UNIVERSITY OF EDINBURGH

Edinburgh Regional Computing Centre

# The IMP80 Language

A description of the programming language IMP80

First Edition
December 1982

EDINBURGH REGIONAL COMPUTING CENTRE

# The IMP80 Language

A reference manual for the programming language IMP80,
with notes on the two main implementations, IMP77 and EMAS IMP80

by

Felicity Stephens

John Murison

CONTENTS

# PREFACE

This manual describes the programming language IMP80, which is a common subset of several extant versions of IMP. The reason for the existence of these different versions is explained in the "Historical Introduction" (below).

It is intended that for the foreseeable future implementations of IMP will follow IMP80 as far as possible; however, this does not preclude implementation dependent extensions.

Separate sections of Appendix B are provided for each major implementation of IMP80. These detail departures from or extensions to IMP80, and include relevant implementation dependent information. Such implementation dependent information is also included occasionally within the main body of the manual, where it was felt that to have excluded it would have been inconvenient or even misleading. Such material is clearly flagged as not describing IMP80 itself, and is always repeated in the relevant section(s) of Appendix B.

There are two implementations of IMP80 at the time of writing: *EMAS IMP80*, implemented on the ICL 2900 range by Peter Stephens, ERCC; and *IMP77*, implemented on several different machines by Peter Robertson, Lattice Logic Ltd., Edinburgh.

This is primarily a reference manual, and it is hoped that the contents list and index are detailed enough to enable it to be used as such. In addition, however, it is felt that guidance on the use of language features should be given, as their utility might not be apparent from a statement of their syntax and semantics alone. The programming examples included for illustrative purposes are also intended to indicate good programming practice.

This manual specifies what constitutes a legal IMP80 program, and in so doing describes various constructions or usages as "illegal" or "invalid", or simply "wrong". This does not imply that all implementations would fault such constructions or usages. For example, optimising compilers may omit to check that a variable whose value is used has in fact been assigned a value; nonetheless, if an attempt is made to use an unassigned variable then the program is wrong.

When an error is detected in an executing IMP80 program, an *event* is *signalled*. IMP80 provides a mechanism to enable the programmer to specify, optionally, what actions are to be taken on the signalling of any event. The event machanism is described in Chapter 3.

A Backus-Naur Form (simplified) specification of the complete syntax of IMP80 is given in Appendix A. An explanation of the conventions used in the syntax definition is given at the end of Appendix A.

This manual was written by Felicity Stephens and John Murison, apart from the Historical Introduction by Peter Stephens. Much of the text is based on earlier documents, in particular: "The Edinburgh IMP Language Manual" (second edition) edited by Roderick McLeod, ERCC (1974), and "The IMP-77 Language" (third edition) by Peter Robertson, Department of Computer Science, University of Edinburgh (1980).

Please report any suspected errors or omissions in this manual to the ERCC Advisory Service, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ.

John M. Murison
Editor
December 1982

## HISTORICAL INTRODUCTION

## EVOLUTION AND PHILOSOPHY OF IMP

### Evolution

The IMP language evolved from Atlas Autocode, which itself is a direct descendant of Algol 60. Although Algol 60 had only moderate success as a programming language - it was hardly used in the United States - no other language before or since has achieved more than a fraction of its influence on programming language design.

At the same time as Algol was being devised and revised, in Manchester another event was taking place which was also to have wide influence. The University was building its fourth machine, the Atlas, which - true to tradition - was at the very limit of the technology of the time. This machine was to introduce paging to the world; the idea being that memory management, provided by the operating system with hardware support, was cheaper and more efficient than allowing each programmer to overlay or shoehorn his program into the space available. After a slow start, this idea was to change the appearance of computing. Since Atlas was a revolutionary machine, Manchester had to write their own software, as they had done for the Manchester Mark 1 and Mercury. This led them to consider the attractions and disadvantages of Algol.

Algol's principal attraction is its block and stack structure: by collecting space together on a stack and re-using it for successive procedure calls, an Algol program causes less paging than the same program written in (say) Fortran.

The disadvantages of Algol are the lack of standard input/output and the difficulties that some features of the language present to the compiler writer. The tragedy of Algol was that so little was gained from the features which presented most of the problems. Almost nothing of real power was gained either from call by substitution or failure to specify formal procedures adequately, and little was gained by the enormous generality of the **for** statements. Yet the problems these areas posed caused all early Algol compilers to produce comparatively low quality object code.

All this and more was obvious to the Manchester team and although Algol was to be implemented on Atlas, the prime language was a new one - Atlas Autocode. This was a simplified Algol with changes to the block, loop and procedure structures to remove the worst problem areas. It contrived to deliver 90% of the power of Algol to the programmer while only requiring 25% as much effort from the compiler writer. (Further details can be obtained from *The Computer Journal, Vol 8, pp 303-310 (1965/66)*.) In retrospect, the name was unfortunate since autocodes were normally low level languages, and "Atlas" indicated quite wrongly a degree of machine dependence.

Edinburgh University started its computing with a data link to the Manchester Atlas, and this happy accident began the long association between the University and the language. When Glasgow and later Edinburgh obtained KDF9 computers it was necessary to write a compiler for Atlas Autocode; this was carried out in a short time by Mr (now Professor) Harry Whitfield and his associates. This compiler was in advance of its time in that it was written entirely in Atlas Autocode and developed on Atlas. It was transferred to KDF9 by the elegant technique of self-compilation. The compiler thus produced compared exceedingly well with the manufacturer's Algol compiler, both in compilation time and in object code efficiency. This project also confirmed that Atlas Autocode was free of implementation trouble spots and very suitable for large scale system programming.

In 1966 there began a large scale project with a joint University/Manufacturer team to write a time-sharing operating system for the ICL 4-75 computer. The project was based in Edinburgh and the final system was required to support Atlas Autocode, among other languages. The recent success of the Atlas Autocode compiler project led to the decision to implement the time-sharing system (later called EMAS) in a high level language called IMP. IMP was to be a superset of Atlas Autocode, containing additional features for system programming. It was at this point that almost all the main language changes were made and the distinctive philosophy of IMP originated.

### Philosophy and Style

IMP was to be primarily a system programming language; in 1966 that was perceived to require:

* *efficient object code*. System programs are liable to be executed millions of times. Thus features that could not be implemented efficiently should be omitted.

* *early compiler availability*. The compiler should be available as soon as the hardware, otherwise programmers would program in something else. Consequently features that would or might cause implementation trouble spots must not creep into the language.

* *minimum run-time support*. Some system programs like supervisors and loaders have to run in an environment almost devoid of support software. The language should be free of features requiring run-time support. (The Atlas Autocode *fault* statement conflicted with this aim and had to be banned from system programs, although retained in the language for user programs.)

* *readability*. System programs have a long life and require maintenance. It is more important that the program be easy to understand than quick to write. Optional keyword omission or abbreviations should be banned. (The language should be "verbose rather than obscure".)

* *access to bizarre hardware features*. System programs require access to funny features - to blow the hooter or ring the gong on hardware failure, for example. However the language would not be compromised here. Instead machine coding would be allowed at any point in the program, with access to the IMP variables and arrays.

Even with fifteen years of hindsight this list still seems relevant, although the need for efficient object code was more pressing then than now. The real key to the long life of IMP is the last item in the list. By allowing machine code *in extremis* almost no machine-dependent features were included except the underlying one of a byte address structure. Consequently IMP has been successful on a dozen or so machines, unlike the main competitors of the era (PL360, Burroughs Extended Algol).

In accordance with the language philosophy, the following changes were made to Atlas Autocode to produce IMP:

* The internal character code was changed to ISO.

* Logical operations were added to the language.

* The additional declarators **byte integer** and **short integer** were introduced.

* Structured data objects - records - were introduced.

* Text handling features were added specifically to aid the writing of command interpreters.

* Reference variables and additional features were added to enable programmers to operate on storage areas outwith the compiler controlled stack.

It proved possible to write EMAS, a high performance multi-access operating system, almost entirely in IMP. (Those requiring further information should read *The Computer Journal, Vol 17, pp 216-223 (1974)*.)

## Recent Developments

After the successful completion of EMAS, responsibility for maintenance of the system and the IMP compiler passed from the department of Computer Science to the Edinburgh Regional Computing Centre. Over the next ten years the language evolved very slowly - even a move of EMAS from ICL 4-75 to ICL 2900 hardware scarcely disturbed the stability of the language.

However the Computer Science department, in pursuing its diverse research interests, encountered a variety of machines and wrote IMP compilers for a substantial proportion of them. As befits an academic department these compilers contained novel features and gradually diverged from EMAS IMP and from each other. In 1980 a stock-taking was instituted, from which there gradually emerged the common core of features described in this manual. Most compilers will have some additional features but will support this common base, and it should be possible to write most programs in the common subset. Such programs should be readily portable in the Edinburgh environment.

P.D. Stephens

# CHAPTER 1

# ELEMENTS OF THE LANGUAGE

IMP80 is a high-level block-structured language. Relative to Algol 60, it adds program structuring, data structuring, event signalling and text handling facilities, but removes (or retains in a modified form) intrinsically inefficient features such as the Algol 60 name-type parameter.

## 1.1 Character set

An IMP80 program is a sequence of *statements* constructed using the ISO seven bit character set extended with a special alphabet. The special alphabet is used to construct *keywords*.

The ISO seven bit character set is given on the next page. The use of the character set is summarised in the sections below and described in detail in the rest of the manual. However, some general points about certain characters should be noted at the outset:

a) Quotes

Several language constructions call for one or more characters to be enclosed in quotes. Within quotes all characters are significant and stand for themselves; thus, for example, space, newline, and percent characters may appear between quotes and stand for space, newline, and percent.

Two quote characters are used:

```
    '        -  character quote
    "        -  string quote
```

Examples:

```
    'A', ';', '"', '!'
    "String", "sealing wax", "!"
```

If it is required to include the delimiting quote within the text it must be represented by two consecutive quotes.

Examples:

```
    ''''              -  the character quote
    "A ""big"" dog"   -  a string of eleven characters
```

However, note '"' and "it's mine".

b) Spaces

Except when used to terminate keywords or when between quotes, *space* characters are ignored and may be used to improve the legibility of the program.

c) Lower case letters

Except when enclosed in quotes, lower case letters are equivalent to the corresponding upper case letters.

d) Non-graphic characters

The ISO characters with no graphic representation have code values less than 32 or greater than 126. Any non-graphic characters can be included within quotes in an IMP80 program, but only one, the *newline* character (NL), is included in the syntactic definition of the language (see Appendix A). This has the ISO code value 10.

*The handling of non-graphic characters by different implementations is detailed in Appendix B.*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | NUL | 32 | SPACE | 64 | @ | 96 | ` |
| 1 | SOH | 33 | ! | 65 | A | 97 | a |
| 2 | STX | 34 | " | 66 | B | 98 | b |
| 3 | ETX | 35 | # | 67 | C | 99 | c |
| 4 | EOT | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ | 37 | % | 69 | E | 101 | e |
| 6 | ACK | 38 | & | 70 | F | 102 | f |
| 7 | BEL | 39 | ' | 71 | G | 103 | g |
| 8 | BS | 40 | ( | 72 | H | 104 | h |
| 9 | HT | 41 | ) | 73 | I | 105 | i |
| 10 | LF(NL) | 42 | * | 74 | J | 106 | j |
| 11 | VT | 43 | + | 75 | K | 107 | k |
| 12 | FF | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO | 46 | . | 78 | N | 110 | n |
| 15 | SI | 47 | / | 79 | O | 111 | o |
| 16 | DLE | 48 | 0 | 80 | P | 112 | p |
| 17 | DC1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN | 54 | 6 | 86 | V | 118 | v |
| 23 | ETB | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN | 56 | 8 | 88 | X | 120 | x |
| 25 | EM | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | 59 | ; | 91 | [ | 123 | { |
| 28 | FS | 60 | < | 92 | \ | 124 | \| |
| 29 | GS | 61 | = | 93 | ] | 125 | } |
| 30 | RS | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US | 63 | ? | 95 | _ | 127 | DEL |

## 1.2 Statement components

An IMP80 program consists of an ordered sequence of statements. IMP80 statements are made up of *atoms*. An atom is an *identifier*, a *constant*, a *special symbol* or a *keyword*.

### Identifiers

An identifier is a sequence of any number of letters and digits, starting with a letter. For example: C1900 TO 1970, NUMBER OF BLOCKS, MAX, x, item1, Item 2, item 2b. Spaces within identifiers are permitted but ignored, as is the case of the letters; thus Item 2 and ITEM2 are not distinguished.

*It is recommended that meaningful identifiers be used whenever possible, to improve the clarity of programs.*

*In this manual, identifiers are always given in upper case. It is stressed that this is merely a convention.*

Identifiers are used to name the following entities:

Arithmetic, string and record variables and arrays

Reference variables

Record sub-fields

Procedures (i.e. routines, functions, maps)

Record and array formats

Named constants

Simple labels and switch vectors

Identifiers in an IMP80 program must be declared before they may be used. The only exception to this general rule is that simple labels are not declared. Further rules relating to the uniqueness of identifiers are given in the relevant sections.

### Constants

IMP80 includes the following types of constant:

| | |
|---|---|
| Integer constants (decimal) | e.g. 2243, -16, 1 000 000 |
| Base constants | e.g. 2_1001, 8_7720, 16_A06C, B'1011', X'1A69' |
| Real constants (decimal) | e.g. 120.0, 120, 1.2@2, 12@1, 1200@-1 |
| Character constants | e.g. 'A', 'a', '+', '"', ' ', '''', '6' |
| Multi-character constants | e.g. M'Four', M'MAX', M'1+1=', M'"#%' |
| String constants | e.g. "Here is a string", "A", "123", "a ""good"" boy" |
| Named constants | e.g. **constant integer** PRICE=23 |

## Special symbols

These are as follows:

| | | | | |
|---|---|---|---|---|
| ! | !! | ′ | " | # (or \= or <>) |
| ## (or \==) | & | ( | ) | * |
| \ (or ^) | \\ (or ^^) | + | , | - |
| -> | . | / | // | : |
| ; | < | <- | << | <= |
| = | == | > | >= | >> |
| @ | \ (or ~) | _ | { | } |
| newline | space | | | |

The special symbols are used in IMP80 in various ways:

* as operators (arithmetic, string, record, relational, logical)
* as separators, e.g. (..) {..} : ; , newline space
* in constants, e.g. 3_201, +27@4, -17.6, X'8F', "Constant"
* in record subfield references, e.g. TAX(MONTH)_OVERTIME
* ! as an alternative to the keyword **comment**

## Keywords

The following is a list of all the IMP80 keywords:

| | | | | |
|---|---|---|---|---|
| **alias** | **dynamic** | **half** | **of** | **signal** |
| **and** | **end** | **if** | **or** | **spec** |
| **array** | **event** | **include** | **own** | **start** |
| **begin** | **exit** | **integer** | **program** | **stop** |
| **byte** | **external** | **list** | **real** | **string** |
| **c** | **file** | **long** | **record** | **switch** |
| **comment** | **finish** | **map** | **repeat** | **system** |
| **const** | **fn** | **monitor** | **result** | **then** |
| **constant** | **for** | **name** | **return** | **unless** |
| **continue** | **format** | **not** | **routine** | **until** |
| **cycle** | **function** | **on** | **short** | **while** |

A *keyword* is a sequence of letters in a special alphabet. In programs stored in a computer in text form ("source" programs), representation of this alphabet is achieved by using the percent character (%), which is defined as causing the subsequent letters (upper or lower case) to be shifted into the special alphabet. The effect of the % character is terminated by *any* non-alphabetic character, including space and newline.

Hence the following statements are equivalent:

```
        %string(7) %array %name P
        %STRING (7) %ARRAYNAME P
```

and both represent **string(7)array name** P

*In this manual keywords are rendered in bold type, and by convention in lower case.*

## 1.3 Statements

An IMP80 program is composed of *statements*, and typically has the following general form:

```
begin

   declaration statements
   and procedure descriptions

   on event .... start
      executable statements
   finish

   executable statements

   procedure descriptions

end of program
```

Only the delimiting *keywords* **begin** and **end of program** must be present. However, assuming that all the types of statement above are represented in an IMP80 program, the effect of running the program on a computer is as follows:

First, storage space for the declarations is obtained. The procedure descriptions are skipped, as are the statements in the **on event** block, and the first of the executable statements is obeyed. Thereafter, the statements themselves define the sequence of execution ("flow of control"). Control passes to the **on event** block if one of the events defined in the **on event** statement occurs whilst executing a statement. The **on event** block is optional and often omitted.

Here is an IMP80 program:

```
begin

   integer A, B, SUM, DIFFERENCE, MAX V, MIN V,
           MAX N, MIN N, N
   N = 0

   cycle
      READ(A)
      exit if A=-1; ! End of input - exit from loop.
      N = N+1
      READ(B)

      WRITE(N, 2)
      PRINTSTRING("   Input values: ")
      WRITE(A,1); PRINTSYMBOL(',')
      WRITE(B,2)

      SUM = A+B; DIFFERENCE = A-B
      PRINTSTRING(" ..... Sum is ")
      WRITE(SUM, 1)
      PRINTSTRING(",   Difference is ")
      WRITE(DIFFERENCE, 1); NEWLINE
```

{Continued on next page}

```
          if N=1 or SUM>MAX V start
             MAX V = SUM; MAX N = N
          finish
          if N=1 or SUM<MIN V start
             MIN V = SUM; MIN N = N
          finish
       repeat
       ! Now print out the maximum and minimum of the sums of all
       ! the pairs of numbers read in.
       if N=0 start
          PRINTSTRING("No data read.")
          NEWLINE
       finish else start
          NEWLINE
          PRINTSTRING("Maximum sum is ")
          WRITE(MAX V,1)
          PRINTSTRING(",  data pair no. "); WRITE(MAX N, 1)
          NEWLINE
          PRINTSTRING("Minimum sum is ")
          WRITE(MIN V,1)
          PRINTSTRING(",  data pair no. "); WRITE(MIN N, 3)
          NEWLINE
       finish

    end of program
```

The different types of statement in IMP80 are listed below, with references to the chapters describing them.

| | Chapter |
|---|---|
| **comment** | 1 |
| **include** | 1 |
| **list** & **end of list** | 1 |
| declaration | 2 |
| assignment | 2 |
| **begin** | 3 |
| **end of program** | 3 |
| **on event** | 3 |
| procedure specification | 3 |
| procedure heading | 3 |
| procedure call | 3 |
| **end of file** | 3 |
| jump & label | 4 |
| **start/finish** | 4 |
| **cycle/repeat** | 4 |

## Continuation

Statements are normally terminated by a newline or semi-colon character. However, a statement may extend over several physical lines provided that each line break occurs after a comma, or is preceded by the keyword **c** (which is otherwise ignored). In addition, blank lines between the lines of a continued statement are ignored.

Thus:

```
if X=Y then P = 1 c
      else P = 0
```

is exactly equivalent to

```
if X=Y then P =1 else P = 0
```

and

```
own integer array X(1:10) = 10, 9, 8, 7, 6,

                            5, 4, 3, 2, 1
```

is exactly equivalent to

```
own integer array X(1:10) = 10,9,8,7,6,5,4,3,2,1
```

## 1.4 Miscellaneous statements

Certain statements are described here rather than in subsequent chapters, as they are different in kind from the other statements of the language and do not fit into the categories used in the rest of the manual.

### Comments

Textual comments can be included in an IMP80 program. They have no effect on the program when it runs, but may be used to render it meaningful to the originator and to others. A comment is a statement; it must be separated from the preceding statement by a newline or semi-colon, and from the succeeding statement by a newline - *not* by a semi-colon. The keyword **comment** is used to introduce the text of the comment, which may contain any character except newline. **comment** may be replaced by the symbol '!'.

*EMAS IMP80: Note that the continuation rules described above apply to comment statements.*

*IMP77: comments cannot be continued onto subsequent lines by any of the methods described.*

Example:

```
        begin
           comment This program sorts a list of names
           comment into alphabetical order

             :
             :

        end of program
```

The IMP80 language also allows comments to be embedded within other statements. An embedded comment is any sequence of characters, excluding '}' and newline, enclosed in a pair of braces, '{' and '}'. This form of comment may appear between any two atoms, but may not occur within an atom. The closing brace may be replaced by a newline.

Example:

```
        ! This is a portion of a test program
        COUNT = 0; ! Note the semi-colon before the '!'; note this one
                   ! (in the text of the comment).
        LIMIT = 100       {Only 100 cases}
        MINIMUM = 0       {all positive
        PROCESS(X {cases}, Y {total cost})
```

This fragment of program is exactly equivalent to

```
        COUNT = 0
        LIMIT = 100
        MINIMUM = 0
        PROCESS(X,Y)
```

The {...} type of comment never counts as a statement in its own right, even when given on a line by itself. Thus it can be interposed between lines of a continued statement.

### *include* statement

A file of IMP80 source statements (terminated by the statement **end of file**) may be included in a source program by giving a statement of the form

**include** *file specification*

where *file specification* is a string constant representing an implementation dependent file name. *Refer to the relevant section of Appendix B for details of any implementation dependent limitations on the use of **include**.*

**include** is particularly useful for inserting procedure specifications.

Example:

```
        begin
           include "WXYZ83.EXTSPECS"
           integer TIME, DISPLACEMENT, ...
             :
             :
```

In this case WXYZ83.EXTSPECS is the name of a file containing (presumably) external procedure specifications (see Chapter 3).

*Note that the **include** statement normally appears in the program listing generated by the compiler, followed by the contents of the file referred to (including the **end of file**). The **include** statement should not be followed on the same line of the source program by other statements, since the complete source line might appear in the program listing* before *the statements included, while in the actual program the statements following the **include** statement* follow *the statements included.*

## *list* and *end of list*

Normally the compiler generates a listing of a program as it compiles it. The precise layout of this listing is implementation dependent. The production of the listing can normally by controlled by setting a system option prior to the compilation. In addition, however, the programmer can control the generation of the listing by use of the statement **end of list**, which inhibits the listing until the end of the program or the statement **list** is encountered. The default is for the listing to be generated.

The statements **end of list** and **list** may appear anywhere within an IMP80 program.

Note that when listing is inhibited, incorrect statements and the accompanying fault messages are still listed.

*IMP77: **end of list** and **list** are nested - see Appendix B.*

# CHAPTER 2

# TYPES, VARIABLES, CONSTANTS AND EXPRESSIONS

## 2.1 Types

Each data item (i.e. each constant or variable) used by an IMP80 program has a *type* associated with it which determines what sort of item it is or can have as a *value*.

There are five categories of type:

1) *Arithmetic*

These comprise integer and real types, as follows:

**byte integer**
**short integer**
**half integer**
**integer**
**long integer**
**real**
**long real**
**long long real**

*Some implementations of IMP80 might not provide all of these types; see Appendix B for details.*

The size of an item of type **integer** always corresponds to the word length of the computer.

