for such systems one needs to have a sound understanding of the range of implementation languages available and of the facilities which they offer. The range of languages with features for real-time use continues to grow, as do the range and type of features offered. In this chapter we concentrate on the fundamental requirements of a good language for real-time applications and will illustrate these with examples drawn largely from Modula-2 and Ada. It is not the purpose of the chapter to compare languages or to offer a complete discussion of all the features of Modula-2 and Ada; such material can be found in the books listed in the Bibliography.

Choosing a language for writing software for embedded real-time systems is an important and serious matter. The choice has implications for the safety, reliability and costs of the final system. The major purpose of this chapter is to provide an understanding of some of the features of programming languages that can influence the choice. This chapter needs to be read in conjunction with the next chapter on Operating systems since some important real-time features – access to the underlying hardware, support for concurrency, intertask communication – are determined by the interface between the language and the operating system.

The aims and objectives of this chapter are to:

- List, explain and prioritise the major requirements for a real-time language.
- Describe the features that assist in the construction of safe, reliable software.
- List and explain techniques used to support the division of code into modules.
- List and explain the major facilities required to support concurrency.
- Assess a language for its suitability for a particular type of application.

5.1 INTRODUCTION

Producing safe real-time software places heavy demands on programming languages. Real-time software must be reliable: the failure of a real-time system can

be expensive both in terms of lost production, or in some cases, in the loss of human life (for example, through the failure of an aircraft control system). Real-time systems are frequently large and complex, factors which make development and maintenance costly. Such systems have to respond to external events with a guaranteed response time; they also involve a wide range of interface devices, including non-standard devices. In many applications efficiency in the use of the computer hardware is vital in order to obtain the necessary speed of operation.

Early real-time systems were necessarily programmed using assembly level languages, largely because of the need for efficient use of the CPU, and access interface devices and support interrupts. Assembly coding is still widely used for small systems with very high computing speed requirements, or for small systems which will be used in large numbers. In the latter case the high cost of development is offset by the reduction in unit cost through having a small, efficient, program. Dissatisfaction with assemblers (and with high-level languages such as FORTRAN which began to be used as it was recognised that for many applications the advantages of high-level languages outweighed their disadvantages) led to the development of new languages for programming embedded computers. These languages included CORAL 66, RTL/2 and more recently C. The limitation of all of them is that they are designed essentially for producing sequential programs and hence rely on operating system support for concurrency.

The features that a programmer demands of a real-time language subsume those demanded of a general purpose language and so many of the features described below are also present (or desirable) in languages which do not support real-time operations. Barnes (1976) and Young (1982) divided the requirements that a user looked for in a programming language into six general areas. These are listed below in order of importance for real-time applications:

- Security.
- Readability.
- Flexibility.
- Simplicity.
- Portability.
- Efficiency.

In the following sections we will examine how the basic features of languages meet the requirements of the user as given above. The basic language features examined are:

- Variables and constants: declarations, initialisation.
- Data types – including structured types and pointers.
- Control structures and program layout and syntax.
- Scope and visibility rules.
- Modularity and compilation methods.
- Exception handling.

A language for real-time use must support the construction of programs that exhibit

concurrency and this requires support for:

- Construction of modules (software components).
- Creation and management of tasks.
- Handling of interrupts and devices.
- Intertask communication.
- Mutual exclusion.
- Exception handling.

Modern real-time languages differ in how they provide the above facilities. There are two basic approaches: one is to provide a minimum set of language mechanisms with the ability to extend the set; the other is to provide a more extensive set of mechanisms from which the programmer can choose. The first approach gives a simple clean language but at the expense of standardisation as each user creates additional facilities (Modula-2 is an example of this approach), the second approach leads to a more complex, but standardised, language. Ada, CONIC and CUTLASS are examples of this latter approach.

5.1.1 Security

Security of a language is measured in terms of how effective the compiler and the run-time support system is in detecting programming errors automatically. Obviously there are some errors which cannot be detected by the compiler regardless of any features provided by the language: for example, errors in the logical design of the program. The chance of such errors occurring is reduced if the language encourages the programmer to write clear, well-structured, code.

Language features that assist in the detection of errors by the compiler include:

- good modularity support;
- enforced declaration of variables;
- good range of data types, including sub-range types;
- typing of variables; and
- unambiguous syntax.

It is not possible to test software exhaustively and yet a fundamental requirement of real-time systems is that they operate reliably. The intrinsic security of a language is therefore of major importance for the production of reliable programs. In real-time system development the compilation is often performed on a different computer than the one used in the actual system, whereas run-time testing has to be done on the actual hardware and, in the later stages, on the hardware connected to plant. Run-time testing is therefore expensive and can interfere with the hardware development program.

Economically it is important to detect errors at the compilation stage rather than at run-time since the earlier the error is detected the less it costs to correct it. Also checks done at compilation time have no run-time overheads. This is important

as a program will be run many more times than it is compiled. Reliance on run-time checking frequently requires additional code to be inserted in the program (normally done by the compiler) and this leads to an increase in program size and a decrease in execution speed. In general strong typing gives good security at compilation time.

5.1.2 Readability

Readability is a measure of the ease with which the operation of a program can be understood without resort to supplementary documentation such as flowcharts or natural language descriptions. The emphasis is on ease of reading because a particular segment of code will be written only once but will be read many times. The benefits of good readability are:

- Reduction in documentation costs: the code itself provides the bulk of the documentation. This is particularly valuable in projects with a long life expectancy in which inevitably there will be a series of modifications. Obtaining up-to-date documentation and keeping documentation up to date can be very difficult and costly.
- Easy error detection: clear readable code makes errors, for example logical errors, easier to detect and hence increases reliability.
- Easy maintenance: it is frequently the case that when modifications to a program are required the person responsible for making the modifications was not involved in the original design – changes can only be made quickly and safely if the operation of the program is clear.

Factors which affect readability are manifold and to some extent readability depends on personal preference. The co-operation of the programmer is also required: it is possible to write unreadable programs in any language. The readability of a program is improved by the adoption of a clear layout which emphasises the structure, and by the careful choice of variable names. The syntax of the language and the layout of the code statements are the two most important factors that affect the readability.

The major disadvantage of using languages that support good readability and of writing readable code is that the source code is longer and it takes longer to write; for all except short-lived software this is a small price to pay for the added security and maintainability of the software.

5.1.3 Flexibility

A language must provide all the features necessary for the expression of all the operations required by the application without requiring the use of complicated constructions and tricks, or resort to assembly level code inserts. The flexibility of a language is a measure of this facility. It is particularly important in real-time

systems since frequently non-standard I/O devices will have to be controlled. The achievement of high flexibility can conflict with achieving high security. The compromise that is reached in modern languages is to provide high flexibility and, through the *module* or *package* concept, a means by which the low-level (that is, insecure) operations can be hidden in a limited number of self-contained sections of the program.

5.1.4 Simplicity

In language design, as in other areas of design, the simple is to be preferred to the complex. Simplicity contributes to security. It reduces the cost of training, it reduces the probability of programming errors arising from misinterpretation of the language features, it reduces compiler size and it leads to more efficient object code.

Associated with simplicity is consistency: a good language should not impose arbitrary restrictions (or relaxations) on the use of any feature of the language.

5.1.5 Portability

Portability, while desirable as a means of speeding up development, reducing costs and increasing security, is difficult to achieve in practice. Surface portability has improved with the standardisation agreements on many languages. It is often possible to transfer a program in source code form from one computer to another and find that it will compile and run on the computer to which it has been transferred. There are, however, still problems when the wordlengths of the two machines differ and there may also be problems with the precision with which numbers are represented even on computers with the same wordlength.

Portability is more difficult for real-time systems as they often make use of specific features of the computer hardware and the operating system. A practical solution is to accept that a real-time system will not be directly portable, and to restrict the areas of non-portability to specific modules by restricting the use of low-level features to a restricted range of modules. Portability can be further enhanced by writing the application software to run on a virtual machine, rather than for a specific operating system. A change of computer and operating system then requires the provision of new support software to create a virtual machine on the new system.

5.1.6 Efficiency

In real-time systems, which must provide a guaranteed performance and meet specific time constraints, efficiency is obviously important. In the early computer control systems great emphasis was placed on the efficiency of the coding – both in terms of the size of the object code and in the speed of operation – as computers

were both expensive and, by today's standards, very slow. As a consequence programming was carried out using assembly languages and frequently 'tricks' were used to keep the code small and fast. The requirement for generating efficient object code was carried over into the designs of the early real-time languages and in these languages the emphasis was on efficiency rather than security and readability.

The falling costs of hardware and the increase in the computational speed of computers have changed the emphasis. Also in a large number of real-time applications the concept of an efficient language has changed to include considerations of the security and the costs of writing and maintaining the program; speed and compactness of the object code have become, for the majority of applications, of secondary importance. There are, however, still application areas where compactness and speed do matter: in the consumer market where production runs may be 100 000 per year the ability to use a slower, cheaper CPU or to keep down the amount of memory used can make a significant difference to the viability of the product. Other areas in which speed matters are in the control of electromechanical systems, aircraft controls and in the general area of signal processing, for example speech recognition. The efficiency of a language depends much more on the compiler and the run-time support than on the actual language design.

5.2 SYNTAX LAYOUT AND READABILITY

The language syntax and its layout rules have a major impact on the readability of code written in the language. Consider the program fragment given below:

```
BEGIN
NST := TICKS() + ST;
T:=TICKS()+ST;
LOOP
WHILE TICKS()< NST DO (* nothing *) END;
T:=TICKS();
CC;
NST := T+ST;
IF KEYPRESSED()THEN EXIT;
END;
END;
END;
```

Without some explanation and comment the meaning is completely obscure. By using long identifiers instead of, for example `NST` and `ST`, it is possible to make the code more readable.

```
BEGIN
NEXTSAMPLETIME := TICKS()+SAMPLETIME;
TIME:=TICKS()+SAMPLETIME;
LOOP
WHILE TICKS()< NEXTSAMPLETIME DO (* NOTHING *)
END;
TIME:=TICKS();
CONTROLCALCULATION;
NEXTSAMPLETIME:=TIME+SAMPLETIME;
IF KEYPRESSED()THEN EXIT;
END;
END;
END;
```

The meaning is now a little clearer, although the code is not easy to read because it is entirely in upper case letters.

We find it much easier to read lower case text than upper case and hence readability is improved if the language permits the use of lower case text. It also helps if we can use a different case (or some form of distinguishing mark) to identify the reserved words of the language. Reserved words are those used to identify particular language constructs, for example repetition statements, variable declarations, etc. In the next version we use upper case for the reserved words and a mixture of upper and lower case for user-defined entities.

```
BEGIN
NextSampleTime := Ticks()+SampleTime;
Time:=Ticks()+SampleTime;
LOOP
WHILE Ticks()< NextSampleTime DO (* nothing *)
END;
Time:=Ticks();
ControlCalculation;
NextSampleTime := Time+SampleTime;
IF KeyPressed()THEN EXIT;
END;
END;
END;
```

The program is now much easier to read in that we can easily and quickly pick out the reserved words. It can be made even easier to read if the language allows embedded spaces and tab characters to be used to improve the layout.

```
BEGIN (* Main program *)
   NextSampleTime := Ticks() + SampleTime;
   Time := Ticks() + SampleTime;
   LOOP
      WHILE Ticks() < NextSampleTime DO
         (* nothing *)
      END (* of WHILE *);
      Time := Ticks();
      ControlCalculation;
      NextSampleTime := Time + SampleTime;
      IF KeyPressed() THEN EXIT;
      END (* IF *);
   END (* of LOOP *);
END (* MAIN *).
```

The exact form of layout adopted is a matter of house style or of personal preference. Modula-2 does not allow names to contain embedded spaces or characters such as the underscore as separators, so it is usual to capitalise the first letter of each word in a compound name.

Modula-2 requires reserved words to be written in upper case. It is also case sensitive; for example, `Time` and `time` would be treated as different entities. Ada is case insensitive but a convention has been established that reserved words are written in lower case and application entities in upper case, with the underscore character used as the separator for compound names. Adopting this approach the code fragment given above would be written as shown below.

```
begin (* Main program *)
   NEXT_SAMPLE_TIME := TICKS() + SAMPLE_TIME;
   TIME := TICKS() + SAMPLE_TIME;
   loop
      while TICKS() < NEXT_SAMPLE_TIME do
         (* nothing *)
      end (* of while *);
      TIME := TICKS();
      CONTROL_CALCULATION;
      NEXT_SAMPLE_TIME := TIME + SAMPLE_TIME;
      if KeyPressed() then exit;
      end (* if *);
   end (* of loop *);
end (* main *).
```

Cooling (1991, p. 269) reports that some Ada programmers consider the adoption of this convention to have been a mistake in that it makes it more difficult to read the code. In the examples illustrating some features of Ada which are given later in this chapter, Modula-2 style is used with upper case for the reserved words.

