Appendix A

Modula-2 for Pascal Programmers

A.1 Introduction

Modula-2 is a programming language invented by Niklaus Wirth, who was also the inventor of Pascal. Thus, it inherits many of the features of Pascal, and from one point of view it can be viewed as an upgraded Pascal. Whether or not one agrees with this last statement, it is certainly true that Modula-2 is a particularly easy language to learn for those who know Pascal.

For the sake of brevity, it is assumed in this Appendix that the reader is already familiar with Pascal. Programmers experienced with some other language, for example Fortran, might also find the explanations here sufficient, although this is less certain. Readers with little or no computer programming experience would be better off consulting an introductory text on Modula-2.

A.2 Lexical Elements

Modula-2 uses essentially the same character set as Pascal, and the same rules about things like space, tab, and end-of-line characters. (For example, a statement may be be spread over several lines, the only restriction being that a line break may not come in the middle of a multi-character symbol such as an identifier or, for example, the two-character symbol ":=".)

One major difference is that the *case* of a letter is significant. Both upper case and lower case letters may be used in forming an identifier, but you have to be consistent. For example, "Abc" and "abc" are two different identifiers. A commonly used convention is to use upper case to make certain letters stand out, as for example in "ThisIsALongName". Most Modula-2 compilers will allow the underscore character to be used inside an identifier (example: "current_value"), but on the whole the upper case approach seems to lead to more readable programs.

There is no restriction on the length of an identifier, but many compilers fail to check the trailing part of very long identifiers. For example, the compiler used in preparing the software for this book considers two identifiers to be equivalent if their first 31 characters are equal.

Upper case is used for all the keywords, all of the built-in function names, and some (not all) of the common library procedures. For example, in the statement

```
REPEAT
    Refine (k, result); INC (count)
UNTIL (count = 20) OR (result < epsilon)
```

the words REPEAT, INC, UNTIL, and OR must all be in upper case. This convention is already used by some Pascal programmers; and it soon becomes second nature to those who have not been used to it.

Comments start with (* and finish with *). (The braces {} are used for sets, therefore they are not available as comment delimiters.) Unlike Pascal, Modula-2 allows comments to be nested. For example, one may have the comment

```
(* The symbols (* and *) are used as comment delimiters *)
```

This particular example is a rather frivolous use of nested comments. A more significant use of nesting arises when you want to "comment out" a section of code while testing a program. In that case, you do not have to worry about whether the "commented out" part itself contains comments.

Character strings may use the single quote (') or the double quote (") as the delimiter. You must use the same delimiter at the start and end of the string. For example,

```
"a"
'b'
'This is a string'
"This is another string, it's a bit longer"
```

With this convention, there is no need for a special arrangement for a quote mark inside a string.

A.3 Statements

As in Pascal, the semicolon ";" is used to terminate declarations and procedure bodies, and to separate adjacent statements (whether or not they occur on the same line). Modula-2 is marginally more flexible than Pascal about redundant semicolons, so that for example it permits you to put a semicolon before the word ELSE in an IF statement (this is illegal in Pascal).

The concept of a "compound statement" − that is, a sequence of statements bracketed by BEGIN and END − does not exist in Modula-2. It was needed in Pascal in order to allow the body of, for example, a WHILE−DO loop to be arbitrarily long. In Modula-2, this effect is obtained by requiring an explicit terminating keyword for every structured statement; that is, for everything except assignments and procedure calls. To illustrate, consider the REPEAT-UNTIL statement, which exists in both Modula-2 and Pascal. This has the form

```
REPEAT
    group of statements
UNTIL BooleanExpression
```

and there is no need for a bracketing BEGIN-END around the group of statements because the UNTIL clearly shows where the group finishes. In Modula-2, this principle is extended to all structured statements. In the majority of cases, the terminator is the keyword END. This will be seen when the individual statement types are described.

The keyword BEGIN exists in Modula-2, but it is used only to mark the start of a procedure body, or to mark the start of the "main program" part of a module.

A.3.1 Assignment statements

An assignment statement has the form

```
variable := expression
```

exactly as in Pascal. In fact, the only obvious difference between a Pascal assignment and a Modula-2 assignment is that Modula-2 has a slightly different list of built-in functions. Occasionally, one needs to read the rules about type compatibility and assignment compatibility very carefully to determine what is legal in terms of mixing types, but the average programmer is unlikely to notice any difference here.