The modifiers **byte**, **short**, etc. relate to the size or precision of items of the appropriate type. For byte-addressed machines, the ranges and (where appropriate) precisions associated with arithmetic types are as follows:

| *type* | *normal storage allocation* | *range (inclusive)* | *precision (decimal digits)* |
|---|---|---|---|
| **byte integer** (or just **byte**) | 8 bits | 0 to 255 | |
| **short integer** (or just **short**) | 16 bits | -32 768 to 32 767 | |
| **half integer** (or just **half**) | 16 bits | 0 to 65 535 | |
| **integer** | 16 bits *or* 32 bits | -32 768 to 32 767 *or* -2 147 483 648 to 2 147 483 647 | |
| **long integer** | 64 bits | -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807 | |
| *Note that integer arithmetic involving values outside the range which a variable of type* ***integer*** *can hold may not be permitted.* | | | |
| **real** | 32 bits | $-7.2\mathrm{x}10^{75}$ to $-1.2\mathrm{x}10^{-77}$, 0, $1.2\mathrm{x}10^{-77}$ to $7.2\mathrm{x}10^{75}$ | 7 digits |
| **long real** | 64 bits | | 16 digits |
| **long long real** | 128 bits | | 36 digits |

2) *String*

The type **string** relates to sequences of characters. An item of this type has two numbers associated with it, the actual and the maximum number of characters which the item comprises or can have as a value.

3) *Record*

An item of type **record** is a composite of (in general) several sub-items, each of which has an associated type. When a variable of type **record** is declared, its precise composition must be specified.

4) *Array*

An item of array type is a composite of (in general) several sub-items, each with the same associated type, which must be one of those listed above. The name of the array type is obtained by appending the keyword array to the name of the type in question.

For example:

| *type* | *corresponding array type* |
|---|---|
| **byte integer** | **byte integer array** |
| **record(***format***)** | **record(***format***)array** |
| **string(***n***)** | **string(***n***)array** |

5) *Reference*

An item of this type is, or has as its value, a *reference* to a variable of a specified type. For each of the foregoing types there is a corresponding reference type, the name of which is obtained by appending the keyword **name** to the name of the type in question.

For example:

| *type* | *corresponding reference type* |
|---|---|
| **byte integer** | **byte integer name** |
| **record(***format***)** | **record(***format***)name** |
| **string(***n***)** | **string(***n***)name** |
| **real array** | **real array name** |

Variables of this type are known as *reference variables* or *pointer variables*.

## 2.2 Variables

Variables are named store locations used to hold numeric or textual information. Each variable must be defined in a declaration statement which specifies its type and an identifier to name it. The amount of storage allocated to a variable depends on its type. All variables must be declared at the head of the block in which they are to be used, or in an outer block (see Chapter 3).

A value can be associated with a variable when it is created, but usually it is "unassigned". The method of indicating that a variable is unassigned is implementation dependent; in some implementations a specific bit pattern is taken by convention to be the "unassigned pattern", and any variable with this pattern as its value is said to be unassigned. See Appendix B for details. Any attempt to use a variable which is unassigned will cause an event to be signalled.

*IMP77: all variables can be assigned initial values; see Appendix B.*

Variables can be divided into four categories:

Arithmetic

String

Record

Reference

The following standard integer function is provided (a 'standard' procedure is one which is predefined; see Chapter 7):

**integer function** SIZE OF(**name** A)

The number of storage units occupied by the given variable is returned. The variable can be of any type. The storage unit is implementation dependent but commonly is a byte.

## Arithmetic variables

Arithmetic variables may be of two types: **integer** and **real**. The first holds whole numbers and the second holds numbers with fractional parts. For more efficient use of store, **short**, **half**, **byte** and **long** integer types may be provided, whilst greater precision can be obtained when **long real** and **long long real** variables are available. However, the provision of different lengths is necessarily hardware dependent, though type **integer** will always be available; it corresponds to the word length of the machine. Furthermore, it is possible that on some restricted implementations for small machines, type **real** will not be supported.

*Where implementation is on a byte-addressed machine (e.g. IBM 370, ICL 2900 ranges)* ***short integer*** *and* ***half integer*** *would be 16 bits in length.* ***integer*** *16 or 32 bits* ***long integer*** *64 bits; and* ***real*** *types would be 32 bits, 64 bits and 128 bits in length. On such machines,* ***real*** *variables can only hold values to a precision of 7 significant decimal digits whereas* ***long real*** *variables are precise to 16 decimal digits and* ***long long real*** *variables (if available) to 36 decimal digits. Further details of the representation of variables can be found in the relevant hardware manuals.*

The following example illustrates the declaration of various arithmetic variables.

Example:
```
begin
   integer I, J
   real P, Q

   begin
      byte integer Z
       :
       :
      P = I+J+Q
      Z = 0
       :
       :
   end
    :
    :
end of program
```

Arithmetic variables can be grouped into arrays. The array bounds separated by the symbol ':' are given in brackets after the array identifier when the array is declared.

Example:
```
integer array IN(1:10), OUT(1:20)
```

Where two or more arrays of the same type are to have the same bounds, the bounds need only be specified once, after the list of array identifiers.

Example:
```
long real array X, Y, Z(1:4, -13:0)
```

Multi-dimensional arrays can be used. The maximum number of dimensions is implementation dependent.

Example:

```
            integer array BITLIST(-4:4, 1:2, 10:100, 1:2)
```

When an individual array element is accessed, the array identifier is followed by an ordered list of integer expressions (one for each dimension) enclosed in brackets. These integer expressions are called subscripts.

Examples:

```
            BITLIST(3, 1, 54, J)

            BITLIST(I+K, I, 10, 1)
```

Each of these integer expressions must evaluate to an integer which lies within the range described by the bounds for the relevant dimension. The program is faulty if the array bounds are exceeded.

An array can be declared with integer variables instead of constants for the array bounds. Obviously, the variables used in such a declaration must be declared and given a value before the array declaration occurs.

Example:

```
            begin
               integer TOP

               READ(TOP); ! TOP is given a value from the input data.

               begin
                  integer array TABLE(1:TOP)
                   :
                   :
               end

                :
            end
```

## String variables

A string variable is one which holds textual information. The maximum length of a string is implementation dependent but is not normally less than 255 characters. When the string variable is declared, the maximum number of characters which the string may hold is specified, in parentheses, after the keyword **string**.

Example:

```
            string(24) S
             :
             :
            S = "Results of last test"; !  Length of S is now 20.
```

As the string of characters may vary in length within the given location while the program executes, an indication of the current length is stored along with the current contents.

*In all implementations on byte-oriented machines to date, the current length is held in an extra byte at the front of the string location. Thus the string S declared in the above example would be allocated 25 bytes of storage.*

As can be seen from the above example, the character " (double quote) is used to delimit textual information to be stored in a string location. Where one double quote character is part of the text, two consecutive double quote characters should occur in the text, to distinguish it from the terminating delimiter.

Example:

```
            MESSAGE = "Peter says ""No"" "
```

String variables can be grouped into arrays; the maximum number of dimensions is implementation dependent. Each element of a string array must have the same maximum length, which is specified when the array is declared.

Example:

```
        string(63)array FIELDS (1:5)
```

FIELDS consists of five strings, each of maximum length 63 characters.

The occurrence in an IMP80 program of a semi-colon or newline character normally terminates a statement. However, both are permitted in string constants.

Example:

```
        S = "A; B; C"
        SNL = "
        "
```

A string may be regarded as having a value based on the ISO code values of the characters of the string. Thus the relational operators >, <, =, #, <=, >= may be used to compare strings. In particular, strings composed entirely of alphabetic characters can be regarded as having a dictionary ordering for the purposes of comparison.

Example:

```
        "AB" < "C"            is true
        "AB" < "ABC"          is true
        "IMP80" < "FORTRAN"   is false
```

## Record variables

A record is a variable comprising a collection of entities which may be of different types. It has an identifier which refers to the whole collection and each entity has an identifier; an entity, or "sub-field", can be referenced by using the record identifier together with the required sub-field identifier.

The collection of sub-fields which makes up the record is described in a *record format* statement, which specifies the identifier and type of each sub-field.

Arrays with constant bounds may be used as sub-fields.

Example:

```
        constant integer FROM=3, TO=5
        record format F(byte integer A, string(8) S,
                   integer array M, F(FROM:TO, 1:400), real Y)
```

Record format statements are placed with other declarations at the head of a block or procedure. They do not cause allocation of storage; they cannot be passed as parameters to procedures.

Note that the sub-field identifiers need not be distinct from other identifiers in use in the program, as they are always associated with a specific record identifier. They must, however, be unique within the record format in which they are defined.

Records are declared at the head of a block or procedure. Space is allocated according to the associated record format statement whose identifier occurs in parentheses in the record declaration. The amount of space occupied by a record of given format is the total number of bytes occupied by all the sub-fields plus the minimum number of *'padding'* bytes required to achieve alignment of sub-fields appropriate to their types. *This alignment is implementation dependent.* The actual space occupied by a record can be determined by use of the standard integer function SIZE OF, described at the start of Section 2.2.

Example:

```
        record format F(integer A, string(8) S)
        record (F) R
        record (F) PP, QQ, RR
```

Note that, as in this example, more than one record may be declared having the same record format.

Each sub-field of a record can be referenced as a location of the appropriate variable type by subscripting the record identifier with the sub-field identifier, the two being separated by the underline character '_'.

Examples:

```
          record format F(integer A, string(8) S)
          record (F) R
           :
           :
          if R_S="INC" then R_A = R_A+1
           :


          record format PE(integer I, real array X(0:10))
          record (PE) P
          integer J
           :
           :
          P_X(J+1) = P_X(J)*2
           :
```

A record format may include *alternatives*.

Example:

```
          record format AS(byte integer array CHAR(0:12) or string(12) TEXT)
          record (AS) A
```

The space allocated to record A can be regarded as holding a byte integer array of 13 elements or a string of maximum length 12 characters.

Alternatives provide a means of imposing different interpretations on all or part of a record. Where only part of a record is to have an alternative format, brackets must be used within the record format statement to enclose the alternatives.

Example:

```
          record format AT(real X,
                           (byte integer A, B, C  or  real R  or  integer E),
                           string(10) F)
```

Every sub-field in the record format must be distinct. Each alternative will start at the same address within the record and will be padded out to the size of the largest. (In the example above padding will be added, as the three alternatives are not all of the same size). The amount of padding required depends on the amount of store allocated to each sub-field, which is implementation dependent (as is any correspondence between elements in different alternatives).

Where a procedure (see Chapter 3) is required to perform several different tasks, on different calls, a record whose format contains alternatives can be passed to it as a parameter. Within the record a sub-field common to all the alternatives can be used to indicate for the particular call which alternative is to apply. It is thus possible to use a record to pass different sets of parameters to a procedure.

A record format may contain several sets of alternatives, and alternatives may be nested to any depth.

Arrays of records are permitted; they are analogous to the arithmetic and string types of arrays already described. Each element of a record array is a record of format specified in the record array declaration.

Example:

```
          record format F(integer A, real array X(1:5))
          record (F) array RA(1:100)
```

The fifth element of sub-field X in the 76th element of the record array may be referenced as follows:

```
          RA(76)_X(5)
```

As with other types of array, several record arrays having the same bounds and format may be defined in a single declaration.

Example:

```
          record (F) array RR1, RR2(1:100)
```

A sub-field may be of type **record**. In this case its format must have been already declared. In particular, its format may not be the record format being described (cf. record name sub-fields, described below).

Example:

```
          record format P(integer array X(0:4), integer I)
          record format F1(integer A, B, record (P) D)
          record format F2(record (P) J, K)

          record (F1) ENT
          record (F2) JAK
```

An arbitrary depth of subscription can thus obtain. Using the above declarations, the following are valid references to record elements:

```
          ENT_D_X(1)
          JAK_J_I
```

*A sub-field of type record is word-aligned in most implementations, irrespective of the format of the sub-field.*

Whilst sub-fields of records may be used exactly like IMP80 entities of corresponding type, it is also possible to assign a whole record from one location to another. Two assignment operators are permitted: '=' and '<-'. Both operators require that operands on each side of the assignment refer to **record** locations, except in the case where zero occurs on the right-hand side of the '=' operator: this results in the space allocated to the record referenced by the left-hand operand being set to binary zeros.

When the '=' operator is used, the record formats associated with the left-hand and right-hand operands must be the same.

*EMAS IMP80: differences in the two formats are not faulted if they have the same overall length.*

The '<-' assignment operator transfers as many bytes from the record referenced by the right-hand operand as will fit into the record referenced by the left-hand operand. The '<-' operator takes no account of record formats.

Example:

```
          record format F(integer X, Y, Z)
          record format Q(byte integer array B(0:15))
          record (F) J
          record (Q) array K(1:100)
           :
           :
          J <- K(1)
          K(1) = 0; ! K(1)_B(0), K(1)_B(1), ..., K(1)_B(15) all set to binary zeros.
```

*IMP77: A **record format spec** statement is provided to enable a record format to be referred to before it has been declared. See Appendix B for details.*

## Reference variables

A reference variable (or "pointer variable") is one which has as its value, not a constant, but a *reference to a variable* of a specified type. Reference variables are declared in the same way as the variables to which they can refer, but with the suffix **name** added.

Example:

```
          integer A
          integer name AREF
```

When a reference variable is declared, space is allocated for a reference to a variable of the corresponding type and precision. The operator "==" is used to establish the reference. Once a reference is established, all references to the reference variable will be redirected to the variable which it references. Note that the reference can be established before the referenced variable has been assigned a value. Reference variables are often used in conjunction with store mapping facilities (Chapter 6).

Example:

```
            integer name N
            integer B

            N == B; ! Reference established
              :
              :
            N = 10; ! Assigns 10 to B
              :
```

In exactly the same way, a reference to an array can be set up in an **array name** variable of the appropriate type.

Example:

```
            real array name P
            real array Q(0:27)
            real name Z

             P == Q; ! Reference established.
              :
              :
             P(25) = 10.3; ! This puts 10.3 into Q(25).
             Z == P(27)
              :
              :
             Z = 0; ! This sets the 27th element of Q to zero.
              :
```

The examples above have been of reference variables of arithmetic types. However, string and record reference variables may also be used.

Example:

```
            string(20)name SREF
            string(20) S
              :
              :
            SREF == S
              :
```

A maximum size must be specified for a string reference variable, as for a string variable. A string reference variable can only refer to a string variable whose maximum size is equal to that of its own. However, to accommodate the situation where a string reference variable may be required to refer to several strings of different maximum sizes, the form

            **string(*)name** *var*

is provided.

In the case of record reference variables, the format of the record to be referenced must be specified in the reference variable declaration.

Example:

```
            record format RECFORMR(.....)
            record (RECFORMR) REPORT
            record (RECFORMR) name REP2
```

A reference variable of type **record name** is assigned to by using the '==' operator (as before), where the right-hand operand is a reference to a record location with the same format as that specified for the reference variable.

Example:
```
        record format F(.....)
        record (F) name F1
        record (F) Q, R
        record (F) array A(1:10)
        record (F) array name Z, W

        F1 == Q;      ! Makes F1 a synonym for record Q
        F1 == A(10);  ! Makes F1 a synonym for the 10th element of A
        Z == A;       ! Makes Z a synonym for A
```

Records may contain sub-fields of type **record name**. In this case the format of the record name sub-field may be the record format being described (cf. sub-fields of type **record**).

The following example illustrates how the recursive nature of the format and sub-field format definitions facilitates the creation of a list structure:

Example:
```
        record format F(integer DATA, record (F) name LINK)
        record (F) array P(1:1000)
```

The structure may be initialised as follows so that the 'link' field of each element of the record array P 'points' to the subsequent element:

```
        record format F(integer DATA, record (F) name LINK)
        record (F) array P(1:1000)
        record (F) END
        integer J

        P(J)_LINK == P(J+1) for J=1,1,999
        P(1000)_LINK == END
         :
        if P(J)_LINK == END then ....
```

Note how the link field of the last record in the chain is set to point to the record END.

Arrays of reference variables are not available in IMP80.
*IMP77: Such arrays are available; see Appendix B.*

## *own*, *constant* and *external* variables

Additional properties can be given to variables by means of the prefixes **own**, **constant** (which can be abbreviated to **const**) or **external** added to the type in their declarations.

An **own** variable is allocated storage in such a way that it preserves its value between successive entries to the block or procedure in which it is declared. It can be initialised in its declaration statement. An **own** variable can be used in any circumstances in which a normal variable of the corresponding type can be used.

A **constant** variable is declared in a similar manner to an **own** variable, but it cannot be changed from its initial value. Constant variables are also known as "named constants", which better describes them, in that they have all the attributes of constants. Note that they do not have addresses (see the function ADDR, described in Chapter 6).

Wherever a constant is permitted in an IMP80 program, a "constant expression" can be used instead. A constant expression is one which can be evaluated at compile-time, i.e. its operands are constants or named constants.

Example:
```
        string (73) DELIVERY
```

can be replaced by

```
        constant integer MAXNAME=20, MAXADDRESS=52
        string (MAXNAME+1{for the newline}+MAXADDRESS) DELIVERY
```

The constant integer NL is predefined: it contains the code value for the newline character.

The constant long real PI is predefined. It is the value of pi to the long real precision of the implementation; where 64 bits are used to hold a long real, this is 3.141592653589793.

The initial values to be assigned to **own**, **constant** and **external** variables are specified when the variables are declared. A variable identifier can be followed by *=cexpr* , where *cexpr* is a constant or constant expression of the appropriate type. The variable is initialised to the value of *cexpr*.

If no initial value is specified, the value assigned by default is implementation dependent; see Appendix B.

Examples:

```
constant byte integer NUL=0, CR=13, DEL=127, FF=12

own long real  RMIN = -3.5@-4,
               RMAX = 17.23614@10
```

An **external** variable is a special form of **own** variable which is used to provide communication between sections of program compiled separately (see Section 3.4). An **external** variable can be used in any circumstances in which a normal variable of the corresponding type can be used.

Any identifier being declared as **external** may be given an "alias". The details of this facility are described in Section 3.4.

An **own**, **constant** or **external** array is initialised by appending a list of values to its declaration. Only one array may be declared per statement. Each element of the array must have a corresponding value with which it is to be initialised. In order to simplify this, each value may be followed by a repetition count in parentheses, and an asterisk, (*), may be used to represent the number of remaining elements of the array.

Examples:

```
external integer array VALUES(-3:7) = c
               17,  4, 23, -2,  3(4),  7,  1(2)

constant integer RED=1, ORANGE=2, YELLOW=4, GREEN=8,
                 BLUE=16, INDIGO=32, VIOLET=64, WHITE=127

own byte integer array COLOUR(1:22) = c
             RED, VIOLET(3), BLUE+GREEN, VIOLET, INDIGO+ORANGE+BLUE,
             YELLOW(2), WHITE(*)
```

**own**, **constant** and **external** arrays are normally one-dimensional, but need not be. If the array is multi-dimensional, the order in which the array elements are assigned the initial values is implementation dependent.

*IMP77: **own**, **constant** and **external** arrays can only be one-dimensional.*

*EMAS IMP80: in a two-dimensional array whose first element was (1,1), the order would be (1,1), (2,1), (3,1), i.e. first subscript changing fastest.*

The {...} form of comment is useful for commenting array initialisation.

Example:

```
own integer array OPCODE(0:20) =  c                           {opcode values}
           16_5800,     16_4800,     16_5000,     16_4000,
{             L            LH           ST           STH
           16_5A00,     16_5B00,     16_5C00,     16_5D00,
{             A            S            M            D
           16_1A00,     16_1B00,     16_1C00,     16_1D00,
{             AR           SR           MR           DR
           -1(*)                                              {all the rest}
```

**own**, **constant** and **external** strings can be likewise initialised with string constants or constant expressions at the time of their declaration.

Example:

```
        own string(19) FILENAME = "ERCC00.TEST"
        constant string(5)array F(0:4) = "Peter", "Mac", ""(3)
```

Records may also be declared as **own**, **constant** or **external**, but these cannot be initialised at the time of declaration.

*Variables of type* ***constant record*** *cannot be assigned at all, unless some implementation dependent method is provided; see Appendix B.*

Reference variables may also be declared as **own**, **external** or **constant**. *Initialisation of such variables, if permitted, establishes the initial reference in an implementation dependent manner; see Appendix B.*

## 2.3 Constants

Constant values can be assigned to variables. In general, the type of the constant must be the same as the type of the variable, although an automatic type conversion is carried out on a constant of *integer* type before assignment to a variable of *real* type.

### Decimal constants

Decimal constants are written in a straightforward notation:

```
        2.538        1        .25
```

The exponent, where present, consists of the symbol @ followed by an optional sign and decimal digits:

```
        -17.28@-1    1@7
```

The type of a decimal constant depends on its value. It is of *integer* type if it has no fractional part, i.e. no decimal point in its specification *and* the exponent (if present) is non-negative; otherwise it is of *real* type. The particular *real* or *integer* type depends upon the magnitude or precision of the constant.

### Base constants

A base constant may be constructed by using the prefix *decimal constant*_ to specify the base (up to a maximum of 36) of the subsequent constant. The letters A, B, ..., Y, Z are used in the constant to represent the digits 10, 11, ..., 34, 35.

Example:

```
        2_11 010        twenty six in Binary
        8_32            twenty six in Octal
        16_1A           twenty six in Hexadecimal
```

Spaces are allowed in this form of constant, and the case of any letter used (see the last example) is not significant.

An alternative form is provided for constants to bases 2, 8 and 16. The constant is written with the digits enclosed in single quotes and preceded by a code letter for the base, the codes being B for base 2 (Binary), K for base 8 (Octal) and X for base 16 (Hexadecimal).

Example:

```
        B'11010'        twenty six in Binary
        K'32'           twenty six in Octal
        X'1A'           twenty six in Hexadecimal
```

Either upper or lower case letters may be used in this form of constant, but spaces may not occur within the quotes.

*When a program is to be used on other machines, care should be taken in the use of constants as the values of the constants may vary, particularly in a transfer from a machine using ones-complement arithmetic to one using twos-complement arithmetic and vice-versa.*

Base constants are of type **integer**.

## Character constants

The ASCII code value of any character may be obtained as an integer value by enclosing the character in single quotes. When the required character is a single quote, it must be represented by two consecutive single quotes.

Examples:

```
        'A', 'a', '+', 'o', '"', '''', ' ', '
        '
```

Note the last three examples which represent the code values for single quote, space and newline, respectively. The predefined named constant NL may be used in place of the rather cumbersome form of a newline character enclosed in quotes.

The code values for several characters may be packed together to form a single integer constant by enclosing the characters in single quotes and giving the prefix M. This is known as a *multi-character constant.*

Examples:

```
        M'over', M'MAX', M'1+2', M'*@@#'
```

The value of the constant is calculated by evaluating the expression:

```
        (..(c1<<b + c2)<<b + c3)<<b + ...
```

where c1, c2... are the characters in the order specified, and b is an implementation dependent constant (commonly 8). The number of characters which can be packed into a variable of any integer type in this way is

```
        (no. of bits in the variable)//b
```

A compile-time fault normally occurs if a multi-character constant contains more characters than this.

Character constants are of type **integer**.

## String constants

A **string** constant is a sequence of characters enclosed in double quote characters, a double quote being represented inside a string constant by two consecutive double quotes. The maximum number of characters allowed in the string is implementation dependent, but is not usually less than 255.

Examples:

```
        "starting time"
        "x-y*4+2"
        "red"
        "HOOD"
```

The null string, a string of no characters, is permitted and is represented by two consecutive double quote characters ("").

A string constant of *n* characters is of type **string**(*n*).

## Named constants

These are treated in this manual as variables with the attribute **constant**; they are described along with **own** and **external** variables at the end of Section 2.2.

## 2.4 Operators and expressions

### Arithmetic operators

There are two assignment operators for use with arithmetic expressions:

| | |
|---|---|
| = | equals |
| <- | jam transfer |

Where the = operator is used, the expression on the right-hand side is evaluated and the value obtained is assigned to the destination indicated by the left-hand side, provided that the lengths and types are compatible. The program is faulty if an attempt is made to assign too large a value to a variable by use of this operator. *In most implementations an event will be signalled.*

Where the <- is used, only as many bits as will fit the location designated by the left hand side are assigned, starting with the least significant bits.