5.3 DECLARATION AND INITIALISATION OF VARIABLES AND CONSTANTS

5.3.1 Declarations

The purpose of declaring an entity used in a program is to provide the compiler with information on the storage requirements and to inform the system explicitly of the names being used. Languages such as Pascal, Modula-2 and Ada require all objects to be specifically declared and a type to be associated with the entity when it is declared. The provision of type information allows the compiler to check that the entity is used only in operations associated with that type. If, for example, an entity is declared as being of type `REAL` and then it is used as an operand in logical operation, the compiler should detect the type incompatibility and flag the statement as being incorrect.

Some older languages, for example BASIC and FORTRAN, do not require explicit declarations; the first use of a name is deemed to be its declaration. In FORTRAN explicit declaration is optional and entities can be associated with a type if declared. If entities are not declared then implicit typing takes place: names beginning with the letters I–N are assumed to be integer numbers; names beginning with any other letter are assumed to be real numbers.

Optional declarations are dangerous because they can lead to the construction of syntactically correct but functionally erroneous programs. Consider the following program fragment:

```
100 ERROR=0
    ...
200 IF X=Y THEN GOTO 300
250 EROR=1
300 ...
    ...
400 IF ERROR=0 THEN GOTO 1000
    ...
```

In FORTRAN (or BASIC), `ERROR` and `EROR` will be considered as two different variables whereas the programmer's intention was that they should be the same – the variable `EROR` in line 250 has been mistyped. FORTRAN compilers cannot detect this type of error and it is a *characteristic* error of FORTRAN. Many organisations which use FORTRAN extensively avoid such errors by insisting that all entities are declared and the code is processed by a preprocessor which checks that all names used are mentioned in declaration statements.

The implicit typing which takes place in FORTRAN can also lead to confusion

and misinterpretation. Consider the program fragment

```
REAL KP,KD,KI
INTEGER DACV,ADCV
...
100    DACV=KP*(KD/KI)
...
END
```

A programmer reading the statement with the label 100, without reference to the declaration statement, might assume that the variables on the right-hand side of the statement are all integers and that the resultant value is then floated and assigned to a real variable, whereas the statement is doing just the opposite: the variables KP, KD and KI are real and the resultant is truncated and assigned to an integer variable.

A common method of avoiding this problem is to insist that all variable names conform to the implicit typing rules. Meaningful names that do not conform are prefixed with an appropriate letter. Therefore a normally implicit real name used for an integer is prefixed I and a normally implicit integer name used for a real number is prefixed X. The above fragment is written

```
REAL XKP, XKD, XKI
INTEGER IDACV, IADCV
100 IDACV = XKP*(XKD/XKI)
```

The most notorious example of the lack of security in programs written in FORTRAN is the error that caused the misdirection of the Voyager spacecraft. The FORTRAN statement intended was

```
DO 20 I = 1,100
```

which is a loop construct. What was typed was

```
DO 20 I = 1.100
```

Because embedded spaces in names are ignored and variables need not be declared, the FORTRAN compiler treated the characters to the left of the assignment operator as a variable name DO20I, and because the name begins with the character D treated it as a real number and assigned the value 1.100 to it.

5.3.2 Initialisation

It is useful if a variable can be given an initial value when it is declared. It is bad practice to rely on the compiler to initialise variables to zero or some other value.

This is not, of course, strictly necessary as a value can always be assigned to a variable. In terms of the security of a language it is important that the compiler checks that a variable is not used before it has had a value assigned to it. The security of languages such as Modula-2 is enhanced by the compiler checking that all variables have been given an initial value. However, a weakness of Modula-2 is that variables cannot be given an initial value when they are declared but have to be initialised explicitly using an assignment statement.

5.3.3 Constants

Some of the entities referenced in a program will have constant values either because they are physical or mathematical entities such as the speed of light or π, or because they are a parameter which is fixed for that particular implementation of the program, for example the number of control loops being used or the bus address of an input or output device. It is always possible to provide constants by initialising a variable to the appropriate quantity, but this has the disadvantage that it is insecure in that the compiler cannot detect if a further assignment is made which changes the value of the constant. It is also confusing to the reader since there is no indication which entities are constants and which are variables (unless the initial assignment is carefully documented).

Pascal provides a mechanism for declaring constants, but since the constant declarations must precede the type declarations, only constants of the predefined types can be declared. This is a severe restriction on the constant mechanism. For example, it is not possible to do the following:

```
TYPE
   AMotorState = (OFF,LOW,MEDIUM,HIGH);
CONST
   motorStop = AMotorState(OFF);
```

A further restriction in the constant declaration mechanism in Pascal is that the value of the constant must be known at compilation time and expressions are not permitted in constant declarations. The restriction on the use of expressions in constant declarations is removed in Modula-2 (experienced assembler programmers will know the usefulness of being able to use expressions in constant declarations). For example, in Modula-2 the following are valid constant declarations:

```
CONST
   message = 'a string of characters';
   length = 1.6;
   breadth = 0.5;
   area = length * breadth;
```

In Ada the value of the constant can be assigned at run-time and hence it is more

appropriate to consider constants in Ada as special variables which become read only following the first assignment to them.

5.4 MODULARITY AND VARIABLES

5.4.1 Scope and Visibility

The scope of a variable is defined as the region of a program in which the variable is potentially accessible or modifiable. The regions in which it may actually be accessed or modified are the regions in which it is said to be visible.

Most languages provide mechanisms for controlling scope and visibility. There are two general approaches: languages such as FORTRAN provide a single level of locality whereas the block-structured languages such as Modula-2 provide multi-level locality.

In the block-structured languages entities which are declared within a block may only be referenced inside that block. Blocks can be nested and the scope extends throughout any nested blocks. This is illustrated in Example 5.1 which shows the scope for a nested `PROCEDURE` in Modula-2.

EXAMPLE 5.1

```
MODULE ScopeExample1;
VAR
   A,B: INTEGER;
PROCEDURE LevelOne;
VAR
   B,C: INTEGER;
BEGIN
   (*
      LevelOne.B and LevelOne.C visible here
   *)
END (* LevelOne *);
BEGIN
   (*
      A and B visible here but not LevelOne.B and
LevelOne.C
   *)
END ScopeExample1.
```

The *scope* of variables `A` and `B` declared in the main module `ScopeExample1` extends throughout the program, that is they are global variables. However, because

the variable name `B` is reused in `PROCEDURE LevelOne` it hides the global variable `B` while procedure `LevelOne` is executing. The scope of variables `LevelOne.B` and `LevelOne.C` extends over `PROCEDURE LevelOne` and both are visible within the procedure.

As in this example, in a block-structured language an entity declared in an inner block may have the same name as an entity declared in an outer block. This does not cause any confusion to the compiler which simply provides new storage for the entity in the inner block and the entity in the outer block temporarily *disappears*, to reappear when the inner block is left.

In Modula-2, as in most block-structured languages, variables declared inside a procedure have storage allocated to them only while that procedure is being executed. When the procedure is entered storage in RAM is allocated from available memory (often referred to as *heap*); if no memory is available a run-time error is generated. When an exit is made from the procedure the memory is released and can be reused.

This dynamic allocation and release of memory leads to two problems:

- variables declared within a procedure cannot be used to hold values for reuse on the next entry to the procedure; and
- if a procedure is called recursively it is possible that the program may fail because there is no more memory available.

Modula-2 overcomes the first of these problems through its use of a program unit `MODULE` which has some specific properties. It is because of the second problem that recursively called procedures should not be used in real-time systems, particularly if failure will compromise the safety of people or equipment (even if the procedures do not declare any variables each call makes demands on the stack space for storage of its volatile environment). To construct a safe system it must be possible to predict the maximum memory requirements.

5.4.2 Global and Local Variables

Although the compiler can easily handle the reuse of names, it is not as easy for the programmer and the use of deeply nested `PROCEDURE` blocks with the reuse of names can compromise the security of a Pascal or Modula-2 program. As the program shown in Example 5.2 illustrates the reuse of names can cause confusion as to which entity is being referenced.

EXAMPLE 5.2

Loss of Visibility in Nested Procedures

```
MODULE ScopeL2;
VAR X, Y, Z : INTEGER;
PROCEDURE L1;
   VAR Y : INTEGER;
   PROCEDURE L2;
      VAR X : INTEGER;
      PROCEDURE L3;
         VAR Z : INTEGER;
         PROCEDURE L4;
         BEGIN
            Y : = 25; (* L1.Y NOT L0.Y*)
         END L4;
      BEGIN
         (* L1.Y, L2.X, L3.Z visible *)
      END L3;
   BEGIN
      (* L1.Y, L2.X, L0.Z visible *)
   END L2;
BEGIN
   (* L0.X, L1.Y, L0.Z visible *)
END L1;
BEGIN
(* ... *)
END ScopeL2.
```

It is very easy to assume in assigning the value 25 to `Y` in `PROCEDURE L4` that the global variable `Y` is being referenced, when in fact it is the variable `Y` declared in `PROCEDURE L1` that is being referenced.

The argument over global or local declaration of entities has been almost as fierce as that over the use of `GOTO` statements. The proponents of local declarations argue that it is good practice to introduce and name entities close to where they are to be used and thus to limit the scope of the entity and its visibility. Those arguing in favour of global visibility of names claim that it is the only way in which consistency and control of the naming of entities can be achieved for large systems being developed by a team of programmers. They argue that local declaration leads to duplication of names and difficulties in subsequent maintenance of programs. A sensible compromise position is probably to declare globally the names of all entities which directly relate to the outside world, that is to the system being modelled or controlled, and to use local declaration for the names of all internal entities. Many

of the difficulties disappear if the language permits explicit control over the scope and visibility of entities and does not rely on default rules.

5.4.3 Control of Visibility and Scope

Modern approaches to software design place a lot of emphasis on modularity and information hiding, both of which contribute to security. For effective use of modularisation in program design a language must provide mechanisms that enable the programmer to control explicitly the scope and visibility of all entities. This includes the ability to extend the scope of entities declared within a program unit to areas outside that unit. These problems have been addressed by a number of language designers. Program units which provide the necessary facilities have been devised; they are known by several different names, for example a *class* in SIMULA, a *module* in Modula-2, a *segment* in ALGOL and a *package* in Ada.

5.4.4 Modularity

In Modula-2 the main program unit is a `MODULE` and local modules can be nested within the main module. The nesting of modules is illustrated in Example 5.3.

EXAMPLE 5.3

Scope and Visibility Control in Modula-2

```
MODULE ImportExport;
(*
Title : Example of Import and Export of objects
File : sb1 Importex.mod
LastEdit:
Author : S. Bennett
*)
```

```
VAR a,b,c : INTEGER;
(* ... *)
   MODULE L1;
      IMPORT a;
      EXPORT d,g;
      VAR d,e,f : INTEGER;
         MODULE L2;
            IMPORT e,f;
            EXPORT g,h;
            VAR g,h,i : INTEGER;
            (* ...
               e,f,g,h,i visible here
            *)
         END L2;
      (* ...
         a,d,e,f,g,h are visible here
      *)
   END L1;
(* ...
a,b,c,d,g are visible here
*)
END ImportExport.
```

To allow entities which are declared within the body of a module to be visible outside the module they must be listed in an `EXPORT` list. Similarly entities which are declared outside the module must be specifically imported into the module by naming them in an `IMPORT` list. It should be noted that all entities can be imported and exported, that is variables, constants, types and procedures. Entities which are declared in a module are created at the initialisation of the program and remain in existence throughout the existence of the program, that is they are not like entities created inside a procedure which cease to exist when the procedure body is left.

The more general concept of a module allows a program to be split into many separate units, each of which can be compiled separately. The facilities for separation also allow for the construction of program libraries in which the library segments are held in compiled form rather than as source code. In order to do this the module is split into two parts:

`DEFINITION MODULE` – this contains information about entities which are exported from the module;

`IMPLEMENTATION MODULE` – this is the body of the module which contains the code which carries out the functions of the module.

The `DEFINITION` part is made available to the client program in source form, but the `IMPLEMENTATION` part is only provided in object form and its source code remains private to the module designer. The separation provides an excellent method of hiding implementation details from a user, and the actual

implementation can be changed without informing the user, providing that the **DEFINITION** part does not change. An example of a **DEFINITION MODULE** is given in Example 5.4.

