

Edinburgh Regional Computing Centre

Edinburgh IMP Language Manual

A description of the IMP Language
as implemented by ERCC

Second edition
June 1974

E D I N B U R G H R E G I O N A L C O M P U T I N G C E N T R E

February 1982

IMP80 on EMAS 2900: Differences from IMP9

Contents

		Page
1.	Compiler name	2
2.	Lower case input	2
3.	Continuation	2
4.	Comments	2
5.	== and ##	3
6.	Available types	3
7.	Keyword and operator alternatives	3
8.	own initialisation	3
9.	Switch labels	4
10.	Cycles	5
11.	start/finish blocks	6
12.	Constants	7
13.	Strings	8
15.	Records	9
15.	external items	10
16.	Procedures as parameters	11

Introduction

This document is intended for users of the programming language IMP on EMAS 2900 who wish to know how the new version of IMP, IMP80, differs from the current version, IMP9.

It should be noted that IMP80 on EMAS 2900 differs in certain respects from other implementations of IMP80, and that this document should not be trusted as far as other implementations are concerned.

Some of the features of IMP80 described below exist in IMP9. They are included here either to help explain some other feature or for completeness.

1. The command invoking the compiler is IMP80, not IMP.

2. Except within single or double quotes, lower case text is not distinguished from upper case. Thus

```
%integer a      and      %INTEGER A
```

are both acceptable and treated as equivalent. Note that

```
%integer Item 1a
```

is not distinguished from

```
%INTEGER ITEM1A
```

The convention in this document is that IMP keywords are underlined and given in lower case, with identifiers in upper case. Thus:

```
integer A
```

3. Continuation of statements. Statements can be continued on the next line by terminating the current line with c. The c is not required if the break comes immediately after a comma. (This applies to all statement types, not just own array initialisations.)

Examples:

```
if A=23 and K<=14 then c
   L=17 and M=18

integer A, B, C, D, E,
        F, G, H, I, J
```

A blank line following a line terminated by c is ignored.

4. Comments. A semi-colon does not terminate a comment - it can only be terminated by a newline. Comment statements can be continued by use of c, OR BY BEING BROKEN AFTER A COMMA (see 3 above).

A new type of comment is introduced; it is delimited by curly brackets, '{' and '}'. Such a comment can appear between atoms of a statement (an atom is an identifier, constant, keyword, operator or delimiting symbol).

Example:

```
A(I{month}, J{salary}) = 927.4
```

The comment text can contain any symbols except '}' and newline. The closing '}' can be omitted, in which case the comment is terminated by the next newline.

{...} comments are particularly useful for explaining own array initialisations.

5. == and ## (or \==)

The == operator can be used in conditions:

Example:

```
if A == B then .....
```

The condition is only true if A and B refer to the same variable; i.e. address and type equivalence is required. The operator ## (or \==) can be used to express the inverse condition:

```
if A ## B then .....
```

Note that == and ## can only be used to compare references to scalar variables, not to arrays.

6. Available types

byte integer half integer integer long integer real long real long long real string (n) record (format)	all of these can be followed by array or name or array name

A half integer variable requires 16 bits (2 bytes) of storage. It holds an unsigned integer value, in the range 0-65535.

The statements reals long and reals normal are not available in IMP80.

7. Keyword and operator alternatives

```
! or | for comment
fn for function
const for constant
byte for byte integer
half for half integer
\ for ** (real exponentiation)
\\ for **** (integer exponentation)
<> or \= for #
~ for \ (logical 'not')
\== for ##
```

8. own initialisation

a) The statement

```
own integer A
```

declares an own integer variable A and initialises it to 0 (the default value when no value is specified).

The statement

```
own integer X, Y, Z=4
```

declares X, Y and Z and initialises them to 0, 0 and 4 respectively. In IMP9 this statement causes X, Y and Z to be set to 4, 4 and 4. Note the difference!

It is bad practice to rely on default initialisation values, especially in IMP80, where existing implementations do not have the same defaults. The statements above should have been given as

```
own integer A=0
own integer X=0, Y=0, Z=4
```

These are unambiguous, whichever version of IMP is used.

b) For convenience, constants used in own array initialisations can be followed by a repeat count, in brackets. This repeat count can be given as '(*)' where * represents the number of remaining array elements to be initialised.

Example:

```
own integer array VALUES (1:50) = %c
 17, 4, 6(3), 9, 22(17),
 100(*) {all the rest}
```

This also applies, of course, to constant and external array initialisation.

c) Own arrays can be multi-dimensional. As before, the bounds must be constants or constant expressions. The order in which array elements are assigned the initialising values is such that the first subscript changes fastest. Thus, for an array A(1:2,1:3), the order of assignment would be A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3).

9. Switch labels

Consider the following:

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

Instead of using a constant to specify a specific element of a switch vector, * can be used. It represents all the elements of the switch vector not defined elsewhere. Note that it does not have to come after the specifically defined switch labels.

10. Cycles

The permissible forms of cycle are these:

a) cycle (endless cycle)
:
repeat

b) while condition cycle
:
repeat

c) cycle
:
repeat until condition

d) for var = init, inc, final cycle
:
repeat

The unconditional instructions continue and exit can be used inside a cycle of any type. continue causes a branch to the next repeat; exit causes a branch to the statement following the next repeat.

Notes on the cycle types:

b) while cycles are executed zero or more times. When the cycle body consists of a single statement, the form

statement while condition

can be used.

Example:
SKIP SYMBOL while NEXT SYMBOL=' '

The IMP9 form while condition then statement is not allowed.

c) until cycles are executed one or more times. The simple form is

statement until condition

The IMP9 form until condition then statement is not allowed.

d) for cycles: the cycle variable must be of type integer; it should not be changed explicitly within the cycle body; (final-init) must be exactly divisible by inc; the cycle body is executed (final-init)//inc + 1 times or zero times, whichever is the greater; if the cycle body is not executed the cycle variable is set to be unassigned. It follows from this that a cycle starting

for I=10,1,8 cycle

will not be executed, but it will not be faulted either. This differs from IMP9, where the equivalent form

cycle I=10,1,8

would be faulted.

The simple form of for is

```
statement for var = init, inc, final
```

Example:

```
A(I)=0 for I=20,-1,1
```

[Going down in steps of -1 to 1 happens to be more efficient on EMAS 2900 than the more usual 1,1,20 form.]

11. start/finish blocks

The general form is

```
if cond 1 then start
     :
     :
     :
finish else if cond 2 then start
     :
     :
     :
     :
finish else if cond n then start
     :
     :
     :
finish else start
     :
     :
     :
finish
```

Notes

* Every start matches with the next occurring finish. If they enclose only one statement then they can be replaced by that statement.
  Example:

```
if cond 3 then start
  statement
finish else if .......
```

  can be expressed as

```
if cond 3 then statement else if ......
```

* then start can be replaced by start.

* if can be replaced by unless, the effect being to negate the condition following.

* Any of the statements starting "finish else" in the general form can be omitted, including the last one.

* If the condition controlling a start/finish block can be determined at compile-time then the IMP80 compiler may do so, and might not generate code for statements that cannot logically be executed. This is known as "conditional compilation".

12. Constants

a) An integer constant of any integer base from 2 to 36 may be specified. The form is

base_constant

where base is a decimal value and constant is an integer expressed with respect to the base. The letters A, B, ..., Y, Z can be used to represent the digits 10, 11, ..., 34, 35 in the integer.

Examples:

2_1010	ten in binary
8_12	ten in octal
16_A	ten in hexadecimal

An alternative form is provided for binary, octal and hexadecimal constants:

B'1010'	ten in binary
K'12'	ten in octal
X'A'	ten in hexadecimal

b) Named constants

Variables of all types can be given the attribute constant. This can be considered a special form of own variable, which cannot be changed from its initial value. However it is probably better to consider such variables as "named constants", since 1) this accords with their intended use, i.e. for replacing arithmetic or string constants within code by meaningful names; and 2) they do not have addresses, unlike other variables (but like constants).