In general the arithmetic assignment instruction assigns the result of evaluating an arithmetic expression to a variable. Only the result of an integer expression may be assigned to an integer variable, but the result of an integer *or* real expression may be assigned to a real variable.

An event is signalled if an attempt is made to assign to an integer variable an integer value outwith the range which the variable can hold (see the table in Section 2.1).

The following operators may be applied to real and integer variables in arithmetic expressions:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | real division |
| // | integer division |
| \ (or ^) | real exponentiation |
| \\ (or ^^) | integer exponentiation |

The established order of precedence for the arithmetic operators is given in the following table, starting with the highest. Operators on the same horizontal line of the table have equal precedence.

| | | |
|---|---|---|
| \ (or ^) | \\ (or ^^) | |
| * | / | // |
| + | - | |

Parentheses may be used to override the natural order of evaluation of an expression or to remove ambiguity. Where operators are of equal precedence, left-hand precedence pertains as in normal mathematical usage.

Examples:

```
A-B+C    is equivalent to (A-B)+(C)
A-(B+C)                   (A)-(B+C)
A/B*C                     (A/B)*(C)
A/(B*C)                   (A)/(B*C)
A\B*C                     (A\B)*(C)
A\(B*C)                   (A)\(B*C)
```

## Arithmetic expressions

An arithmetic expression is a sequence of operators and integer or real operands obeying the elementary rules of algebra. Expressions may be real or integer according to context. Apart from the rules for operator precedence given above, no assumptions may be made about the order of evaluation of expressions.

a) Integer expressions

An expression or sub-expression is evaluated as integer if it consists of integer operands and operators. It may be converted to real, depending on the context. It is not converted to real if it is being assigned to an integer variable, or passed as an integer value parameter, or occurs in a position where an integer expression is mandatory.

All the operands and operators in an integer expression must yield integer values. The operators available for use in integer expressions are:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| // | integer division. This operator always yields an integer result. The result consists of a quotient whose sign is determined algebraically and a remainder which is ignored. Note that dividend and divisor must both be integer expressions. |
| \\ (or ^^) | integer exponentiation. This operator only operates on integer variables and always yields an integer result. The exponent must be a non-negative integer expression. |

The precision used in evaluating integer expressions depends on the operands. Variables of type **byte integer**, **short integer** and **half integer** are expanded to normal integer precision before the operation is carried out. An operation between an integer variable and a long integer variable will be carried out by long integer arithmetic.

The following standard integer functions are provided (a 'standard' procedure is one which is predefined; see Chapter 7):

**integer function** IMOD(**integer** I)

This function returns the modulus (absolute value) of the parameter.

**integer function** INT PT(**long real** L)

This function returns the integer part of the parameter L, any truncation being downwards.

Examples:
```
INT PT(-5.01) = -6
INT PT(-1.8)  = -2
INT PT(0.3)   =  0
INT PT(3.9)   =  3
```

An event is signalled if the result cannot be held in an integer variable.

**integer function** INT(**long real** L)

This function returns the nearest integer to the parameter value. Thus INT(L) = n, where $n-0.5 \leqslant L < n+0.5$ .

Examples:

```
INT(-1.8)   = -2
INT(-1.01)  = -1
INT(-0.99)  = -1
INT(0.3)    =  0
INT(1.499)  =  1
INT(1.501)  =  2
```

**integer function** TRUNC(**long real** L)

This function returns the integer part of the parameter L, any truncation being towards zero. Note the difference between TRUNC and INT PT.

Examples:

```
TRUNC(-5.01) = -5
TRUNC(-1.8)  = -1
TRUNC(0.3)   =  0
TRUNC(3.9)   =  3
```

b) Real expressions

All the operands and operators in a real expression must yield real or integer values, and assignment can only be made to a real variable. Integer values will automatically be converted into their real equivalents before being used.

The operators available for use in real expressions are:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| \ (or ^) | real exponentiation. This always yields a real result. A negative exponent, e.g. X\(-4), is evaluated as 1/(X\4). |

A real expression is evaluated to single precision until a long real variable is encountered. Thereafter the expression is evaluated to double precision. *Double precision work is time and space consuming and should only be used when strictly necessary to preserve accuracy. However, it is often required with floating-point arithmetic where loss of accuracy may occur in addition and subtraction due to cancellation of significant figures.*

The following standard real functions are provided (a 'standard' procedure is one which is predefined; see Chapter 7):

**long real function** FRAC PT(**long real** L)

The fractional part of the parameter L, i.e. L-INT PT(L), is returned as the result. Note that the fractional part is always greater than or equal to zero.

Examples:

```
FRACPT(-5.01) = 0.99
FRACPT(-4.6)  = 0.4
FRACPT(3.9)   = 0.9
```

**long real function** FLOAT(**integer** N)

The floating-point equivalent of the integer parameter is calculated and returned as the result.

**long real function** MOD(**long real** L)

This function returns the modulus (absolute value) of the parameter.

## Logical operators and expressions

Logical operations are performed on bit patterns stored in integer variables, which may be of any of the permitted lengths. Before the operation is carried out, byte, short, and half integer variables are made up to full integer length in one of two ways, according to the type of the initial variable: a) byte and half integers are made up by filling the left hand bits with zeros; b) short integers are made up by sign extension, i.e. the leftmost bit of the variable - the sign bit - is propagated leftwards until the necessary number of bits have been obtained. Where necessary, integers are made up to long integer precision by sign extension.

There are two assignment operators available for logical expressions.

| | |
|---|---|
| = | equals treats the result of the logical operation as a signed integer and attempts to perform an arithmetic assignment to the designated variable. Hence it may be wrong to attempt to put the result into a variable of the same precision as that of an operand in the logical expression. |
| <- | jam transfer copies the bit pattern of the expression indicated by the right hand side into the variable indicated by the left hand side, starting with the least significant bits and stopping when the variable has been filled. |

The following is a list of the logical operators available, excluding the assignment operators discussed above.

| | |
|---|---|
| << | left shift |
| >> | right shift |
| & | and |
| ! | or |
| !! | exclusive or |
| \ (or ~) | not |

The shift operators allow the programmer to move the bit pattern of an integer of any length to the left or right by a number of places less than the number of bits in a variable of type **integer**. Note that a shift of more than this might not cause an event to be signalled but will result in a value which is implementation dependent, *not* necessarily zero-filled.

Example:

```
        J = I>>N
```

This causes integer I to be shifted to the right the number of places specified by N and the result stored in integer J. If I or N are of less than integer precision they will be made up to integer precision, as described above, before the operation takes place.

In a left shift, bit positions vacated at the right hand end are filled with zeros and bits shifted off the left hand end are lost. In a right shift, bit positions vacated at the left hand end are filled with zeros and bits shifted off the right hand end are lost.

The operators &, !, !! are carried out on a bit-by-bit basis between the patterns stored in two integer variables. Where one operand is a long integer, the other will be made up to long integer by sign extension.

| | |
|---|---|
| *and* (&) | produces a pattern containing a 1-bit where the two source patterns both have 1-bits and containing 0-bits elsewhere. |
| *inclusive or* (!) | produces a pattern containing a 0-bit where the two source patterns both have 0-bits and containing 1-bits elsewhere. |

| | |
|---|---|
| *exclusive or* (!!) | produces a pattern containing a 1-bit where the bits in the source patterns are different and contains 0-bits elsewhere. |

These rules are summarised in the following table:

| Operands | | & | ! | !! |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

The *logical not* operator, \, operates on a single operand to invert the value of each bit; that is, 0-bits become 1-bits and vice versa. ~ can be used in place of \.

Example:

| | |
|---|---|
| If X contains the bit pattern | 01....0100110011 |
| then \X is | 10....1011001100 |
| thus \X + X is | 11....1111111111 |

Arithmetic and logical operators may occur in the same arithmetic expression. The established order of precedence, starting with the highest, is:

| | | | |
|---|---|---|---|
| \ (*logical not*) | | | |
| \ | \\ | >> | << |
| * | / | // | & |
| + | - | ! | !! |

Operators given on the same line in the above table have equal precedence.

It is syntactically incorrect to have two operators adjacent to each other in an IMP80 expression. Thus, in cases where two operators would be adjacent, e.g. A&\B, it is necessary to use brackets, e.g. A&(\B).

Example:

```
integer I, J
! Variables of type integer are assumed to have 32 bits in this example.
byte integer array B(0:3)

 :
I = ....
 :
B(J//8) = I>>(24-J) & X'FF' for J = 0,8,24
 :
```

In this example, a 32-bit integer I is copied, in groups of 8 bits, into the byte integer array B.

Note that X'FF' represents a bit pattern of eight 1s in the least significant end of the specified location and zeros elsewhere.

## String operators and expressions

There are three assignment operators available for use with strings: =, <- and ->.

Where = is used, the string expression on the right-hand side is evaluated and assigned to the string location specified on the left-hand side. The string expression must evaluate to a string constant no larger than the maximum which the left-hand string location can hold. An event will be signalled if the left-hand string location overflows.

<- is known as the "jam transfer" operator. It will assign to the location specified on the left-hand side only as many characters of the right-hand string as will fit. Any remaining characters from the right-hand end of the string being assigned will simply be omitted and no error will occur.

-> is known as the "string resolution" operator. It is used exclusively for the manipulation of strings. Its effect is described below.

Another operation exclusive to strings is "concatenation". This allows strings to be joined in a prescribed order. The strings to be concatenated are listed in the required order and separated by the symbol '.' .

Example:

```
begin
   string(100) S
   string(15) NAME1, NAME2, NAME3
   string(20) ADDRESS

   NAME1 = "Peter "
   NAME2 = "John "
   NAME3 = "Smith
"
   ADDRESS = "12 Bothwell Drive"
   S = NAME1.NAME2.NAME3.ADDRESS
   PRINTSTRING(S)
   NEWLINE
end of program
```

String S is assigned the concatenated characters of the four strings NAME1, NAME2, NAME3 and ADDRESS. Thus the printed output would read

```
Peter John Smith
12 Bothwell Drive
```

Note that the string NAME3 has a newline character in the text after the name Smith.

The only type of string expression in IMP80 is that produced by concatenation, for which bracketed expressions are neither required nor permitted.

String resolution allows a string to be searched for a specified substring of characters. If this substring does not occur in the string being resolved, an event is signalled. If, however, the substring is found on searching the string from left to right, then all characters to the left and right of the substring will be stored respectively in two string variables specified to the left and right of the substring in the resolution instruction. An event is signalled if either of the string variables is too small to store the characters being assigned to it by the resolution. The substring, which may be a string expression (a constant, a variable or a concatenation), is enclosed in parentheses with a string identifier on either side, each of these three elements being separated by '.' .

Example:

```
S = "ERCC00.FLAG"
 :
 :
S -> A.(".").B
! A now contains "ERCC00" and B contains "FLAG".
```

Thus string S has been resolved into two smaller strings, neither of which include '.' . The same exercise would be accomplished by the following:

```
S = "ERCC00.FLAG"
 :
 :
X = "."
S -> A.(X).B
```

The substring may be a concatenation:

Example:

```
FIRST NAME = "John"
SURNAME = "Smith"
NAME AND ADDRESS = "Peter John Smith, 12 Bothwell Drive"
NAME AND ADDRESS -> A.(FIRST NAME." ".SURNAME).B
```

The strings used to store the characters which occur before and after the specified substring may be omitted, in which case the characters are discarded.

Examples:

```
S = "ERCC00.FLAG"
S -> (".").B
! B now contains "FLAG".

S = "ERCC00.FLAG"
S -> A.(".")
! A now contains "ERCC00".

S = "ERCC00.FLAG"
S -> (".")
! Resolution would fail if "." did not occur in S.
! There are no other products of this resolution.
```

Since a resolution must either succeed or fail, it may be used as a simple condition.

Examples:

```
S = A.B while S -> A.(" ").B
! This statement removes all spaces from string S.

if S -> A.("/").B then T = A." or ".B else T = S
```

Note that, although the string resolution is used here as a condition, it is nonetheless *carried out* if it can be (i.e. if the condition is true). No event is signalled in this case if the resolution cannot be carried out.

There are four procedures provided for the manipulation of strings.

**byte integer map** CHAR NO(**string**(*)**name** S, **integer** N)

This map returns a reference to the Nth character of string S. An event is signalled if N is less than or equal to zero, or greater than the current length of S.

**byte integer map** LENGTH(**string**(*)**name** S)

The result is a reference to a variable containing the current length of the string S.

Example:

```
string(80) S
byte integer name L
 :
 :
L == LENGTH(S)
L = L-1 while L>0 and CHARNO(S,L) = ' '
! This example shows how to delete trailing spaces from a string.
```

**string(*)function** SUBSTRING(**string(*)name** S, **integer** I, J)

The result is the substring of S comprising the Ith to Jth (inclusive) characters of S. An event is signalled unless

```
1 <= I <= LENGTH(S)  and
0 <= J <= LENGTH(S)  and
I <= J+1
```

If I=J+1 then a null string is returned.

Note that the string parameters to CHARNO, LENGTH and SUBSTRING are all **name**-type. (See Chapter 3 for details of the different types of procedure parameter.) Thus string constants and expressions cannot be passed to these procedures.

**string(1)function** TO STRING(**integer** I)

The result is a string of length 1 comprising the character defined by the least significant byte of integer I.

## Record operators

The only record operators taking complete records, rather than record sub-fields, as their operands are the assignment operators =, <- and ==. These are described above, in Section 2.2.

The integer function SIZE OF, described at the start of Section 2.2, applies to any type of variable but is particularly useful when applied to records.

# CHAPTER 3

# BLOCKS AND PROCEDURES

## 3.1 Block structure and storage allocation

IMP80 is a block-structured language. A block is a sequence of statements of which the first is **begin** or a procedure heading, and the last is **end**. The program itself is regarded as a block, and the first **begin** encountered is interpreted as the start of the program. The statement **end of program**, rather than **end**, is used to indicate that the end of the program has been reached. Blocks may be nested to a depth determined by the particular implementation.

Within each block, variables and constants to be used must be declared at the head of the block (and before any **on event** statement), unless they have already been declared at the head of an outer block. Labels and switch labels are always local to a block; thus it is impossible to jump from one block to another. Keywords which occur in pairs such as **cycle ... repeat** must have both elements within the same block.

Example:

```
begin
   integer I, J
   real array A(1:10)
    :
   real X, Y
    :
    :
   begin
      real Z, P
      I = 4
      J = 10
       :
       :
   end
    :
    :
end of program
```

I and J have been declared at the head of the outermost block and may thus be referred to from any inner blocks of the program.

If, however, a variable is declared which has the same identifier as a variable already declared in an outer block of the program, then use of that identifier will refer to the variable most recently declared.

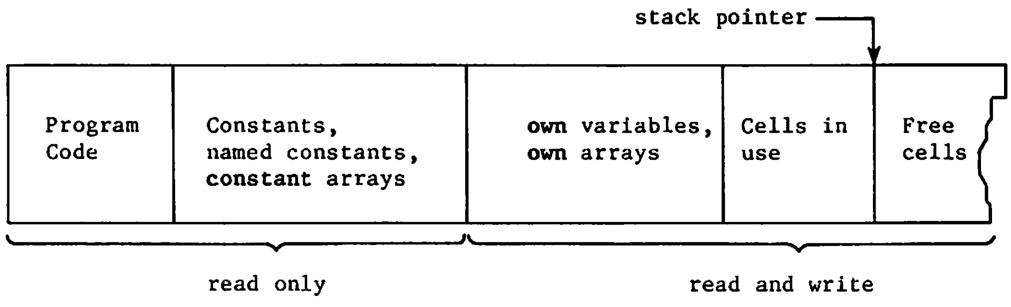
Example:

```
begin
   integer I, J, K
   real X, Y
   I = 23
    :
    :

   begin
      real I
      I = 4.106
       :
       :

      begin
         X = I
          :
          :
      end

   end

end of program
```

I is first declared as an integer type and will be allocated storage accordingly. The first use of I refers to this integer location. Within the next block, however, I is redeclared as a real variable. Now space is allocated for a real variable and within this block (and any deeper blocks of the program - unless I is again redeclared) use of I refers to the real location and the integer location of the outer block remains untouched.

When a variable is accessible it is said to be *in scope*. A variable is *global* to a block within the block in which it is declared if it is accessible to the inner block. In the example above, X, Y, J and K are global to both inner blocks.

The redeclaration of variables is permissible because of the way in which storage space is allocated to a program. Each program can be considered to have space allocated to it on what is called the *stack*. The stack is an area of store and the stack space given to each program has the layout shown in the following diagram.


stack pointer
Program Code
Constants, named constants, constant arrays
**own** variables, **own** arrays
Cells in use
Free cells
read only
read and write

*Note that this description of a stack is for explanatory purposes only - it may not be accurate for specific implementations.*

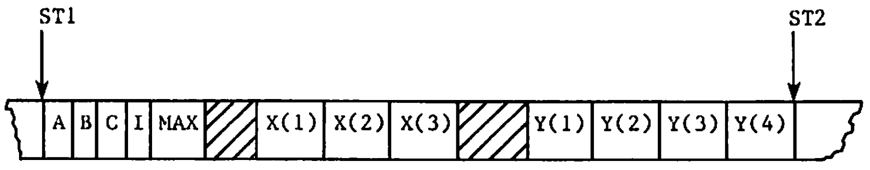
The "read only" area contains the object code of the program as produced by the compiler, and also any of the named constants declared by the program. This area cannot be altered during execution of the program. The "read and write" area has own variables stored first and thereafter space is allocated according to the requirements of the program. Each cell will be as big as required by the entity stored in it: thus an integer variable occupies only a small cell, whilst a complex record variable may require a very large cell.

The stack pointer indicates the next free location in store.

The following example illustrates the stack mechanism.

Example:
```
        begin
           real A,B,C
           integer I,MAX
           real array X(1:3), Y(1:4)
```

As a result of the above declaration, the stack might look like this:



ST1 is the position of the stack pointer before the **begin**, and ST2 is its position after the declarations. The shaded areas indicate portions of store which contain information essential to the program (such as array dimensions) but which are not used directly by the programmer.

The stack pointer may be advanced by any further declarations or by activity initiated by the instructions of the program. On entry to a new block or procedure, the stack pointer is advanced as necessary to cope with new declarations, etc. On exit from the block (or procedure) it returns to its last position prior to entry to that block (or procedure).

Storage space for fixed variables such as **real** or **integer** types is determined at compile time, but arrays with dynamic bounds cannot be allocated space until the values of the bounds are determined at run time.

Since the block structure of a program is so closely related to the allocation of store, skilful use of blocks can lead to economical use of store. Consider the following examples.

Example:

```
begin
   integer N

   cycle
      READ(N)
      exit if N=0

      begin
         integer I
         integer array A(1:N)
         READ(A(I)) for I=1,1,N
          :
          :
      end

   repeat

end of program
```

The required size of the array is read in the outer block and the array itself declared in the inner block. Thus the space used by any one set of data will be recovered when the inner block is left, so allowing one to repeat the process without incurring successively increasing demands for storage space.

Example:

```
begin
    :
    :

   begin
      real array XYZ(1:5000)
       :
       :
   end

    :
    :

   begin
      integer array IJK(1:20, 1:250)
       :
       :
   end

    :
    :
end
```

Since the declarations at the head of a block are cancelled on executing the **end** of the block, it is often possible to economise on storage space if a program consists of several distinct tasks, each requiring large amounts of space. The above example illustrates the point. Procedures can be used in a similar way to economise on store.

## 3.2 Events

During the execution of a program, events may occur which normally cause the program to terminate with an error message. However, there is a mechanism which allows events to be intercepted and used to control the subsequent execution of the program. This mechanism is activated by the use of the **on event** statement.

The **on event** statement (which may occur only once in any block) is used to introduce a group of statements which is only executed if one of the specified events occurs during execution of the code in the block including the **on event** statement. The form of the **on event** group of statements is:

```
on event nlist start
   executable instructions
finish
```

The keyword **event** may be omitted.

*nlist* is a list of integer values in the range 1 to 14 inclusive, where each number refers to a specified class of error, as follows:

| | |
|---|---|
| 1 | Overflow |
| 2 | Excess resource |
| 3 | Data error |
| 4 | Invalid data |
| 5 | Invalid arguments |
| 6 | Out of range |
| 7 | Resolution failure |
| 8 | Undefined value |
| 9 | I/O error |
| 10 | Library procedure error (e.g. SQRT negative) |
| 11-14 | Available for programmer definition |

*Variations to this list and details of related facilities for specific implementations are given in Appendix B.*

Up to 255 sub-events may be defined for each event, but these cannot be specifically intercepted and are necessarily implementation dependent. For example, not all machines distinguish integer overflow from real overflow.

*Sub-events defined for some implementations are listed in Appendix B.*

The **on event** group must follow the declarations at the head of a block and may be regarded as the last declaration of the block. The code within the **start ... finish** is not executed by entry through the head of the block, but is jumped to on the occurrence, during the execution of the block, of an event referenced by the event list. Following such a jump, the flow of control is determined by the contents of the **on event** group; the program does not resume at the point of the failure.

In addition to events such as "integer overflow", "resolution failure", etc. occurring, an event can be forced to occur by the programmer, by use of the instruction

**signal event** *const*, *exprn*

where *const* specifies the event required and *exprn* is an optional integer expression (evaluating to an integer within the range 0-255) which may be used to specify sub-event information.

The use of the **signal event** statement is the only way of causing a user-defined event (i.e. one which is not predefined by the implementation) to occur, although it can also be used with the predefined events (1-10).

If an event occurs (or is caused by a **signal event** statement) in an **on event** group which includes the occurring event in its event list, a branch is not made to the head of that group, since such a branch would probably cause looping. Instead, the event is traced up the stack through each superior block until either an **on event** statement including the occurring event in its list is found, or the user environment is left. If a suitable **on event** statement is found, control is transferred to its **start ... finish** group.

In parallel with these language statements, the following standard integer function is provided to enable the programmer to determine further information when an event occurs. It may only be meaningfully called in a block which has an **on event** statement within it.

**integer function** EVENT INF

This function returns

```
(event no<<8) ! sub-event no
```

for the last event which has occurred.

If an event is not intercepted in the block in which it occurs, then it is traced up the stack through each superior block until either a suitable **on event** statement is encountered or the user environment is left, a diagnostic package (if one is provided in the implementation) being entered in the latter case. When a suitable **on event** statement is encountered in an outer block, program control is transferred to its **start ... finish** group.

As a result of these facilities it follows that, for example, "input ended" may be detected and dealt with from within an external routine or a routine within a main program.

Examples:

*System defined events*

```
integer SUBCLASS
constant string(21)array MESSAGE(1:2) = "Capacity exceeded",
                                        "Array bounds exceeded"
on event 6 start
   SUBCLASS = EVENT INF&255; ! Mask off event number.
   if 1<SUBCLASS<=2 then PRINTSTRING(MESSAGE(SUBCLASS)) else c
     PRINTSTRING("Invalid subclass")
   NEWLINE
   ->ERROR EXIT
finish
 :
 :
ERROR EXIT: ....
```

*User defined events*

```
integer SUBEVENT
on event 12 start
   PRINTSTRING("Event 12 has been intercepted, with subevent")
   WRITE(SUBEVENT, 1)
   NEWLINE
   ->EVENT 12
finish
 :
 :
SUBEVENT = 2
 :
 :
signal event 12, SUBEVENT
 :
 :
EVENT 12: .....
```

```
          on event 10 start

             if EVENT INF&255#8 start; ! enter start/finish if subevent is not 8
                signal event 10, EVENT INF&255
                ! Thus all subevents except 8 are passed to outer block
             finish
              :
              : ! Code to deal with subevent 8 of event 10
              :

          finish
           :
           :
```

## 3.3 Procedures

A procedure takes the form of a block in which the first **begin** is replaced by a procedure heading. However, a procedure can only be entered by execution of a *procedure call* statement, whereas a block is entered when the **begin** statement at the start of the block is executed.