EXAMPLE 5.4

DEFINITION MODULE in Modula-2

```
DEFINITION MODULE Buffer;
(*
Title : Example of definition module
File : sb1 Buffer.mod
*)

EXPORT QUALIFIED
   put, get, nonEmpty, nonFull;
VAR
   nonEmpty, nonFull : BOOLEAN; (* used to test the
status of buffer *)

PROCEDURE put (x : CARDINAL);
   (* used to add items to the buffer *)
PROCEDURE get (VAR x: CARDINAL)
   (* used to remove items from the buffer *)
END Buffer.
```

5.5 COMPILATION OF MODULAR PROGRAMS

If we have to use a modular approach in designing software how do we compile the modules to obtain executable object code? There are two basic approaches: either combine at the source code level to form a single unit which is then compiled, or compile the individual modules separately and then in some way link the compiled version of each module to form the executable program code. Using the second approach a special piece of software called a *linker* has to be provided as part of the compilation support to do the linking of the modules.

A reason for the popularity and widespread use of FORTRAN for engineering and scientific work is that subroutines can be compiled independently from the main program, and from each other. The ability to carry out compilation independently arises from the single-level scope rules of FORTRAN; the compiler makes the assumption that any entity which is referenced in a subroutine, but not declared within that subroutine, will be declared externally and hence it simply inserts the necessary external linkage to enable the linker to attach the appropriate code. It must be stressed that the compilation is *independent*, that is when a main program

is compiled the compiler has no information available which will enable it to check that the reference to the subroutine is correct. For example, a subroutine may expect three real variables as parameters, but if the user supplies four integer variables in the call statement the error will not be detected by the compiler. Independent compilation of most block-structured languages is even more difficult and prone to errors in that arbitrary restrictions on the use of variables have to be imposed. Many errors can be detected at the linking stage. However, because linking comes later in the implementation process errors discovered at this stage are more costly to correct. It is preferable to design the language and compilation system in such a way as to be able to detect as many errors as possible during compilation instead of when linking.

Both Modula-2 and Ada have introduced the idea of *separate* compilation units. Separate compilation implies that the compiler is provided with some information about the previously or separately compiled units which are to be incorporated into a program. In the case of Modula-2 the source code of the `DEFINITION` part of a separately compiled module must be made available to the user, and hence the compiler. This enables the compiler to carry out the normal type checking and procedure parameter matching checks. Thus in Modula-2 type mismatches and procedure parameter errors are detectable by the compiler. It also makes available the scope control features of Modula-2.

The provision of independent compilation of the type introduced in FORTRAN represented a major advance in supporting software development because it enabled the development of extensive object code libraries. Languages which support separate compilation represent a further advance in that they add greater security and easy error checking to library use.

5.6 DATA TYPES

As we have seen above, the allocation of types is closely associated with the declaration of entities. The allocation of a type defines the set of values that can be taken by an entity of that type and the set of operations that can be performed on the entity. The richness of types supported by a language and the degree of rigour with which type compatibility is enforced by the language are important influences on the security of programs written in the language. Languages which rigorously enforce type compatibility are said to be *strongly* typed; languages which do not enforce type compatibility are said to be *weakly* typed.

FORTRAN and BASIC are weakly typed languages: they enforce some type checking; for example, the statements `A$=25` or `A=X$+Y` are not allowed in BASIC, but they allow mixed integer and real arithmetic and provide implicit type changing in arithmetic statements. Both languages support only a limited number of types.

An example of a language which is strongly typed is Modula-2. In addition to

enforcing type checking on standard types, Modula-2 also supports enumerated types. The enumerated type allows programmers to define their own types in addition to using the predefined types. Consider a simple motor speed control system which has four settings `OFF`, `LOW`, `MEDIUM`, `HIGH` and which is controlled from a computer system. Using Modula-2 the programmer could make the declarations:

```
TYPE
   AMotorState = (OFF,LOW,MEDIUM,HIGH);
VAR
   motorSpeed : AMotorState;
```

The variable `motorSpeed` can be assigned only one of the values enumerated in the `TYPE` definition statement. An attempt to assign any other value will be trapped by the compiler, for example the statement

```
motorSpeed := 150;
```

will be flagged as an error.

If we contrast this with the way in which the system could be programmed using FORTRAN we can see some of the protection which strong typing provides. In ANSI FORTRAN integers must be used to represent the four states of the motor control:

```
INTEGER OFF,LOW,MEDIUM,HIGH
DATA OFF/0/,LOW/1/,MEDIUM/2/,HIGH/3/
```

If the programmer is disciplined and only uses the defined integers to set `MSPEED` then the program is clear and readable, but there is no mechanism to prevent direct assignment of any value to `MSPEED`. Hence the statements

```
MSPEED = 24
MSPEED = 150
```

would be considered as valid and would not be flagged as errors either by the compiler or by the run-time system. The only way in which they could be detected is if the programmer inserted some code to check the range of values before sending them to the controller. In FORTRAN a programmer-inserted check would be necessary since the output of a value outside the range 0 to 3 may have an unpredictable effect on the motor speed.

5.6.1 Sub-range Types

Another valuable feature which enhances security is the ability to declare a sub-range of a type. In Modula-2 sub-ranges of ordinal types (that is, `INTEGER`, `CHAR`

and ENUMERATED types) can be defined. The following statements define sub-range types:

```
TYPE
   ADACValue = 0..255;
   ALowerCaseChar = 'a'..'z';
```

and if the variables are defined as

```
VAR
   output : ADACValue;
   character : ALowerCaseChar;
```

then the assignments

```
output := -25;
character := 'A';
```

will be flagged as errors by the compiler. The compiler will also insert run-time checks on all assignment statements involving sub-range types and any assignment which violates the permitted values will generate a run-time error. The use of sub-range types can increase the security of a program, but in a real-time system full use of sub-range types may not be appropriate. They can be used if the run-time system permits the transfer of control to a user-supplied error analysis segment on detection of a run-time error; if it does not and it terminates execution of the program then the security of the system can be jeopardised by their use. Sub-range types can be useful during the development stages of the system as violations of correctly set sub-ranges can indicate logical errors in the code. For this reason many compilers provide an option switch to control the inclusion of sub-range checking.

Sub-range types have been extended in Ada to include sub-ranges of REAL. Again the usefulness is limited because of the extra code introduced in order to check for violations. In applications involving a large amount of computation the use of sub-ranges of REAL can significantly slow down the computation and hence in many applications the efficiency requirements will necessitate the use of explicit range checks at appropriate points rather than the use of compiler-supplied checks through the use of sub-range types.

5.6.2 Derived Types

In many languages new types can be created from the implicit types: these are known as derived types and they inherit all the characteristics of the parent type. The use of derived types can make the meaning of the code clearer to the reader,

for example

```
TYPE
   AVoltage = REAL;
   AResistance = REAL;
   ACurrent = REAL;
VAR
   V1 : AVoltage;
   R3 : AResistance;
   I2 : ACurrent;
BEGIN
   I2 := V1/R3;
END;
```

In the above code the reader can easily see what is implied by the calculation: Ohm's law is being used to calculate the current flowing through a resistance. If the statement read

```
I2 := V1*R3;
```

then because `I2` has been declared as of type `ACurrent` the reader would be suspicious that there was an error in the line.

In a very strongly typed language such as Ada neither of the above statements would be accepted by the compiler for although derived types inherit the properties of the parent type they are treated as distinct types and hence are not compatible. Strong typing of this form has the advantage that the compiler can detect errors such as assigning `V1` which represents a voltage to the variable `I2`, which represents a current, as is shown in the first statement in the program fragment below, but it would also flag as an error the perfectly legitimate operation shown in the second statement:

```
I2 := V1;
I2 := V1/R3
```

In Ada this problem is overcome by a mechanism that permits operators on types to be *overloaded*, that is the operator can be redefined to be compatible with a different set of types. In this case the operator / would have to be defined as performing the division operation with a variable of type `AVoltage` as the dividend, a variable of type `AResistance` as the divisor and a variable of type `ACurrent` as the resultant. It is arguable whether the benefits of the strong typing outweigh the disadvantage of the complexity of overloading.

5.6.3 Structured Types

Many programming languages provide only one structured type, the array. Arrays, though powerful, are limited in that all the elements of the array must be of the same

type. There are many applications in which it would be useful to be able to declare entities which are made up of elements of different types.

EXAMPLE 5.5

```
TYPE
   AController = RECORD
      inputAddress : INTEGER;
      outputAddress : INTEGER;
      maxOutput : REAL;
      name : ARRAY [0..16] OF CHAR;
      status : (OFF,ON);
   END (* RECORD *);
VAR
   airFlow : AController;
   fuelFlow : AController;
   reactant : AController;

PROCEDURE Control(VAR loop : AController);
(* controller that is used to control several loops
*)
END (* CONTROL *);
BEGIN
   LOOP
      Control(fuelFlow);
      Control(reactant);
   END (* LOOP *);
END.
```

In Example 5.5 the whole of the information relevant to a particular control loop for the plant is contained in one variable of type `RECORD`. The advantages of using a structure type such as the record structure is that the programmer does not continually have to consider the details of the way in which the information relevant to the variable is stored: it contributes to the process known as abstraction. In addition to `RECORD` many of the modern languages support types such as `FILE` and `SET`.

5.6.4 Pointers

Pointers provide a mechanism for indirect reference to variables. They are widely used in systems programming and in some data processing applications. They can be used, for example, to create linked lists and tree structures. The unrestricted

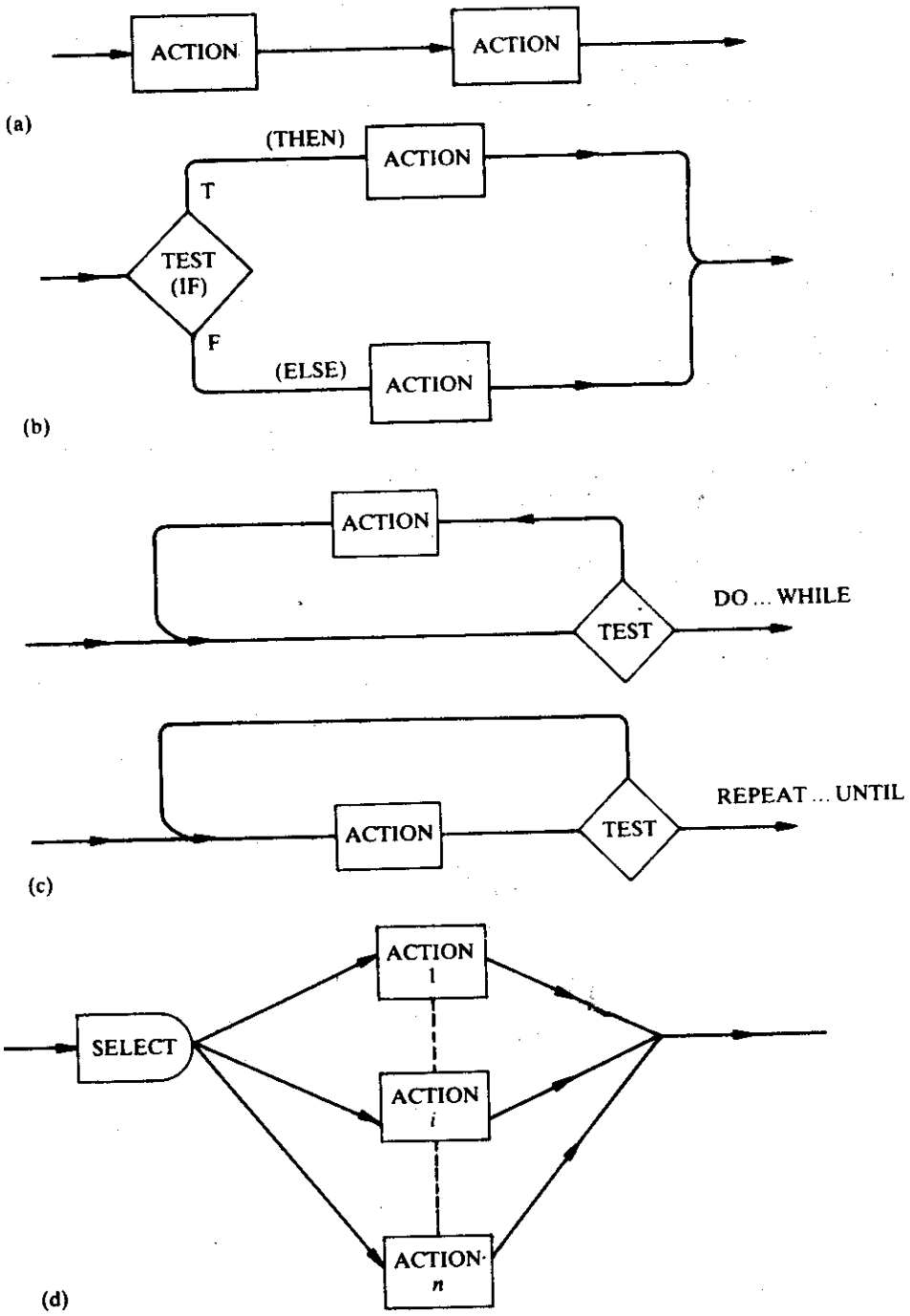

Figure 5.1 Standard structured program constructs: (a) sequence; (b) decision; (c) repetition; (d) selection.

availability of pointers, for example permitting pointers to entities of different types to be interchanged, can give rise to insecurity. Implicit in the use of pointers is the dynamic allocation and consequent deallocation of storage; the overheads involved in the required mechanisms can be considerable and care in the use of pointer types is necessary in real-time applications.