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
```

Example:

```
constant integer NO=0, YES=1,
                 INPUT=1, CALCULATION=2,
                 OUTPUT=3
switch PHASE(INPUT:OUTPUT)
  :
  :
->PHASE(OUTPUT) if DONE=YES
  :
  :
PHASE(OUTPUT): ! Now print the results
  :
```

13. Strings

a) The keyword string may always be followed by a length specification.

Thus `string(10)array name ....`

and `string(255)name ......`

are permitted.

In EMAS 2900 IMP80, no use is made of the maximum length specification for string name and string array name variables.

[In other IMP80 implementations, however, a string name variable must have a maximum length specification and can only refer to ("be pointed at") a string variable of the same maximum length. The forms

```
string(*)array name .....
string(*)name ...........
```

are also provided, however, to enable declarations of reference variables which can point at any string variable.]

b) The string function FROMSTRING is renamed SUBSTRING.

c) A string resolution of the form

```
S -> (A).B
```

succeeds in IMP9 only if string S starts with string expression A. In IMP80, however, the resolution is interpreted as being equivalent to S -> JUNK.(A).B where JUNK is a "hidden" string (255) variable; that is, the resolution will succeed if A appears anywhere within S.

When converting an IMP9 program to IMP80, the following translation is recommended:

```
if S -> (B).C then ...                                    in IMP9
becomes if S -> NS1.(B).NS2 and NS1="" then C=NS2 and ...
                                                          in IMP80
```

[NS1 and NS2 are new string (255) variables]

This translation is still valid when the IMP9 statement is

```
if S -> (B).S
```

i.e. when C is S.

Unconditional resolutions can normally remain unchanged; they might succeed in IMP80 where they would fail in IMP9, but this is not significant unless you are expecting them to fail.

14. Records

a) The syntax of declarations in IMP80 differ from those in IMP9. They are of the form

```
record (format) ident, ...
record (format) array ident, ...
record (format) name ident, ...
record (format) array name ident, ...
```

"format" is either the name of a record format previously described, the name of a record previously declared, or the actual record format itself.

Example:

```
record format RF(integer I, J, K)
record (RF) R
```

and

```
record (integer I, J, K) R
```

are both valid and have the same effect, except that the first version declares a record format with identifier RF, which can be used elsewhere, clash with other identifiers, etc. Either of the above forms could be followed by the statement

```
record (R) P
```

which would declare a record with the same format as that of record R.

To summarise: the keyword record in IMP80 must be followed by the keyword format or by a bracketed format or format reference or record reference.

[This syntax change can cause difficulties when translating IMP9 programs: a routine spec such as

```
routine spec NAME1(record name NAME2, ...)
```

must now be converted to

```
routine spec NAME1(record (FORM2) name NAME2, ...)
```

The record format FORM2 is presumably declared somewhere in the program, since a record of this format is required in order to call the routine; but it might not be in scope at the routine spec statement, and may have to be moved so that it is.]

record spec statements are not allowed in IMP80.

b) The syntax of record format statements has been extended to permit alternative formats, i.e. to enable all or part of a record to be interpreted in different ways.

Example:

```
record format RF(integer A or byteinteger B, C, D c
                 or long real E)
record (RF) R
```

The record R can be considered to consist of an integer or three byte integers or a long real. Each alternative starts at the same address. Thus it follows that in

```
record format RF2 (byteintegerarray A(0:10) or c
                   string(10) S)
record (RF2) R2
```

R2_A(i) holds the ith character of string R2_S.

Note that all the sub-fields in a record format must have distinct identifiers.

In the first example above, the three alternatives were of different sizes. This is permitted: the alternatives have padding bytes appended to them to bring them up to the size of the largest. Thus when calculating the size of a record, use the size of the largest alternative.

When only part of a record is to have alternative formats, the alternatives must be bracketed within the record format statement.

Example:

```
record format RF3(integer TYPE, real RATIO,
                  (byte integer array A(1:20) c
                   or string (10) S c
                   or record (RF2) DATA),
                  string(*) name SN)
```

More than one set of alternatives can be given within a single record format; in addition, they can be nested. Redundant brackets round alternatives are allowed.

c) Records can contain records. The format of such a record must have already been defined, or be explicit. [A record clearly cannot contain a record with the same format as it itself has.]

Records can contain multi-dimensional arrays of fixed bounds, of any type.

Records can contain record names. The format of such a record name can be the same as that of the record containing it; thus

```
record format RF4(integer X, record (RF4) name NEXT)
```

is permitted.

15. external items

a) The IMP9 keyword extrinsic is replaced by external ... spec .

Example:

```
extrinsic integer array A(1:500)          in IMP9
```

becomes

```
external integer array spec A(1:500)      in IMP80
```

External variables can be initialised, like own variables, in declaration statements, but not in specification statements.

Example:

```
external integer array A(1:500) = 25(10), 14(72),
                                  16(22), 63(*)
```

b) External variables or procedures may be given an alias. The form

```
alias "..."
```

can follow the identifier name, in declaration statements or specification statements.

Example:

```
external real function spec SIN alias "MATH$DSIN"(real A)
```

The string constant specifies the string to be used for external linkage (i.e. the external reference). From within the program the item is referred to by its identifier, in the usual way.

16. Procedures as parameters

When a procedure has a procedure parameter the specification of the latter is given in the parameter list, not in a subsequent spec statement.

Example:

```
routine X(integer Y, routine Z(real A), string (10) S)
```

John M. Murison

PREFACE

This edition of the IMP Language Manual is intended to replace in part, the Edinburgh IMP Language Manual published in 1970. Two associated manuals are being prepared - the 'Edinburgh IMP/FORTRAN System Library Manual' which will contain details of all the items in the IMP system library and 'A Syntactic and Semantic Definition of the IMP Language as Implemented at the Edinburgh Regional Computing Centre'. This edition contains some material from its predecessor, and some new material. The sections on Strings, Records, Conditional Instructions and Input/Output facilities have been completely rewritten. The references to Job Control requirements have been removed; the user is referred to the User Manual for the appropriate computer for information on this topic.

The IMP programming language has been implemented on several computers. This manual describes the current version running on ICL 4/75 and IBM 370/158 computers. These machines both use 32 bit words and byte addressing. Other implementations of the language use machines with different word lengths and addressing. The user who is likely to move his IMP programs to other machines should ensure that he is aware of these and other differences between implementations of the language.

The Manual is intended as a reference manual rather than a teaching manual. Little attempt has been made to order material in a sequence suitable for a newcomer to programming. It has been assumed that the reader has some knowledge of programming in IMP or a similar language. On the other hand facilities which are not usually found in other high level languages, e.g. Records and Strings, are described in considerable detail since it is likely that in these areas at least the manual will have to serve as a teaching manual.

The Manual is the work of many people in the Edinburgh Regional Computing Centre. Particular mention should be made of the contributions of Keith Yarwood, Gordon Burns, Peter Stephens and Andrew McKendrick. Anne Tweeddale, Laura Lang and Dorothy Kidd, together with staff of the Reprographics section were all involved in the production of the Manual.

Roderick McLeod
Editor
May 1974

CONTENTS

SECTION	TITLE
1	Basic Language
2	Arithmetic Operations
3	Logical Operations
4	Control of Sequence of Instructions
5	Storage Allocation and Block Structure
6	Routines and Functions
7	Store Mapping
8	Strings
9	Records
10	Input/Output
11	Aids to Program Development
12	Compile Time Faults
13	Run Time Faults
14	Fault Trapping
15	Internal Character Code
16	Routines, Functions and Maps in the IMP Library
	Index

SECTION 1 - THE BASIC LANGUAGE

TYPING CONVENTIONS FOR IMP PROGRAMS

Programs written in IMP are typed on some form of data preparation equipment, for example a card punch or a teletype, according to the following rules:

1. Only the first 72 character positions in a line may be used.

2. Statements must be separated by a newline character or a semi-colon.

3. Apart from text contained within quotes all letters must be in upper case.

4. Spaces are only significant where they appear within quotes, or where they follow a word which is a delimiter .

5. Delimiters, which have a pre-defined meaning in the language, are sequences of symbols. When these are letters they are preceded by a % sign to distinguish them from NAMES. The % indicates to the compiler that the letters which follow are to be treated as a delimiter. The effect of the % ceases at the first character that is not a letter. So, for example, in the declaration of an integer it is essential to have a space between the delimiter and the name being declared.

   ```
   %INTEGER MINE
   ```

   If the space were omitted the compiler would treat it as one delimiter 'INTEGERMINE'.

6. Text is for some purposes enclosed within quotes. In these cases everything within the quotes is treated as part of the text, including spaces and newlines. If the quote character is required in the text it has to be typed as two separate quote characters to distinguish it from the terminators.

7. If it is necessary to continue a statement on to a new line the sequence '%C' should be used at the end of the first line. It may be used at any convenient point in the statement. If it is used within a delimiter a '%' must be used at the beginning of the continued line.

8. A mis-typed character can be deleted by use of the double quote (") delete character, immediately after it. Multiple double quote characters can be used to delete a sequence of wrong characters as far back as the beginning of the line.

NAMES

Names are used in IMP programs for the following entities:-

Arithmetic, Logical and String Variables
Records and Record sub-fields
Routines and Functions
Simple Labels
Record Formats and Array Formats

A name consists of a letter followed optionally by a sequence of up to 254 letters and/or numerals in any order. It is recommended that meaningful names be used where possible in order to improve the legibility of programs. The following are valid IMP names:

ROW1
NUMBER OF BLOCKS
C1900TO1970
END POINT

COMMENTS

Comments should be used to make programs more meaningful both to the originator and to anyone else who needs to work on them. Either the delimiter %COMMENT or ! may be used to introduce a comment. A comment is a statement and must be separated from the statement before it and after it by the usual separators: newline or semi-colon.

DELIMITERS

These are a pre-arranged and pre-defined set of sequences of symbols which have fixed absolute meanings to the compiler. They include:

Operators: arithmetic, assignment, relational, logical, sequential

Separators: e.g. , %COMMENT

Brackets: e.g (,), %BEGIN, %END

Declarators: e.g. %OWN, %LONG, %REAL

Specificators: e.g. %LIST, %RETURN.

NOTE

Some symbols of the language have more than one meaning, but are defined in a context which normally is unambiguous.
For example:

!	or (operator)
!	comment (delimiter)
!	modulus sign (used in pairs)

VARIABLES

Variables are locations in the store of the computer which are used to hold numeric or textual information. Each variable or group of variables used in a program is given a name by the programmer. Variables can be divided into three groups:

Arithmetic variables	-	see below
String variables	-	see Section 8
Records	-	see Section 9

ARITHMETIC VARIABLES

There are five types of arithmetic variable, the first three can only hold whole numbers, the last two can additionally hold numbers which include fractional parts.

Type	Length in bits	Range of Values
%BYTEINTEGER	8	0 : 255
%SHORTINTEGER	16	-32768 : 32767
%INTEGER	32	-2147483648 : 2147483647
%REAL	32	-7@75 : 7@75 (approx.)
%LONGREAL	64	-7@75 : 7@75 (approx.)

The format 7@75 means '7 multiplied by 10 to the power of 75', see below.

The choice of which integer variable to use can be made on the basis of the values it is required to hold. The choice between %REAL and %LONGREAL will depend upon the accuracy required. %REAL variables can only hold values to a precision of between 6 and 7 significant decimal digits, whereas %LONGREAL variables hold values to a precision of between 14 and 15 digits on the ICL 4/75 and between 15 and 16 digits on the IBM 370. Further details of the representation of variables can be found in the hardware manual for the appropriate computer.

CONSTANTS

DECIMAL CONSTANTS

Decimal constants are written in a straightforward notation:

```
2.538   1       .25     -17.28@-1   1@7
```

The last two examples mean -1.728 and 10000000, respectively.

The numerical part (mantissa) can be written in a number of ways:

```
15      015     15.     15.000
```

all of which are equivalent. The exponent, where present, consists of '@' followed by optional sign and decimal digits.

THE CONSTANT π

The symbol 'π' can be used in IMP programs. It has the value 3.141592653589793. It can be used in any expression requiring the value of π, for example:

```
AREA =π*RADIUS**2
```

SYMBOL CONSTANTS

Symbol constants having a numerical value within the range 0 to 127 may be written by enclosing the required symbol within quotation marks:

```
'*'
```

The internal code values of symbols are given in Section 15. A %BYTE can hold 1 symbol, a %SHORTINTEGER 2 symbols and an %INTEGER 4 symbols. A constant containing more than 1 symbol is preceeded by 'm' for example:

```
M'XYER'
```

NOTES

1. Spaces and newlines are always significant between quotes and the numerical values of the symbols (they are not zero) will be included in the value of the constant if any should appear.

2. If a single quotation mark is required as part of a constant it must be replaced by two single quotation marks.

3. Each symbol occupies one byte i.e. 8 bits of the %INTEGER or %SHORTINTEGER location, any unused bytes at the most significant end of the location will be set to zero.

HEXADECIMAL CONSTANTS

A hexadecimal constant consists of a string of hexadecimal digits, enclosed in quotation marks and preceded by the letter X. In addition to 0, 1, 2, ..., 9 which have their usual significance, a hexadecimal digit may also be A, B, C, D, E or F which stand for the decimal numbers 10, 11, 12, 13, 14 or 15 respectively:

```
X'2A'
```

would have the same value as the decimal number 42 i.e. 2 * 16 + 10, since A represents 10 in the scale of 16 i.e. the hexadecimal scale.

Each hexadecimal digit occupies a location of four bits length; hence a %BYTEINTEGER variable can hold two such digits, a %SHORTINTEGER variable four, and an %INTEGER variable eight.

```
%INTEGER I
I = X'6789ABCD'
```

BINARY CONSTANTS

This type of constant consists of a string of binary digits, enclosed in quotation marks and preceded by the letter B. A binary digit, which occupies just one 'bit' of store, may be either 0 or 1. Eight may therefore be stored in a %BYTEINTEGER variable, sixteen in a %SHORTINTEGER variable, and thirty-two in an %INTEGER variable.

```
%BYTEINTEGER M
M = B'01011011'
```

NOTES

1. Hexadecimal and Binary constants may appear in arithmetic expressions; they will, however, be most used in conjunction with the logical operators (see Section 3).

2. If the number of digits or characters which appear in a constant is less than the maximum permissible e.g. a constant of three hexadecimal digits being assigned to a %SHORTINTEGER variable, then the value assumed is the same as if the digits or characters had been right justified in a location of 32 bits and the remaining bit positions filled with zero bits.

   For a %SHORTINTEGER variable N,

   ```
   N = X'2AB'
   ```

   will have the same effect as

   ```
   N = X'02AB'
   ```

3. Both the 4/75 and the 370 computers store integers in twos-complement form. When writing programs that are likely to be moved to other machines programmers should consider carefully the possible change of arithmetic value of binary and hexadecimal constants.

DECLARATION OF VARIABLES

All variables must be declared at the head of the block in which they are used, or at the head of an outer block. (see Section 5). A declaration consists of a type delimiter followed by one name or a list of names separated by commas:

```
%INTEGER FIRST
%LONGREAL TOP, BOTTOM, LARGEST
```

DECLARATION OF ARRAYS

Arrays of variables are declared in a similar manner. The bounds of the array are written after the name, in brackets:-

```
%INTEGERARRAY IN(1:10),OUT(1:20)
```

Multi-dimensional arrays of up to seven dimensions may be declared:

```
%SHORTINTEGERARRAY BITLIST (-4:4,1:2,10:100,1:2)
```

When accessing an individual element of an array the name must be written, followed by an integer expression for each dimension:

```
BITLIST (J+I,J,10,1) = 0
```

The values of the expressions must be within the bounds for the relevant dimension otherwise the run time fault 'ARRAY BOUND FAULT' will occur.

ARRAYS WITH VARIABLE BOUNDS

It is possible to use integer expressions involving variables for the bounds of arrays. The variables referenced should be global to the block containing the declaration. An example of the use of this is:

```
%BEGIN
%INTEGER TOP
    READ (TOP)
        %BEGIN
        %INTEGERARRAY TABLE (1:TOP)
        .
        .
        .
```

%OWN VARIABLES

The delimiter %OWN may be written before the type of a variable. It has the following effects:

1. The variable will remain in existence for the duration of the program. If it is within a routine or function it will retain its value between calls of the routine, which is not the case for other variables local to routines and functions.

2. The variable can be set to an initial value e.g.