There are three forms of procedure - **routine**, **function** (or **fn**) and **map**.

Example:

```
          begin
             integer array A(1:40)
              :
              :

             routine CLEAR A
                integer I
                A(I) = 0 for I=1,1,40
             end {Of routine CLEAR A.}

              :
             CLEAR A
              :
          end of program
```

In this example, a procedure - routine CLEAR A - is described and then used by means of the procedure call statement 'CLEAR A'.

CLEAR A is effectively a named block: if the procedure heading were replaced by **begin** and the procedure description moved down to take the place of the procedure call statement, the effect would be exactly the same. However, by making the block a routine - and thus giving it a name - it is possible to call it at different places in the program without repeating the description each time.

In addition, procedures can have *parameters*, passed to them via a parameter list enclosed in brackets in the procedure call statement.

Example:

```
          begin
             integer F, T
             string(31) array A(0:99)

             routine STRINGSORT(string(*)array name X, integer FROM, TO)
                integer L, U
                string(255) D
                 :
                 :
             end {Of routine STRINGSORT.}

              :
              :
             STRINGSORT(A, F, T)
              :
              :
          end of program
```

In the above example, the procedure heading specifies the form of procedure (**routine**) being described, gives the procedure an identifier (STRINGSORT) and describes the number, order and types of variables passed as parameters, using dummy names. The procedure is entered when the call statement STRINGSORT(A, F, T) is executed.

The first line of a procedure has one of the following forms:

**routine** *name* (*parameter list*)

*type* **function** *name* (*parameter list*)

*type* **map** *name* (*parameter list*)

where *type* is one of the arithmetic types, or **string**(*n*) or **record**(*format*), and (*parameter list*) is optional.

A procedure must be declared before it is called, unless it is a 'standard' procedure; see Chapter 7. If the procedure itself is placed at the head of a block (as above) no further declaration is needed. Otherwise a specification (**spec**) statement must be placed amongst the other declarations at the head of a block, with the procedure included later at the same textual level. The **spec** statement is exactly like the procedure heading with **spec** inserted after the keyword **routine**, **function** or **map**.

Example:

```
routine spec STRINGSORT(string(*)array name X, integer FROM, TO)
```

It is most important to give the specification accurately: in particular, the number and types of the parameters must be correct. However, the names of the parameters in a specification statement are not significant.

*IMP77: the identifiers of parameters in a **spec** statement must be distinct.*

Variables used in procedures may be declared locally (as is the case for L, U and D in the above example), but any information stored in them becomes inaccessible on exit from the procedure. If the information calculated by the procedure is to be preserved for use on subsequent entries, it must be stored in global variables or **own** variables declared locally.

Global variables should be used with care in procedures. Note that such variables must be declared globally to the procedure itself: it is not sufficient that they be declared globally to the call(s) of the procedure.

Example:

```
begin
   integer X, Y

   routine CONVERT
      X = ....
      Y = ....
   end

   CONVERT
    :
    :

   begin
      integer X
       :
       :
      CONVERT
       :
       :
   end

    :
    :
end of program
```

The routine CONVERT, which has no parameters, operates on variables X and Y. The first call of the routine uses X and Y as declared at the head of the program. The second call occurs from within an inner block in which X has been redefined. However, the procedure again uses the X declared at the head of the program, ignoring the redefined X of the inner block.

Information calculated by a procedure and stored in global variables is, of course, accessible on exit from the procedure.

There are three categories of procedure which may be called by a program:

1) *standard* procedures, which are automatically available to all programs (see Chapter 7); for example, READ, INT PT;

2) procedures described within the program;

3) *external* procedures, which are compiled separately from the program (see Section 3.4).

Procedures can be nested: that is, a procedure can be defined inside another procedure. The scope rules apply as before.

Procedures can be used recursively: that is, a procedure can be called from within itself. The example given below of a sorting program demonstrates the recursive use of routine STRINGSORT. Obviously, some criterion within the body of the procedure must eventually prevent the procedure calling itself endlessly. In the example, the recursive calls of STRINGSORT are embedded in conditional instructions, thus providing the necessary opportunity to stop the recursion process.

**Parameters**

When a procedure with parameters is described, the procedure heading contains dummy names for the parameters to be passed. These dummy names, known as *formal parameters*, are allocated local storage within the procedure, and are used within the procedure as though they were local variables.

When a procedure is called, the *actual* parameters are specified in the call statement; they must correspond in number, order and type with the formal parameters in the procedure heading.

Parameters are passed either *by value* or *by name*.

When a parameter is passed by value, the local storage in the procedure allocated to the corresponding formal parameter is initialised to the *value* of the actual parameter, which can thus be an *expression* of the appropriate type.

Example:

```
begin
   real Z, Y

   real function AUX(real X) ; ! Call by value: X is a formal parameter.
        :
        :
   end

    :
    :
   Y = AUX(Z) ; ! Z is an actual parameter.
    :
    :
   Y = AUX(4.5*Z) ; ! 4.5*Z is an actual parameter.
    :
    :
end of program
```

Function AUX has formal parameter X of type **real**. Execution of the first call of the function will assign to X the value to be found in Z. On executing the second call, the expression 4.5*Z will be evaluated and the result assigned to formal parameter X.

Parameters passed "by name", however, are treated differently. The local storage allocated for a parameter passed by name is a reference variable of the appropriate type. In this case, when a procedure is executed, the effect is that the local variable is *pointed at* the actual parameter, which must therefore be a variable (or reference to a variable) of the appropriate type, and not an expression. Thus, every reference in the procedure to the formal parameter is treated as if it were a reference to the actual parameter. Parameters passed by name can be used to preserve information calculated by the procedure, for use on exit.

Note that arrays can *only* be passed by name.

Example:

Given a procedure with heading

```
routine ALPHA(integer BETA, real name GAMMA)
```

the call

```
ALPHA(J*K+4, R(M))
```

implicitly causes the following assignments to be carried out on entry to the procedure:

```
BETA = J*K + 4
GAMMA == R(M)
```

(Note that the normal scope rules do not apply to these assignments, because they use variables in scope at the point of the call to assign to variables local to the routine body.)

Example:

```
begin
   constant string (1) SNL = "
"
   string(31)array NAMES(1:99)
   routine spec STRINGSORT(string(*)array name X, integer F, T)
   integer I, N

   READ(N) until 1 <= N <= 99; ! Reject value if not in range.
   READSTRING(NAMES(I)) for I=1,1,N
   STRINGSORT(NAMES,1,N)
   PRINTSTRING(NAMES(I).SNL) for I=1,1,N

   routine STRINGSORT(string(*)array name X, integer FROM, TO)
      integer L, U
      string(255) D

      return if FROM >= TO
      L = FROM; U = TO
      D = X(U)

      cycle
         L = L+1 while L < U and X(L) <= D
         exit if L = U
         X(U) = X(L)
         U = U-1 while U > L and X(U) >= D
         exit if U = L
         X(L) = X(U)
      repeat
      ! Now L = U.

      X(U) = D
      L = L-1; U = U+1
      STRINGSORT(X,FROM,L) if FROM < L
      STRINGSORT(X,U,TO) if U < TO
   end

end of program
```

This program sorts a set of strings held in array NAMES. Note the **routine spec** statement at the head of the block, with the routine description occurring later at the same textual level.

There are three parameters passed to the **routine** STRINGSORT; the two integers are passed by value, and the string array is passed by name. (Arrays can only be passed by name.) When the routine is called the first time the parameters are treated as follows. The string array NAMES is passed by name and thus all references to the formal parameter X within the body of the routine become references to NAMES; the actual value 1 will be assigned to formal parameter F; and the value stored in N will be assigned to formal parameter T. On exit from the routine, NAMES will have its elements sorted, but the value of N will be unchanged.

Strings may be passed as parameters to procedures. Where a string name parameter is used, the length specified in the procedure heading can be replaced by * so that strings of any length (up to the allowed maximum) can be passed to that procedure. *In this situation run-time overflow checking is applied to the actual string passed to the procedure.*

In IMP80, parameters called by name are assigned at the time of call. Thus if a routine with parameter list (**real name** X, **integer name** I, ...) were called with parameters (A(J), J, ...) where A is the name of a previously declared real array, then on execution of the procedure every reference to X will refer to the element of A determined by the value of J on entry, no matter how J varies during execution of the procedure.

Procedures, too, may be passed as parameters:

Example:

```
begin
   routine ONE(routine PARAM(integer X) )
       :
       :
   end

   routine TWO(integer P)
       :
       :
   end

   routine THREE(real X)
       :
       :
   end

   ONE(TWO)
end
```

Routine ONE has a single parameter, a routine with a single parameter of type **integer**.

Note that routine THREE cannot be passed as parameter to routine ONE because THREE has a parameter of type **real**. That is, the parameter list of the actual procedure passed must correspond with the parameter list of the corresponding formal procedure parameter.

A formal parameter can be any of the following types:

1) any arithmetic type (e.g. **long real**, **byte integer**)

2) **string**(n)

3) **record**(format)

4) any of the above followed by **name** or **array name**

5) any type of procedure (i.e **routine**, **function** of any type, **map** of any type)

Items (1) - (3) correspond to call by value; items (4) and (5) to call by name. The actual parameter in a call by value must be an expression of the appropriate type; in a call by name it must be a "reference" to an entity of the appropriate type.

*EMAS IMP80: those standard procedures which are "intrinsic" (see Appendix B) cannot be actual parameters.*

Note that arrays can only be passed by name, not by value.

The three different forms of procedure will now be examined in more detail.

**Routines**

A routine call may be used exactly like an instruction. When the call is executed, control is transferred to the routine, which executes until either the **end** statement is reached or a **return** statement is encountered. Flow of control is resumed at the statement after the routine call.

Example:

```
 :
 :
integer X,Y

routine CONVERT
   if X < Y start
      X = X+Y
   finish else start
      X = X-Y
   finish
end

 :
 :
CONVERT
 :
 :
CONVERT unless X = 0
 :
 :
```

Note that CONVERT uses global variables X and Y and that the result is stored in X on exit from the routine. Note also the use of CONVERT in a conditional statement.

**Functions**

A function calculates a value of the specified type (integer, real, string or record), and may be used in an expression exactly like an operand of that type. The function terminates when an instruction of the form

**result** = *expression*

is executed, and the value of the expression, which must be of the same type as that of the function, is returned to the statement making the call. The result statement may not be used in an inner block within the function.

Example:

```
integer function SUMSQ(integer A, B)
   result = A**2 + B**2
end

integer X, Y, Z
 :
 :
Z = SUMSQ(X,Y) - 3
 :
```

Unusual side effects may result from changing the values of global variables inside functions. Such side effects are often implementation dependent.

Example:
```
        begin
           integer I, J, K

           integer function SIDE
              I = I+1
              result = J
           end

            :
            :
           K = I+SIDE
            :
            :
        end
```

In the statement K = I+SIDE, there is no defined order of evaluation of operands in the expression on the right hand side of the assignment, but the value of I used in the expression depends upon the order of evaluation. Thus the value of the expression is indeterminate. The actual value computed depends on the implementation of IMP80 used.

Functions may be of **string** type; in this case the maximum length of the string which may be returned by the function is included in the specification and heading of the function.

Example:
```
        string(20)function FIELD(integer I)
```

Functions may be of **record** type; in this case the format of the record returned must be included in the specification and heading of the function.

Example:
```
        record format RFA(integer ONE, TWO, THREE)

        record (RFA) function TRIPLES(integer ITEM)
```

*EMAS IMP80: the format associated with a record function may not exceed 256 bytes in length.*

## Maps

A map (or "mapping function") calculates a reference to a variable of the specified type (integer, real, string or record) and may be used exactly like a variable of that type. The map terminates when an instruction of the form

**result** == *reference to a variable of the same type as the map*

is executed and the address of the given variable is returned to the calling instruction. The "reference to a variable" on the right hand side of the **result** statement can be a normal variable, a reference variable or a mapping function call.

Example:
```
        integer X, Y, K

        integer map MIN
           if X < Y then result == X else result == Y
        end

        MIN = 0
        ! This statement is exactly equivalent to
        !       if X < Y then X = 0 else Y = 0
         :
        K = MIN
         :
```

Note the use of a map on the left-hand side or right-hand side of an assignment statement.

Where a map is of **string** type, the specification and heading must include a length - the precise maximum length of any string to which the map may refer.

Example:

```
          string(3)map XA(integer I)
```

If a string map might refer to strings of varying length, then the procedure heading and declaration may have the symbol '*' in place of a specific length. The map may then reference strings of any length up to the allowed maximum.

Example:

```
          string(*)map XA(integer I)
```

Where a map is of **record** type, the specification and heading must include a record format. The right hand side of a **result** statement within a record map must refer to a record of the same type as the map, or to the standard record map RECORD (described in Chapter 6).

Example:

```
          record format RF(integer X, string(10) TITLE)

          record (RF) map RM(integer I, J)
             record (RF) name CURRENT
               :
               :
             CURRENT == ...
               :
               :
             result == CURRENT if ...
               :
          end {Of record map RM

          string(15) HEAD
          integer P, Q
           :
          HEAD = RM(P,Q+1)_TITLE
           :
          RM(17,3)_X = 24
           :
```

*IMP77: the record format given in the specification or heading can be replaced by (*), meaning that the reference returned by the map may be to a record of any format; the actual record format used depends on the context.*

A number of standard maps are provided. These are described in Chapter 6.

## 3.4 External linkage

A complete program may be divided into separately compiled modules which are linked before (or possibly during) program execution. This section describes the language facilities provided for setting up or accessing a separately compiled module.

A procedure compiled separately from a program which uses it is called an *external procedure*. If a program uses an external procedure it must contain an external specification statement. This is of the same form as the **spec** statement described in Section 3.3, but with the prefix **external**.

Example:

```
          external string(*) function spec WEEKDAY(integer NO)
```

An **external...spec** statement has the same effect as a normal **spec**, except that there is no description of the procedure later in the program. **external ... spec** statements may be given wherever other **spec** statements would be valid.

The keywords **system** and **dynamic** may be used in place of **external**; refer to the relevant system documentation for details of the effects of these keywords.

External variables are also available. If a program uses an external variable it must include an external specification. This is of the same form as the variable declaration statement but with the keywords **external** and **spec** added.

Examples:

```
          external integer spec WAIT, CHOICE

          external real array spec MEAN(-6:6)
```

## External files

A file of external procedures and variables may be compiled. Such a file differs from the structure of a program file (described in Section 1.2) in several respects:

* There is no initial **begin**.
* **end of program** is replaced by **end of file**.
* Variables declared outside any procedures must be **own**, **constant** or **external** (described below).
* The first statement of any procedure description within an external file can be preceded by the keyword **external**; such a procedure can then be made accessible to other programs, as explained above. If a procedure in the file is not **external** then it is accessible only within the file, in accordance with the normal scope rules.
* **begin/end** blocks are not allowed, except within procedures.
* Where an **external ... spec** statement in the file specifies an external procedure or variable described later in the same file, the keyword **external** may be omitted.

An **external** variable has all the properties of an **own** variable, but is declared with the keyword **own** replaced by **external**.

Examples:

```
          external integer CHOICE = 6, WAIT = -5

          external real array MEAN(-6:6) = 2.7(5),0.3,1.5(*)
```

External variables can be declared in an external file or in a normal program file, wherever other declarations are valid. *They are normally declared in the outer block of an external file.*

Note that external variables may be initialised, like **own** variables, when they are declared, but not when they are specified in an **external ... spec** statement.

Example of an external file:

```
          external integer IN=0, OUT=0

          routine GET(integername SYM)
             READSYMBOL(SYM)
             IN = IN+1
          end

          routine PUT(integer SYM)
             PRINTSYMBOL(SYM)
             OUT = OUT+1
          end
```

{Continued on next page}

```
        external routine PROCESS
           integer CH
            :
           GET(CH) until CH='*'
            :
           PUT(CH)
            :
        end {Of PROCESS}

        end of file
```

A program making use of the external file:

```
        begin
           external routine spec PROCESS
           external integer spec IN, OUT
           integer DATA, DMAX
            :
            :
           for DATA = 1,1,DMAX cycle
              IN = 0; OUT = 0
              PROCESS
              PRINTSTRING("Calculation no.")
              WRITE(DATA,2)
              PRINTSTRING(":  ")
              WRITE(IN,1)
              PRINTSTRING(" characters in;")
              WRITE(OUT,1)
              PRINTSTRING(" characters out.")
              NEWLINE
           repeat
            :
            :
        end of program
```

### *alias*

Any identifier given in an external declaration or specification may be followed by **alias** *string constant*, where the string constant specifies the string to be used for external linkage. This has no effect on the use of the identifier within the program.

Examples:

```
        external long real fn spec EIGENVALUE alias "D#REFEIGEN$" c
                                          (long real array name MAT)

        external integer RESULT alias "ICL9CERETURN" = 4
```

Note that the string constant *cannot* be a named constant or constant expression.

# CHAPTER 4

# EXECUTABLE STATEMENTS

## 4.1 Conditions

In IMP80 a *condition* can form part of any executable statement. Conditions are therefore described before the various types of executable statement.

The commonest form of simple condition in IMP80 is made up of two expressions separated by a *comparator*. The expressions are evaluated and compared. The condition is true if the relation specified by the comparator holds. The expressions must yield values of the same type; the only exception to this rule is that an integer expression can be compared with a real expression. Complete records cannot be compared; this restriction *includes* comparison with 0.

Examples:

```
J = 0
A = "END"
X+23.7 <= F(J+2,Y) - 2*Z/P
REF == K
```

The comparators are:

| | |
|---|---|
| = | is equal to |
| # (or \= or <>) | is not equal to |
| < | is less than |
| <= | is less than or equal to |
| > | is greater than |
| >= | is greater than or equal to |
| == | refers to the same variable as |
| ## (or \==) | does not refer to the same variable as |

In the case of the last two comparators, == and ##, the items being compared are references to variables, which must be of identical type. The condition is true if the addresses of the variables referred to are equal (==) or not equal (##). (Note that the address of a reference variable is the address of the variable to which it refers; see Chapter 6.)

The other forms of simple condition are as follows:

a) *expression comparator expression comparator expression*

This is known as a double-sided condition.

Example:

```
A+B <= C+D < E+F
```

This condition is true if A+B <= C+D *and* C+D < E+F .

The second expression is only evaluated once. The third expression is only evaluated if the condition between the first two expressions is true.

The comparators == and ## (or \==) may not be used in double-sided conditions.

b) String resolution

Example:

```
A -> B.(C).D
```

The resolution is attempted. If it succeeds the condition is true *and* the resolution is performed. If it fails the condition is false, but no event is signalled.

c) Compound condition (see below) enclosed in brackets

Example:

```
(A>0 or B<=0)
```

d) Any of the above forms preceded by the keyword **not**

The effect of preceding a simple condition with **not** is to reverse the truth value of the simple condition.

Examples:

```
not A <= 0       (equivalent to A > 0)

not 23 <= I+J <= 99 (equivalent to I+J < 23 or I+J > 99)
```

*Compound conditions* can be produced by combining simple conditions using the keywords **and** and **or**:

*simple condition* **and** *simple condition* **and** ....

*simple condition* **or** *simple condition* **or** ......

Examples:

```
A+B <= C+D and C+D < E+F

I = 20 or A -> B.("..").C or X > Y >= Z+3.47
```

It is *not* valid to combine **and** and **or**, as in

```
X < Y and B = C or D = 24
```

However a compound condition enclosed in brackets is treated as a form of simple condition (see above). Thus

```
(X < Y and B = C) or D = 24
```

is valid. Note that the form

```
not (X < Y and B = C)
```

is also permitted.

By combining **and** and **or** and brackets, conditions of arbitrary complexity can be produced:

Example:

```
(A <= B or C == D) and S -> ("Jim").T and (X <= Y <= Z or (P_K = 23 and Q < 0))
```

The testing of conditions proceeds from left to right, simple condition by simple condition, terminating any clause as soon as an inevitable outcome for the clause has been established. Thus, in the example above, if A <= B were true then C == D would not be evaluated; if A <= B were false and C == D were false then the remainder of the condition would not be evaluated.

Example:

```
        A = 0 or B/A = C
```

If the variable A has the value 0 the whole condition will be true without B/A = C being tested.

```
        B/A = C or A = 0
```

In this case an event will be signalled ("overflow") if variable A has the value 0.

## 4.2 Instructions

An *instruction* is an imperative statement which may be made conditional. The following IMP80 statement types are instructions:

| *statement* | *described in* |
|---|---|
| assignment | Chapter 2 |
| routine call | Section 3.3 |
| **monitor** | Section 4.2 |
| **signal event** | Section 3.2 |
| **return** | Section 3.3 |
| **result=** | Section 3.3 |
| jump | Section 4.2 |
| **stop** | Section 4.2 |
| **exit** | Section 4.4 |
| **continue** | Section 4.4 |

In addition, a *compound instruction* can be formed by use of the keyword **and**:

*instruction* **and** *instruction* **and** ......

Example:

```
        X = 23 and continue
```

The last of the series of instructions in a compound instruction can be any of those listed above; the other component instructions can only be assignments, routine calls or **monitor** statements.

### Conditional instructions

An instruction can be made conditional as follows:

**if** *condition* **then** *instruction*

or

**unless** *condition* **then** *instruction*

In the first form, the instruction is executed if the condition is true; in the second, the instruction is not executed if the condition is true. If the instruction is not executed, nothing is done.

Examples:

```
        if 0 < I <= 9 and K > 0 then B(I) = K

        unless J = I then A(I,J) = 0
```

**if** and **unless** statements can be elaborated to allow specification of an alternative instruction, to be executed if the first one is not:

**if** *condition* **then** *instruction* **else** *conditional instruction*

**unless** *condition* **then** *instruction* **else** *conditional instruction*

Example:

```
if X < 47.2 then Y = Z+3 else Y = 0.0
```

Note that the second instruction in these elaborated **if** and **unless** statements can be conditional.

Example:

```
if STATE < 0 then ERROR = IN else c
    if STATE = 0 then ERROR = CALCULATION else ERROR = OUT
```

The simple forms of **if** and **unless** statements (i.e. those without the **else**) can be made simpler still:

*instruction* **if** *condition*

*instruction* **unless** *condition*

Examples:

```
B(I) = K if 0 <= I <= 9 and K>0

A(I,J) = 0 unless J = I
```

## Labels and jumps

Any statement, excluding declarations, may be given one or more labels. A label is an identifier but it is not declared; however, it does have to be distinct from other local identifiers.

Each label is located immediately to the left of the statement to which it refers, followed by a colon.

Examples:

```
NEXT: P = P+1 if P < 0

ERR1: ERR2: FAULTS = FAULTS+1
```

Control is passed to a labelled statement when a jump instruction of the form

-> *label*

is executed.

Examples:

```
->NEXT

if DIVISOR = 0 then ->ERR1
```

As indicated, jump instructions can be made conditional (as can all instructions).