5.7 CONTROL STRUCTURES

There have been extensive arguments in the past about the use of both conditional and unconditional `GOTO` statements in high-level languages. It is argued that the use of `GOTO`s makes a program difficult to read and it has been shown that any program can be expressed without the use of `GOTO`s as long as the language supports the `WHILE` statement, the `IF...THEN...ELSE` conditional and `BOOLEAN` variables. Most modern languages support such statements. The standard structured programming constructs are shown in Figure 5.1.

From a theoretical point of view the avoidance of `GOTO`s is attractive. There are, however, some practical situations in which the judicious use of a `GOTO` can avoid complicated and confusing coding. An example of such a situation is when it is required to exit from a loop in order to avoid further processing when a particular condition occurs.

EXAMPLE 5.6

Consider the following scenario. A stream of data in character form is received from a remote station over a serial link. The data has to be processed character by character by a routine `ProcessItem` until the end-of-transmission character (EOT – ASCII code = 4) is received. The EOT character must not be processed.

A simple loop structure of the form

```
REPEAT
   get(character);
   ProcessItem(character);
UNTIL character = EOT;
```

cannot be used since the `EOT` character would be processed. Possible solutions are:

1.

```
Get(Character);
WHILE Character <> EOT DO
   ProcessItem(Character);
   Get(Character)
END (* WHILE *);
```

2.

```
Finished := FALSE;
REPEAT
   Get(Character);
   IF Character = EOT THEN
      Finished := TRUE
   ELSE
      ProcessItem(Character)
   END (* IF *);
UNTIL Finished;
```

A much cleaner solution is provided by the use of an EXIT statement as in the fragment below:

3.

```
LOOP
   Get(Character);
   IF Character=EOT THEN
      EXIT
   END (* IF *);
   ProcessItem(Character);
   END (* LOOP *);
(* EXIT causes a jump to statement here if EOT is
detected *)
```

The solution shown in (3) which is possible in Modula-2 is much clearer because all the operations are shown within the LOOP statement, whereas in solutions (1) and (2) some operation has to be performed outside the loop. Either the first character has to be obtained before entering the loop or a Boolean variable has to be set before the loop is entered. The general LOOP...END statement becomes particularly valuable if several different exceptions require an exit from the loop.

A general non-terminating loop statement which can contain one or more EXIT statements that result in control passing to the statement immediately following the end of the loop is useful as is a RETURN statement (for use in PROCEDUREs) that results in an immediate exit from the PROCEDURE. Both are useful additions to control structures. Although not strictly required they can lead to clearer and simpler coding. A program illustrating their use is given in Example 5.7.

EXAMPLE 5.7

Behaviour of RETURN and EXIT statements.

```
MODULE ReturnExitLoop ;
(*
Title : Example of different methods of leaving LOOP
statement
*)
```

```
FROM InOut IMPORT
   WriteString, WriteLn;
IMPORT Terminal;
VAR
   ch: CHAR;
   return : BOOLEAN;

PROCEDURE ReadKey ;
BEGIN
   WriteString ('ReadKey called  ');
   LOOP
      WHILE NOT (Terminal.KeyPressed()) DO
         (* do nothing *)
      END (* while *);
      Terminal.Read(ch);
      IF ch='r' THEN
         RETURN
      END (* if *);
      IF ch='e' THEN
         EXIT
      END (* if *);
   END (* loop *);
   WriteString ('this was EXIT not RETURN');
   WriteLn;
   return:=FALSE
END ReadKey;
BEGIN
   LOOP
      return:=TRUE;
      ReadKey;
      IF return THEN
         WriteString ('this was RETURN not EXIT');
         WriteLn
      END (* if *);
      WriteString ('To repeat type c, to stop any
other character');
      WHILE NOT (Terminal.KeyPressed()) DO
         (* do nothing *)
      END (* while *);
      Terminal.Read(ch);
      IF ch<>'c' THEN
         EXIT
      END (* if *);
      WriteLn
   END (* LOOP *)
END ReturnExitLoop.
```

5.8 EXCEPTION HANDLING

One of the most difficult areas of program design and implementation is the handling of errors, unexpected events (in the sense of not being anticipated and hence catered for at the design stage) and exceptions which make the processing of data by the subsequent segments superfluous, or possibly dangerous. The designer has to make decisions on such questions as what errors are to be detected? What sort of mechanism is to be used to do the detection? And what should be done when an error is detected?

Most languages provide some sort of automatic error detection mechanisms as part of their run-time support system. Typically they trap errors such as an attempt to divide by zero, arithmetic overflow, array bound violations, and sub-range violations; they may also include traps for input/output errors. For many of the checks the compiler has to add code to the program; hence the checks increase the size of the code and reduce the speed at which it executes. In most languages the normal response when an error is detected is to halt the program and display an error message on the user's terminal.

In a development environment it may be acceptable for a program to halt following an error; in a real-time system halting the program is not acceptable as it may compromise the safety of the system. Every attempt must be made to keep the system running.

EXAMPLE 5.8

Error Checking

Consider a boiler control system (similar to the one described in Chapter 2). One control loop uses the ratio of fuel flow and air flow to calculate the set point for the controller. Assume that the values of fuel flow and air flow are read from the measuring instruments and stored in `REAL` variables `FuelFlow` and `AirFlow` respectively. The instruments provide values which are in the range 0.0 to 4096.0 corresponding to the flow ranges 0 to 100%. The control setting is to be held in a real variable, `RatioSetPoint`. The programmer might write the statement:

```
RatioSetPoint := FuelFlow / AirFlow
```

A program using this statement may function correctly for a long period of time, but suppose a fault either on the air flow measuring instrument or in the interface unit results in `AirFlow` being set equal to zero. The program, and hence the system, would halt with an error – attempt to divide by zero.

One method of dealing with this problem is to validate the data prior to executing the statement by adding code to give

```
IF AirFlow > 0 THEN
   RatioSetPoint := FuelFlow / AirFlow
ELSE
   AirFlowAlarm := TRUE;
   RatioSetPoint := DefaultValue
END (* IF *);
```

An alternative method, if the language supports sub-ranges of REAL variables, is to declare AirFlow to be in the range 0.0 to 4096.0 and to allow the compiler to insert the necessary sub-range checks; but what happens when the compiler detects a sub-range violation – does it simply halt the program? If it does, then nothing has been gained.

In Example 5.8 it is easy for the programmer to put in the necessary checks. However, checking for all possible data errors can become very complex, can obscure the general flow of the code, and can slow down execution.

A better solution for error handling in real-time systems is for the language to be designed so as to allow the application software to deal with errors when they are detected, and to pass the error on to the built-in error handling mechanism only if the application software fails to deal with the error. We need the language run-time support software to detect as many types of error as possible, to inform the application software that an error exists, and to contain error handling routines that will deal with the error if the application software does not do so.

One of the first languages to provide error handling in this way was BASIC with its ON ERROR GOTO and ON ERROR GOSUB statements. Use of this type of statement enables error trap routines to be inserted in the application program and can simplify the flow of the code as it permits grouping of the error handling and analysis into one place. The run-time error can be checked and action taken to keep the system running; if it cannot be kept running then at least there may be an opportunity to close it down safely and warn the operator.

EXAMPLE 5.9

Using BASIC the problem given in Example 5.8 can be dealt with as follows:

```
1000 ON ERROR GOSUB 9500;
     (All errors detected in the following lines of
     code will be
     (trapped and referred to the subroutine at line
     9500 until a
     (further ON ERROR statement is executed.
1040
1050 RatioSetPoint = FuelFlow / AirFlow
1060
1070 REM A return from the ERROR subroutine at
     9500 is made here.
     (...
     (Rest of code inserted here
     (...
9500 REM Start of error handler
9510 IF ERRORNO=DivideByZero THEN RatioSetPoint =
       DefaultValue
9520 ELSE GOTO ERRORTRAP
9530 RETURN
```

The error detection mechanism is assumed to return an error number to the program; if this error number matches the attempt to divide by zero error number then `RatioSetPoint` is set to a default value, otherwise the error was not expected at this point and has to be either passed to some more general error analysis routine or returned to the underlying default error handler which will normally halt the running of the program.

The obvious advantage of this approach is that the designer can consider the normal function and the error actions as two separate problems. It also has the advantage that execution speed is improved and in more complex cases the readability of the code is improved. The disadvantage is that it encourages reliance on the run-time system to catch errors rather than careful consideration of the possible errors.

The use of error trapping or exception statements requires careful consideration of the action which follows the execution of the error handling routine. In Example 5.9 it is assumed that execution resumes at the next statement after the statement which generated the error. Alternatively we could have written the example as follows:

```
1000 ON ERROR GOTO 9500;
1040
1050 RatioSetPoint = FuelFlow / AirFlow
1060
    (...
    (Rest of code inserted here
    (...
9500 REM Start of error handler
9510 IF AirFlow < 0 THEN AirFlow = 100.0
9520 GOTO 1050
```

In the above code, return is made to the statement that caused the error, the cause of the error having been corrected (hopefully!). (A potential problem with returning to the statement that caused the error to be raised is that if the correction is incorrect the same error will be raised again and the software will cycle endlessly round the same loop.)

The facility to trap and return information on potentially fatal errors to the application program is only one aspect of error handling. There will be abnormal or error conditions which the underlying language support system will not detect. For example, if in Example 5.9 the valid range of `AirFlow` was 100.0 to 4096.0 then there would be an error if `AirFlow` was set to 50.0. This would not result in system error (unless sub-range violations were being checked); however, the application program designer would be helped if, having dealt with the immediate problem, there was some means by which some further error analysis or error handling could easily be invoked. It is always possible to write such error handling mechanisms explicitly as Example 5.10 illustrates.

EXAMPLE 5.10

Suppose that the measured values of air flow and fuel flow obtained from the instruments have to be converted to actual flow rates by taking the square root of the input value. An instrument error causing a negative input value results in an error return from the square root function. We wish to protect against this error and also want to notify other parts of the system that an instrument error has been detected. Hence we might code the conversion as follows:

```
BEGIN
   IF RawFuelFlow < 0 THEN
      RawFuelFlow := MinValueRawFuelFlow;
      RaiseError('RawFuelFlow')
   END (* IF *);
   FuelFlow := SQRT(RawFuelFlow);
   IF RawAirFlow <= 0 THEN
      RawAirFlow := MinValueRawAirFlow;
      RaiseError('RawAirFlow')
   END (* IF *);
   AirFlow : = SQRT(RawAirFlow);
   RatioSetPoint := FuelFlow/AirFlow;
END

PROCEDURE RaiseError(VAR ErrorMessage: STRING);
(*
   Procedure to raise alarms according to error
message value
*)
END RaiseError;
```

In Ada there is explicit language provision for handling errors. The above could be written as follows:

```
PROCEDURE Convert_Flows;
BEGIN
   IF Raw_Fuel_Flow < 0 THEN
   RAISE Fuel_Error;
   Fuel_Flow := SQRT(Raw_Fuel_Flow);
   IF Raw_Air_Flow <= 0 THEN
   RAISE Air_Error;
   Air_Flow : = SQRT(Raw_Air_Flow);
   Ratio_Set_Point := Fuel_Flow/Air_Flow;
   EXCEPTION
      WHEN Fuel_Error =>
         ALARM(Fuel);
      WHEN Air_Error =>
         ALARM(Air);
   END Convert_Flows;
```

The major problem in exception handling occurs when procedures are nested and a report of the error has to be passed from one procedure to another.