A.3.2 Procedure and function calls

These are the same as in Pascal, except for the special case of a function with no parameters. If F1 is a function with no parameters, then to call it you must write F1(). The parentheses are essential, even when there is nothing inside them.

A.3.3 The IF statement

The most general form of the IF statement is

```
IF BooleanExpression
THEN
    group of statements
ELSIF AnotherBooleanExpression THEN
    group of statements
ELSIF ...
    (* As many ELSIF parts as you like *)
ELSE
    group of statements
END (*IF*)
```

The (*IF*) on the last line is my comment, not part of the language. (I use it because I believe that the terminator should have been ENDIF, but unfortunately Wirth and I don't have precisely the same ideas about language design.) The terminating END is the important difference between Modula-2 and Pascal. Be careful here: it is needed even in simple cases like

```
IF a > b THEN
    x := 1
END (*IF*)
```

The ELSIF and ELSE parts are optional, but the END is compulsory. Notice that ELSIF is written as a single word. (If it had been written as ELSE IF, then we would have had a nested IF statement, and an extra END would have been needed.)

A.3.4 The CASE statement

An example of the CASE statement is

```
CASE ch OF
    "a":        y := 5;
                z := x - y;
    |
    "b".."f":   z := Lookup (ch);
    |
    "g","m":    x := 1; y := 2;
                INC (z);
    |
    ELSE
                ErrorFlag := TRUE;
END (*CASE*);
```

Notice that

(a) As with the IF statement, the word END marks the end of the statement;

(b) The character "|" is used to separate the cases;

(c) Because of the explicit "|" separator, there is no limit to the number of statements in each section;

(d) An (optional) ELSE part may be included, to be executed when none of the other cases applies;

(e) There may be several labels on an alternative, specified as a list separated by commas, where each element of the list is either a simple value or a range in the form low..high;

(f) The semicolon before the "|" is optional, and the "|" before ELSE is optional.

A.3.5 Loops

There are four ways of specifying a loop in Modula-2, and three of these are almost identical to Pascal loops. The REPEAT-UNTIL loop, already mentioned, is exactly the same as in Pascal. This is the only structured statement which does not finish with an END. The WHILE-DO form of loop is written as

```
WHILE BooleanExpression DO
    group of statements
END (*WHILE*)
```

and the FOR loop has the form

```
FOR variable := expr TO expr BY expr DO
    group of statements
END (*FOR*)
```

In this construct, the BY part is optional, and is omitted in the most common case where a step of $+1$ is desired. The expression after BY may be negative, therefore there is no need for the DOWNTO alternative which exists in Pascal. The control variable should not be a formal parameter, an imported variable, or a component of a structured variable.

In addition to these, there is a general loop construct which has the form

```
LOOP
    group of statements
END (*LOOP*)
```

In the absence of any further specification, this is an endless loop, but in fact there is a way to leave the loop. The EXIT statement, which can be used in the loop anywhere a statement is legal, causes a jump to the statement just after the end of the loop. In the case of nested loops, EXIT leaves the innermost LOOP statement in which it appears. An EXIT which is not inside a LOOP statement is of course meaningless, and any compiler would flag this as an error. There can be several loop exits, as the following example illustrates.

```
LOOP
    Read (ch);
    IF ch = ControlZ THEN
        EXIT (*LOOP*)
    END (*IF*);
    AddToTable (ch); INC (count);
    IF count = MaxCount THEN
        EXIT (*LOOP*)
    END (*IF*)
END (*LOOP*)
```

NOTE: The EXIT statement may *not* be used to leave a FOR loop, a WHILE-DO loop, or a REPEAT-UNTIL loop.

Strictly speaking, the WHILE-DO and REPEAT-UNTIL forms of loop are unnecessary, since one can always do the same thing with the LOOP statement. These constructs are provided because they are often superior, in terms of clarity, to the pure LOOP statement.

A.3.6 The WITH statement

This is the same as in Pascal, except that a terminating END is needed. That is, the statement has the form

```
WITH RecordVariable DO
    group of statements
END (*WITH*)
```

Some compilers allow more than one variable to be specified after the WITH (for example, “WITH a,b DO”), but this is a nonstandard extension.