## Switches

A set of labels, known as a *switch*, may be declared in a manner similar to a one-dimensional array, but using the keyword **switch**. The switch must have bounds which can be determined at compile-time (i.e. constants, or integer expressions comprising constants and named constants).

Examples:

```
switch TYPE(4:9)

switch S1, S2(1:10), S3('A':'Z')
```

Once declared, switch labels may be located in the same way as simple labels, the particular label of the switch being specified by an integer constant.

Example:

```
          switch SW(4:9)
          constant integer FAULTY = 6
           :
           :
          SW(4): CHECK VALUE(1)
           :
          SW(FAULTY): ERROR FLAG = 1
           :
           :
          LAST: SW(9): ! All finished.
           :
```

Control is passed to a statement labelled by a switch label when a jump instruction of the form

-> *switch* ( *integer expression* )

is executed.

Not all of the labels in the switch need be located; in the example above SW(5), SW(7) and SW(8) are not, and do not need to be elsewhere in the program. An event is signalled if an attempt is made to jump to an undefined switch label.

Instead of an integer constant, an asterisk (*) may be used when locating a switch label. This has the effect of defining any labels in the switch not defined elsewhere in the program block.

Example:

```
          switch LET('a':'z')
           :
           :
          LET('a'):LET('e'):LET('i'):LET('o'):LET('u'):
          ! Deal with vowels here.
           :
           :
          LET(*): ! All the rest, i.e. consonants
           :
```

Notes

* A label can appear on a line without being followed by a statement. This is sometimes done to improve readability. *Strictly, the label is labelling a null statement.*
  Example:
  ```
            LAST STAGE:
                      if Y < 23.7 and ....
  ```

* The use of both types of label is limited to the block in which they are defined, *excluding* any blocks described therein. That is, labels cannot be global to a block and therefore it is not possible to jump into or out of a block.

* The identifiers used for labels must be distinct from other local identifiers.

* It is wrong to jump into a **for** loop (see Section 4.4); the effect is implementation dependent.

### *stop*

This instruction causes the execution of the program to be terminated.

### *monitor*

This instruction passes control to a run-time diagnostic package which then generates a trace of the state of the program. *In implementations of IMP80 without a run-time diagnostic package* **monitor** *is a null instruction. In some implementations the amount of trace information can be controlled by the programmer.*

Following the trace the program resumes at the point where it left off.

## 4.3 *start/finish*

**start/finish** statements are used to make the execution of a *group* of statements conditional. The most general type of conditional group is a sequence of statements of the form:

> **if** *condition 1* **then start**
>
> statements to be executed if *condition 1* is true
>
> **finish else if** *condition 2* **then start**
>
> statements to be executed if *condition 1* is false and *condition 2* is true
>
> **finish else if** *condition 3* **then start**
>
> :
> :
>
> **finish else start**
>
> statements to be executed if all the previous conditions are false
>
> **finish**

Any or all of the statements containing the keyword **else** may be omitted. When they are all omitted, the form becomes:

> **if** *condition 1* **then start**
>
> statements to be executed if *condition 1* is true
>
> **finish**

Other simple forms:

> **if** *condition 1* **then start**
>
> statements to be executed if *condition 1* is true
>
> **finish else start**
>
> statements to be executed if *condition 1* is false
>
> **finish**

> **if** *condition 1* **then start**
>
> statements to be executed if *condition 1* is true
>
> **finish else if** *condition 2* **then start**
>
> statements to be executed if *condition 1* is false and *condition 2* is true
>
> **finish**

Notes

* "**if** ... **start**" and "**finish else if** ... **start**" are complete statements in their own right and as such must be terminated by a newline or semicolon.

* Each **start** and the next **finish** effectively bracket the statements between them, which are all controlled by the same set of conditions.

* **start/finish** groups may be nested to any depth.

* **then start** may be elided into **start.**

* **finish else start** may be elided into **else.**

* If a **start** and the next **finish** enclose only one instruction then the complete **start...finish** sequence can be replaced by that instruction.

  Example:

  ```
   :
  else if K = 0 then start
     X = 4*SQRT(Y**2 + Z**2 - 4)
  finish else start
   :
  ```

  can be written as

  ```
   :
  else if K = 0 then X = 4*SQRT(Y**2 + Z**2 - 4) else start
   :
  ```

* The keyword **if** may always be replaced by **unless**, with the effect of negating the subsequent condition. Thus the following two statements are equivalent:

  ```
  if X = 0 then Y = 1 else Z = -1

  unless X = 0 then Z = -1 else Y = 1
  ```

* Where a small group of unconditional instructions is to be made conditional, a compound instruction (Section 4.2) can be used instead of a **start/finish** construction.

  Example:

  ```
  if K<0 then X = 4 and return
  ```

  is equivalent to

  ```
  if K<0 start
     X = 4
     return
  finish
  ```

## 4.4 *cycle/repeat*

**cycle** and **repeat** statements are used to bracket statements which are to be repeated. In the simplest case, a group of statements may be repeated indefinitely by enclosing them between the statements **cycle** and **repeat.**

Example:

```
cycle
   READ DATA; ! Get next set of data
   PROCESS DATA; ! Carry out calculation
   OUTPUT DATA; ! Print results
repeat
```

The statements between a **cycle** statement and the corresponding **repeat** statement are known as the *cycle body*. **cycle/repeat** groups can be nested to any depth, i.e. a cycle body can contain further **cycle/repeat** groups.

## Conditional repetition

The number of times the cycle body is executed can be controlled by modifying the **cycle** statement or the **repeat** statement.

a) **while** *condition* **cycle**

```
 :
 :
repeat
```

The specified condition is tested *before* each execution of the cycle body. If the condition is true the cycle body is executed; otherwise control is passed to the statement following the matching **repeat**.

The cycle body will be executed zero or more times.

b) **for** *control variable* = *init* , *inc* , *final* **cycle**

```
 :
 :
repeat
```

where *control variable* is a variable of type **integer**, and *init*, *inc* and *final* are all integer expressions.

This is a special form of **while...cycle**, in which an integer variable takes a series of regularly spaced values in a specified range. Each value corresponds to an execution of the cycle body.

Example:

```
for I = 1,1,10 cycle
   A(I) = I+1
   B(I) = 0
repeat
```

The cycle body is executed 10 times, the control variable I taking the values 1, 2, 3, ..., 9, 10 in succession.

In detail, the execution of a **for** loop entails the following stages. A test is first made that *inc* is non-zero and that (*final* - *init*) is exactly divisible by *inc*. An event is signalled if it is not. This test may be carried out at compile-time, if the three expressions permit; a compile-time fault may then be given.

If the test is successful, the total number of times that the cycle body could be executed is calculated; this is ((*final* - *init*) // *inc*) + 1.

*EMAS IMP80: if this number is less than or equal to zero, the control variable is set to be unassigned and control is passed to the statement following the matching* ***repeat****. Otherwise the control variable is assigned the value of* init *and the cycle body is executed. When the matching* ***repeat*** *is reached the value of the control variable is tested, and if it is equal to* final*, control is passed to the statement following the* ***repeat****. Otherwise the control variable has* inc *added to its value and the cycle body is executed again.*

*IMP77: an event is signalled if the total number of times that the cycle body could be executed is less than 0. Otherwise the control variable is assigned the value* init - inc*. Before each execution of the cycle body the value of the control variable is compared with* final*. If they are equal control is passed to the statement following the matching* ***repeat****; otherwise* inc *is added to the control variable and the cycle body is executed. It follows that if the cycle execution is terminated by means of this test, the control variable will thereafter be equal to* final *in all cases.*

It follows that the cycle body will be executed zero or more times.

Notes

* *inc* can be negative.
* The control variable *must* be of type **integer** - a byte integer or long integer etc. is not allowed.
* It is wrong to jump into a **for** loop; the effect is undefined.
* It is wrong to change the value of the control variable within the cycle body; the effect is undefined.
* The three expressions *init*, *inc* and *final* are evaluated once only, before the cycle body is executed. It is permissible, within the cycle body, to change the value of any variable appearing in one of these expressions, but it has no effect on the execution of the **for** loop.

c) **cycle**
```
 :
 :
```
**repeat until** *condition*

After each execution of the cycle body the condition is tested. If false the cycle body is executed again.

**until** loops always execute the cycle body at least once (cf. **while** loops).

## Simple forms of loop

If the cycle body comprises only one instruction (which can be a compound instruction), the loop may be written in the form:

*instruction loop clause*

i.e.

*instruction* **while** *condition*
*instruction* **for** *control* = *init*, *inc*, *final*
*instruction* **until** *condition*

Examples:

SKIPSYMBOL **and** SP = SP+1 **while** NEXTSYMBOL = ' '

A(J) = 0 **for** J = 1,1,20

READSYMBOL(S) **until** S = NL

## *exit* and *continue*

Two instructions are provided to control the execution of a cycle from within the cycle body.

a) **exit**

This instruction causes execution of the cycle to be terminated. Control is passed to the statement following the matching **repeat**.

b) **continue**

This instruction causes control to be passed to the matching **repeat**, where any **until** condition will be tested. The effect is thus to terminate this execution of the cycle body, but not of the complete cycle.

Notes

* **exit** and **continue** can only be used within **cycle/repeat** loops.

* **exit** and **continue** are instructions, and can thus be made conditional.
  Examples:

  ```
              continue if P_J = 0

              X = 5 and exit if CPU TIME > 50
  ```

* Within nested loops, **continue** and **exit** operate with respect to the innermost loop in which they are contained; i.e. control is passed to the **repeat** of the innermost loop containing the **continue** or **exit** or to the statement following it, respectively.

* **continue** and **exit** may not be used inside a block defined within the relevant cycle body.

# CHAPTER 5

# INPUT/OUTPUT FACILITIES

Input/Output (I/O) facilities are provided to enable programs to read data from files or input devices, and send data to files or output devices. These facilities take the form of standard procedures, described below. (As explained in Chapter 7, standard procedures do not require to be specified before use; they are defined implicitly). All the procedures described refer to logical I/O channels to which numbers in the range 0-99 have been assigned. In some implementations certain channel numbers are reserved for system use.

*EMAS IMP80: channels 0 and 81-99 are reserved for system defined devices. Channel numbers in the range 1-80 can be used for purposes defined by the user, subject only to the rule that a channel number can only refer to one channel at a time.*

*IMP77: input channel* n *and output channel* n *are logically distinct.*

Each implementation of the language provides facilities for linking these logical channels to particular files or I/O devices; refer to the relevant User Guide for information on this subject.

The primary I/O facilities in IMP80 use character information, that is, information that can be represented as sequences of characters.

All the routines and functions use an Internal Character Code based on the ISO Code for the interchange of data (see Chapter 1).

Facilities are also provided to handle binary information, as a direct copy of the representation of values in the computer store. Two types of binary I/O are provided: Sequential Access for use when data is accessed (read or written) in the order in which it is held in the store, and Direct Access for use when data is accessed randomly from the store. All the binary I/O routines and functions are explicit, and so must be declared in each program in which they are used (see Section 5.2).

## 5.1 Character I/O

All character handling routines and functions operate with respect to either the currently selected *input stream* or the currently selected *output stream*. An input or output stream corresponds to a logical I/O channel (described above), and is referenced by the appropriate channel number. All character I/O streams act upon continuous streams of ISO characters. Thus an input stream consists of a continuous stream of characters with an associated pointer which at any instant points to the *next* character to be input. Similarly, an output stream is able to accept a continuous stream of ISO characters. Naturally, there has to be some implementation dependent mechanism for transferring these characters to and from files or physical devices, but this is hidden from the user.

*IMP77: IMP77 works with streams of 8-bit codes.*

On entry to a program, default streams are selected for input and output. At any point in the program it is possible to redirect character input or output by a call of the standard procedure

```
                routine SELECT INPUT(integer I)
            or
                routine SELECT OUTPUT(integer I)
```

Each of these routines takes one integer parameter, the value of the logical channel number corresponding to the stream being selected. An event is signalled if the specified channel does not correspond to a valid input or output stream, as appropriate.

Examples:

```
        SELECT OUTPUT(3)
        SELECT INPUT(I+17)
```

After a call of SELECT INPUT, all calls of character input routines will operate on the selected input stream, until another call of SELECT INPUT is made or the end of the program is reached. The same rule applies to SELECT OUTPUT and character output routines and functions.

## Input of character data

The procedures described below operate on the currently selected input stream. In all cases an event is signalled if an attempt is made to read beyond the end of the input stream.

*EMAS IMP80: the following ISO codes (see Chapter 1) are ignored by these procedures: DEL (value 127), and all codes with values less than 32 apart from NL (10), EM (25) and SUB (26). If it is desired to read these codes, rather than ignore them, then the procedures READ CH and NEXT CH (described below) should be used.*

### **routine** READ SYMBOL(**name** A)

This routine transfers the internal value of the next symbol from the current input stream into the integer variable parameter, which may be of any permissible length (byte, long, etc).

Example:

```
READ SYMBOL(IN); ! IN has been declared as an integer variable.
```

*IMP77: eight bits are transferred from the input stream to the parameter.*

*IMP77: the actual parameter supplied can be of type* ***string****, of any length, in which case a single character is read and held internally as a one-character string.*

*EMAS IMP80: all non-printing characters, excluding NL (ISO code 10) but including CR (ISO code 13) are ignored by READ SYMBOL. Cf. READ CH, described below.*

### **integer function** NEXT SYMBOL

This integer function returns the internal value of the next symbol on the current input stream. It does not move the pointer to the stream, so the next call of this or any other character input routine will access the same character again.

Example:

```
if NEXT SYMBOL = 'a' then .....
```

### **routine** SKIP SYMBOL

This routine moves the pointer to the current input stream along one symbol without transferring any information to store. In the following example, SKIP SYMBOL and NEXT SYMBOL are used together to skip over a series of space characters.

Example:

```
SKIP SYMBOL while NEXT SYMBOL = ' '
```

### **routine** READ STRING(**string(*)name** S)

This routine is used to read a string into the string variable location specified by the parameter. The string should be in the format of an IMP80 string constant. Any spaces and newline characters which precede the double quote character at the start of the string will be ignored. An event will occur if the input stream does not contain a string constant, or if the input string overflows the location specified by the parameter.

*IMP77: READ STRING is not provided. Instead the routine READ, described below, is extended to accept a string name parameter and to read a string into it.*

**routine** READ ITEM(**string**(*)**name** S)

This routine reads the next symbol from the current input stream and stores it as a string of length 1 in the string variable specified by the parameter.

*IMP77: READ ITEM is not provided. Instead the routine READ SYMBOL has been extended, as described above, to accept a string name parameter.*

**string**(*)**function** NEXT ITEM

This function reads the next symbol from the currently selected input stream without moving the pointer to the input stream (as with the function NEXT SYMBOL). The symbol is returned as a string of length 1.

*IMP77: NEXT ITEM is not provided.*

*EMAS IMP80: the following '... CH' procedures are provided. They have the same effect as their '... SYMBOL' counterparts described above, except that they do not ignore any ISO characters.*

***routine** READ CH(**name** I)*

*This routine transfers the internal value of the next symbol from the current input stream into the integer variable parameter. All characters of the ISO character set, including control characters, are passed back by READ CH.*

*READ CH will pass back EM (the end message character) when end-of-file is reached and will signal an event if further reads on this file are requested.*

***integer function** NEXT CH*

*This function reads the next character from the input stream without moving the pointer to the stream (as with NEXT SYMBOL). All characters of the ISO character set, including End Message character (EM), are passed back by NEXT CH.*

**Input of numeric data**

**routine** READ(**name** I)

This routine is used to read numeric data into an arithmetic variable. The single parameter should be of type **integer** or **real** and of any length. The numbers being read should be written as described for decimal constants, except that no space characters may be included in the constants. Any space or newline characters which precede the number will be skipped by the routine; thus spaces and newlines can be freely used as separators between numbers. The number is terminated by any character in the input stream other than digit, +, -, . or @. On return from READ, the input stream pointer will be pointing to the character immediately following the number just read.

Note that if an attempt is made to read either a real number, or a number with an exponent, into an integer variable, then the reading will stop at the decimal point or the "@" symbol, but no event will be signalled. However, an event will be signalled if the first character of a number is neither a sign, nor a decimal digit nor a decimal point. The same event will occur if the initial character is a decimal point but the parameter passed to READ is an integer.

The range of values which can be read depends on the given parameter and on the implementation.

*IMP77: the parameter to READ can be the name of a string, of any length. In this case READ reads in a string of characters, as follows: any ISO control characters plus spaces and newlines are first skipped, then all characters excluding DEL, up to but not including the next control character or space or newline are read and built up into a string which is assigned to the string parameter. Enclosing quote delimiters are "not" required.*

**Output of character data**

A number of procedures are provided to write individual characters or sequences of characters to the currently selected output stream.
*Note that when an interactive terminal is used, the symbols will normally be transmitted only when a 'trigger' character, usually newline (NL) or form feed (FF), is sent to the output stream.*

The basic character output procedure is PRINT SYMBOL; most of the other character output procedures operate via this procedure.

**routine** PRINT SYMBOL(**integer** I)

The parameter passed to this routine must be an integer expression which is evaluated by the routine. Any character whose value lies in the range 0-255 is transmitted to the output stream.

*EMAS IMP80: The least significant 7 bits only of the parameter are used by the routine. If the value of the expression corresponds to that of a symbol which could be read by READ SYMBOL (see above) then it is sent to the output stream; if not, the character SUB (ISO code value 26) is sent.*

Examples:

```
PRINTSYMBOL('A')
PRINTSYMBOL(I+J+32)
```

*EMAS IMP80:*
***routine** PRINT CH(**integer** I)*

*This routine behaves exactly like the standard version of PRINT SYMBOL, i.e. any character whose code value lies in the range 0-255 is transmitted to the output stream.*

**routine** PRINT STRING(**string**(255) S)

This routine transmits to the currently selected output stream the string expression specified by the string parameter.

*EMAS IMP80: PRINT STRING operates effectively as a series of PRINT SYMBOL calls. Thus certain characters will be replaced by SUB, as explained above.*

There are a few other standard procedures provided to simplify the output of text.

| | |
|---|---|
| SPACE | transmits one space character to the currently selected output stream |
| SPACES(N) | transmits N space characters to the currently selected output stream, where N is an integer expression. No spaces are transmitted if N is negative. |
| NEWLINE | transmits one newline character to the currently selected output stream. |
| NEWLINES(N) | transmits N newline characters to the currently selected output stream, where N is an integer expression. |
| NEW PAGE | transmits one Form Feed (FF) character to the currently selected output stream. No newlines are transmitted if N is negative. |

*EMAS IMP80: in the case of the procedures SPACES and NEWLINES, only the least significant byte of the parameter is used. Thus a maximum of 255 spaces or newlines can be transmitted by a single call.*

## Output of numeric data

Three routines are provided to allow output of numeric information. WRITE is used to transmit the value of integer expressions; PRINT and PRINTFL transmit the values of real expressions in floating point form.

### routine WRITE(integer I, J)

This routine transmits the value of the integer expression I to the currently selected output stream. The second parameter specifies the number of positions to be used. To simplify the alignment of positive and negative numbers, an additional position is allowed for a sign, but the sign is only printed in the case of negative numbers. If the number to be printed needs more positions than are specified by the second parameter, then more positions are used.

Examples:

```
            WRITE(I, 4)
            WRITE(TOTAL+SUM+ROW(I), 6)
            WRITE(SNAP, POS+4)
```

*IMP77: the total number of print positions to be used is defined by the* modulus *of the second parameter. If this parameter is negative, no space character is output before a positive value.*

### routine PRINT(long real X, integer I, J)

This routine transmits to the currently selected output stream the value of the real expression specified by the first parameter. The second and third parameters should be integer expressions specifying the number of places to be allowed before and after the decimal point. If the integer part needs more positions than are specified by the parameter, then more positions will be taken. One position is allowed for a sign which is printed only in the negative case. If necessary, the fractional part will be rounded.

Examples:

```
            PRINT(A, 2, 3)
            PRINT(COS(A-B), 1, 10)
```

*IMP77: The second parameter is interpreted in the same way as the second parameter of WRITE (described above).*

### routine PRINT FL(longreal X, integer I)

This routine transmits to the currently selected output stream the value of the real expression specified by the first parameter. The second parameter specifies the number of places to be allowed after the decimal point. The printed number takes up the specified number of places, plus 7 additional places.

Example:

```
            PRINT FL(X,4)
```

If X has the value 17.63584, this would be printed as  1.7636@ 1. The number is standardised in the range 1 <= X < 10. One position each is allowed for signs for mantissa and exponent; in each case the sign is only printed when negative.

## Closing streams

All input and output streams are closed automatically when the program terminates. However, if it is necessary to close a stream during the running of a program, then a call of the appropriate routine is necessary.

*EMAS IMP80:*

*__routine__ CLOSE STREAM(__integer__ I)*

*The parameter must be an integer expression which will evaluate to the number of the stream to be closed. The currently selected input or output stream cannot be closed, and any attempt to do so will cause an event to be signalled.*

*EMAS IMP80:*

*If a stream is closed and then reselected for input, the pointer to the stream will be positioned at the start of the file. Thus a file can be re-read, or a file written earlier in the program can be read.*

*In the same way, if a stream is closed and then reselected for output, the pointer to the file is left at the start of the file. Thus any information in the file at the time it is reselected will be overwritten by data transmitted to the file after reselection.*

*IMP77:*

**routine** *CLOSE INPUT*
*or*
**routine** *CLOSE OUTPUT*

*In each case the routine closes the* current *stream, input or output as appropriate. The input or output stream then becomes null (except for stream 0, which is unaltered), until another channel is selected by use of SELECT INPUT or SELECT OUTPUT.*

**routine** *RESET INPUT*

*This routine resets the current input stream to the start of the file.*

**routine** *RESET OUTPUT*

*This routine throws away all output on the current stream.*

*In some systems it is possible to specify that output is to be appended to an existing file. In this case the effect of closing and then reopening a corresponding output stream might not be as described above.*

## 5.2 Binary I/O

*Binary I/O facilities are provided by procedures which must be specified explicitly. They do not form part of the IMP80 language, but a list and brief explanation is given in Appendix B for some implementations. A more detailed description of the procedures can be found in the appropriate User Guide or Library Manual.*

# CHAPTER 6

# STORE MAPPING

It is possible to give an alternative name to a variable declared in an IMP80 program. It is also possible to refer to an arbitrary store location, *not* necessarily associated with a declared variable, and operate on it as though it held a variable of a specified type.

"Store mapping" is the name given to this technique. It is useful for the following reasons:

* to enable entities not declared within a program to be accessed and operated on by the program, without recourse to machine code or other machine specific features

* to save space in main store (see first example below)

* to access a variable both as declared, and as a set of sub-variables; for example, a variable of type **integer** can be accessed as several separate variables of type **byte integer**

* to access an array element as a simple variable, with consequent saving of machine time

* to improve the clarity of a program

The store mapping facilities in IMP80 are provided by means of several components:

* reference variables

* user-written mapping functions

* the ADDR function

* the standard mapping functions

## 6.1 Reference variables and user-written mapping functions

Two of the fundamental entities from which programs are constructed are constants and variables (Chapter 2). A variable comprises an *identifier* (its name), and a *type* which specifies what type of *value* it may have. Its value is a constant, or a reference to a variable. Thus, the value of a variable of type **integer** is an integer constant, while the value of a variable of type **integer name** is a reference to an integer variable.