```
%OWNINTEGER MAXIMUM = 527
```

   The value must be expressed as a constant. If no initial value is provided the variable will be set to zero.

3. %OWN arrays may be declared, but only of single dimension and with constant bounds. Elements may be initialised by writing a list of values, separated by commas and newlines. Note that in this situation the %C continuation delimiter is not required, even if the list extends on to several lines, so long as the line is terminated with a comma.

```
%OWNBYTEINTEGERARRAY HEXTAB (0:15) = '0','1',
'2', '3', '4', '5', '6', '7', '8', '9',
'A', 'B', 'C', 'D', 'E', 'F'
```

   Note that there must be the same number of constants as there are elements in the array. If one constant is to be repeated it may be written once, followed by a count in brackets:

```
%OWNINTEGERARRAY IN(1:10) = 1,0(9)
```

Apart from these points %OWN variables are used in exactly the same way as normal variables.

%CONST VARIABLES

The delimiter %CONST may be used in place of %OWN if the variables are to have a constant value for the duration of the program. Any attempt to assign values to them other than in the declaration statement will result in a compile time fault. They are initialised in the same way as %OWN variables.

%EXTERNAL VARIABLES

The delimiter %EXTERNAL written before a variable gives it all the attributes of an %OWN variable, and additionally makes it available to other, seperately compiled, programs or routines. (See Section 6)

%REALSLONG

This statement can appear at any point in a program. Its effect is to cause the compiler to interpret any subsequent %REAL statements in declarations, function types, and parameter lists for routines or functions as %LONGREAL. The statement %REALSNORMAL causes the compiler to revert to its default mode.

SECTION 2 - ARITHMETIC OPERATIONS

ARITHMETIC OPERATORS

The following operators may be applied to real and integer variables in arithmetic expressions. Logical operators are described in Section 3.

The Operators

Symbol	Meaning
+	addition
-	subtraction
*	multiplication
/	division
//	integer division
**	exponentiation (i.e. Y**3 = Y cubed)

NOTE

Implied multiplication should be avoided; in IMP it is only accepted in the case of a constant followed by a name, e.g. 34X. Thus, whereas in common mathematical notation XY may mean X multiplied by Y, the IMP compiler will correctly interpret this as the name XY and will fault the line as variable name not set. The correct presentation is X * Y.

PRECEDENCE OF ARITHMETIC OPERATORS

Rules, following normal practice, have been established to which the IMP compilers conform and for arithmetic operators the order of precedence is given below, whilst for logical and mixed arithmetic logical expressions the rules are given in Section 3.

Highest precedence	(1)	**
	(2)	* or / or //
Lowest precedence	(3)	+ or -

The programmer may override the natural order of evaluation by using brackets. Extra brackets are acceptable to ensure clarity and to remove doubts. The left hand precedence between operators otherwise of equal precedence agrees with normal mathematical usage.

Expression:	Meaning:
A - B + C	(A - B) + C not 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

Expressions may be real or integer according to context. An expression is evaluated as real if it is being assigned to a real variable or passed as a real value parameter (see Section 6). An expression is evaluated as integer 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.

INTEGER EXPRESSIONS

An integer expression may contain integer constants and variables declared to be of type integer; these may be simple or subscripted variables, or integer functions (see Section 6).

There are two division operators in IMP:

Arithmetic Division (/)

Integer Division (//)

Arithmetic Division may occur in real expressions; Integer Division may only occur in integer expressions.

1. Arithmetic Division (/).
   If this operator occurs is an integer expression, it must always yield an integer result.
   ```
            N*(N-1)/2 is always satisfactory,
   but      N*((N-1)/2) fails if N is even.
   ```

Note that Integer Division is preferred in the context of integer expressions.

2. Integer Division (//).
   This operation always yields an integer result and is exactly comparable to the Algol integer division. The result consists of a quotient whose sign is determined algebraically, and a remainder which is ignored. Note that both the dividend and the divisor for an integer division must be integer expressions.
   ```
       9//2    = 4
   100//15     = 6
   (-9)//2     =-4
   ```

The definitions of division given above ensure that integer expressions always yield an integer result. Integer expressions are always evaluated single length (32 bits) and integer overflow will occur if at any point in evaluation the capacity of a single length variable is exceeded. If the calculation requires a wider range of numbers, real arithmetic must be used.

Exponentiation is carried out by repeated multiplication. In integer expressions the exponent must be an integer expression with a value in the range $0<=n<=63$.

THE INTRINSIC INTEGER FUNCTIONS 'INT' AND 'INT PT'

The built-in integer function INT yields as its result the nearest integer from a real expression, and may be used in an integer expression. The integer function INT PT yields as its result the the value of the integral part of the quantity specified on entry. Strictly the result is the integer that is less than or equal to the expression, hence:

INT PT(-3.7) is -4

REAL EXPRESSIONS

A real expression may have, as operands, simple variables or subscripted variables declared to be of type real or integer but the result must be assigned to a real variable (see below).

In evaluating a real expression, the compiler will work to single precision unless a double precision variable (i.e. long real) is encountered; the working will then be in double precision. Hence, double precision should only be used where an estimate of the accuracy attainable relative to the input data requires it. Double precision working is time and space consuming. Nevertheless, it should be noted that floating point arithmetic does not guard against loss of accuracy due to cancellation of significant figures in addition and subtraction. This loss of accuracy is reduced by use of double precision, although in the case of the 4/75 this is slightly less effective than in the case of the 370 computer.

Exponentiation in real expressions is carried out as for integer expressions except that the exponent must be an integer expression with a value in the range -255<=n<=255.

A real expression which contains only integer operands and the operators +, -, *, is evaluated in integer mode and subsequently converted to real. Otherwise, each integer is converted to real before being used.

SUB-EXPRESSIONS

Generally, a bracketed sub-expression is treated as an expression in its own right and the rules of precision given above apply. Therefore, a real sub-expression containing only single precision variables will be evaluated single precision and the result converted to double precision if necessary.

ARITHMETIC ASSIGNMENTS

The two assignment operators are:

1. =

Here, the RHS is evaluated and the value is assigned to the destination given by the LHS provided that the lengths are compatible. The Run Time Fault 'CAPACITY EXCEEDED' will occur if an attempt is made to assign too large a value to a variable using this operator.

2. <-

Here, the least significant 8 or 16 bits of the RHS are assigned respectively to the byte integer or short integer location on the LHS. The remaining bits of the RHS are ignored. For assignment to 32 or 64 bits, <- is treated exactly as =.

The general arithmetic assignment instruction assigns the result of evaluating an arithmetic expression to a variable. An integer variable may only have assigned to it the result of an integer expression but either an integer or real expression may be assigned to a real variable.

Examples of valid assignments:

```
A(P,Q) = 1 + 2*COS (2 * N*(X + Y))
X = (U + V)/(Z + W) + F(M,N)
I = I + 1
```

where A, F are real arrays
X, Y, U, V, Z, W are real variables
I,P,Q,M,N are integer variables

THE INTRINSIC REAL FUNCTION 'FRAC PT'

This built in function returns as its result the fractional part of a real expression. Note that the fractional part is always treated as being greater than or equal to zero. Hence:

FRAC PT(-4.6) is .4

MODULUS OF EXPRESSIONS

Two methods are provided for calculating the modulus or absolute value of an expression. Modulus signs (!) used before and after an expression bracket the expression and calculate the modulus without changing its type:

```
%INTEGER I,J
%LONGREAL X,Y
    I=!J+I!
    X=3.4*!Y+SIN(Y)!
```

In the first example the expression is left as an integer expression and in the second as a real expression. The alternative method is to use the built in %LONGREALFN MOD. This always returns a real result, hence can only be used in a real expression.

ASSIGNMENT OF SYMBOLS

Instructions to assign symbols to integer variables are written in a form very similar to those which assign numbers, but the symbol concerned is written between a pair of quotation marks:

```
%INTEGER I, J, K
    I = '*'
    J = M'ABCD'
    K = '7'
```

Note that the last two instructions assign the SYMBOLS ABCD and 7 to J and K respectively. On the other hand, the instructions:

```
J = ABCD
K = 7
```

assign to J the NUMERICAL value currently stored in the variable named ABCD, and to K the NUMBER 7.

SECTION 3 - LOGICAL OPERATIONS

LOGICAL OPERATORS

The logical operators are as follows:

left shift	<<
right shift	>>
and	&
or	!
exclusive or	!!
not	¬ (or~)
assignment	<-

Logical operations are performed on bit patterns stored in integer variables, including elements of integer arrays, but not in real variables.

The IMP programmer may specify these operations on, by, or between byte integer, short integer or integer variables and may similarly so assign the results, but he must take precautions to understand the implications.

The IMP Compiler always makes up the bit pattern to 32 bits before carrying out the operation, thus:

Byte Integers	-	the left hand 24 bits are filled with zeros
Short Integers	-	the left hand 16 bits are filled with copies of the sign bit (the most significant bit of the %SHORTINTEGER

As with Arithmetic operations both '=' and '<-' are available as assignment operators but care should be taken on the assignment of the result of a logical operation.

1. The assignment operator '=' treats the result of the logical operation as a 32 bit signed integer and attempts to perform an arithmetic assignment to the designated variable. Hence it is not always possible to put the result back in a variable of the same type as that in which the operand was originally held, (if this was a byte or short integer).

2. The assignment operator '<-', however, simply copies the requisite bit pattern to the designated variable so that 32 bits are assigned to an integer; the 16 least significant bits to a short integer; the 8 least significant bits to a byte integer.

The choice of assignment operator depends on the context of the program.

SETTING UP BIT PATTERNS IN INTEGER VARIABLES

A bit pattern is normally specified in the program as a hexadecimal constant.

1. The compiler sets up the pattern right justified and makes it up to 32 bits by padding with zeros at the most significant end of the location and stores it in a table of constants in the User's program. Assignment of a constant to a short integer or byte integer location therefore needs care.

2. The arithmetic assignment operator = will always copy a constant (set up as in 1.) into a 32-bit integer variable.

3. The arithmetic operator = will only copy from the 32-bit constant location into a short integer or byte integer if by so doing the numerical value remains unchanged during the operation:

```
%SHORTINTEGER I
I = X'F000'
```

   will fail, 'CAPACITY EXCEEDED' (Fault 30)

4. The assignment operator <- will treat the constant as a 32-bit pattern (form of (1) above) and will copy the 32, 16, or 8 least significant bits to the designated integer variable. Thus, contrary to the example in 3:

```
%INTEGER J
%SHORTINTEGER I
I <-X'F000'
J = I
```

   The first instruction will not fail, but will set up in I the bit pattern:

   1111 0000 0000 0000

   The second instruction will set up in J the bit pattern:

   1111 1111 1111 1111 1111 0000 0000 0000

SHIFT OPERATORS

The IMP programmer may specify a first operand (I) to be shifted by a second operand (N) and the result to be placed either back in the location of the first operand or in a new location (J). He may declare I, N or J to be byte, short or integer variables but he must check that these are meaningful.

The Compiler makes the operands (I) and (N) up to a 32 bit pattern (as described above) before the operation takes place, and then selects the six least significant bits of the second operand (N) to determine the amount of shift. The remaining 26 bits of N are ignored, thus the shift will be positive in the range 0 to 63.

I >> N has the effect of I >>(N &63)

NOTES

Left Shift

1. Any bit positions vacated at the right-hand of the 32-bit pattern are filled with zeros.

2. Any bits shifted off the left-hand end of the 32 bit pattern are lost.

Right Shift

1. Any bit positions vacated at the left-hand end of the 32-bit pattern are filled with zeros.

2. Any bits shifted off the right-hand end of the 32 bit pattern are lost.

The explicit library function SHIFTC is provided for shifting a bit pattern cyclically. It takes two integer expressions as parameters and returns as a result the bit pattern in the first expression shifted by the number of places specified by the second parameter. Any bits shifted off one end of the 32 bit pattern re-appear at the other end. A positive shift is to the left and vice verca.

THE OPERATOR 'NOT'

This is represented by ¬ and operates on a single operand (I). The IMP programmer may declare I to be a byte, short or integer variable. The compiler expands this to 32 bits in the computer location before the operation is carried out.

The operator 'not' inverts the values of the bits:

0's are changed to 1's
1's are changed to 0's

If I contains the bit pattern

0.....01 00 11 00 11

then ¬I gives 1.....10 11 00 11 00

i.e. ¬I + 1 = -I

THE OPERATORS 'AND', 'OR', 'EXCLUSIVE OR'

These operations are carried out between the bit patterns stored in two integer variables. Whether the IMP program has declared these as byte, short or integer variables the compiler expands each bit pattern to 32 (as explained above) and the result, a single bit pattern, is held in a 32 bit location. The programmer must choose an integer of suitable length in to which to assign the result, and should note the effects of the '=' and '<-' operators in this context.

'and' (&) result pattern contains a 1-bit where the two source patterns both have 1-bits and contains 0-bits elsewhere.

'inclusive or' (!) result pattern contains a 0-bit where the two source patterns both have 0-bits and contains 1-bits elsewhere.

'exclusive or' (!!) result pattern contains a 1-bit where the bits in the source patterns are different and contains 0-bits elsewhere.

These rules may be summarised thus:

	&	!	!!
0 : 0	0	0	0
0 : 1	0	1	1
1 : 0	0	1	1
1 : 1	1	1	0

PRECEDENCE FOR MIXED ARITHMETIC AND LOGICAL OPERATIONS

Arithmetic and logical operators may be mixed in an integer expression. The rules of precedence are then:

```
¬          (most binding)
** >> <<
* / // &
+ - ! !!    (least binding)
```

Example of use of operators:

To 'unpack' the contents of an integer location I, into four sections each of 8 bits length and store them in the positions of a byte integer array B:

```
%INTEGER I,J
%BYTEINTEGERARRAY B(0:3)
    .....
    I=........
    .....
    %CYCLE J=0, 8, 24
        B(J/8)= I>>(24-J)& X'FF'
    %REPEAT
    ......
    ......
```

NOTES

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

2. The shift, having a higher precedence than the &, is effected first.

THE FUNCTION 'BITS'

The explicit library function BITS takes one parameter which must be an integer expression. It returns as a result the number of bits in the evaluated expression. For example if the integer IN contained a character read from a binary paper tape and it was required to check whether it had odd parity one could write:

```
->ODD %IF BITS(IN)&1=1
```

SECTION 4 - CONTROL OF SEQUENCE OF INSTRUCTIONS

INTRODUCTION

All programs require facilities to control the order in which instructions are obeyed. A variety of facilities is provided in IMP.

1. Routines and Functions: these are used to group instructions together and give them a name - READ and SIN are examples of Routines and Functions (see Section 6).

2. Conditional Statements: these are used to make the execution of single instructions or groups of instructions, dependent on the result of one or more tests (see below).

3. Conditional Repetition and Cycles: these are used to execute a sequence of instructions repeatedly. The number of repetitions can be controlled by a variable or by a condition.

4. Jumps to Switch Labels: these are used where a program is required to take one of many different paths depending on the value of an expression.

5. Jumps to Simple Labels: these are available as alternatives to 2, 3 and 4 above.

CONDITIONS

Conditions are described here in the context of a simple conditional statement for reasons of clarity but they can also be used in more complex conditional statements and cycles as described later. An example of a simple conditional statement is:

```
%IF A>2 %THEN PRINT(A,5,3)
```

which can be represented as:

```
%IF <condition> %THEN <unconditional instruction>
```

The following types of statement can be made the subject of a condition and are known as 'unconditional instructions'. The phrase 'unconditional instruction' is used to denote a single instruction which is executed once each time it is reached. Examples of instructions in this group are:

Type	Example	Notes
Assignments	A = 27	
Routine Calls	Print (A,2,3)	
Jumps	->27	See below
Special Jumps	%RETURN	See Section 6
	%RESULT=	See Section 6
	%STOP	See below
	%MONITORSTOP	See Section 11
	%MONITOR	See Section 11
	%EXIT	See below

The <condition> part is made up of two expressions separated by a relational operator. The relational operators and their meanings are:

```
=    equal
>    greater than
<    less than
>=   greater than or equal
<=   less than or equal
#    not equal
```

A double sided condition may be used, in which case the whole condition is true only when both sides of the condition are true, for example:

```
%IF 9>=I>=0 %THEN %PRINTTEXT'IN RANGE'
```

The %PRINTTEXT instruction will only be executed if I is less than or equal to 9 and at the same time I is greater than or equal to 0.

More generally multiple conditions may be linked using the %AND and %OR operators. These take their logical meanings; example:

```
%IF A=1 %OR A=10 %THEN NEWLINE
```

Note that if both %AND and %OR are used in the same conditional statement, then in order to avoid ambiguity the conditions they link must be separated from each other by brackets

```
%IF A=10 %AND (S='NOW' %OR S='SOON' %OR S='LATER') %THEN A=0
```

In future examples <condition> implies any of the above forms.

FURTHER USE OF CONDITIONS

The unconditional instruction which was made conditional in the first example can be replaced by a sequence of unconditional instructions enclosed in the bracket pair %START and %FINISH. In this case the %THEN may be omitted, if preferred.

```
%IF <condition> [%THEN] %START
        A = 1
        NEWLINE
        .
        .
        .
        .
        %FINISH
```

Alternatively where a small number of instructions is involved they can be linked with the operator %AND. Note that here %AND is as in common usage, and the linked instructions are obeyed in the sequence in which they appear in the text of the statement.

Example: `%IF <condition> %THEN NEWPAGE %AND LINE = 0`

ALTERNATIVE PATHS

The %ELSE operator can be used to indicate the path to be taken when the condition is found to be not true

```
%IF <condition> %THEN A=1 %ELSE A=0
```

and hence

```
%IF <condition>     [%THEN] %START
                    .......
                    .......
                    %FINISH %ELSE %START
                    .......
                    .......
                    %FINISH
```

USE OF %UNLESS

In all the examples above %IF can be replaced by %UNLESS. This has the effect of testing that the condition is not true. Hence

```
%IF A#B %THEN A=0        and
%UNLESS A=B %THEN A=0
```

will have the same effect.

FURTHER SIMPLIFICATIONS

The simple condition can be written with the unconditional instruction first if preferred; example:

```
A=0 %IF A=B
```

Note that %THEN is no longer needed, but note also that neither %START %FINISH, %ELSE nor linking of instructions with %AND can be used with this form.

REPEATED EXECUTION OF INSTRUCTIONS AND CYCLES

In many situations it is useful to execute a single instruction or group of instructions repeatedly, the number of repetitions being controlled by a condition or by a control variable.

CONDITIONAL REPETITION

The simplest form is

```
%WHILE <condition> %THEN <unconditional instruction>
```

The condition is tested and if found to be true then the unconditional instruction is executed. The whole operation is repeated until the condition ceases to be true. For example

```
%WHILE NEXTSYMBOL= ' ' %THEN SKIPSYMBOL
```

has the effect of skipping any space characters on the input stream. The inverse form is

```
%UNTIL <condition> %THEN <unconditional instruction>
```

In this case the unconditional instruction is executed first and then the condition is tested. The whole operation is repeated until the condition is found to be true. Note that when using %UNTIL the unconditional instruction is always obeyed at least once whereas when using %WHILE the condition is tested before executing the unconditional instruction and may be false at the first test.

EXTENSIONS TO CONDITIONAL REPETITION

The unconditional statement in the above examples can be replaced by a group of unconditional statements linked by %AND, or enclosed by %CYCLE and %REPEAT; examples:

```
%UNTIL J=999 %THEN READ(HOLD(J)) %AND J=J+1

%WHILE NEXTSYMBOL#NL [%THEN] %CYCLE
    READSYMBOL (I)
    S=S.TOSTRING(I)
%REPEAT
```

An alternative form where only one unconditional instruction is involved is

```
<unconditional instruction> %WHILE <condition>
```

for example

```
SKIPSYMBOL %WHILE NEXTSYMBOL=NL
```

%CYCLES WITH CONTROL VARIABLES

Instead of using a condition to control the number of repetitions of a %CYCLE a control variable may be used. The control variable must be a variable declared to be an %INTEGER. (A %SHORTINTEGER or %BYTEINTEGER may not be used).

The cycle is written:

```
%CYCLE <control variable> = <start>,<step>,<final>
```

where <start>, <step> and <final> are all integer expressions; example:

```
%CYCLE I=1, 1, 10
        IN(I)=0
        IN1(I)=0
%REPEAT
```

On entry to the %CYCLE statement a check is made that <step>#0 and that (<final> - <start>)/ <step> is a positive integer.

If the test fails the fault 'INVALID CYCLE' will occur which will normally cause the program to terminate (see Section 13). Otherwise the control variable is set to the value <start>. The instructions between %CYCLE and %REPEAT are executed, a test is made for equality between the control variable and <final> and if unsuccesful the control variable is incremented by <increment> and the sequence is repeated until the control variable reaches the value <final>.

The control variable can be used in expressions within the cycle but it should not have anything assigned to it. The effect of doing so is undefined.

THE %EXIT INSTRUCTION

This may be used at any point within a %CYCLE - %REPEAT block. The effect is to go to the instruction following the %REPEAT, preserving the current value of the %CYCLE variable.

```
%CYCLE N=4, -1, -100
    .
    .
    .
    %IF IN(N)=' ' %THEN %EXIT
    .
    .
    .
%REPEAT
```

INDEFINITE CYCLES

The delimitors %CYCLE and %REPEAT may also be used without a condition or a control variable. The effect is to repeat the instructions between the %CYCLE and %REPEAT indefinitely. An %EXIT or a jump should be included among the instructions.

NESTING CONDITIONS AND CYCLES

Conditions and Cycles can be nested to any depth so long as all of the instructions relating to the nested condition or cycle are contained between the %START - %FINISH or %CYCLE - %REPEAT of the outer condition on cycle; example:

```
READSTRING(TEST)
%WHILE TEST # 'END' %CYCLE
    !CHECK FOR NON-ALPHA CHARACTERS IN NAMES
    %CYCLE I=1, 1, LENGTH(TEST)
        %UNLESS 'A'<= CHARNO(TEST,I)<='Z' %START
            %PRINTTEXT'INVALID NAME'
            NEWLINE
            %STOP
        %FINISH
    %REPEAT
    NAME(POINTER) = TEST
    POINTER=POINTER+1
    READSTRING(TEST)
%REPEAT
```

SWITCH VECTORS

Switch vectors are used in situations where it is necessary for a program to take one of several paths depending on the value of an expression. Switch vectors must be declared at the beginning of the block or routine in which they are used, together with declarations of variables. The declaration consists of a name followed by a pair of integer constants which define the range of vectors to be used, for example:

```
%SWITCH     SWA(1:10)
```

At the point at which the branching is to take place a statement such as

```
->SWA (I+J)
```

should be used. The expression I+J can be replaced by any suitable integer expression which has a value in the range declared for SWA.

Finally at the points to which control is to pass the label should be written, example

```
SWA(3):     PRINT(N,3,4)
```

Note that the maximum range allowed for switch vectors is -32767 to +32767. It is not necessary to include labels for all positions declared. If an attempt is made to jump to a non-existent label a run time fault 'SWITCH VECTOR NOT SET' occurs.

JUMPS TO SIMPLE LABELS

Simple labels may be used in IMP. EitherIMP NAMES or unsigned positive integers in the range 1 to 16383 may be used. In order to improve the legibility of programs the use of meaningful names for labels is recommended. The label declaration comprises the identifier followed by a colon, for example:

```
ENDFILE:
27:
2222:
```

Note that if NAMES are used they do not conflict with names of variables, routines etc. because they are held in a seperate list by the compiler.

Jumps may be made to labels from anywhere in the same block; examples:

```
->ENDFILE
->2222  %IF     FLAGS=1
```

The following restrictions exist in relation to the use of labels and jumps.

1. All declarations of variables, arrays etc., must precede any labels or jumps in the same block.

2. No attempt should be made to enter a %CYCLE - %REPEAT block other than through the %CYCLE.

3. When using the sequence:-

```
....%START
    .
    .
    .
    .
    .
    %FINISH %ELSE......
```

Then no attempt should be made to jump into the block between %START and %FINISH.

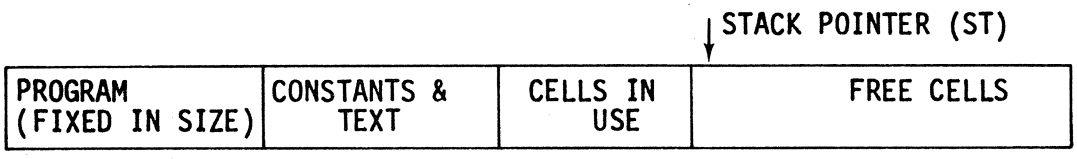
%STOP AND %MONITORSTOP

The Unconditional Instruction %STOP can appear at any point in a program. When it is reached the program is terminated and control is returned to the operating system. A %STOP statement is effectively compiled at the statement %ENDOFPROGRAM. The statement %MONITORSTOP has the same effect as %STOP with the added feature of printing a trace of the program and values of scalar variables before termination. (see Section 11)

SECTION 5 - STORAGE ALLOCATION AND BLOCK STRUCTURE

THE STACK

At compile time the Compiler translates the user's IMP program into machine instructions and this object code is loaded by the Loader which then passes control to the user's program. The data space which is required by the user's program is then allocated in a dynamic fashion as the program proceeds. The following simplified description of the data store (the stack) illustrates the principles of storage allocation.



Each cell represents a unit of computer store and can be used to hold an arithmetic or string variable or a %RECORD. A cell may be imagined as being of variable size from 8 bits upwards, appropriate to the length of the entity being held.

At any time during the running of a program the stack pointer (ST) points to the next available location - that is, it contains the address of the next free cell.

In the following examples shaded areas represent locations which hold information essential to the program, such as array dimensions, but of no interest to the user since he cannot access them. Cells which are allocated to variables are indicated by the name given to the variable.

STORAGE ALLOCATION DECLARATIONS

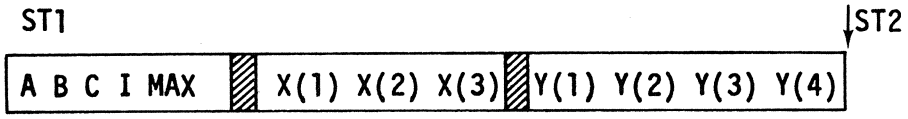
The following declarations are typical of those which allocate storage space:

%REAL %INTEGER %REALARRAY %INTEGERARRAY

To illustrate the stack mechanism the following example is considered:

```
%BEGIN
%REAL A, B, C ; %INTEGER I, MAX
%REALARRAY X(1:3),Y(1:4)
```

After the above declarations the stack would be:



ST1 is the position of ST before %BEGIN and ST2 its position after the declarations. Any further declarations advance ST by the appropriate amount, likewise any activity initiated by the instructions in the body of the block may cause ST to be advanced (either explicitly or implicitly) still further. Finally when %RETURN (in routines) or %END or %ENDOFPROGRAM is executed, ST reverts to ST1.

Variables declared by %REAL and %INTEGER (and associated types) are called FIXED VARIABLES, because the amount of storage space required is determined at compile time. Array declarations, however, may have general integer expressions as the parameters and hence have dynamic significance.

For example, the space allocated by a declaration such as

```
%REALARRAY X,Y (1:M, 1:N)
```

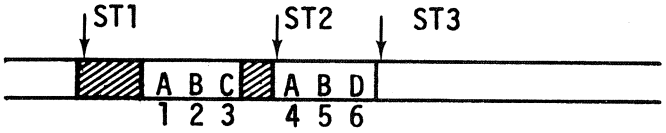
will depend on the computed values of M and N and cannot be determined at compile time. The stack pointer, ST, is thus advanced in several stages following the initial step which reserves space for all the fixed variables.

BLOCK STRUCTURE OF PROGRAMS

This is illustrated by the following example:

```
%BEGIN
%REAL A,B,C
A=1 ; B=2 ; C=A+B
    %BEGIN
    %REAL A,B,D
    A=2; D=1; B=C; C=4
    %END
A=A+B+C
%END
```

The associated stack is:



Before the first %BEGIN, ST is at ST1 and moves to ST2 on entering the outer block. After the second %BEGIN, ST is at ST3 and reverts to ST2 when %END is executed. At the second %END, corresponding to the first %BEGIN,ST assumes its original position, ST1.

In the diagram, positions 1,2,3 correspond to the declarations of the outer block and 4,5,6 to those of the inner block. After the instruction C=A+B, the value 3 is left in position 3; while the inner block instructions leave the values 2,1,3,4 in the positions 4,6,5,3 respectively. The last instruction of the outer block leaves the value 7 in position 1. This example indicates the importance of understanding the scope of influence of the declarations made in the inner and outer blocks.

The variables A,B of the inner block do not conflict with A,B of the outer block and are termed LOCAL names; a reference to C in the inner block is taken to refer to the variable of that name declared in the outer block and is a NON-LOCAL or GLOBAL name to the inner block.

Note also that the information stored in the variables of the inner block is lost, when the block is left, unless it is a variable of type %OWN, and that one cannot refer in the outer block to any variable declared in the inner block.

These simplified concepts are amplified in the following notes:

1. Blocks may contain any number of sub-blocks and blocks may be nested to a depth of 10.

2. Names declared in a block take on their declared meaning in the block and in any sub-blocks unless re-declared in the sub-block. Thus global variables must be used to communicate between blocks.

3. Declarations must appear at the start of a block before any instructions which may cause a jump to occur or any labels.

4. Labels and switch labels, unlike variables, are always local to a block. Thus a block may only be entered through its head and it is impossible to jump from one block to another.

5. Each loop control pair, e.g. %CYCLE - %REPEAT must be in the same block as must %START - %FINISH pairs.

6. The outermost block of a program is terminated by %ENDOFPROGRAM which causes the process of compilation to be terminated and transfers control to the next stage.

USE OF BLOCK STRUCTURE

It is often convenient to regard a complete block as one compound instruction. With this view of an inner block in mind, the following are among the reasons for nesting blocks of program.

1. It is often necessary to use an array whose effective bounds are not known until some stage in the execution of the program. Rather than declare an array whose size will always be adequate the following example indicates an economical use of available storage:

```
%BEGIN
%INTEGER N
    %CYCLE
        READ(N)
            %BEGIN
            %INTEGER I
            %INTEGERARRAY A(1:N)
                %CYCLE I= 1,1,N
                    READ (A(I))
                %REPEAT
                ......
                ......
            %END
    %REPEAT
%ENDOFPROGRAM
```

Here, the required size of the array is read in the outer block and the necessary array declared in the inner block. Note that the space used by any one set of data will be recovered when the inner block is left, thus allowing us to repeat the process without incurring successively increasing demands for space.

It might be imagined that a simpler solution is to declare an array, after all other declarations, which uses the whole of the remaining space in the machine. While this is probably true on a system where the user has the whole machine to himself, core store is the most expensive form of memory on a computer system and operating systems on large general purpose machines normally attempt to optimise its use by a variety of techniques. Thus on such systems, a requirement for large amounts of core storage (which is often largely unused) will incur penalties both in priority and cost.

2. 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 store. The general procedure is illustrated in the following example in which each task is written as a distinct block.