A.4 Procedures and Functions

The main visible difference between a Modula-2 procedure and a Pascal procedure is that, in Modula-2, the procedure name must be repeated after the final END. An example is

```
PROCEDURE Proc1 (x: INTEGER; VAR y: CHAR);

    VAR z: BOOLEAN;

    BEGIN
        the body of the procedure;
    END Proc1;
```

The usual way to return from a procedure is by “falling out the bottom”. However, there is a RETURN statement which forces a premature return.

The keyword PROCEDURE is also used to introduce a function. (Unlike Pascal, there is no separate “function” keyword.) The differences between a procedure and a function are:

(a) The type of a function result must be given in the header, just after the parameter list. This is the same as in Pascal.

(b) A RETURN statement is compulsory in the function body. The word RETURN is followed by an expression giving the value to be returned as the result. A function must not return by falling out of the bottom of its code.

As an example, here is a function to find the greater of two real numbers.

```
PROCEDURE Maximum (a, b: REAL): REAL;

    BEGIN
        IF a > b THEN RETURN a ELSE RETURN b
        END (*IF*)
    END Maximum;
```

Some Modula-2 compilers retain the Pascal rule which states that the result type of a function must not be a structured type (that is, that it may not be something like an array or a record). In most of the newer compilers there is no restriction on the type of the result.

When a function has no parameters, the parentheses around the (empty) parameter list are still required. For example,

```
PROCEDURE Random (): REAL;
```

This function would be called with a statement like

```
    x := Random();
```

There is a special provision for variable-length arrays in the parameter list of a procedure (or function) declaration. For example, the procedure heading might be

```
    PROCEDURE InnerProduct (A, B: ARRAY OF REAL): REAL;
```

where there is nothing to indicate the size of arrays A and B. Inside the procedure body, the lower subscript is assumed to be 0, and the upper subscript can be obtained with the built-in function HIGH. That is, the elements of A are A[0] up to A[HIGH(A)]. This facility is available only for procedure parameters, it works only for singly-subscripted variables, and iterated forms like ARRAY OF ARRAY OF ... are not supported.

When a procedure is declared FORWARD, the full parameter list must be given both in the forward declaration and in the actual procedure heading (and they must agree). (In Pascal, the parameter list is omitted from the second declaration, which can be a nuisance when one is trying to read a program listing.) Some compilers permit forward calls even without the FORWARD declaration.

A.5 Constants, Variables, and Types

The declarations of constants, variables, and types is much the same as in Pascal, although there are some subtle differences. One major difference is that declarations can be made in any order; that is, one can have a VAR declaration, followed by a TYPE declaration, followed by another VAR declaration, and so on. (In fact, CONST, TYPE, and VAR declarations are allowed even between procedure declarations; but one generally avoids doing that, for the sake of clarity.) Forward references should be avoided, except in the special case (also allowed in Pascal) where a pointer type is being declared, and the type to which it points is not yet defined.

The predefined types are BOOLEAN, CHAR, INTEGER, CARDINAL, REAL, and BITSET. A CARDINAL is an unsigned integer, and a BITSET is a set of small cardinals. In principle, BITSET variables can be used to manipulate the bits within a word. For example, if B is a variable of type BITSET, then the assignment "B := {3,6,7}" sets bits 3, 6, and 7 of B. In practice this feature is not very useful, because (a) the width of B varies from one implementation to another, and (b) compiler writers cannot agree on how to number the bits (is bit 0 the leftmost bit, or the rightmost?).

As a nonstandard extension, some compilers support the "double precision" types LONGINT, LONGCARD, and LONGREAL, and the "short" types SHORTINT and SHORTCARD. In addition, there is a special module SYSTEM which exports the types BYTE, WORD, ADDRESS, and possibly LONGWORD. Anything exported by SYSTEM must be treated as non-portable, and used only in cases where low-level machine-dependent programming is justified; for example, when writing low-level device driver software.

Enumeration types and subranges are supported, as in Pascal, but there is a slight difference in the way in which subranges are written. Consider the following example.

```
    TYPE
        Day = (Monday, Tuesday, Wednesday, Thursday,
                        Friday, Saturday, Sunday);
        Weekday = [Monday..Friday];
        Digits = ["0".."9"];
```

As the example shows, square brackets must be used in defining a subrange type. This leads to a slight difference, as compared to Pascal, in the way in which arrays are declared. Some examples are:

```
    TYPE row = [0..24]; column = [0..79];

    VAR CurrentRow: ARRAY column OF CHAR;
        ScreenImage: ARRAY row, column OF CHAR;
        FilterCoeffs: ARRAY [1..10],[1..10] OF REAL;
```

This is almost like Pascal, but the brackets are considered to be part of the subrange definitions rather than part of the array declarations. Note particularly that this makes a difference when an array has more than one subscript. However, the executable part of the program still uses multiple subscripts in the form FilterCoeffs[i,j].