A *function* (Chapter 3) is similar to a variable in that it has an identifier, and a type which specifies what sort of value it returns. A *mapping function* (or *map*) is analogous to a reference variable in that its type (the type of the value that it returns) is a reference to a variable; for this reason maps are sometimes called "name functions".

The difference between a variable and a function, of course, is that a variable has its value stored while a function computes its result.

**Examples**

1) Use of a reference variable with a map.

   In this program, a two-dimensional array is accessed by means of a map, because the array is symmetrical: i.e. X(i,j) = X(j,i) for all valid i and j. Thus to save storage space only the values of X(i,j) with i >= j are stored. By keeping only these values in a one-dimensional array A - which is global to the map - we can make economical use of store without losing the symmetrical appearance of the array X.

   The array is first assigned values and then various references to it involving **integer** ALPHA and **integer name** BRAVO are made.

```
begin
integer array A(1:210)
   integer ALPHA, I, J
   integer name BRAVO

   integer map X(integer I,J)
      signal event 6 unless 1<=I<=20 and 1<=J<=20; ! Array bound check.
      result == A(I*(I-1)//2 + J) if I>J
      result == A(J*(J-1)//2 + I)
   end

   for I = 1,1,20 cycle
      X(I,J) = I\\2 + 2*I*J + J\\2 for J = 1,1,I
   repeat

   ALPHA = X(17,10)       {ALPHA assigned the value of the variable returned
                          {by the call X(17,10) of map X.
   BRAVO == ALPHA         {BRAVO made to refer to ALPHA (thus BRAVO is now
                          {synonymous with ALPHA).
   BRAVO = 4              {BRAVO (i.e. ALPHA) assigned the value 4.
   BRAVO == X(16,9)       {BRAVO made to refer to the variable returned by
                          {the call X(16,9) of map X.
   BRAVO = 4              {BRAVO (i.e. the variable it currently refers to)
                          {assigned the value 4.
   ALPHA = BRAVO          {ALPHA assigned the value of the variable to which
                          {BRAVO currently refers. From the previous statement,
                          {this value is 4.
   X(6,15) = 17           {The variable to which reference is returned by
                          {the call X(6,15) of map X is assigned the
                          {value 17.
   ALPHA = X(15,6)        {ALPHA set to the value of the variable returned
                          {by the call X(15,6) of the map X. From the
                          {previous statement, and bearing in mind the
                          {symmetry of X, this value is 17.
    :
    :
end of program
```

2) Reference variable used as a parameter to a procedure

When a procedure is called, its formal parameters - in the example below SIZE and STATUS - are assigned initial values according to their types and to the actual parameters used in the call. SIZE is of type **integer** and so has the current value of TOTAL assigned to it; STATUS is of type **integer name** and so is "pointed at" the integer variable RETURN.

*This example is included because procedures with parameters called by name provide a method for renaming variables, and as such represent a common application of store mapping techniques.*

```
routine spec TEST(integer SIZE, integer name STATUS)

integer TOTAL, RETURN
 :
 :
TOTAL = 3
TEST(TOTAL, RETURN)
! TEST called: formal parameters assigned values.
! TOTAL cannot be changed as a result of this procedure call.
! But RETURN can be, since it is called by name.
 :
 :
```

3) Mapping a direct access file

The following map enables the programmer to treat a direct access file held on backing store as though it were a real array with declaration **real array** FILE(1:NBLOCK, 0:255), where NBLOCK is the number of blocks in the file.

In use, the only difference from a normal array is that a closing call *real var* = FILE(0,*n*) must be made.

It is assumed that the direct access file has been associated with channel number 1 prior to the execution of this program.

*In this implementation of IMP80, real variables are 32-bit entities, and each block of a direct access file consists of 1024 8-bit bytes.*

*Note that when a virtual memory operating system is being used, a simpler and better method for mapping arrays onto files may be available - see example in the EMAS IMP80 section of Appendix B.*

```
real map FILE(integer BLOCK, ELEMENT)

   ! The external routines specified below relate to the use of direct-access
   ! (DA) files.  They are the standard routines provided in most implementations
   ! of IMP80, but do not form part of the language definition. Further details
   ! for some implementations are given in Appendix B.
   externalroutinespec OPEN DA(integer CHANNEL)
   externalroutinespec CLOSE DA(integer CHANNEL)
   externalroutinespec READ DA(integer CHANNEL, integername BLOCK,
                               name START, FINISH)
   externalroutinespec WRITE DA(integer CHANNEL, integername BLOCK,
                                name START, FINISH)
   constant integer NO=0, YES=1
   own integer CURRENT BLOCK = 0, BLOCK CHANGED = NO
   own integer LAST ELEMENT
   own real LAST VALUE
   own real array BUF(0:255) = 0(256)

   unless BLOCK>0 and 0<=ELEMENT<=255 start
      ! Could be an error, or the closing call.
      ! In either case, tidy up and close the file.
      WRITE DA(1, CURRENT BLOCK, BUF(0), BUF(255)) if BLOCK CHANGED=YES
      CURRENT BLOCK = 0

      if BLOCK=0 start; ! Closing call.
         CLOSE DA(1)
         result == BUF(0); ! Irrelevant in this case.
      finish

      ! Error - parameters out of range.
      signal event 6
   finish

   if CURRENT BLOCK=0 start; ! First call.
      OPEN DA(1)
      CURRENT BLOCK = BLOCK
      READ DA(1, CURRENT BLOCK, BUF(0), BUF(255))
   finish else start; ! Not the first call.
      ! Has the value of the last element returned by the map been
      ! changed?  If so, note the fact so that the block is written back when
      ! no longer required by the map.
      BLOCK CHANGED = YES if BUF(LAST ELEMENT) # LAST VALUE

      if CURRENT BLOCK # BLOCK start
         ! Block required differs from that currently held in BUF.
         ! Write currently held block back if it has been changed.
         WRITE DA(1, CURRENT BLOCK, BUF(0), BUF(255)) if BLOCK CHANGED=YES

         ! Now get block required.
         CURRENT BLOCK = BLOCK
         READ DA(1, CURRENT BLOCK, BUF(0), BUF(255))
         BLOCK CHANGED = NO; ! Reset change flag.
      finish

   finish

   LAST ELEMENT = ELEMENT
   LAST VALUE = BUF(ELEMENT)

   result == BUF(ELEMENT)

end; ! Of real map FILE.
```

## 6.2 ADDR and the standard mapping functions

A variable has an identifier, a type and a value. A fourth attribute of an accessible variable is its *address*. This is an integer which uniquely specifies its location in the computer store. The address of a variable can be obtained by use of the standard **integer function** ADDR.

Example:
```
          string(27) S
          integer ADDRESS OF S
            :
            :
          ADDRESS OF S = ADDR(S)
```

The parameter of ADDR can be a variable or array element of any type; in the case of a reference variable parameter, ADDR returns the address of the variable to which it refers, *not* the address of the reference variable itself.

Given a variable, it is thus possible to find its address. Conversely, given an address, it is possible to *construct* a variable of any type, subject to certain restrictions detailed below. This is achieved by the use of the *standard mapping functions*.

For each arithmetic and string type there is an associated standard map whose name is the same as that of the type; thus BYTE INTEGER, LONG REAL, STRING, etc. A standard map has a single parameter of type **integer**, which is an address; it returns a reference to a variable of the appropriate type, located at that address.

The standard map RECORD is described later in this Chapter.

Example:
```
          integer I, J
          byte integer array B(0:3)
            :
            :
          I = M'ABCD'
            :
          B(3-J) = BYTE INTEGER(ADDR(I)+J) for J = 0,1,3
            :
```

In this example the integer I has been unpacked into its four component byte integers (*implementation using 32-bit integers assumed*).

Example:
```
          byte integer array IN(0:80)
          string(*)name LINE
            :
          LINE == STRING(ADDR(IN(0)))
            :
```

From this point onward the array can be referenced either as an array of bytes or as the string LINE. Obviously the length byte of the string, normally the first byte of the string location, will have to be set to an appropriate value.

There are a number of points to note about the use of the standard mapping functions:

* They are efficient.

* An address error will occur if the address passed to a standard mapping function is not *aligned* correctly with respect to the type of reference implied.
  *On some byte addressed machines, for example:*

  *SHORT INTEGER requires the address to be divisible by 2*
  *INTEGER (if 32-bit integers used) requires the address to be divisible by 4*
  *REAL requires the address to be livisible by 4*

  *The maps for the longer arithmetic types may have corresponding requirements.*

* In Chapter 3 it was explained that variables declared in a program are allocated storage on the *stack*. However, if the computer system being used enables an IMP80

program to obtain store addresses of items not on the stack (e.g. items within the operating system, or a connected file in a virtual memory system), then the standard mapping functions can be used to refer to them as variables. This is an extremely powerful facility.

* It is possible to cause a location in which, for example, a real constant is stored to be treated as an integer or a string, etc. While this facility may be useful on occasions it should be used with care, because: 1) misuse can lead to errors which are hard to diagnose; 2) it requires a knowledge of the precise format of data types and is necessarily implementation dependent; 3) the purpose of providing types in IMP80 is to enable checks to be made that dissimilar items of data are not being erroneously combined. Use of the standard mapping functions merely to avoid such checks is bad programming practice.

*EMAS IMP80: a standard map, ARRAY, is provided for mapping all array types. It is described in Appendix B.*

RECORD

The main use of the standard map RECORD is to assign a reference to a record reference variable. The format of the record in question is that of the reference variable on the left-hand side.

Example:

```
record format F(integer A, B, C, real D, string(11) E)
record (F) R
record (F) array RA(1:50)
record (F) name N1, N2, N3
integer ADDRESS
 :
ADDRESS = address of some store location
 :
N1 == R;              ! Record name N1 now synonymous with record R.
N2 == RA(20);         ! Record name N1 now synonymous with element 20 of
                      ! record array RA.
N3 == RECORD(ADDRESS); ! Record name N3 now synonymous with a record of format
                      ! F located at address ADDRESS.
 :
```

Example:

```
integer J
integer array II(1:100)
record format A(byte integer I, J, K, L or half integer P, Q)
                                        ! Implementation dependent type
record(A)name X
 :
X == RECORD(ADDR(II(J)))
 :
```

Now, for example:

X_I is a reference to a byte of II(J)
X_P is a reference to a half-word of II(J)

*Implementation dependent notes concerning the standard map RECORD are given in Appendix B.*

# CHAPTER 7

# STANDARD PROCEDURES

The standard procedures are those provided automatically, i.e. no specification (**spec**) statements are required for them. The standard procedures are listed below, with an indication of where their descriptions may be found: either a chapter number of this manual, or "I-S" meaning that the procedure's precise effect is implementation-specific. The relevant documents for various implementations of IMP80 are given in Appendix B.

Notes

* At the time of writing, existing implementations of IMP80 do not have a complete common set of standard procedures. The first list below includes procedures common to all the implementations. The other lists relate to specific implementations.

* A standard procedure will not be provided in an implementation if it is a function or map, or has a parameter, of a type not provided in the implementation.

* Implementations of IMP80 may provide procedures other than those listed here which do not require specification statements. In addition, various libraries of external procedures may be provided. These are not included here; consult the relevant documentation for details.

* Two named constants are predefined:

  **constant integer** NL = {code value for newline character. The ISO value is} 10

  **constant long real** PI = 3.141592653589793 {if 64 bits allocated to a **long real**}

* *EMAS IMP80: a standard procedure can be* intrinsic. *An intrinsic procedure is one which is compiled as part of the program (for reasons of efficiency) rather than being separately compiled. Intrinsic procedures cannot be passed as parameters. The standard procedures which are intrinsic are noted in Appendix B.*

Standard Procedures

| **Type** | **Name and parameter list** | **Reference** |
|---|---|---|
| **integer function** | ADDR(**name** A) | 6 |
| **byte integer map** | BYTE INTEGER(**integer** I) | 6 |
| **integer map** | CHAR NO(**string(*)name** S, **integer** I) | 2 |
| **long real function** | FLOAT(**integer** I) | 2 |
| **long real function** | FRAC PT(**long real** A) | 2 |
| **half integer map** | HALF INTEGER(**integer** I) | 6 |
| **integer function** | IMOD(**integer** I) | 2 |
| **integer function** | INT(**long real** A) | 2 |
| **integer function** | INT PT(**long real** A) | 2 |
| **integer map** | INTEGER(**integer** I) | 6 |
| **integer map** | LENGTH(**string(*)name** S) | 2 |
| **long integer map** | LONG INTEGER(**integer** I) | 6 |
| **long long real map** | LONG LONG REAL(**integer** I) | 6 |

| Type | Name and parameter list | Reference |
|---|---|---|
| long real map | LONG REAL(integer I) | 6 |
| long real function | MOD(long real A) | 2 |
| routine | NEW LINE | 5 |
| routine | NEW LINES(integer I) | 5 |
| routine | NEW PAGE | 5 |
| integer function | NEXT SYMBOL | 5 |
| routine | PRINT(long real A, integer I, J) | 5 |
| routine | PRINT FL(long real A, integer I) | 5 |
| routine | PRINT STRING(string(255) S) | 5 |
| routine | PRINT SYMBOL(integer I) | 5 |
| routine | READ(name A) | 5 |
| routine | READ SYMBOL(name I) | 5 |
| real map | REAL(integer I) | 6 |
| record map | RECORD(integer I) | 6 |
| routine | SELECT INPUT(integer I) | 5 |
| routine | SELECT OUTPUT(integer I) | 5 |
| short integer map | SHORT INTEGER(integer I) | 6 |
| integer function | SIZE OF(name A) | 2 |
| routine | SKIP SYMBOL | 5 |
| routine | SPACE | 5 |
| routine | SPACES(integer I) | 5 |
| string(*)map | STRING(integer I) | 6 |
| string(*)function | SUBSTRING(string(*)name S, integer I, J) | 2 |
| string(*)function | TO STRING(integer I) | 2 |
| routine | WRITE(integer I, J) | 5 |

*EMAS IMP80-specific standard procedures*

| Type | Name and parameter list | Reference |
|---|---|---|
| long real function | ARC COS(long real A) | I-S |
| long real function | ARC SIN(long real A) | I-S |
| long real function | ARC TAN(long real A, B) | I-S |
| array map | ARRAY(integer I, array format A) | 6 |
| routine | CLOSE STREAM(integer I) | 5 |

| Type | Name and parameter list | Reference |
|---|---|---|
| long real function | COS(long real A) | I-S |
| long real function | COT(long real A) | I-S |
| long real function | EXP(long real A) | I-S |
| integer function | EVENT INF(integer I) | 3 |
| integer function | EVENT LINE(integer I) | 3 |
| long integer function | LENGTHEN I(integer I) | I-S |
| long long real function | LENGTHEN R(long real A) | I-S |
| long integer function | LINT(long long real A) | I-S |
| long integer function | LINT PT(long long real A) | I-S |
| long real function | LOG(long real A) | I-S |
| integer function | NEXTCH | 3 |
| string(*)function | NEXT ITEM | 5 |
| routine | PRINT CH(integer I) | 5 |
| long real function | RADIUS(long real A, B) | I-S |
| routine | READ CH(name I) | 5 |
| routine | READ ITEM(string(*)name S) | 5 |
| routine | READ STRING(string(*)name S) | 5 |
| integer function | SHORTEN I(long integer I) | 2 |
| long real function | SHORTEN R(long long real A) | 2 |
| long real function | SIN(long real A) | I-S |
| long real function | SQRT(long real A) | I-S |
| long real function | TAN(long real A) | I-S |

*IMP77-specific standard procedures*

| Type | Name and parameter list | Reference |
|---|---|---|
| routine | CLOSE INPUT | 5 |
| routine | CLOSE OUTPUT | 5 |
| routine | RESET INPUT | 5 |
| routine | RESET OUTPUT | 5 |
| record format<br>record(EVENT FM)map | EVENT FM(integer EVENT, SUB, EXTRA)<br>EVENT | 3 |
| integer function | REM(integer A, B) | I-S |
| integer function | TYPE OF(name A) | I-S |

# APPENDIX A

## IMP80 SYNTAX

```
letter      ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
                a | b | c | d | e | f | g | h | i | j | k | l | m |
                n | o | p | q | r | s | t | u | v | w | x | y | z

digit       ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

space       ::= {space character}

newline     ::= {newline character}

bar         ::= {| character}

character   ::= ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - |
                . | / | : | < | = | > | ? | @ | [ | ] | \ | ^ | _ |
                ` | ~ | ; | } | { | letter | digit | space | bar

comstart    ::= comment | !

comment     ::= comstart [character]... newline

integer     ::= digit...

letdig      ::= letter | digit

name        ::= letter [letdig]...

namelist    ::= name [, name]...

int const   ::= [plusminus] integer

frac        ::= . [integer]

dec const   ::= [int const] [frac] [@ int const]

base const  ::= int const _ letdig... | letter ' letdig... '

charnl      ::= character | newline

char const  ::= ' charnl ' | M ' charnl... '

str const   ::= " [charnl]... "

const       ::= int const | dec const | base const | char const |
                str const

unop        ::= + | - | \ | ~

assop       ::= = | == | <- | ->

plusminus   ::= + | -

comp1       ::= comp2 | -> | == | \== | ##

comp2       ::= = | # |  \= | <> | > | >= | < | <=

op          ::= * | / | // | + | - | \ | \\ | ^ | ^^  | >> | << |
                ! | !! | & | .

app         ::= ( expr [, expr]... )

recel       ::= _ name [app]

operand     ::= name [app] [recel]... | const | ( expr )
```

| | | |
|---|---|---|
| *expr* | ::= | [*unop*] *operand* [*op operand*]... |
| *count* | ::= | ( *expr* ) \| (*) |
| *exprcount* | ::= | *expr* [*count*] |
| *constlist* | ::= | = *exprcount* [, *exprcount*]... |
| *narr* | ::= | **array** |
| *arrf* | ::= | *narr* |
| *decln* | ::= | *arrf adecln* \| [[*narr*] **name**] *namelist* |
| *adecln* | ::= | *namelist bpair* [, *namelist bpair*]... |
| *rfelmnt* | ::= | [[*narr*] **name**] *namelist* \| *narr adecln* |
| *rfdeclistor* | ::= | ( *rfdeclist* [**or** *rfdeclist*]... ) |
| *rfdeclist* | ::= | *rfdec* [, *rfdec*]... |
| *rfdec* | ::= | *type rfelmnt* \| *rfdeclistor* |
| *rfref* | ::= | ( *name* ) |
| *btype* | ::= | **integer** \| **real** \| **long real** |
| *type* | ::= | **integer** \| **real** \| **long** *btype* \| **byte** [**integer**] \| **short** [**integer**] \| **half** [**integer**] \| **string** *count* \| **record** *rfref* |
| *namea* | ::= | *name* [**alias** *str const*] |
| *namealist* | ::= | *namea* [, *namea*]... |
| *ownlist* | ::= | *namealist* [= *expr*] |
| *owndec* | ::= | [[*narr*] **name**] [**spec**] *ownlist* [, *ownlist*]... \| *arrf* [**spec**] *namea bpair* [*constlist*] |
| *fm* | ::= | **fn** \| **function** \| **map** |
| *rt* | ::= | **routine** \| *type fm* |
| *fpdel* | ::= | *type* [[**array**] **name**] *namelist* \| *rt* [**name**] *namelist* [*fpp*] \| **name** *namelist* |
| *fpp* | ::= | ( *fpdel* [, *fpdel*]... ) |
| *range* | ::= | *expr* : *expr* |
| *bpair* | ::= | ( *range* [, *range*]... ) |
| *xown* | ::= | **own** \| **external** \| **constant** \| **const** |
| *%sed* | ::= | **system** \| **external** \| **dynamic** |
| *endlist* | ::= | **of program** \| **of file** \| **of list** |
| *sc* | ::= | *expr comp1 expr* [*comp2 expr*] \| ( *sc restofcond* ) \| **not** *sc* |
| *restofcond* | ::= | [**and** *sc*]... \| [**or** *sc*]... |
| *%wf* | ::= | **while** *sc restofcond* \| **for** *name* = *expr* , *expr* , *expr* |
| *%iu* | ::= | **if** \| **unless** |

```
ui          ::=  name [app] [recel]... [assop expr] [and ui] |
                 -> name [app] |
                 return |
                 result assop expr |
                 monitor [and ui] |
                 stop |
                 signal [event] const [, expr] |
                 exit |
                 continue

ci          ::=  %iu sc restofcond restofiu

restofiu    ::=  [then] start | then ui [else]

else        ::=  else start |
                 else ci |
                 else ui

restofss    ::=  %iu sc restofcond |
                 until sc restofcond |
                 %wf

s           ::=  ; | newline

ss          ::=  begin s |
                 type decln s |
                 record format name rfdeclistor s |
                 [%sed] rt [spec] name [fpp] s |
                 xown type owndec s |
                 include str const |
                 switch namelist bpair [, namelist bpair]... s |
                 on [event] [, integer]... start s |
                 ui [restofss] s |
                 ci s |
                 finish [else] s |
                 else s |
                 [%wf] cycle s |
                 repeat [until sc restofcond] s |
                 name [app] : |
                 name (*) : |
                 list s |
                 end [endlist] s |
                 comment |
                 s
```

Notes

* In the above syntax, items in italics are non-terminals, '|' separates alternatives, '::=' means 'is defined as', items enclosed in [..] brackets are optional, and items followed by '...' can be repeated one or more times. All other characters stand for themselves, except that {...} indicates a "descriptive" definition, involving a newline character or a character used in the statement of the syntax itself.

* If a string of characters does not satisfy the definition of *ss* then it is not a valid IMP80 source statement.

* The case of letters is ignored in IMP80 programs except within quoted constants.

* Space characters are only significant within character, string or base constants or following a keyword (**real**, **if**, **unless**, **end**, **not**, etc.).

* Text preceded by '{', terminated by '}' and appearing between syntactic elements of a program is treated as a comment. Such a comment is also terminated by a newline character; it may not include a newline or '}' character.

* IMP80 statements are continued on the next line if the current line is terminated with the keyword **c**. The **c** is not required if the break comes immediately after a comma. *IMP77: comment statements cannot be continued.*

* There may be differences between the syntax given above and that used by specific implementations. See Appendix B for details. For example: 1) the definition of *type* may not be as given above, since not all the types are available in some implementations; 2) implementation dependent extensions may involve modifications to the above syntax.

# APPENDIX B

# IMPLEMENTATION-SPECIFIC INFORMATION

## B1: EMAS IMP80

### B1.1 Compile-time errors

The EMAS IMP80 compiler can generate any of the following messages while compiling a program. The following points should be noted:

* The symbols '#' and '##' appearing in any of the messages below are replaced by appropriate integer values when the message is output by the compiler.

* The symbols '&' and '&&' appearing in any of the messages below are replaced by appropriate names (of program variables, routines, switches, etc.) when the message is output by the compiler.