EXAMPLE 5.11

Consider the following system:

```
PROCEDURE A;
BEGIN
...
B;          ----------------
            PROCEDURE
            B;
            BEGIN

            C;
            ----------------  ------------------------------
                              PROCEDURE C;
                              BEGIN
                              IF RawFuelFlow < 0 THEN
                                RawFuelFlow :=
                              MinValueRawFuelFlow;
                                RaiseError('RawFuelFlow')
                              END (* IF *);
                              FuelFlow :=
                              SQRT(RawFuelFlow);
                              IF RawAirFlow <= 0 THEN
                                RawAirFlow :=
                              MinValueRawAirFlow;
                                RaiseError('RawAirFlow')
                              END (* IF *);
                              AirFlow : =
                              SQRT(RawAirFlow);
                              FuelFlow :=
                              SQRT(RawFuelFlow);
                              END C;
            ----------------  ------------------------------
            ...
            END B;
            ----------------

...
(* value of AirFlow is to be used here *)
END A;
```

In Example 5.11, if the value of `RawAirFlow` is found to be in error in `PROCEDURE C` then the code in `PROCEDURE B` and in `PROCEDURE A` may not be

relevant if it requires a correct value for `RawAirFlow`. The standard means of passing back the information that an incorrect value of `RawAirFlow` has been used is to pass an error flag value in the procedure calls, that is to change the simple procedure calls to ones with the form `PROCEDURE ((* arguments *), VAR ErrorMessage:STRING)`.

Ada offers an alternative solution in that when an exception is raised it is passed from block to block until the appropriate exception handler is found. Thus in the above example, if the exception was raised in `PROCEDURE C` and there was no exception handler in `C`, then `C` would be terminated and control would pass to `B`; if there were no handler in `B` it would also be terminated and control passed to `A`.

One special purpose language, CUTLASS, which was developed for real-time use, offers an interesting approach to this problem. It marks data as *bad* by using one bit in the storage used for data values as a *tag* bit. When an error is detected the data value is marked as bad and then allowed to propagate normally through the system. Any action needed to deal with bad data is then taken at the point at which it is used. The rules used in the propagation of bad data produce a result which is marked as bad if any of the operands in an arithmetic operation are bad. An example illustrating the technique is given in Section 5.16.2.

5.9 LOW-LEVEL FACILITIES

In programming real-time systems we frequently need to manipulate directly data in specific registers in the computer system, for example in memory registers, CPU registers and registers in an input/output device. In the older, high-level languages, assembly-coded routines are used to do this. Some languages provide extensions to avoid the use of assembly routines and these typically are of the type found in many versions of BASIC. These take the following form:

PEEK(address) – returns as INTEGER variable contents of the location address.
POKE(address, value) – puts the INTEGER value in the location address.

It should be noted that on eight-bit computers the integer values must be in the range 0 to 255 and on 16 bit machines they can be in the range 0 to 65 535. For computer systems in which the input/output devices are not memory mapped, for example Z80 systems, additional functions are usually provided such as

INP(address) and
OUT(address, value).

A slightly different approach has been adopted in BBC BASIC which uses an 'indirection' operator. The indirection operator indicates that the variable which follows it is to be treated as a pointer which contains the address of the operand

rather than the operand itself (the term indirection is derived from the indirect addressing mode in assembly languages). Thus in BBC BASIC the following code

```
100 DACAddress=&FE60
120 ?DACAddress=&34
```

results in the hexadecimal number 34 being loaded into location FE60H; the indirection operator is '?'.

In some of the so-called Process FORTRAN languages and in CORAL and RTL/2 additional features which allow manipulation of the bits in an integer variable are provided, for example

```
SET BIT J(I),
IF BIT J(I) n1,n2 (where I refers to the bit in
variable J).
```

Also available are operations such as AND, OR, SLA, SRA, etc., which mimic the operations available at assembly level. The weakness of implementing low-level facilities in this way is that all type checking is lost and it is very easy to make mistakes. A much more secure method is to allow the programmer to declare the address of the register or memory location and to be able to associate a type with the declaration, for example

```
VAR charout AT OFE60H :CHAR;
```

which declares a variable of type CHAR located at memory location OFE60H. Characters can then be written to this location by simple assignment

```
charout := 'a';
```

Note that the compiler would detect and flag as an error an attempt to make the assignment

```
charout := 45;
```

since the variable is typed. Both Modula-2 and Ada permit declarations of the above type.

Modula-2 provides a low-level support mechanism through a simple set of primitives which have to be encapsulated in a small nucleus coded in the assembly language of the computer on which the system is to run. Access to the primitives is through a module SYSTEM which is known to the compiler. SYSTEM can be thought of as the software bus linking the nucleus to the rest of the software modules. SYSTEM makes available three data types, WORD, ADDRESS, PROCESS, and six procedures, ADR, SIZE, TSIZE, NEWPROCESS, TRANSFER, IOTRANSFER. WORD is the data type which specifies a variable which maps onto

one unit of the specific computer storage. As such the number of bits in a WORD will vary from implementation to implementation; for example, on a PDP-11 implementation a WORD is 16 bits, but on a 68000 it would be 32 bits. ADDRESS corresponds to the definition TYPE ADDRESS = POINTER TO WORD, that is objects of type ADDRESS are pointers to memory units and can be used to compute the addresses of memory words. Objects of type PROCESS have associated with them storage for the volatile environment of the particular computer on which Modula-2 is implemented; they make it possible to create easily process (task) descriptors (see Chapter 6 for detailed information on task descriptors).

Three of the procedures provided by SYSTEM are for address manipulation:

```
ADR(v)    returns the ADDRESS of variable v
SIZE(v)   returns the SIZE of variable v in WORDs
TSIZE(t)  returns the SIZE of any variable of type t
          in WORDs.
```

In addition variables can be mapped onto specific memory locations. This facility can be used for writing device driver modules in Modula-2. A combination of the low-level access facilities and the module concept allows details of the hardware device to be hidden within a module with only the procedures for accessing the module being made available to the end user.

Example 5.12 shows the DEFINITION MODULE for the analog input and output module for an 11/23 computer system. For normal use the two procedures ReadAnalog and WriteAnalog are all that the user requires. Additional information and procedures are made available to the expert user, including information on the actual hardware addresses (these may vary from system to system).

EXAMPLE 5.12

DEFINITION MODULE for a Device Handler

```
DEFINITION MODULE AnalogIO;
(*-------------------------------------------------------
*)
(* Analog Input/Output
*)
(* Version 0, S. White, 2-Jul-85, adapted from ADC and DAC *)
(*            modules.                                   *)
(*-------------------------------------------------------
*)
```

```
FROM SYSTEM IMPORT
   ADDRESS;
EXPORT QUALIFIED
   moduleName, moduleVersion,
   ReadAnalog, WriteAnalog;
CONST
   moduleName = 'AnalogIO';
   moduleVersion = 1;
PROCEDURE ReadAnalog( adcNum: CARDINAL; VAR val: CARDINAL);
PROCEDURE WriteAnalog( dacNum; CARDINAL; val: CARDINAL);
(* Hardware configuration - expert use only *)
TYPE
AReadProc=PROCEDURE (CARDINAL, VAR CARDINAL );
AWriteProc=PROCEDURE (CARDINAL, CARDINAL );
PROCEDURE InitAnalog(
   initProc: PROC; readProc: AReadProc;
   writeProc: AWriteProc; doneProc: PROC);
(*------------------MNCAIO--------------------*)
PROCEDURE InitMNCAIO;
PROCEDURE ReadMNCAD( adcNum: CARDINAL; VAR val: CARDINAL );
PROCEDURE WriteMNCAA( dacNum: CARDINAL; val: CARDINAL );
PROCEDURE DoneMNCAIO;
VAR
   MNCADReg: POINTER TO
      RECORD
         CSR: BITSET; (* Control/Status *)
         DBE: CARDINAL (* BufferPreset *)
      END;
   MNCADVec;
      RECORD
         ConversionComplete: ADDRESS;
         Error: ADDRESS;
      END;
CONST
   MNCADBaseReg = 171000B;
   MNCADBaseVec = 400B;
VAR
   MNCAAReg: POINTER TO
      RECORD
         DAC0: CARDINAL;
         DAC1: CARDINAL;
         DAC2: CARDINAL;
         DAC3: CARDINAL;
      END;
CONST
   MNCAABaseReg=171060B;
END AnalogIO.
```

5.10 COROUTINES

In Modula-2 the basic form of concurrency is provided by coroutines. The two procedures NEWPROCESS and TRANSFER exported by SYSTEM are defined as follows:

```
PROCEDURE NEWPROCESS(ParameterlessProcedure:PROC;
            workspaceAddress: ADDRESS;
            workspaceSize: CARDINAL;
            VAR coroutine: ADDRESS (* PROCESS *));

PROCEDURE TRANSFER(VAR source, destination : ADDRESS
             (*PROCESS*));
```

Any parameterless procedure can be declared as a PROCESS. The procedure NEWPROCESS associates with the procedure storage for the process parameters (the process descriptor or task control block – see Chapter 6) and some storage to act as workspace for the process. It is the programmer's responsibility to allocate sufficient workspace. The amount to be allocated depends on the number and size of the variables local to the procedure forming the coroutine, and to the procedures which it calls. Failure to allocate sufficient space will usually result in a stack overflow error at run-time.

The variable coroutine is initialised to the address which identifies the newly created coroutine and is used as a parameter in calls to TRANSFER. The transfer of control between coroutines is made using a standard procedure TRANSFER which has two arguments of type ADDRESS (PROCESS). The first is the calling coroutine and the second is the coroutine to which control is to be transferred. The mechanism is illustrated in Example 5.13. In this example the two parameterless procedures Coroutine1 and Coroutine2 form the two coroutines which pass control to each other so that the message

```
coroutine one coroutine two
```

is printed out 25 times. At the end of the loop, Coroutine2 passes control back to MainProgram.

EXAMPLE 5.13

Example Showing the Use of Coroutines

```
MODULE CoroutinesExample ;
(*
Title : Example of use of coroutines
*)
FROM SYSTEM IMPORT
   ADDRESS, WORD, NEWPROCESS, TRANSFER, ADR, SIZE, PROCESS;
FROM InOut IMPORT
   WriteString, WriteLn;
```

```
VAR
   coroutine1Id, coroutine2Id, MainProgram :
   PROCESS;
   worksp1, worksp2 : ARRAY [1..600] OF WORD;
PROCEDURE Coroutine1;
BEGIN
   LOOP
      WriteString('coroutine one');
      TRANSFER(coroutine1Id, coroutine2Id);
   END (* loop *);
END Coroutine1;
PROCEDURE Coroutine2 ;
VAR
   count : CARDINAL;
BEGIN
   count : = 0;
   LOOP
      WriteString ('  coroutine two');
      WriteLn;
      IF count=25 THEN
         TRANSFER(coroutine2Id, MainProgram);
      ELSE
         INC(count);
         TRANSFER(coroutine2Id, coroutine1Id)
      END (* if *);
   END (* loop *);
END Coroutine2;
BEGIN
   NEWPROCESS(Coroutine1, ADR(worksp1),
SIZE(worksp1), coroutine1Id);
   NEWPROCESS(Coroutine2, ADR(worksp2),
SIZE(worksp2), coroutine2Id);
   TRANSFER(MainProgram, coroutine1Id)
END CoroutinesExample.
```

The fact that the concurrent programming implementation is based on coroutines has led some commentators to state that Modula-2 cannot be used for real-time systems. It cannot be used directly, but it is possible to use the primitive coroutine operations to create a real-time executive as is described in Chapter 6.

5.11 INTERRUPTS AND DEVICE HANDLING

If full I/O device support is to be provided by a high-level language then the language must provide support for the handling of interrupts. This demands support for some minimum form of concurrent operation since an interrupt causes a

suspension of the running program (task) and the execution of some other code. There are two basic approaches in language design to doing this: one is to provide a set of low-level primitive operations which can be used either directly or to build higher-level constructs; the other is to provide a high-level set of primitive operations which must be used. Modula-2 is typical of the first method and Ada of the second. In the following sections Modula-2 is used to illustrate the various requirements.

Hardware interrupts can be handled from within a Modula-2 program. A device handling process can enable the external device and then suspend itself by a call to a procedure `IOTRANSFER`. This procedure is similar to `TRANSFER` but has an additional parameter which allows the hardware interrupt belonging to the device to be identified. When an interrupt occurs control is passed back to the device routine by a return from `IOTRANSFER`.

The procedure, `IOTRANSFER`, has the format

```
IOTRANSFER (VAR interruptHandler : PROCESS;
            interruptedProcess : PROCESS;
            interruptVector : CARDINAL)
```

The action of `IOTRANSFER` is to save the current status of `interruptHandler` and to resume execution of `interruptedProcess`, that is to wait for an interrupt. When an interrupt occurs the equivalent of

```
TRANSFER (interruptedProcess, interruptHandler)
```

occurs. A skeleton interrupt handler would thus take the form

```
BEGIN
   LOOP
...
   IOTRANSFER (interruptHandler, interruptedProcess,
interruptVector);
(* interrupt handler waits at this point for interrupt *)
...
   END Loop
END;
```

The interrupt handler code is placed inside the `LOOP ... END` construct, and is initiated by an explicit `TRANSFER` operation; it then waits for an interrupt at the `IOTRANSFER` statement. The first time this statement is executed control is returned to the initiating task. Subsequent executions will be after an interrupt has occurred and a return will be made to whichever task was interrupted.