Record declarations are the same as in Pascal, except for differences in any variant parts which might be present. For example,

```
    TYPE Person =   RECORD
                        name: NameType;
                        sex: (Male, Female);
                        age: CARDINAL;
                        CASE employed: BOOLEAN OF
                            FALSE: LastEmployed: Date;
                          |
                            TRUE: employer: EmployerType;
                        END (*CASE*);
                        address: AddressType;
                    END (*RECORD*);
```

Notice that a variant part has its own END, and as a result it is not compulsory to put the variant part last. (In fact, it is legal to have several variant parts, if desired.) The alternatives in the variant part are separated by "|", as in the CASE statement, so there is no need for parentheses to show where a variant starts and ends.

To declare a pointer type, use the words "POINTER TO", as in the example

```
VAR p: POINTER TO REAL;
```

In this example, the real number to which p points can be accessed as p^, just as in Pascal. That is, the caret or up-arrow is used in executable statements, but not in declarations.

Sets in Modula-2 are basically the same as sets in Pascal, but there are some minor syntactical differences. The declarations are in the form

```
TYPE CharSet = SET OF CHAR;
     CountSet = SET OF [0..200];
```

and so on. To use a set in an expression, one can have statements like the following.

```
x := CharSet {"a","f","g"};
y := CountSet {3, 6..25, 92..99, 107};
```

Notice that the type name is written in front of the set; this is needed to remove ambiguities. If you omit the type name, BITSET will be assumed, and this is usually not what you want. This rule also holds in constant declarations, such as

```
CONST Letters = CharSet {"A".."Z", "a".."z"};
```

(which can be very useful in constructs like "IF ch IN Letters THEN ...").

The maximum possible size of a set varies from compiler to compiler. Typically, SET OF CHAR is possible, but SET OF INTEGER is too big.

Procedures are themselves first-class objects in Modula-2, so that you can have variables of type "procedure". More precisely, you can have declarations like

```
VAR f: PROCEDURE (REAL): REAL;
```

and then use statements like

```
f := SIN;
y := f(x);
```

This facility is most useful in the case of a procedure which takes a procedure name as one of its parameters. Notice that the declaration of a procedure type must include the type of its parameters, and the type of its result if it happens to be a function. The predefined type PROC may be used in the case of procedures with no parameters and no result.

A.6 Modules

The most important feature of Modula-2, indeed the key factor which makes Modula-2 an improvement on Pascal, is the facility for dividing a program into modules. This topic appears towards the end of these notes only because of a wish to take advantage of the reader's existing knowledge of Pascal. If the topics had been arranged in order of importance, then this section would have come first.

A program written in Modula-2 consists of a main module, and any number of imported modules. The imported modules can be library modules, or they can be modules written by the user. The main module has the form:

```
MODULE ModuleName;
    import specifications;
    declarations;
BEGIN
    statements;
END ModuleName.
```

Notice that the module name must be repeated after the final END. The module name is any identifier chosen by the programmer. Most compilers adopt the convention that the file name for a module consists of the first eight or nine characters of the module name (the exact number depending on what the file system allows for file names). Thus, it is usual to make the name of the main module the same – at least in the first eight or so characters – as the intended name of the overall program.

The "declarations" section consists of constant, type, variable, and procedure declarations, just as in Pascal; and the statements between BEGIN and END form the main program, again as in Pascal.

The import specifications are the novel part. These are used as in the following example.

```
FROM InOut IMPORT WriteString, WriteLn;
FROM Maths IMPORT SIN, COS, SQRT;
```

This says that this module imports names from two other modules: two procedures called WriteString and WriteLn from a module called InOut, and functions called SIN, COS, and SQRT from a module called Maths. All external procedures, including library routines, which are used in the current module must be imported using an IMPORT specification. Most commonly, it is procedures which one imports, though it is also possible to import types, constants, and variables. The general rule is that IMPORT is used for any object which exists in another module (and is exported by it), and to which there is a reference in the current module.