* Messages numbered 1-100 relate to standard compile-time errors. Messages numbered 101-200 relate to various compile-time limits (compiler table sizes, etc.) being exceeded. Messages with numbers greater than 200 are warnings - they do not in themselves indicate an error in the program being compiled.

| | |
|---|---|
| 1 | **repeat** is not required |
| 2 | Label & has already been set in this block |
| 4 | & is not a Switch name at current textual level |
| 5 | Switch name & in expression or assignment |
| 6 | Switch label &(#) set a second time |
| 7 | Name & has already been declared |
| 8 | Routine or fn & has more parameters than specified |
| 9 | Parameter # of & differs in type from specification |
| 10 | Routine or fn & has fewer parameters than specified |
| 11 | Label & referenced at line # has not been set |
| 12 | **cycle** at line # has two control clauses |
| 13 | **repeat** for **cycle** at line # is missing |
| 14 | **end** is not required |
| 15 | # **ends** are missing |
| 16 | Name & has not been declared |
| 17 | Name & does not require parameters or subscripts |
| 18 | # too few parameters provided for & |
| 19 | # too many parameters provided for & |
| 20 | # too few subscripts provided for array & |
| 21 | # too many subscripts provided for array & |
| 22 | Actual parameter # of & conflicts with specification |
| 23 | Routine name & in an expression |
| 24 | Integer operator has real operands |
| 25 | Real expression in integer context |
| 26 | # is not a valid **event** number |
| 27 | & is not a routine name |
| 28 | Routine or fn & has specification but no body |
| 29 | **function** name & not in expression |
| 30 | **return** outwith routine body |
| 31 | **result** outwith fn or map body |
| 34 | Too many textual levels |
| 37 | Array & has too many dimensions |
| 38 | Array & has upper bound # less than lower bound |
| 39 | Size of Array & is more than X'FFFFFF' bytes |
| 40 | Declaration is not at head of block |
| 41 | Constant cannot be evaluated at compile time |
| 42 | # is an invalid repetition factor |
| 43 | **constant** name & not in expression |
| 44 | Invalid constant initialising & after # items |
| 45 | Array initialising items expected ## items given # |
| 46 | Invalid **external**, **extrinsic** or variable **spec** |
| 47 | **else** already given at line # |
| 48 | **else** invalid after **on event** |
| 49 | Attempt to initialise **extrinsic** or **format** & |
| 50 | Subscript of # is outwith the bounds of & |

| | |
|---|---|
| 51 | **finish** is not required |
| 52 | **repeat** instead of **finish** for **start** at line # |
| 53 | **finish** for **start** at line # is missing |
| 54 | **exit** outwith **cycle repeat** body |
| 55 | **continue** outwith **cycle repeat** body |
| 56 | **externalroutine** & at wrong textual level |
| 57 | Executable statement found at textual level zero |
| 58 | Program among external routines |
| 59 | **finish** instead of **repeat** for **cycle** at line # |
| 61 | Name & has already been used in this **format** |
| 62 | & is not a **record** or **record format** name |
| 63 | **record** length is greater than # bytes |
| 64 | Name & requires a subname in this context |
| 65 | Subname & is not in the **record format** |
| 66 | Expression assigned to record & |
| 67 | Records && and & have different formats |
| 69 | Subname && is attached to & which is not of type **record** |
| 70 | String declaration has invalid max length of # |
| 71 | & is not a string variable |
| 72 | Arithmetic operator in a string expression |
| 73 | Arithmetic constant in a string-expression |
| 74 | Resolution is not the correct format |
| 75 | String expression contains a sub expression |
| 76 | String variable & in arithmetic expression |
| 77 | String constant in arithmetic expression |
| 78 | String operator '.' in arithmetic expression |
| 80 | Pointer variable & compared with expression |
| 81 | Pointer variable & equivalenced to expression |
| 82 | & is not a pointer name |
| 83 | && and & are not equivalent in type |
| 86 | Global pointer && equivalenced to local & |
| 87 | **format** name & use in expression |
| 90 | Untyped name & used in expression |
| 91 | **for** control variable & not integer |
| 92 | **for** clause has zero step |
| 93 | **for** clause has noninteger number of traverses |
| 95 | Name & not valid in assembler |
| 96 | Operand # not valid in assembler |
| 97 | Assembler construction not valid |
| 101 | Source line has too many continuations |
| 102 | Workfile of # Kbytes is too small |
| 103 | Dictionary completely full |
| 104 | Dictionary completely full |
| 105 | Too many textual levels |
| 106 | String constant too long |
| 107 | Compiler tables are completely full |
| 108 | Condition too complicated |
| 109 | Compiler inconsistent |
| 201 | Long integers are inefficient as subscripts |
| 202 | Name & not used |
| 203 | Label & not used |
| 204 | Global **for** control variable & |
| 205 | Name & not addressable |
| 206 | Semicolon in comment text |
| 255 | Compiler failure: contact Computing Centre Advisory Service |

## B1.2 Event numbers

| *Failure* | *Event/Subevent* |
|---|---|
| Integer overflow | 1/1 |
| Real overflow | 1/2 |
| Zero divide | 1/3 |
| SIN, COS, TAN arg out of range or inappropriate, COT arg out of range or inappropriate | 1/4 |
| TAN too large | 1/5 |
| EXP arg out of range | 1/6 |
| Int pt too large | 1/7 |
| Program too large | 2/1 |
| SUB character in data | 3/1 |
| Symbol in data | 4/1 |
| Symbol instead of string | 4/2 |
| Illegal cycle | 5/1 |
| SQRT arg negative | 5/2 |
| LOG arg negative or zero | 5/3 |
| Illegal exponent | 5/5 |
| Array inside out | 5/6 |
| Capacity exceeded | 6/1 |
| Array bounds exceeded | 6/2 |
| Resolution fault | 7/1 |
| Unassigned variable | 8/1 |
| Switch label not set | 8/2 |
| Input ended | 9/1 |
| ARCSIN arg out of range | 10/1 |
| ARCCOS arg out of range | 10/2 |
| ARCTAN args zero | 10/3 |
| HYPSIN arg out of range | 10/4 |
| HYPCOS arg out of range | 10/5 |
| RADIUS args too large | 10/6 |
| Graph fault | 11/1 |

## B1.3 Differences from IMP80

The following notes are given in the same order and under the same headings as the descriptions in the body of the manual of the IMP80 items to which they refer. The notes attempt to describe the EMAS IMP80 departures from and extensions to IMP80. They will be updated from time to time as necessary.

## CHAPTER 1: ELEMENTS OF THE LANGUAGE

### 1.1 Character set

All characters with code values less than 32 are treated as spaces, apart from

```
NL  (ISO code 10)
EM  (ISO code 25) - terminates program
SUB (ISO code 26) - causes a syntax fault
```

It ignores DEL (ISO code 127) and maps all codes greater than 127 according to the extended Regional Network Code (which is detailed in Regional Communications Memo JID/78/M20.4).

## 1.2 Statements

### Continuation

Blank lines between the lines of a continued statement are ignored.

## 1.3 Statement components

### Special symbols

\ and \\ can be replaced by ** and **** respectively.

## 1.4 Miscellaneous statements

### Comments

Note that the continuation rules apply to comment statements.

### Machine code

Any source statement starting with an asterisk (*) is taken to be a machine code instruction. The use of machine code within IMP80 programs is beyond the scope of this manual.

# CHAPTER 2: TYPES, VARIABLES, CONSTANTS AND EXPRESSIONS

## 2.1 Types

The types provided in EMAS IMP80 are as follows:

1) **byte integer**
   **half integer**
   **integer**
   **long integer**
   **real**
   **long real**
   **long long real**
   **string**(*n*)
   **record**(*format*)

2) Reference types corresponding to each of the types given in (1), designated by the keyword **name** appended in each case.

3) Array types corresponding to each of the types given in (1), designated by the keyword **array** appended in each case.

4) Array reference types corresponding to each of the types given in (1), designated by the keywords **array name** appended in each case.

## 2.2 Variables

Arrays (all types) can have up to 12 dimensions.

### Record variables

Alignment within record formats.

In general the layout of sub-fields within a record is such as to require the minimum number of padding bytes consistent with sub-field alignment requirements (detailed below). This can mean that the relative alignment of alternatives is dependent on the preceding part of the format - it cannot be deduced by examining the alternatives alone. Padding bytes may be appended to a record format to ensure that the alignment would be correct if it were used by a record array; the amount of this padding depends on the size of the format and the types of the sub-fields it contains.

If it is necessary to know the size of a record whose format is complex, then the standard integer function SIZE OF should be used.

### Sub-field alignment requirements

| | |
|---|---|
| *(Word boundary:* | *address divisible by 4* |
| *Half-word boundary:* | *address divisible by 2)* |

Byte integer and string: no alignment requirements.

Half integer: must start on a half-word boundary.

Integer, long integer, real, long real, long long real, record, reference variable of any type: must start on a word boundary.

Arrays: requirements as for corresponding scalars.

It is stressed that these alignments are carried out automatically; they are only given here to enable programmers to determine how much padding, if any, will be used in a record format.

In a record declaration it is permissible to give the actual record format instead of giving the name of a record format.

Example:

```
        record (integer I, J, string(7) S) A, B, C
```

It is also permissible to give the name of a record already declared in place of a record format name. The format of the specified record is then taken as the required format for the record being declared.

Example:

```
        record (A) D; ! A is the record declared in the previous example.
```

In record assignments using '=', differences in the formats of the left-hand and right-hand operands are not faulted if they have the same overall length.

## Reference variables

In the declaration of string reference variables, the string variable maximum size may be omitted. Whether it is or not, the EMAS IMP80 compiler treats the size specification as '(*)', meaning that can refer to a string variable of any maximum size.

When a record reference variable is assigned to by use of the '==' operator, the record location referenced by the right-hand operand does *not* have to have the same format as the record reference variable. In such a case the sub-fields associated with the reference variable are those in its own format, not those in the format of the right-hand operand.

## *own*, *constant* and *external* variables

**own**, **constant** and **external** variables which are not assigned initial values are set to binary zeros.

In a two-dimensional array whose first element was (1,1), the order of elements would be (1,1), (2,1), (3,1), i.e. first subscript changing fastest. It is necessary to know this when initialising such arrays.

The space allocated to records declared as **own**, **external** or **constant** is filled with binary zeros, unless the declaration is followed by = *n*, where *n* is an integer constant in the range [0-255]. In this case every byte of the record is initialised to the given value.

Reference variables may be declared as **own**, **external** or **constant**. They are initialised as shown in the following example:

**constant integer name** K INST PER SECOND = X'80C000C0'

The value to which the integer name variable is being initialised specifies a particular storage location, and is therefore system dependent.

An own, external or constant array name is initialised to refer to a particular location if its declaration is followed by = *n*, where *n* is the address of the location.

Example:

**constant integer array name** A = x'AA0022'

The array referred to is one-dimensional, with bounds (0 : $2^{23}$-1).

## 2.3 Constants

### Character constants

The following form of character constant is additionally provided:

| | |
|---|---|
| C'....' | a character constant, like M'....' but held internally in EBCDIC code instead of ISO code. |

### String constants

The following form of string constant is additionally provided:

| | |
|---|---|
| E"...." | a string constant, like "...." but held internally in EBCDIC code rather than ISO code. |

### Decimal constants

The following forms of decimal constant are additionally provided:

| | |
|---|---|
| D'*decimal constant*' | like *decimal constant* but held internally as a 128-bit constant (i.e. as a **long long real**). |
| R'*hexadecimal constant*' | like *hexadecimal constant* but held internally as a *real* constant of length 32, 64 or 128 bits depending on the number of hexadecimal digits specified, which must be exactly 8, 16 or 32. |

## 2.4 Operators and expressions

### Arithmetic expressions

\\ is anomalous in that *long integer\\integer* is carried out by repeated long integer multiplication but *integer\\long integer* is carried out by repeated integer multiplication.

Where the exponent is an integer expression, the operation is carried out by repeated multiplication.

Where the exponent is a real expression, the result is obtained by using the standard functions LOG and EXP, and events relating to these functions may be signalled.

**String operators and expressions**

Multiple resolution is permitted; it is treated as a series of simple resolutions.

Example:

```
            S = "WINSTON SPENCER CHURCHILL"
            S -> A.(" ").B.(" ").C
```

The above resolution will be treated as

```
            S -> A.(" ").PRIV and PRIV -> B.(" ").C
```

The standard string function SUB STRING(S,I,J) returns a null string if I>J - an event is *not* signalled.

# CHAPTER 3: BLOCKS AND PROCEDURES

## 3.1 Block structure and storage allocation

**Events**

Event 11 is used by the ERCC Graphics Package.

The following two functions are provided:

**integer function** EVENT INF

This function returns

```
            (event no<<8) ! sub-event no
```

for the last event which has occurred. An error occurs at compile time if the function is called in a block with no **on event** statement, and an undefined value will result at run time if no event has in fact occurred when the function is called.

**integer function** EVENT LINE

This function returns the program line number at which the last event occurred during execution of the block in question (provided the program was compiled with line number updating; otherwise 0 will be returned). If no event has occurred, an undefined value will result.

When EMAS IMP80 programs are "optimised" (a compilation option), it is possible that an event signalled immediately prior to exiting from a block might *not* cause a branch to an **on event** statement in that block, even when it specifies the relevant event number. This usually affects **result** statements only. It is a consequence of the computer hardware design ("pipelining").

## 3.2 Procedures

Those standard procedures which are "intrinsic" (see notes on Chapter 7, below) cannot be actual parameters.

**Functions**

The format associated with a record function cannot exceed 256 bytes.

### Maps

A result statement in a map can be of the form

**result** = *integer expression*

The effect is to return a reference to a variable of the same type as the map, with address specified by the integer expression (which usually includes a call of the function ADDR, described in Chapter 6).

# CHAPTER 4: EXECUTABLE STATEMENTS

## 4.1 Conditions

The == and ## comparators cannot be used to compare references to arrays.

## 4.2 Instructions

### Labels and jumps

The rule that label identifiers have to be distinct is not enforced, but compliance is recommended.

## 4.4 *cycle/repeat*

For technical reasons on ICL 2900 Series computers, cycles which count down to unity are more efficient than other types of cycle. Thus

```
A(I) = 0 for I = N,-1,1
```

is more efficient than the more common

```
A(I) = 0 for I = 1,1,N
```

### Conditional repetition

The number of times that a cycle body could be executed is calculated prior to execution of the cycle. If this number is less than or equal to zero, the control variable is set to be unassigned and control is passed to the statement following the matching **repeat**. Otherwise the control variable is assigned the value of *init* and the cycle body is executed. When the matching **repeat** is reached the value of the control variable is tested, and if it is equal to *final*, control is passed to the statement following the **repeat**. Otherwise the control variable has *inc* added to its value and the cycle body is executed again.

# CHAPTER 5: INPUT/OUTPUT FACILITIES

I/O channels 0 and 81-99 are reserved for system defined devices. Channel numbers in the range 1-80 can be used for purposes defined by the user, subject only to the rule that a channel number can only refer to one channel at a time.

## 5.1 Character I/O

Note that in addition to the standard I/O procedures described in Chapter 7, some I/O procedures which require to be explicitly may also be available in the implementation. See the current System Library Manual for details.

### Input of character data

The following ISO codes (see Chapter 1) are ignored by the character input procedures: DEL (value 127), and all codes with values less than 32 apart from NL (10), EM (25) and SUB (26). If it is desired to read these codes, rather than ignore them, then the procedures READ CH and NEXT CH (described below) should be used.

**routine** READ SYMBOL(**name** A)

All non-printing characters, excluding NL (ISO code 10) but including CR (ISO code 13) are ignored by READ SYMBOL. Cf. READ CH, described below.

The following '... CH' procedures are provided in EMAS IMP80. They have the same effect as their '... SYMBOL' counterparts described above, except that they do not ignore any ISO characters.

**routine** READ CH(**integer name** I)

This routine transfers the internal value of the next symbol from the current input stream into an **integer** variable. All characters of the ISO character set are passed back by READ CH.

READ CH will pass back EM (the end message character) when end-of-file is reached and will signal an event if further reads on this file are requested.

**integer function** NEXT CH

This function reads the next character from the input stream without moving the pointer to the stream (as with NEXTSYMBOL). All characters of the ISO character set, including End Message character (EM), are passed back by NEXT CH.

### Output of character data

**routine** PRINT SYMBOL(**integer** I)

The least significant 7 bits only of the parameter are used by the routine. If the value of the expression corresponds to that of a symbol which could be read by READ SYMBOL (see above) then it is sent to the output stream; if not, the character SUB (code value 26) is sent.

**routine** PRINT CH(**integer** I)

This routine behaves exactly like the standard version of PRINT SYMBOL, i.e. any character whose code value lies in the range 0-255 is transmitted to the output stream.

**routine** PRINT STRING(**string**(255) S)

This routine operates effectively as a series of PRINT SYMBOL calls. Thus certain characters will be replaced by SUB, as explained above.

```
routine SPACES(integer N)
routine NEWLINES(integer N)
```

In the case of the procedures SPACES and NEWLINES, only the least significant byte of the parameter is used. Thus a maximum of 255 spaces or newlines can be transmitted by a single call.

**Closing streams**

```
routine CLOSE STREAM(integer I)
```

The parameter must be an integer expression which will evaluate to the number of the stream to be closed. The currently selected input or output stream cannot be closed, and any attempt to do so will cause an event to be signalled.

## 5.2 Binary I/O

The following binary I/O procedures are provided in the implementation. They have to be explicitly specified. Full details of their use can be found in the current System Library Manual.

```
external routine OPEN SQ(integer I)
external routine CLOSE SQ(integer I)
external routine READ LSQ(integer I, name K, L, integer name J)
external routine READ SQ(integer I, name K, L)

external routine OPEN DA(integer I)
external routine CLOSE DA(integer I)
external routine READ DA(integer I, integer name J, name K, L)
external routine WRITE DA(integer I, integer name J, name K, L)
```

# CHAPTER 6: STORE MAPPING

## 6.2 ADDR and the standard mapping functions

### ARRAY

There is only one standard mapping function, ARRAY, for all the *array* types. It takes two parameters: an address, and an *array format* to specify the type of the array being referenced. **array format** statements are analogous to **record format** statements in that they enable an identifier to be associated with a type and a structure. They can then be referenced by name in subsequent statements.

**array format** statements declared outside any procedures in an external file (see Section 3.3) must be given the attribute **own**.

ARRAY can only be used to assign to a reference variable of the appropriate **array** type.

Example:

```
integer array AONE(1:10 000)
integer array name ATWO
integer array format AFORM(1:100, 1:100)
 :
ATWO == ARRAY(ADDR(AONE(1)), AFORM)
 :
ATWO(27, 43) = 928
 :
```

The **array format** statement is used to describe the characteristics of the array ATWO - i.e. the number of dimensions and bounds for each dimension.

As an alternative to using an array format for the second parameter, the name of another array (of the appropriate type) can be used, if one with suitable characteristics has been declared and is in scope.

In order to map record arrays, the second parameter of ARRAY must either be an existing record array variable, or a record array format. A record array format must specify both the dimensions of the array and the format of each (record) element of the array:

**record** (*record format*) **array format** *name* (*array dimensions*)

Example:

*This example is of a record array format being used in an IMP80 program running under the operating system EMAS 2900. The integer function SMADDR returns the address of a file connected in the virtual memory of the user process.*

```
routine PAYCHECK(integer CHANNEL, RECNO)
integer I, J, K, START, LENGTH
string(11) NAME
externalintegerfnspec SMADDR(integer CHANNEL, integername L)
record format PAYF (string(11) SURNAME, integer AGE, SEX, YEAR,
                    integerarray SALARY (1:12))
! Each record thus formatted contains 72 bytes.
record(PAYF)array format PAYAF(1:RECNO)
record(PAYF)array name PAY
 :
 :
! Assume that a file was associated with channel CHANNEL,
! prior to the execution of the routine.

START = SMADDR(CHANNEL,LENGTH); ! File now connected.

if LENGTH < 72*RECNO start
   PRINTSTRING("File too small:")
   WRITE(LENGTH,1); NEWLINE
   PRINTSTRING("Must be at least")
   WRITE(72*RECNO,1); NEWLINE
   return
finish

PAY == ARRAY(START,PAYAF)
! Now record array name PAY has been mapped onto the file.
! Note that START was set by the SMADDR call.
 :
 :
NAME=PAY(I)_SURNAME
if PAY(I)_SALARY(J)>350 ....
 :
 :
PAY(K)_YEAR=1978
 :
 :
end; ! Of routine PAYCHECK.
```

## RECORD

The standard record map RECORD returns a reference to a record whose format is unspecified but "very large". This is acceptable when assigning a reference to a reference variable, as described above, since in EMAS IMP80 the format of the reference on the right-hand side of such an assignment statement does not have to match that of the reference variable on the left-hand side. However, a record variable *cannot* be assigned by a statement of the form

```
RECVAR = RECORD(ADDRESS)
```

Since the sizes of the records on each side are not equal. On the other hand, the form

```
RECVAR <- RECORD(ADDRESS)
```

is permitted; as many bytes as RECVAR can hold (determined by its format) will be transferred.

# CHAPTER 7: STANDARD PROCEDURES

A standard procedure can be *intrinsic*. An intrinsic procedure is one which is compiled as part of the program (for reasons of efficiency) rather than being called by the normal mechanism. Intrinsic procedures cannot be passed as parameters. The standard procedures which are intrinsic are as follows:

```
integer function ADDR(name I)
array map ARRAY(integer I, array format name J)
byte integer map BYTE INTEGER(integer I)
byte integer map CHARNO(string(*)name S, integer I)
long real function FRAC PT(long real A)
integer function IMOD(integer I)
integer function INT(long real A)
integer map INTEGER(integer I)
integer function INT PT(long real A)
byte integer map LENGTH(string(*)name S)
long integer map LONG INTEGER(integer I)
long long real map LONG LONG REAL(integer I)
long real map LONG REAL(integer I)
long real function MOD(longreal A)
routine NEWLINE
routine NEWLINES(integer I)
routine NEWPAGE
integer function NEXT CH
string(1)function NEXT ITEM
integer function NEXT SYMBOL
routine PRINT CH(integer I)
routine PRINT STRING(string S)
routine PRINT SYMBOL(integer I)
routine READ CH(name I)
routine READ ITEM(string(*)name S)
routine READ SYMBOL(name I)
real map REAL(integer I)
record map RECORD(integer I)
routine SELECT INPUT(integer I)
routine SELECT OUTPUT(integer I)
routine SKIP SYMBOL
routine SPACE
routine SPACES(integer I)
string(*)map STRING(integer I)
string(1)function TO STRING(integer I)
```

## EMAS IMP80-specific standard procedures

Note that the given reference is either the number of a chapter of the manual or it is "I-S", meaning "Implementation-Specific". In this case refer to the "System Library Manual for 2900 Compilers" (ERCC 1978) for further details.

| *Type* | *Name and parameter list* | *Reference* |
|---|---|---|
| long real function | ARC COS(long real A) | I-S |
| long real function | ARC SIN(long real A) | I-S |
| long real function | ARC TAN(long real A, B) | I-S |
| array map | ARRAY(integer I, array format A) | App B |
| routine | CLOSE STREAM(integer I) | 5 |
| long real function | COS(long real A) | I-S |
| long real function | COT(long real A) | I-S |
| long real function | EXP(long real A) | I-S |
| integer function | EVENT INF(integer I) | 3 |
| integer function | EVENT LINE(integer I) | App B |
| long integer function | LENGTHEN I(integer I) | I-S |
| long long real function | LENGTHEN R(long real A) | I-S |
| long integer function | LINT(long long real A) | I-S |
| long integer function | LINT PT(long long real A) | I-S |
| long real function | LOG(long real A) | I-S |

| *Type* | *Name and parameter list* | *Reference* |
|---|---|---|
| **string(1)function** | NEXT ITEM | 5 |
| **routine** | PRINT CH(**integer** I) | 5 |
| **long real function** | RADIUS(**long real** A, B) | I-S |
| **routine** | READ CH(**name** I) | 5 |
| **routine** | READ ITEM(**string(*)name** S) | 5 |
| **routine** | READ STRING(**string(*)name** S) | 5 |
| **integer function** | SHORTEN I(**long integer** I) | 2 |
| **long real function** | SHORTEN R(**long long real** A) | 2 |
| **long real function** | SIN(**long real** A) | I-S |
| **long real function** | SQRT(**long real** A) | I-S |
| **long real function** | TAN(**long real** A) | I-S |

## APPENDIX A: IMP80 SYNTAX

The syntax of the machine code instructions mentioned in the EMAS IMP80-specific notes on Section 1.2 are not included in the IMP80 syntax given in Appendix A.