An example of a Modula-2 program which uses the low-level facilities is given in Example 5.14. This program illustrates a further requirement for low-level support from the high-level language, namely the ability to handle interrupts. In the example shown `SuspendUntilInterrupt` is a high-level procedure provided by

the module Processes which is part of a real-time support library (further information about this is given in Chapter 6).

EXAMPLE 5.14

```
MODULE TermOut[4 (* interrupt priority of device *)];
FROM PROCESSES IMPORT
   SuspendUntilInterrupt;
EXPORT
   PutC;

CONST
   readyBit = 7;
   interruptEnableBit = 6;
   interruptVector = 64B;
VAR
   ttyReg[177564B] : BITSET;
   ttyBuf[177566B] :CHAR;
PROCEDURE PutC(c: CHAR);
BEGIN
   IF NOT(readyBit IN ttyReg) THEN
      INCL(ttyReg, InterruptEnableBit);
      (* high processor priority will fend off
      the interrupt until ...*)
      SuspendUntilInterrupt(interruptVector);
      EXCL(ttyReg, interruptEnableBit);
      END; (* IF *)
   ttyBuf := c;
END PutC;
BEGIN
END TermOut;
```

This example also shows how Modula-2 handles bit-level manipulation. It is possible to declare a register as of type BITSET and perform set operations on the register. The operators INCL and EXCL are respectively the operations of including a bit in the set, that is setting a bit in the register, and excluding a bit from the set, that is resetting a bit in the register.

5.12 CONCURRENCY

Wirth (1982) defined a standard module Processes which provides a higher-level mechanism than coroutines for concurrent programming. The module makes no assumption as to how the processes (tasks) will be implemented; in particular it does

not assume that the processes will be implemented on a single processor. If they are so implemented then the underlying mechanism is that of coroutines. The DEFINITION MODULE for Processes is as follows:

```
DEFINITION MODULE Processes ;
TYPE
SIGNAL;
    (* opaque type: variables of this type are used to
    provide synchronisation between processes. The
    variable must be initialised by a call to Init before use
    *)
PROCEDURE StartProcess(P:PROC;
    workSpaceSize:CARDINAL);
    (* start a new process. P is parameterless procedure
    which will form the process, workSpaceSize is the number
    of bytes of storage which will be allocated to the
    process
    *)
PROCEDURE Send(VAR s: SIGNAL);
    (* If no processes are waiting for s, then SEND has no
    effect. If some process is waiting for s then that
    process is given control and is allowed to proceed
    *)
PROCEDURE WAIT(VAR s: SIGNAL);
    (* The current process waits for the signal s. If at some
    later time a SEND(s) is issued by another process then
    this process will return from wait. Note if all
    processes are waiting the program terminates
    *)
PROCEDURE Awaited(s:SIGNAL):BOOLEAN;
    (* Test to see if process is waiting on s, if one or more
    processes are waiting then TRUE is returned
    *)
PROCEDURE Init(s: SIGNAL);
    (* The variable s is initialised, after initialisation
    Awaited(s) returns FALSE
    *)
END Processes.
```

Concurrency and multi-tasking will be dealt with in more detail in Chapter 6.

5.13 RUN-TIME SUPPORT

An important contributor both to the efficiency of implementation of a language and to the run-time security are the run-time support mechanisms that are provided by the particular implementation of the language. A problem with run-time support, however, is that security and efficiency frequently come into conflict. One way of trying to resolve the conflict is to provide optional run-time error checking and

trapping mechanisms. For test purposes the error traps are switched in but once the software has been tested the error traps are switched out and the software runs faster.

A typical example is the checking of array bounds. Many compilers allow the insertion of check code as an option which is selected when the program is compiled. This is an acceptable approach for standard programming: during the initial testing the array bound check is selected but once the program appears to function correctly it is omitted and hence the program runs faster. Although the facility is often used in this way for real-time systems it is not a reliable solution. (Why not?)

There are two ways to do the array checking. Consider an array of 20 numbers denoted by *array*[*i*] where *i* is an integer variable. For security the system must ensure that whenever *array*[*i*] is used the condition $1 \leqslant i \leqslant 20$ is satisfied. The first way of checking is to insert a check before every array access to ensure that the condition is not violated. An alternative technique, used by the more modern compilers, is to check the condition whenever an assignment is made to *i*. The use of the second method requires the co-operation of the programmer who must specify the permitted range of variable *i* when it is declared. Provided that the checking of the assignment of values to *i* is done then there is no need to check access to *array*[*i*]. A further reduction in run-time checking is also found with this technique in that a large number of the assignments to *i* can be tested during compilation. The second method relies on the language being strongly typed. For a secure, real-time system, the second method of array bound checking is obviously preferable as it allows checking to remain in the final version of the software without imposing large run-time overheads on the execution time.

For real-time systems the other important feature of the run-time support software is whether it allows the application software to intercept the error traps. This was discussed in the section above where we dealt with exception handling.

5.14 OVERVIEW OF REAL-TIME LANGUAGES

The best way to start an argument among a group of computer scientists, software engineers or systems engineers is to ask them which is the best language to use for writing software. Rational arguments about the merits and demerits of any particular language are likely to be submerged and lost in a sea of prejudice.

Since 1970 high-level languages for the programming and construction of real-time systems have become widely available. Early languages include: CORAL (Woodward *et al.*, 1970) and RTL/2 (Barnes, 1976) as well as modifications to FORTRAN and BASIC. More recently the interest in concurrency and multi-processing has resulted in many languages with the potential for use with real-time systems. These include Ada (see Young, 1982; Burns and Wellings, 1990), ARGUS (Liskov and Scheifler, 1983), CONIC (Kramer *et al.*, 1983), CSP (Hoare, 1978), CUTLASS (CEGB, see Bennett and Linkens, 1984), FORTH (Brodie, 1986),

Modula-2 (Wirth, 1982), occam (Burns, 1988), PEARL and SR (Andrews, 1981, 1982). The most widely used language for programming embedded real-time systems is probably C. These languages range from comprehensive, general purpose languages such as Ada and Modula-2 to more restrictive, special purpose languages such as CONIC and CUTLASS. The advantage of the general purpose languages is that they provide flexibility that can support the building of virtual machines; for example, Budgen (1985) has shown how Modula-2 can be used to create a MASCOT virtual machine. However, general purpose, flexible languages can be too complex and permit operations that can compromise the security of a real-time system.

A language suitable for programming real-time and distributed systems must have all the characteristics of a good, modern, non-real-time language; that is, it should have a clean syntax, a rational procedure for declarations, initialisation and typing of variables, simple and consistent control structures, clear scope and visibility rules, and should provide support for modular construction. The additions required for real-time use include support for concurrency or multi-tasking and mechanisms to permit access to the basic computer functions (usually referred to as low-level constructs).

Modula-2 and Ada represent two very different approaches to language design: Modula-2 is based on a small set of mandatory facilities which can be extended using library modules, whereas in Ada all facilities considered necessary are mandatory and extensions are prohibited. The benefits of Ada are that it is standardised and hence application software should be highly portable; the disadvantages are size and complexity. The reverse is true of Modula-2: although the core is standardised there are many different sets of libraries supplied by different compiler implementers. For example, input and output routines are not provided as part of the language but have to be supplied as a library module and consequently the range and type of routines differ according to the supplier. The advantages of the Modula-2 approach are that the language is kept simple and the user can choose to add the features necessary and appropriate to the type of application. The major disadvantage is the loss of standardisation and hence portability.

5.15 APPLICATION-ORIENTED SOFTWARE

A large number of software packages and languages have been developed with the intention of providing a means by which the end user can easily write or modify the software for a particular problem. The major reason for wanting the end user, rather than a specialist programmer, to write the software is to avoid the communication problem. A large proportion of 'errors' in a system arise from a misunderstanding of the operation or structure of a plant by the programmer. This is not always the programmer's fault; often the engineers and managers responsible for the plant do not communicate their requirements clearly and precisely.

The misunderstanding can largely be avoided if the engineers responsible for the plant can themselves write the software.

The engineers are not, however, expected to be specialists in computer programming and hence they must be provided with simple programming tools which reflect the particular application. Because for a given type of application, for example process control, the range of facilities required is small and predictable, it is not too difficult to devise special application software.

Three main approaches are used:

1. table-driven;
2. block-structured; and
3. specialised languages.

These are considered in more detail in the following sections.

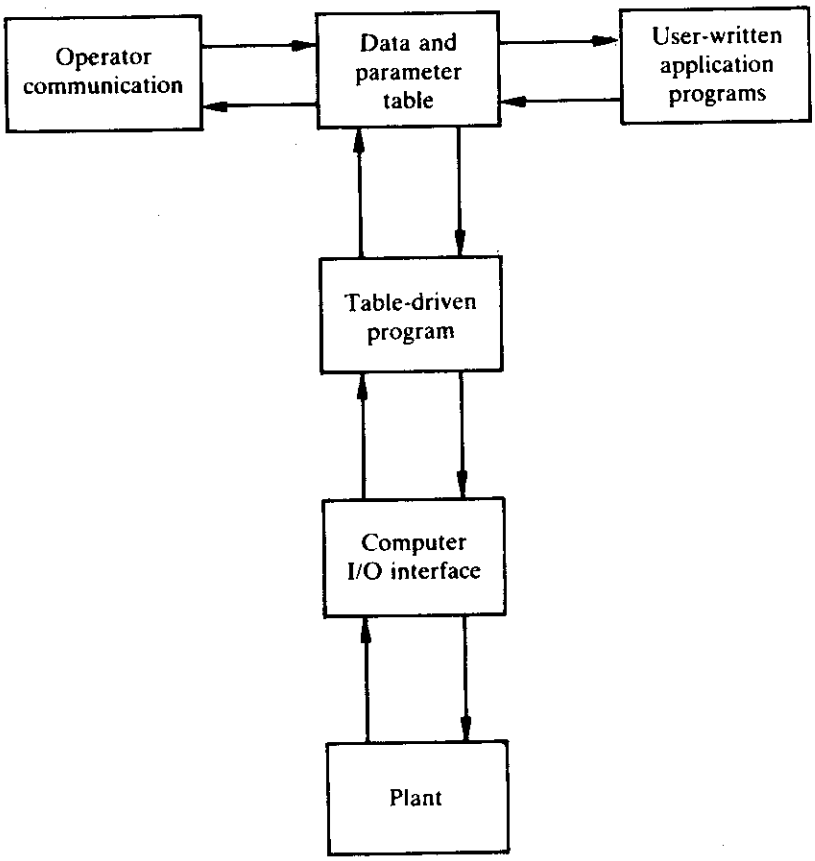

Figure 5.2 Simple table-driven system.

5.15.1 Table-driven Approach

The table-driven approach arose out of systems programmed in assembly code. It was soon realised that many code segments were used again and again in different applications as well as in the particular application. For example, the code segment used for PID control need only be written once if it contains only pure code and all references to parameters and data are made indirectly.

A simple table-driven system is illustrated in Figure 5.2. As well as allowing the control program to communicate with the data and parameter table, provision is also made for the operator to obtain information from the table (and in some instances to change values in the table). In addition some systems allow the user to write application programs in a normal computer language (usually FORTRAN or BASIC) which can interact with the table-driven software. The actual table-driven program is supplied as part of the system and cannot be modified by the user.

An alternative approach is shown in Figure 5.3 in which a database manager program is inserted between the data table and all users; access to the data table is now controlled by the database manager. The use of a database manager to control access places additional overheads on the system and can slow it down. It has the advantage of improving the security of the system in that checks can be built into it, for example to limit the items which can be changed from the operator's console, or to restrict the access of user-written application programs to certain areas of the data table. Typically a database manager program would provide, by means of a password or a key-operated switch on the console, different access rights to the operator and the plant engineer. .

A crucial factor in the usefulness of table-driven software is the method of setting up and modifying the tables. There are three main methods:

1. direct entry into the data tables;
2. use of language DATA statements; and
3. the filling in of forms.