An alternative approach is simply to write

```
IMPORT Mname;
```

where Mname is the name of a module. This automatically imports everything that is exported by module Mname. This requires, however, that any reference to an imported object must be qualified by the module name. For example, if you simply import module Maths, without using the FROM form, then you will have to use statements like

```
y := Maths.SIN(x) + Maths.COS(x);
```

when referring to functions SIN and COS from the Maths module. If the FROM form of import is used, then this statement simplifies to

```
y := SIN(x) + COS(x);
```

In the majority of cases, the FROM option is superior. Importing a module by just giving its name is justified only in special cases, such as when two procedures, from two different modules, happen to have the same name.

A main module, of the kind just described, does not export anything. Most modules do, however, export something, and are broken into two parts: a definition part, and an implementation part. The definition part defines the information − such as procedure headers − which is to be exported. The implementation part contains most of the detail, and that detail will not be exported. For a concrete example, suppose we want a procedure, in its own module, which reverses the order of the characters in a character string. One way to write this is as follows.

```
DEFINITION MODULE Reverse;

    PROCEDURE ReverseCharacters (VAR string: ARRAY OF CHAR);

        (* Reverses the order of the characters in array "string" *)

END Reverse.
```

This is the definition part of the module. It gives the procedure header for procedure ReverseCharacters, thus implicitly specifying that other modules may import this procedure. It does not, however, contain the procedure body. That is contained in the implementation part (a separate file) which is shown below.

```
IMPLEMENTATION MODULE Reverse;

PROCEDURE Swap (VAR x,y: CHAR);

    (* Swaps two characters. Note that this is a purely *)
    (* internal procedure; it is not exported.          *)

    VAR temp: CHAR;

    BEGIN
        temp := x; x := y; y := temp;
    END Swap;

(**************************************************************)

PROCEDURE ReverseCharacters (VAR string: ARRAY OF CHAR);

    (* Reverses the order of the characters in array "string" *)

    VAR j, last: CARDINAL;

    BEGIN
        last := HIGH(string);
        FOR j := 0 TO (last-1) DIV 2 DO
            Swap (string[j], string[last-j])
        END (*FOR*);
    END ReverseCharacters;

END Reverse.
```

This example is slightly atypical, in that the module consists entirely of procedures. An implementation module may contain anything that a main module may contain: import lists, declarations of constants, types and variables, procedure declarations, and even a main program part. The main program part, if present, acts as an "initialisation section" for the module. When a program first starts, the main program parts of all imported modules are executed before the main program of the main module is executed.

(Strictly speaking, Modula-2 does not have the concept of a "main program". It simply has modules with initialisation sections. The initialisation section of the top-level module – i.e. the one that is initialised last – corresponds to what would be called a "main program" in other languages.)

A definition module, on the other hand, never contains any executable code. It contains procedure headings but not their bodies. It may contain IMPORT specifications (though these are not often needed), and it may contain declarations of constants, types, and variables. (If so, these declarations are not repeated in the implementation module.) Anything declared in the definition module is automatically exported. It is very common to export types, but much less common to export constants or variables. Notice that an exported variable would be known globally to all modules (or, at least, all of those which wish to import it), in the same way as a Fortran COMMON variable; and the use of such variables is generally considered to be bad programming practice.

In an early version of the Modula-2 language, there was a requirement that every definition module have a special EXPORT declaration, listing all the identifiers to be exported. This requirement was deleted in subsequent revisions of the language, but it may still be found in some old compilers. With up-to-date compilers, the word EXPORT is used only in nested modules, which for the sake of brevity are not described here.

One special feature of the language, which is very useful in practice, is the ability to declare "private types", also known as "opaque types". A private type is declared in the definition module, but the details of the type definition are omitted. (Those details are, of course, given in the implementation module.) An example is:

```
DEFINITION MODULE Example;
TYPE Abc;       (* private type *)
PROCEDURE CreateObject (VAR x: Abc);
PROCEDURE Insert (VAR x: Abc; newinfo: CHAR);
END Example.
```

Here, the type Abc must be mentioned in the definition module, because it is used in the parameter lists of the exported procedures. Another module can import type Abc, but that other module has no way of knowing the implementation details of that type. Thus, all operations on variables of type Abc must be performed by procedures in the defining module.

Most compilers impose the restriction that all opaque types must be pointer types. (This is done to simplify the compiler's job of allocating memory for variables.) Although this might be seen to be an unreasonable restriction, it can always be circumvented – at the cost, admittedly, of a more obscure programming style – by making the opaque type a pointer to the type which one really wants to define.