The definitions of some phrases in EMAS IMP80 differ from those given in Appendix A:

```
type        ::=  integer | real | long btype | byte [integer] |
                 half [integer] | string [count] | record rfref

op          ::=  * | / | // | + | - | \ | \\ | ^ | ^^  | >> | << |
                 ! | !! | & | . | ** | ****

arrf        ::=  narr [format]
```

(This permits **array format** declarations, used in conjunction with the standard map ARRAY.)

```
rfref       ::=  ( name ) | rfdeclistor

char const  ::=  ' charnl ' | M ' charnl... ' | C ' charnl... ' |
                 E " charnl... " | D ' dec const ' | R ' letdig... '
```

## B2: IMP77

### B2.1 Compile-time messages

During the compilation of a program the compiler may generate messages which are generally sent to the listing file and possibly to an interactive report stream. These messages are either error indications or warnings.

#### Errors

An error message indicates that the current statement does not obey the rules ot the language or that a necessary statement has been omitted from the previous statement sequence.

Once an error has been detected the compiler ignores the rest of the faulty statement and continues compiling with the next. This may result in consequential errors which will disappear once the original error is corrected. For example, the compiler will fault the following declaration:

**integer** A,B,,C,D

The extra comma will cause the declaration of C and D to be ignored and so subsequent references to them will be faulted ("not declared"). In general it is good practice to correct errors in the order in which they occur in the listing.

Error messages start with an asterisk (*), and where possible they contain a marker which points into the offending statement at the position at which the compiler detected the error.

The error messages are as follows:

| | |
|---|---|
| Atom | An unknown atomic element has been encountered. This is commonly caused by mistyping a keyword.<br>Example:<br>**integer, rutine, strat** |
| Bounds | The size of an array or switch vector is negative.<br>Example:<br>**switch** S(10:1)<br>**own integer array** X(-1:-10) |
| Context | An otherwise correct statement has been given in a context where it is meaningless.<br>Examples:<br>**exit** not contained within a **cycle/repeat** loop<br><br>**return** not inside a **routine** |
| Context *recf* | *recf* is the identifier of a record format which has been used to define a record or record array within the definition of *recf* itself. Note that it *is* valid to declare record name and record array name variables in this context.<br>Example:<br>**record format** F(**integer** X, **record**(F) Y) |
| Duplicate | A local identifier is being redeclared.<br>Example:<br>**real** SUN, MON, TUE, WED, THUR, FRI, SAT, SUN |
| Form | An unexpected atom has been encountered. This is usually caused by the omission of an atom or the insertion of an extra atom.<br>Examples:<br>**integer** A, B,, C<br><br>PRINTSTRING("BYE") NEWLINE {semicolon missing} |

Format — A record with a format which is currently undefined has been used.
Example:

```
record format spec FM
record(FM)name PT
 :
PT = 0
```

Index — A switch label has been given an index outwith the declared bounds.
Example:

```
switch S(1:5)
 :
S(6):
```

Match — The definition of a procedure does not match a previous specification.
Example:

```
routine spec PROC(integer X)
 :
routine PROC(real X)
```

Not a variable — An attempt has been made to use an object with a constant value in a context where it could be modified. This is commonly caused by using named constants as though they were variables.
Example:

```
constant integer TEN = 10
 :
TEN = TEN+1
```

Not declared — An undeclared identifier has been used. This error is also commonly generated by omitting the percent from the beginning of certain keywords (usually **if**, **finish** and **repeat**).
Example:

```
integer SWOP
 :
SWAP = 0
```

Note the following common error: **string**(7)**name** P
This declares a simple string variable NAMEP instead of what was probably intended, a string reference variable P.

Order — This is similar to Context but is reserved for statements which are given before they are valid or after other statements which invalidate them. There are three common causes:

1) The declaration of variables (other than **own** or **external**) global to the outermost block of a program.
   Example:

```
integer X
begin
 :
```

2) The declaration of an array following a label.
   Example:

```
begin
LAB: integer array A(1:5)
```

3) Declarations following an **on event** statement.
   Example:

```
on event 7 start
    stop
finish
 :
integer array XX(2:7)
```

Size — A constant has a value outwith the permitted range.
Example:

```
string(300) S
```

Too complex — The statement is too large or complicated to be analysed. This error is quite rare and can invariably be cured by splitting the offending statement into two or more simpler statements.

Note that putting redundant continuations (c) at the end of each line of a large list of array initialising constants may provoke this error.

Type — The type of a given variable or expression does not match the type of object required by the context.
Example:

```
integer X
byte integer name P
  :
P == X
  :
X = 1.2
```

**begin** missing — An **end** has been found which has no matching **begin** (or procedure heading).

**cycle** missing — A **repeat** has been found which does not have a matching **cycle** in the current block.

**end** missing — The end of the program file has been reached before all blocks have been terminated.

**finish** missing — The end of a block has been reached and it contains a **start** which has no matching **finish**.

**repeat** missing — The end of a block has been reached and it contains a **cycle** which has no matching **repeat**.

**result** missing — This occurs at the end of a function, map, or predicate when it is not evident that control will always be passed back from the procedure at run-time.
Examples:

```
integer function F(integer X)
    result = 0 if X <= 0
end

predicate EVEN(integer N)
    true if N&1=0
    false if N&1#0
    ! This will give the error as the compiler
    ! is unlikely to be clever enough to detect
    ! the 'completeness' of the conditions.
end
```

**start** missing — The compiler has found a **finish** for which there is no matching **start**.

*proc* missing — The procedure identified by *proc* has been specified in the preceding block (by a **spec** statement) but has not subsequently been defined.
Example:

```
begin
    routine spec CHECK
      :
    CHECK
      :
end
```

## Warnings

A warning indicates that the compiler has detected something which, although not an error in itself, may indicate a logical error elsewhere.

Warning messages start with a question mark (?) and are as follows:

Access — Control cannot reach the current statment. That is, the previous executable statement was or implied an unconditional transfer or control, and the current statement is not labelled.

Non-local — The control variable of a for loop is not local to the current block. Such use of globals can lead to unexpected infinite loops.
Example:

```
integer P
 :
routine R
    for P = 1,1,10 cycle   {P is global to routine R.}
     :
end
 :
R for P = 1,1,20
```

*ident* unused — The given identifier has been declared but never used.

## Catastrophic errors

Under certain circumstances the compiler will be unable to continue after discovering an error, usually because its tables will have been filled or corrupted.

These errors are as follows:

Compiler error — There is a fault in the compiler itself. Contact the Computing Centre Advisory Service.

Switch vector too large
A switch vector has been declared with a very large number of elements.

Too many names — The compiler has no room left to describe new named objects.

Dictionary full
The compiler has no room left to hold the text of new identifiers. This is usually caused by declaring a large number of long identifiers.

Input ended — The end of an input file has been reached without **end of file** or **end of program** being detected. This is most commonly caused by mistyping **end of program**, or leaving out a closing string quote.
Some IMP77 compilers may choose to treat this as a warning and complete the compilation.

String constant too long
A string constant has been discovered to contain more than 255 characters. This is commonly caused by leaving out the terminating quote.

Included file *file* does not exist
The compiler cannot gain access to a file specified in an **include** statement.

Too many faults!
This is generated when the compiler discovers a high fault rate in the program. It is used to terminate compilations which would otherwise generate a large number of faults. This is commonly caused by faulty declarations, or by attempting to compile something which is not an IMP80 program.

## B2.2 Event numbers

The details of an event can be obtained when it occurs by use of the **record map** EVENT, described in the IMP77-specific notes on Section 3.1. Where appropriate, EVENT_EXTRA is given in brackets in the descriptions below.

| **event** | **sub-class** | **meaning (+extra information)** |
|---|---|---|
| 0 | | TERMINATION |
| | -1 | abandon program |
| | 0 | normal termination; equivalent to **stop** |
| | >0 | user generated error |
| 1 | | OVERFLOW |
| | 1 | integer overflow |
| | 2 | real overflow |
| | 3 | string overflow |
| | 4 | division by zero |
| 2 | | EXCESS RESOURCE |
| | 1 | not enough store |
| | 2 | output exceeded |
| | 3 | time exceeded |
| 3 | | DATA ERROR |
| | 1 | symbol in data (+symbol) |
| 4 | | CORRUPT DATA |
| | 1 | data transmission error |
| | 2 | corrupt control variable |
| 5 | | INVALID ARGUMENTS |
| | 1 | for cannot terminate |
| | 2 | illegal exponent (+exponent) |
| | 3 | array inside-out |
| | 4 | string inside-out |
| | 5 | illegal parameter |
| 6 | | OUT OF RANGE |
| | 2 | array bound fault (+index) |
| | 3 | switch bound fault (+index) |
| | 4 | Illegal event signal (+event) |
| | 5 | CHARNO out of range (+index) |
| | 6 | TOSTRING out of range (+symbol) |
| 7 | | RESOLUTION FAILS |
| 8 | | UNDEFINED VALUE |
| | 1 | unassigned variable |
| | 2 | no switch label (+index) |
| | 3 | for variable corrupt |
| 9 | | I/O ERROR |
| | 1 | input ended |
| | 2 | illegal stream (+stream no) |
| | 3 | file does not exist |
| 10 | | LIBRARY PROCEDURE ERROR |
| 11-15 | | GENERAL PURPOSE |

## B2.3 Differences from IMP80

The following notes are given in the same order and under the same headings as the descriptions, in the body of the manual, of the IMP80 items to which they refer. The notes attempt to describe the IMP77 departures from and extensions to IMP80. They will be updated from time to time as necessary.

# CHAPTER 1: ELEMENTS OF THE LANGUAGE

## 1.1 Character set

Except for NL all characters not enclosed in quotes and with ISO codes outwith the range 32 to 126 inclusive are treated as spaces, but will be sent to the listing unaltered. The character FF (form feed) may be used to cause a new page to be taken in program listing files.

## 1.2 Statements

### Continuation

Continuation is implied if a line break occurs after the keywords **or** or **and**.

## 1.4 Miscellaneous statements

### Comments

Comments cannot be continued onto subsequent lines by any of the methods described.

### *list* and *end of list*

**end of list** and **list** are nested, so that two **end of list** statements require two matching **list** statements to switch on the listing again. This means that it is possible within an included file to control the listing of the contents of the file, without changing the listing status of the rest of the program.

### Machine code

There are two methods of adding in-line machine code sequences to an IMP77 program:

1) *=*integer constant*

   Statements of this form plant the given integer constant as an instruction.

2) **machine code*

   Statements of this form enable pseudo assembler statements to be included which can make use of program-declared objects. Refer to the relevant implementation notes for details of the syntax of *machine-code*.

# CHAPTER 2: TYPES, VARIABLES, CONSTANTS AND EXPRESSIONS

## 2.1 Types

The types provided in IMP77 are as follows:

1) **byte integer**
   **short integer**
   **integer**
   **long integer**
   **real**
   **long real**
   **string**(*n*)
   **record**(*format*)

2) Reference types corresponding to each of the types given in (1), designated by the keyword **name** appended in each case. A general reference type, designated by the single keyword **name**, is also provided.

3) Array types corresponding to each of the types given in (1) *and* in (2), designated by the keyword **array** appended in each case.

4) Reference types corresponding to each of the types given in (3), designated by the keyword **name** appended in each case.

## 2.2 Variables

All variables can be assigned initial values. The syntax is as for **own** variable initialisation. Stack variables are re-initialised whenever they are re-created.

### Record variables

A **record format spec** statement is provided to enable a record format to be referred to before it has been declared. A statement of the form

**record format spec** *name*

specifies a record format identifier. Until the format is declared fully in a **record format** statement the identifier may only be used in the declaration of record reference variables.

Example:

```
record format spec Y
record format X(record (Y) name P, real VALUE)
record format Y(record (X) name Q, integer VALUE)
```

Record formats of this sort are useful in list processing when the items in the list are records of alternating format X, then Y, then X, etc.

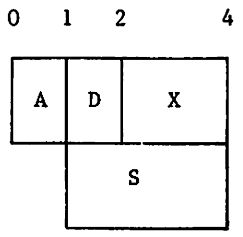
Alignment of record format alternatives.

The start of each of a set of alternatives within a record format is aligned as though its first element were the most 'demanding' within all the alternatives, so far as alignment is concerned. Thus, if a set of alternatives contains a long real element anywhere within it, the alternatives are aligned as though they started with the long real. This can of course cause padding bytes to be required in the record just before the bracketed alternatives, and it may result in further padding being required within the alternatives. It is done this way because it means that the alignment of alternatives is not affected by anything outside the brackets, and so curious effects concerning alignment within the alternatives do not occur as a result of some other part of the record format being changed.

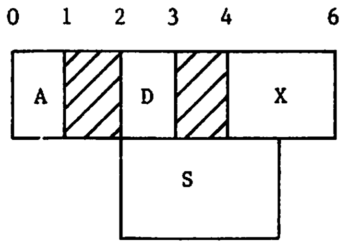
Example:

```
        record format(byte integer A,
                      (byte integer D, short integer X or c
                       string(2) S)
```

would not be aligned



as might be expected, but as follows



The integer function SIZE OF should be used with a record of the format in question as parameter if there is any doubt about the amount of space required.

## Reference variables

When a record reference variable is declared, the format can be specified as (*), meaning that the variable can refer to a record of any format. Such a reference variable has no associated sub-fields; it is only of use when it is pointed at subsequently by a reference variable with a specific format.

When an array reference variable is being declared, the number of dimensions of the array to which it can refer is specified in brackets:

Example:

```
        real array (4) name SPACETIME
```

The bracketed integer can be omitted if the dimensionality is 1.

In IMP77 reference variables can be grouped into arrays.

Examples:

```
        integer name array FREQ('A':'Z')

        record (CARFM) name array TABLE(-5:22)
```

Furthermore, reference variables which can point at such arrays are provided.

Examples:

```
        integer name array name FREQP

        record (CARFM) name array name TABLE REF
```

IMP77 also provides a 'general' reference variable type which can be pointed at a variable of any type. Arrays of general reference variables, and reference variables which can point at such arrays, are also provided.

Examples:

```
name NA, NB

name array WHAT(0:6)

name array name GEN POINTERS
```

### *own*, *constant* and *external* variables

**own**, **constant** and **external** variables which are not assigned initial values are undefined, and any attempt to use them before they are assigned will cause an event ("unassigned variable") to be signalled.

The '<-' assignment operator can be used in **own**, **constant** and **external** variable initialisation.

**own**, **constant** and **external** arrays can only be one-dimensional in IMP77.

The values of the sub-fields of records declared as **own**, **external** or **constant** are undefined, unless the whole record is initialised to binary zeros in the declaration.

Reference variables may be declared as **own**, **external** or **constant**. They are initialised as shown in the following example:

```
constant integer name CLOCK == 16_3C
```

The value to which the integer name variable is being initialised specifies a particular storage location, and is therefore system dependent.

## 2.3 Constants

### Base constants

The base can have *any* positive integer value. However, if it is greater than 36 then not all the digits will be representable by 0...9 and A...Z.

Example:

```
I = 256_1234
```

where I is a four-byte integer, assigns the values 1, 2, 3 and 4 respectively to the four bytes of I.

A base constant can include a decimal part, in which case it is of type **real**.

Examples:

```
3_0.1 {= 1/3}

16_3.102A9
```

## 2.4 Operators and expressions

### Arithmetic expressions

The modulus of an integer expression can also be obtained by enclosing the expression between vertical bars; e.g. |I-J| .

The modulus of a real expression can also be obtained by enclosing the expression between vertical bars; e.g. |X-Y| .

# CHAPTER 3: BLOCKS AND PROCEDURES

## 3.1 Block structure and storage allocation

### Events

There are two extensions to the range of event numbers:

```
Event 0  is defined - Termination (see Section B2.2)
Event 15 is available for user definition
```

The form

**on event * start**

is permitted. It is a shorthand way of specifying *every* event number, i.e. 0, 1, ..., 14, 15.

A standard record map is provided to enable the programmer to determine further information when an event occurs.

**record format** EVENT FM(**integer** EVENT, SUB, EXTRA)
**record**(EVENT FM)**map** EVENT

EVENT returns a reference to a system-provided record which contains the parameters of the last event to have been signalled. If no event has been signalled then all the fields of the record are set to zero.

## 3.2 Procedures

The identifiers of parameters in a **spec** statement must be distinct.

### Maps

The keyword **map** may be replaced by **name function** or **name fn.**

The record format given in the specification or heading can be replaced by (*), meaning that the reference returned by the map may be to a record of any format; the actual record format used depends on the context.

Example:

**record**(*)**map** SURVEY(**integer** I, **real** X)

The standard map RECORD (described in Chapter 6) is of this sort.

### Predicates

A predicate is a procedure which tests the validity of an hypothesis and may be used wherever a simple condition is required. When a predicate is called its statements are executed until either the instruction **true** is executed, in which case the predicate returns and the simple condition it constitutes is true, or the instruction **false** is executed, in which case the predicate returns and the simple condition is false.

Note that a predicate does not return any value.

The first line of a predicate has the following form

**predicate** *name* (*parameter list*)

Example:

```
            integer N

            predicate SINGLE DIGIT
                true if 0 <= N <= 9
                false
            end

              :
            N = N//10 unless SINGLE DIGIT
              :
```

## 3.3 External linkage

### External files

A single **begin**/**end** block at the outer level is allowed, so long as it immediately precedes the **end of file** statement. The block's **end** statement and the **end of file** statement can then be replaced by the single statement **end of program**.

# CHAPTER 4: EXECUTABLE STATEMENTS

## 4.2 Instructions

### *stop*

**stop** is event 0 and can therefore be trapped.

## 4.4 *cycle/repeat*

### Conditional repetition

An event is signalled if the total number of times that a cycle body could be executed is less than 0. Otherwise the control variable is assigned the value *init - inc*. Before each execution of the cycle body the value of the control variable is compared with *final*. If they are equal control is passed to the statement following the matching **repeat**; otherwise *inc* is added to the control variable and the cycle body is executed. It follows that if the cycle execution is terminated by means of this test, the control variable will thereafter be equal to *final* in all cases.

The control variable of a cycle can be an integer of any type.

Any statement containing the keyword **cycle** can be matched with any statement containing the keyword **repeat**.

Example:

```
            while I<=100 cycle
              :
              :
            repeat until J=4
```

The cycle-termination test implied by each statement is made when that statement is executed.

# CHAPTER 5: INPUT/OUTPUT FACILITIES

Input channel *n* and output channel *n* are logically distinct, and can thus be open simultaneously.

## 5.1 Character I/O

IMP77 works with streams of 8-bit codes.

### Input of character data

**routine** READ SYMBOL(**name** A)

Eight bits are transferred from the input stream to the parameter.

The actual parameter supplied can be of type **string**, of any permitted maximum length, in which case a single character is read and held internally as a one-character string.

**routine** READ STRING(**string**(*)**name** S)

READ STRING is not provided. Instead the routine READ, described below, is extended to accept a string name parameter and to read a string into it.

**routine** READ ITEM(**string**(*)**name** S)

READ ITEM is not provided. Instead the routine READ SYMBOL has been extended, as described above, to accept a string name parameter.

**string**(*)**function** NEXT ITEM

NEXT ITEM is not provided.

### Input of numeric data

**routine** READ(**name** I)

The parameter to READ can be the name of a string, of any length. In this case READ reads in a string of characters as follows: any ISO control characters plus spaces and newlines are first skipped, then all characters excluding DEL, up to but not including the next control character or space or newline, are read and built up into a string which is assigned to the string parameter. Enclosing quote delimiters are "not" required.

### Output of Numeric Data

**routine** WRITE(**integer** N, J)

The total number of print positions to be used is defined by the *modulus* of the second parameter. If this parameter is negative, no space character is output before a positive value.

**routine** PRINT(**long real** X, **integer** J, K)

The second parameter is interpreted in the same way as the second parameter of WRITE (described above).

**Closing streams**

```
            routine CLOSE INPUT
         or
            routine CLOSE OUTPUT
```

In each case the routine closes the *current* stream, input or output as appropriate. The input or output stream then becomes null, until another channel is selected by use of SELECT INPUT or SELECT OUTPUT.

```
            routine RESET INPUT
```

This routine resets the current input stream to the start of the file.

```
            routine RESET OUTPUT
```

This routine throws away all output on the current stream.

# CHAPTER 6: STORE MAPPING

## 6.2 ADDR and the standard mapping functions

### RECORD

The standard map RECORD is of the form

```
            record(*)map RECORD(integer AD)
```

i.e. the reference returned by RECORD may be to a record of any format; the actual format is determined by the context. Thus RECORD can be used to assign to a complete record.

Example:

```
            record format AF(integer array X(1:20), string(10) TITLE)
            record(AF) A
            integer AD
             :
             :
            A = RECORD(AD)      {The record format used is that of A, i.e. AF}
             :
```

Note that statements of the form

```
            RECORD(I) = RECORD(J)
```

are *not* allowed, since no format is indicated from the context.

# CHAPTER 7: STANDARD PROCEDURES

## IMP77-specific standard procedures

Note that the given reference is either the number of a chapter of the manual or it is "I-S", meaning "Implementation-Specific". In this case refer to "The IMP-77 Language", by Peter S. Robertson, Internal Report CSR-19-77, Department of Computer Science, University of Edinburgh, for further details.

| *Type* | *Name and parameter list* | *Reference* |
|---|---|---|
| **routine** | CLOSE INPUT | 5 |
| **routine** | CLOSE OUTPUT | 5 |
| **routine** | RESET INPUT | 5 |
| **routine** | RESET OUTPUT | 5 |
| **record format** | EVENT FM(**integer** EVENT, SUB, EXTRA) | App B |
| **record**(EVENT FM)**map** | EVENT | |
| **integer function** | REM(**integer** A, B) | I-S |
| **integer function** | TYPE OF(**name** A) | I-S |

# APPENDIX A: IMP80 SYNTAX

Comment statements cannot be continued onto subsequent lines by any of the continuation methods described in Chapter 1.

The syntax of the machine code sequences described in the IMP77-specific notes on Section 1.2 are not included in the IMP80 syntax given in Appendix A.

The definitions of some phrases in IMP77 differ from those given in Appendix A:

| | | |
|---|---|---|
| *operand* | ::= | *name* [*app*] [*recel*]... \| *const* \| (*expr*) \| ! *operand* ! |
| *btype* | ::= | **integer** \| **real** |
| *type* | ::= | **integer** \| **real** \| **long** [*btype*] \| **byte** [**integer**] \| **short** [**integer**] \| **string** [*count*] \| **record** *rfref* |
| *narr* | ::= | [**name**] **array** [( *integer* )] |

(This permits *type* **name array** declarations, i.e. arrays of reference variables, and of reference variables which can be pointed at such arrays.)

*ss*  *add* the following alternative:
  [*narr*] **name** *namelist*

(This permits the declaration of general reference variables, described elsewhere in the IMP77-specific notes.)

*rt*  ::= **routine** | **predicate** | *type fm*

*ui*  *add* the following alternatives:
  **true** | **false**

# INDEX