In the simplest systems the data has to be entered into specified memory locations, but this method is rarely used nowadays. It is more normal for the entry program to allow names to be used for the locations. Thus, for example, in a system with eight analog inputs the conversion values for each channel may be set by entering from a keyboard statements such as

```
ANCONV (1) = 950.2
ANCONV (6) = 0.328
```

which would set the conversion factors for the signals coming in on channels 1 and 6. On some systems it is possible to specify signal names rather than use the table index numbers. For example,

```
ANINP (1) = FEEDFL
ANINP (2) = FEEDTP
```

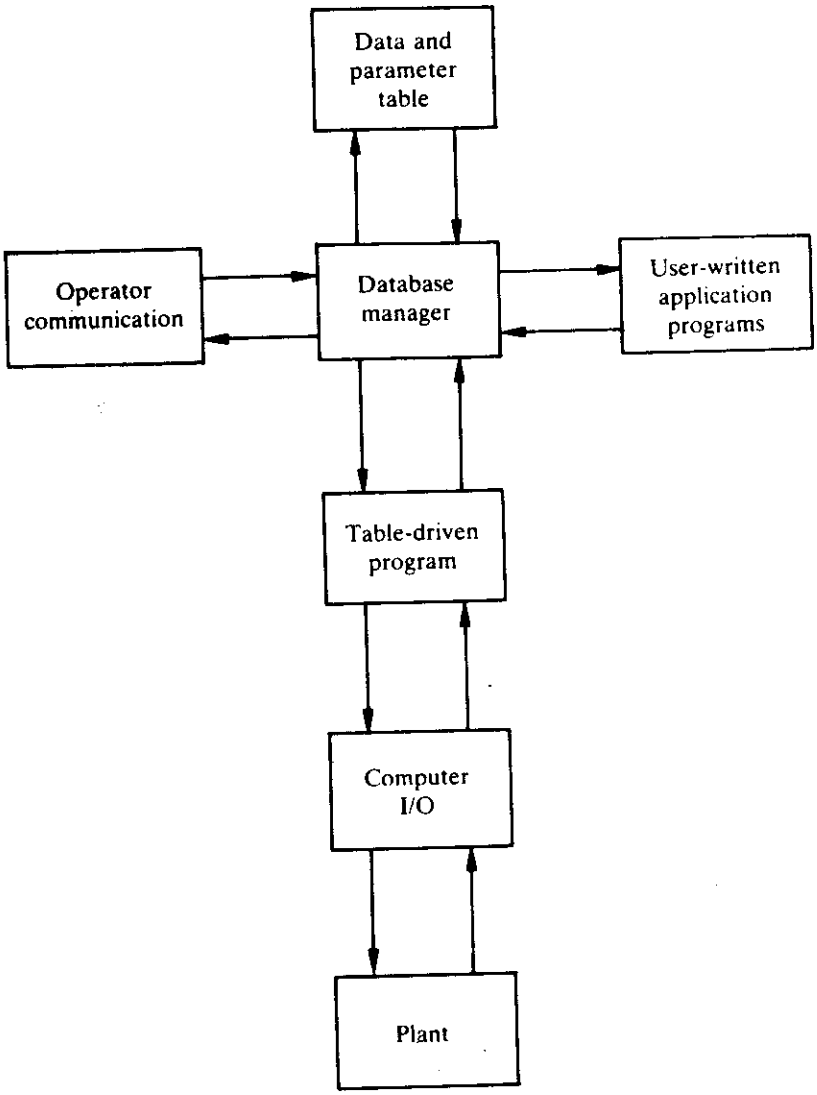

Figure 5.3 Use of table-driven system with database manager.

would specify that the signals on analog inputs 1 and 2 are to be known as FEEDFL (representing, say, feedwater flow rate) and FEEDTP (representing feedwater temperature). Once such names have been declared they can be used elsewhere to reference the particular signals. This is a feature which is useful when user-written application software is used, say, to produce specialised displays.

The use of language statements to set up the data tables is useful in large systems for initially setting up the system. It is normally supplemented by allowing changes

to be made, from the operator keyboard, either of all entries or of selected entries. It has the advantage that meaningful names can be given to the entries at initial setting-up time; subsequent modifications are then made using the plant names rather than the table entry index.

The most commonly used approach is to provide a form into which data relating to the system is entered. In the early systems data was transcribed onto punched cards and then read into the system; now the form is presented on the screen and the data entered directly. Usually some type of 'forms' processor is provided which checks data for consistency as it is being entered and prompts for any missing data (Figure 5.4).

Table-driven software is very simple to use; it is, however, restrictive in that maximum numbers for each type of control (loop, input and output) have to be inserted when the system is configured and these cannot be changed by the user. Because of this a similar, but more flexible, approach based on the use of function blocks has been developed.

5.15.2 Block-structured Software

The software supplied in a block-structured system consists of a library of function

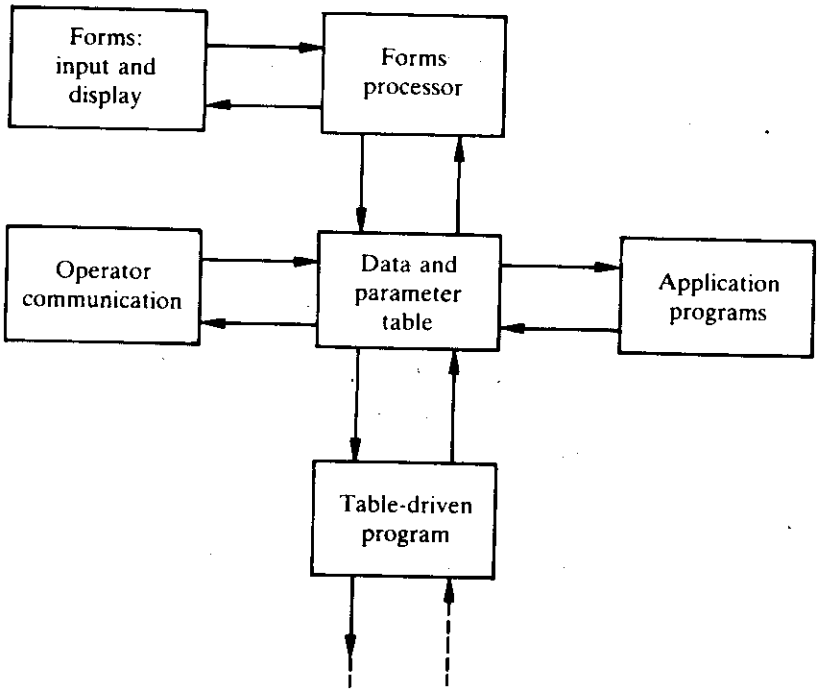

Figure 5.4 Forms processor for table-driven system.

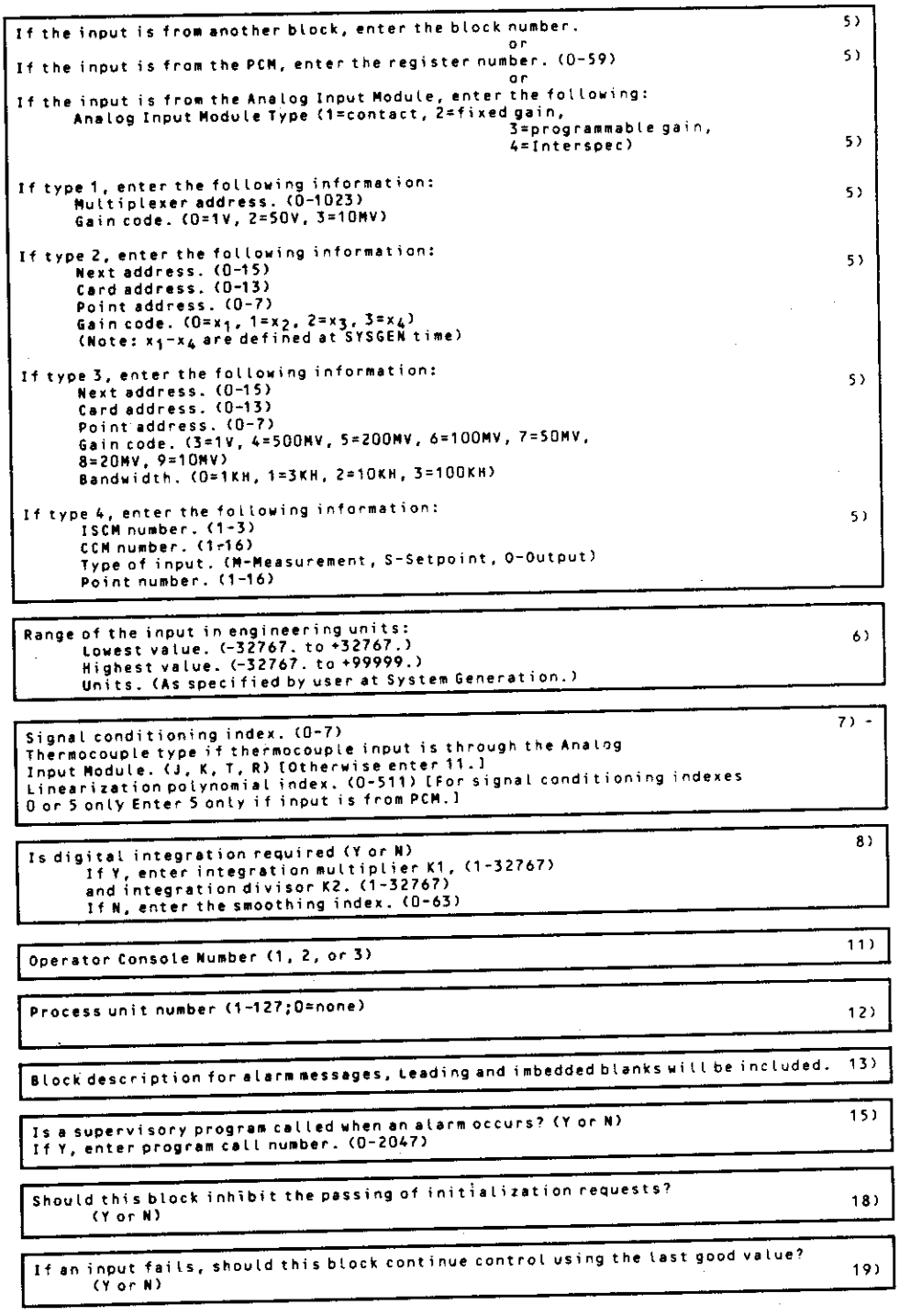

Figure 5.5 Typical layout sheet for forms entry (adapted from Mellichamp, *Real-time Computing*, Van Nostrand Reinhold (1983)).

blocks (scanner routines, PID control, output routines, arithmetic functions, scaling blocks, alarm routines and display routines), a range of supervisory programs and programs for manipulating the block functions. The engineer programs a control scheme by connecting together the various function blocks which he or she requires and entering the parameters for each block. This is typically done using a VDU with a graphical representation of the block connections. An example of the information which has to be supplied is shown in Figure 5.5.

The block-structured approach is used in a wide range of systems, from large process control systems with several hundred loops and multiple operator display stations to simple programmable controllers used for sequence control. The range of controllers available is shown in Figure 5.6.

5.15.3 Application Languages

Application languages range from simple interpreters which allow for interaction with table-driven or functional block systems to complex high-level languages which have to be compiled. The major feature of such languages is that they provide a syntax which reflects the nature of the application. In the following section a large, complex application language is described in outline.

Software functions:	*Basic*	*Advanced*	*Process*
	Boolean	Block transfer	Signalling
	Timers	Jump	Monitoring
	Counters	File	PID control
	Data move	Shift registers	Communication
	Comparison	Sequencers	Logging
	Arithmetic	Floating point	Display
Hardware functions:	*Small PCs*		*Large PCs*
Inputs	16		4096
Outputs	16		4096
Timers	8		256
Counters	8		256
User program	2K		48K
Cycle time (per 1K)	100 ms		1 ms

Figure 5.6 Block functions in programmable controllers.

5.16 CUTLASS

CUTLASS is a high-level language which is oriented towards use by the engineer rather than the professional programmer. It has been developed by the UK Central Electricity Generating Board with the aim of enabling engineering staff to develop, modify and maintain application software independently of professional software support staff.

The major requirements which CUTLASS had to meet are:

1. It should be suitable for a wide range of applications within power stations. All applications packages should operate within the same general framework so that future developments can easily be incorporated without rendering obsolete previous work. To achieve these aims the language was developed in the form of a number of compatible subsets which cover the following functions:
   (a) modulating control;
   (b) data logging;
   (c) data analysis;
   (d) sequence control;
   (e) alarm handling;
   (f) visual display; and
   (g) history recording.
   These subsets operate within a framework which provides:
   (a) a real-time executive – TOPSY;
   (b) communications network management;
   (c) support facilities; and
   (d) integrated I/O and file handling.
2. The software should have a high degree of independence from any particular computer type. This is a particularly important requirement for software which is expected to have a long lifetime – 20 to 30 years – during which period the actual control computers may have to be replaced and it is important that this can be done without having either to use obsolete technology or to incur high program modification costs. In order to achieve this the CUTLASS language has been written in CORAL.
3. The software should be simple and safe to use so that engineers based in the power station can produce and modify programs. This has been achieved by providing within the language an extensive range of subroutines and by hiding the detailed operational and security features from the user.