There is an implied constraint on the order in which modules may be compiled. If module A imports something from module B, then the definition part of B must be compiled before module A is compiled. (And if a definition module is re-compiled, any module which imports that module must be re-compiled.) However, there is no constraint on the order of compilation of implementation modules. This means that a module may be replaced by an upgraded version, and – provided that the definition module is not changed – there is no need to re-compile other modules.

Most Modula-2 compilers come with a "make" facility which can work out the inter-module dependencies and recompile any out-of-date modules.

A.7 The Module SYSTEM

Every implementation of Modula-2 provides a special module called SYSTEM, which defines some low-level types and procedures. This module should be used sparingly, since programs using it will probably be non-portable. However, it is needed when writing machine-dependent software such as device drivers.

SYSTEM is actually a pseudo-module; you will not find a listing of it among the library modules, even if you have access to the source listings. Its details are in fact buried inside the compiler. In effect, importing something from SYSTEM is an instruction to the compiler to relax its type-checking rules and to permit the use of some machine-dependent operations.

The types defined by SYSTEM usually include BYTE, WORD, and ADDRESS, whose meanings will be obvious to those familiar with low-level programming. Their main use is to bypass type checking. Like Pascal, Modula-2 is a strongly typed language, which means, roughly speaking, that an assignment "a := b" is illegal unless a and b are of the same type (or of closely related types; for example, the two types may be subranges of the same type). The types imported from SYSTEM provide exemptions from the type compatibility rules. For example, the assignment "p := q" is legal when p has type ADDRESS and q is any pointer type.

The following three functions are usually available from SYSTEM:

ADR(x)	Returns the address of x (which may have any type). The result has type ADDRESS.
SIZE(x)	Returns the size of variable x. This is usually given as the number of bytes of memory occupied by x.
TSIZE(t)	Like SIZE, but t is a type name rather than a variable name. For example, TSIZE(REAL) returns the number of bytes needed to hold a real variable.

A.8 Standard Functions and Procedures

The following functions are built in to the language, and do not have to be imported from anywhere before being used.

ABS(x)	Absolute value of x, which may be REAL or INTEGER. The result is of the same type as x.
CAP(c)	Converts the CHAR c to upper case, if it happens to be a lower case letter. If c is not a lower case letter, the result is just c, with no change.
CHR(x)	Returns the character with collating sequence x, where x has type CARDINAL.
FLOAT(x)	Converts CARDINAL to REAL.
MIN(t)	Returns the minimum value that can be taken by a variable of type t. The argument t may be INTEGER, CARDINAL, CHAR, BOOLEAN, or any subrange or enumeration type.
MAX(t)	As for MIN, but returns the maximum value. Notice that the Pascal predefined constant "Maxint" is written as MAX(INTEGER) in Modula-2.
ODD(x)	Returns TRUE if the CARDINAL x is odd, and FALSE if x is even.
ORD(x)	Returns a CARDINAL giving the ordinal value of x, where x is any variable whose type is acceptable as an argument to MIN.
TRUNC(x)	Converts REAL to CARDINAL. The result is truncated, not rounded.
VAL(t,x)	Returns the value from type t whose ordinal value is x. VAL is the inverse function of ORD. Notice that VAL(CHAR,x) is the same as CHR(x). Many compilers also permit VAL to be used with non-ordinal types such as REAL.

Because of the type compatibility rules, any function which has argument type CARDINAL will also accept arguments of type INTEGER, provided that the value of the integer is nonnegative.

It is also possible to use the name of a type as if it were a function name, to perform conversion from one type to another. For example, if you have

```
TYPE Grade = (first, second, third);
     Group = (one, two, three);
VAR x: Grade; y: Group;
```

and then execute the assignment statements

```
x := second; y := Group(x);
```

then y will be assigned the value "two". This sort of type conversion does not involve any computation; it simply takes the machine representation of the argument, and re-interprets the value as the value of a variable of the new type. Because it is machine-dependent, it should be used with great care.

As well as the built-in functions, there are some built-in procedures.

DEC(x,n)	Subtract n from x. In the special case n=1, you may write DEC(x) instead of DEC(x,1).
INC(x,n)	Add n to x. In the special case n=1, you may write INC(x) instead of INC(x,1).
INCL(x,e)	Include the element e in the set x. If x has type S, this is the same as x := x + S{e}.
EXCL(x,e)	Remove the element e from the set x. If x has type S, this is the same as x := x − S{e}.
HALT	Terminate the program.
NEW(p)	As in Pascal.
DISPOSE(p)	As in Pascal.