5.16 1 General Features of CUTLASS

The basic unit of a CUTLASS program is a SCHEME which is an independently

compilable unit. A SCHEME is defined as

```
<subset type> SCHEME <name>
GLOBAL <data>
COMMON <data>
TASK <qualifying data>
...
TASK <qualifying data>
ENDSCHEME
```

A scheme may contain any number of tasks and a program may contain any number of schemes. The schemes may run on different computers in a distributed network. The software is developed on a host machine and downloaded to the target machine(s); a typical system is shown in Figure 5.7. During the operation of the overall system the host may remain connected to the target systems.

In addition to the schemes generated by the programmer, support software – including the TOPSY executive – is loaded into the target machine. The schemes can be enabled and disabled by the user from a keyboard connected to the target machine or from the host machine or from within a supervisory task.

As an example of the division of the software for an application into schemes and tasks, consider the hot-air blower system described in Chapter 1. For this system it is required that the temperature measurement be filtered by taking a running average over four samples at 10 ms intervals. The actual control is to be a PID

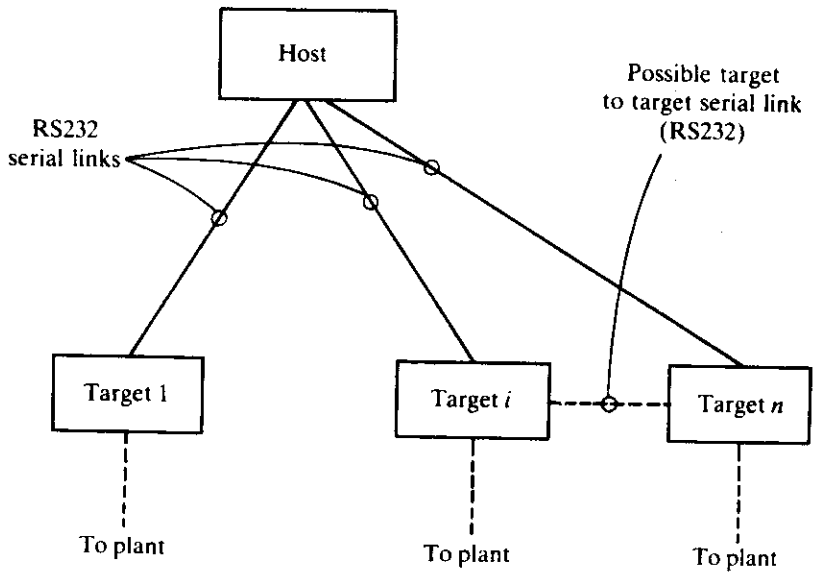

Figure 5.7 CUTLASS host–target configuration.

controller running at 40 ms intervals. The display of the input, output and error is to be updated at 5 s intervals.

EXAMPLE 5.15

Scheme and Task Outline

```
1. TEMPERATURE CONTROL SCHEME
   1A TASK FILTER RUN EVERY 10 MSECS
      Reads temperature from heater and computes the
      running average.
   1B TASK TEMPCON RUN EVERY 40 MSECS
      Uses the average temperature obtained from
      filter and computes, using a PID control
      algorithm, the output for the heater.
2. DISPLAY SCHEME
   2A TASK DISPLAY UPDATE RUN EVERY 5 SECS
      Update the display with the values of
      temperature, error and heater output.
```

A possible arrangement is shown in Example 5.15 in which the program is split into two schemes: `TEMPCON`, used for the control of the temperature in which there are two tasks (`FILTER` and `TMPCON`), and `DISPLAY` in which there is one task, `UPDATE`.

The timing of the tasks is controlled by TOPSY and the timing requirement is specified as part of the task qualifying data. The syntax is

```
TASK <identifier> PRIORITY = <priority level> RUN
EVERY <integer> <time interval>
identifier = name
priority level = 1..250
time interval = (MSECS:SECS:MINS:HOURS:DAYS)
```

The outline of the `TEMPCON` scheme would be as follows:

```
DDC SCHEME TEMPCON
;
; variable declarations placed here
;
TASK FILTER PRIORITY=240 RUN EVERY 10 MSEC
;
; task local declarations placed here
;
START
;
; task body
;
ENDTASK
```

```
;
TASK TMPCON PRIORITY=225 RUN EVERY 40 MSECS
;
; task declarations
;
START
;
; task body
;
ENDTASK
ENDSCHEME
```

As is seen from the above statements, the TOPSY executive supports CYCLIC- or CLOCK-based tasks; it also supports a `DELAY` timing function, but the `DELAY` function can be used only within the `GEN` subject, and then only for tasks which are non-repetitive.

5.16.2 Data Typing and Bad Data

As in Pascal, CUTLASS requires all variables to be associated with a data type when they are declared. The types supported are: logical, integer, real, text, string, array and record.

An additional feature which is used in CUTLASS is that variables can be flagged as good or bad. The concept of 'bad' data is useful in an environment in which a large number of values are derived from plant measurements. Instruments on the plant can become faulty and supply data which is incorrect. It is often easy to detect when an instrument is supplying false information as when, for example, typical plant transducers supply signals in the range 4–20 mA; if the signal falls below 4 mA it is an indication that the transducer is faulty. The difficulty arises in transmitting through the system an indication that the reading from that particular instrument is faulty.

The ultra-safe approach would be to put the whole scheme into manual mode and allow the operator to take over until the instrument is repaired or replaced. In many circumstances such extreme action may be unnecessary; the instrument may only be supplying an operator display, or there may be an alternative measurement available.

The approach adopted is to mark the data value as bad and allow the bad data to propagate through the system, action, if necessary, being taken elsewhere in the system. The rules used in the propagation of bad data produce a result which is marked as bad if any of the operands in an arithmetic operation are bad. For example,

```
A:= B + C
E:= D/C
```

would result in E being marked as bad if B were bad. The rules for propagation through logic operations are slightly more complex in that a logic variable which is bad is treated as a 'don't know' condition. For example,

```
MODEA := (TEMP > 30.0) OR FANRUN
```

would produce a valid result if TEMP was bad and FANRUN was true, but a bad value if FANRUN was false. An advantage of the bad data flagging is that large sections of the code can be written on the assumption that the data is good: if it is not good then the fact that it is bad will be automatically propagated through the system to the point at which action must be taken.

5.16.3 Language Subsets

The language is divided into four subsets which share some common features but which perform essentially different functions. A scheme may use only instructions from one subset and the subset being used is declared as part of the scheme heading. The subsets are:

1. GEN A general purpose subset which is used to support text input and output; a scheme which uses only the features common to all subsets is also referred to as a GEN scheme.
2. DDC This subset provides the set of instructions used for direct digital control algorithms.
3. SEQ A subset which supports the construction of sequential control algorithms; there is some restriction on the use of the common language features within an SEQ scheme.
4. VDU A subset which provides graphical and text support for a range of VDUs including both colour and monochrome devices.

Some indication of the power of the DDC subset can be obtained from the list of instructions supported which is shown in Figure 5.8.

5.16.4 Scope and Visibility

The CUTLASS system has very simple scope and visibility rules. Objects declared within a task are local to that task: the scope extends throughout the task and the object is visible within the body of the task. The object is not visible within a subroutine called by the task.

Variables may be shared between tasks by declaring them in a COMMON block in the scheme declarations. The compiler restricts variables declared within the COMMON block to the sharing of information between tasks and prevents their use

Function Output

Non-history-dependent functions

AVE	average of all good inputs
EAVE	average of all inputs; if any input is bad then output is bad
MIN	minimum of good inputs
AMIN	absolute minimum of good inputs
EMIN	minimum of inputs; if any input is bad then output is bad
EAMIN	absolute minimum of inputs; if any is bad then output is bad
MAX	maximum of good inputs
AMAX	absolute maximum of good inputs
EMAX	maximum of inputs; if any is bad then output is bad
EAMAX	absolute maximum of inputs; if any bad then output bad
INHIBIT	depends on condition of inhibit raise and lower flags
INCS	converts integer value to incremental units
LIMIT	limit range of real variable
BRAND	deadband function

History-dependent functions

BUCKET	counts increments supplied and used
FLIPFLOP	logical flipflop triggered at twice the task repeat interval
CHANGE	logic function A_n=A_n FOR A_{n-1}
RISING	logic function A_n=A_n AND HCT (A_{n-1})
FALLING	logic function A_n NOT (A_n) AND A_{n-1}
ACC	sets accumulator to an initial condition
ACCLIMIT	sets hard limits on an accumulator

History- and time-dependent functions

INT	approximate integrator
DELTA	approximate differentiator
FIRST	first-order filter
IMCPID	incremental PID control algorithm, includes roll-off filter
PID	absolute PID algorithm including roll-off filter
EFORM	controller expressed as a polynomial in z
RAMP	limits rate of change of a variable.

Figure 5.8 Example of a CUTLASS DDC scheme (reproduced with permission from Bennett and Linkens, *Real-time Computer Control*, Peter Peregrinus (1984)).

as local variables as well by enforcing the following rules:

1. Only one task may write to a COMMON variable. For this purpose arrays are treated as indivisible objects so that if a task writes to an individual element of an array then all the elements become write only for that task.
2. Tasks are not allowed read access to variables to which they write.

Communication between tasks in different schemes is by means of GLOBAL variables. These variables can be created only by use of a special utility which is run

by a privileged user. GLOBAL variables are owned by the user who created them and may be removed only by the owner. The same access rules as for common variables are applied to GLOBALs with the added restriction that only a task belonging to the owner of a GLOBAL variable may write to that variable.

The rules are relaxed for tasks which belong to users with privileged status such that:

1. GLOBALs may be declared as private to the user.
2. GLOBALs may be written to by more than one task (subject to the ownership rule).
3. Tasks may have both read and write access to the variable.

5.16.5 Summary

CUTLASS represents an attempt to resolve many of the problems which arise in real-time systems. It does not purport to be a general purpose language and hence the solutions which are adopted are not always particularly elegant. The emphasis in the system is on enabling inexperienced computer users to write reliable, secure, programs. In this the language is successful.

However, the overall system is complex and setting up and maintaining it requires the support of experienced computer staff. This does not detract from one of the aims of the system, which was to allow engineers to write their own application programs; this has been achieved. It would be difficult, however, to operate the CUTLASS system in an organisation which did not have expert computer staff.

Although the language syntax is an improvement on CORAL 66, readability is not high in that identifiers are restricted to nine characters and must be in upper case. A further restriction is that user subroutine libraries cannot be developed; all user (as opposed to system) subroutines must be included within the TASK declaration area in source code form. The language supports constants in a useful form, variables can be preset and such variables are then treated as read-only variables.

5.17 A NOTE ON BASIC

BASIC (Beginners' All-purpose Symbolic Instruction Code) was developed as a language which would provide a reasonably powerful range of facilities but which would be easy for the novice to learn and use. In particular the use of an interpretative mode of implementation was intended to make the writing, running and debugging of programs as quick and easy as possible. There is little doubt that BASIC has achieved the designer's aims and its widespread availability on microprocessor systems has also contributed to its use.

There are now many versions of BASIC with numerous extensions to the original language, some of which make them suitable for certain types of real-time systems, for example experimental, prototype and development work where the speed and ease with which BASIC programs can be written and debugged is a great advantage. These BASICs can also be used for small systems where safety is not a major issue.

The simplest extensions to BASIC which provide support for embedded system use are the provisions of low-level access mechanisms. Other extensions include functions for handling input and output to analog-to-digital and digital-to-analog converters (see Mellichamp, 1983), event handling (interrupts) and multi-tasking.

The simplest form of event handling allows the use of statements of the form

```
ON EVENT GOSUB <n>
```

where <n> is a specified line number. This may be extended in some BASICs to allow several different events:

```
ON INTERRUPT0 GOSUB <n1>
ON INTERRUPT1 GOSUB <n2>
ON TIMEOUT GOSUB <n3>
```

The GOSUB is used so that a RETURN statement can be used to indicate the end of the section of code for the particular event. Further extensions provide support for multiple tasks.

Each task body is indicate by statements

```
TASK <name>
...
EXIT
```

which are used to bracket the statements which form the task. The task remains dormant (in terms of the task states described in Chapter 6, existent but not active) until an ENABLE statement is executed:

```
ENABLE <name> EVERY 10cs
```

or

```
ENABLE <name> WHEN event
```

The tasks can be disabled by the statement DISABLE <name>.

The major advantage of BASIC is the ease with which the language can be learnt. If well-trained programmers with experience of other languages are not available then BASIC may be a very sound choice and result in better, more secure software than could be achieved by using an inherently more secure language.

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