The calls NEW(p) and DISPOSE(p) are translated by the compiler into ALLOCATE(p,SIZE(p^)) and DEALLOCATE(p,SIZE(p^)), respectively. Thus, any module which uses NEW and DISPOSE must import ALLOCATE and DEALLOCATE from module Storage (or from another module, if you are prepared to write your own versions of ALLOCATE and DEALLOCATE). You may, if you wish, bypass NEW and DISPOSE by calling ALLOCATE and DEALLOCATE directly.

Any Modula-2 compiler will almost certainly have associated with it a collection of library modules, supplying a number of useful procedures. These can be imported in the same way as importing from user-written modules.

Note in particular that mathematical functions such as SIN and COS, and input-output procedures such as Write and Read, are not built into the language, but must be imported from the appropriate library modules.

A.9 Task Switching

Another feature found in most implementations of Modula-2 is a collection of procedures which can help in writing multitasking software. These are usually found in module SYSTEM; but there is no fundamental reason for this, and in some implementations they will be found in some other library module or modules.

The basic task-switching operation is

```
TRANSFER (from, to)
```

where variable "from" denotes the calling task, and variable "to" specifies the task to which control is to be transferred. Both parameters have type ADDRESS. (Or type PROCESS if using an older compiler, where PROCESS is a type imported from SYSTEM.) The initial value of "from" is not important; procedure TRANSFER will use it to save information about the current task, information which will later be used when that variable is used as the second argument in a call to TRANSFER. The initial value of "to" must be set up by procedure NEWPROCESS, which is the procedure used to introduce a new task to the system. The NEWPROCESS call has the form

```
NEWPROCESS (StartAddress, work, size, descriptor)
```

The first parameter, which has type PROC, specifies the procedure where the new task is to start executing the first time it is invoked by a call to TRANSFER. The second parameter is the address of a block of memory which can be used for the new task's stack, and the third parameter is the size (in bytes) of this block. The final parameter is the variable which will be used in subsequent calls to TRANSFER.

For interrupt handling, there is a procedure

```
IOTRANSFER (from, to, InterruptNumber)
```

which is like TRANSFER, except that it also sets up an interrupt vector, before transferring control to the task specified by the "to" parameter. When the interrupt occurs, the state of the interrupted task is saved in the second parameter, and execution continues at the statement following the IOTRANSFER. The intention is that the interrupt handler should be written as a loop, with the IOTRANSFER call inside that loop. Thus, the next time that IOTRANSFER is executed, control will be transferred back to the interrupted task.

To support inter-task coordination, there are also library procedures to perform operations on variables of type SIGNAL (not to be confused with a semaphore Signal). A SIGNAL is a variable which is somewhat similar to, but simpler than, a semaphore. For brevity, those details are not described here.

The procedures described in this section are not suitable for use directly by high-level software, because the caller of TRANSFER must explicitly specify the task to which control is to be transferred. That is, one cannot say something like "transfer control to the next available task"; it is up to the caller to keep track of which tasks are ready to run. The real point of these procedures is that they are primitive operations to be used as building blocks in writing a kernel.

As it happens, they were *not* used as the basis for the software in this book. The decision to bypass the multitasking features of Modula-2 was made for three reasons:

(a) If one uses procedure TRANSFER without knowing the details of how it is implemented, then one is left a little in the dark as to how the task switch actually happened. A major goal of this book is to unveil that mystery, and this is best achieved by developing software right down to the lowest level, rather than relying on a built-in shortcut provided by the language.

(b) In some cases, the supplied procedures are less efficient than one would like for real-time applications.

(c) Some (not all) compilers implement TRANSFER and IOTRANSFER via calls to the underlying operating system. This is not very helpful if one is trying to write the operating system.

(d) Some compilers do not support TRANSFER and IOTRANSFER.

Another aid to multiprogramming − again, not one which is not used in this book − is the ability to declare a module *priority*. The priority is a number which is written after the module name in the heading of an implementation module. The processor priority is set to that value when executing code in that module, if the processor is one which supports interrupt priorities. With processors which do not support that feature, interrupts are disabled while code in the module is being executed. Thus, module priorities are a crude way of ensuring that there will be no unexpected task switch.