```
%BEGIN
......
......
    %BEGIN
    %REALARRAY XYZ(1:5000)
    ......
    ......
    %END
    ......
    ......
    %BEGIN
    %INTEGERARRAY IJK(1:20, 1:250)
    ......
    ......
    %END
......
......
%END
```

3. In developing a complicated program it is often a great advantage if each sub-block can be developed separately. A program is generally much clearer if its sub-blocks are related to the blocks of its flow diagram. A closely related method of breaking a program into sub-units is by the use of routines which are discussed in Section 6.

SECTION 6 - ROUTINES AND FUNCTIONS

INTRODUCTION

There are many occasions when it is necessary to perform a similar operation several times in different sections of a program, or even in distinct programs (perhaps written by different people). It has already been explained that a block may be regarded as a single compound instruction. Instead of writing out this block in full every time it is required one may give it a name which is then written (as a single instruction) each time the block is to be executed. Such a named block is called a ROUTINE and this Section contains a discussion of this basic concept and its extensions.

ROUTINES WITHOUT PARAMETERS

There are three operations involved in incorporating a routine into a program:

1. Declaration (or specification)
2. Calling
3. Description

Consider the example which uses a routine to interchange the values of two variables X and Y.

1. Declaration

```
%ROUTINESPEC INTERCHANGE
```

The name INTERCHANGE is to be the title for a routine (a block of declarations and instructions) which will be described later.

2. Call

```
INTERCHANGE
```

Carry out the routine which has the title INTERCHANGE

3. Description

```
%ROUTINE INTERCHANGE
%REAL Z
Z=X ; X=Y ; Y=Z
%END
```

The routine INTERCHANGE consists of the single declaration and three instructions.

NOTES

1. A Routine description has the same structure as a block except that %BEGIN is replaced by %ROUTINE followed by its name.

2. In the example X and Y are global variables.

3. The first line of the description is always the same as the declaration but with %SPEC omitted.

4. A routine may be called in any block interior to the one in which it is declared (and described). In this way one can think of local and global routines, in just the same way as local and global variables.

5. The compiler inserts instructions to jump round a routine description at run time. Thus the instructions constituting the routine are only obeyed as the result of a call on that routine.

6. A routine call is an instruction and may be made conditional:

```
%IF P = 10 %THEN INTERCHANGE
```

7. Normally, instructions in the routine are obeyed in sequence until reaching %END. If it is desired to return from the routine at some other point, the instruction %RETURN may be used. This is equivalent to jump to %END and hence cannot be used in an inner block of the routine. %RETURN may be made conditional.

8. Declaration of a %ROUTINE is not required if the description precedes the call.

Example: Interchange X and Y and square them if they are both positive.

```
%ROUTINE INTERCHANGE AND SQUARE
%INTEGER Z
    Z = X; X = Y; Y = Z
    %IF X <= 0 %OR Y <= 0 %THEN %RETURN
    X = X*X; Y = Y*Y
%END
```

NOTE

A second %RETURN could be written immediately before %END, but would be redundant.

GLOBAL VARIABLES IN ROUTINES

When using global variables within routines, it is necessary for them to be global to the routine description. It is not sufficient for them to be global to the call, as shown by the following example:

```
%BEGIN
%INTEGER A
%ROUTINESPEC SQUARE
    ..........
    A = 10
        %BEGIN
        %INTEGER A
            A = 5
            SQUARE
        %END
    %ROUTINE SQUARE
        A = A**2
    %END
%END
```

NOTES

1. It is the variable A of the outer block which is global to the routine description, so the result of the SQUARE instruction above will be to set A = 100.

2. The above remarks apply equally to other global names e.g. routine and function names.

ROUTINES WITH PARAMETERS

The previously described routine 'INTERCHANGE' will exchange the values of X and Y, but will be of no use to interchange any other pair of variables.

In IMP, to facilitate the use of the same routine in different contexts within a program, the user is permitted to write the routine using formal (or dummy) names for some or all of the variables global to it. In each call of the routine, these formal names are replaced by the appropriate actual names.

If formal names are used in the writing of a routine, then the following modifications must be made to the procedures for declaring, describing, and calling the routine:

1. In the declaration and descrition of the routine, its name must be followed by a bracketed list of the formal parameters used, together with a statement of their type.

2. In calling the routine, the name must be followed by a bracketed list of the actual parameters which are to replace the formal parameters on this occasion.

The designation 'parameter' has been used above in anticipation of facilities which permit quantities other than names (for example, elements of arrays and arithmetic expressions) to be passed on to routines.

Example 1:

```
%REAL U,V
%INTEGER I
%REALARRAY A(1:10)
%ROUTINESPEC INTERCHANGE (%REALNAME X,Y) ; ! DECLARATION
    ..........
    ..........
    INTERCHANGE (U,V)                        ; ! CALL 1
    %CYCLE I = 1,1,5
        INTERCHANGE (A(I) , A(11-I))         ; ! CALL 2
    %REPEAT
    ..........
        %ROUTINE INTERCHANGE (%REALNAME X,Y) ; ! DESCRIPTION
        %REAL Z
            Z = X; X = Y; Y = Z
        %END
```

NOTES

1. Here X and Y are the formal parameters.

2. The actual parameters must be placed in the same order as the formal parameters to which they correspond. In call 1, X is replaced by U and Y by V. In call 2, X is replaced by A(I) and Y by A(11-I).

3. In the example the type %REALNAME was used. In an analogous fashion any valid type may be specified as formal parameters. The actual parameters must, of course, correspond in type to the formal parameters.

4. The statement of parameter type is omitted in calling the routine but the compiler checks the actual parameters listed and will generate a compile time fault message if tney do not correspond to the declaration (see Section 12).

Parameter N in the example below illustrates the use of a different type of formal parameter, that called by value.

Example 2:

```
..........
%INTEGER SHRIEK
%ROUTINESPEC FACTORIAL (%INTEGERNAME Y, %INTEGER N)
    ..........
    ..........
    FACTORIAL (SHRIEK, 10)
    ..........
    ..........
        %ROUTINE FACTORIAL (%INTEGERNAME Y, %INTEGER N)
        %INTEGER I
            Y = 1 ;I = 1
            %WHILE I<=N %THEN Y=I*Y %AND I=I+1
        %END
```

The difference between the formal parameter types used in Examples 1 and 2 is important and must be carefully noted.

In Example 1 the formal parameters X, Y are of type %REALNAME and are the names of the variables to which the results are assigned, and the corresponding actual parameters must be names, in this case the names of %REAL variables.

A reference to Y inside the routine is essentially a reference to the non-local variable named by the actual parameter.

In Example 2, on the other hand, the formal parameter %INTEGER N can be replaced by an integer arithmetic expression, which is evaluated and assigned to the local variable N which is specially created in addition to any local variables declared in the routine. N is an essentially local quantity which is lost on exit from the routine. Consequently the routine should place the information it produces in variables which are called by NAME (such as X and Y), or in variables which are global to the routine. The formal parameter N is said to be called by VALUE in so far as it is only the value of the corresponding actual parameter which is of interest.

Note that a value is assigned to the local variable N by use of the '=' operator. Thus a CAPACITY EXCEEDED error could occur if the formal parameter is of type %SHORT (or %BYTE) %INTEGER, or %STRING.

However, when it is necessary to pass complete arrays to routines these may only be passed by means of NAME parameters. This is because the creation of local arrays and the necessary copying of them is both time and space consuming. Example 3 illustrates the use of %REALARRAYNAME parameters:

Example 3:

```
%ROUTINE MATMULT (%REALARRAYNAME A,B,C %INTEGER P,Q,R)
%INTEGER I,J,K ; %REAL T
    %CYCLE I = 1,1,P
        %CYCLE J = 1,1,R
            T = 0
            %CYCLE K = 1,1,Q
                T=T+A(I,K)*B(K,J)
            %REPEAT
            C(I,J)=T
        %REPEAT
    %REPEAT
%END
```

This forms the product of a 'P x Q' matrix A and a 'Q x R' matrix B. The result, a 'P x R' matrix, is accumulated in C. The routine assumes that the first element of each matrix has the suffix (1,1). A typical call sequence might be:

```
%REALARRAY H(1:20,1:20),X,Y,(1:20,1:1)
    .
    .
    MAT MULT (H, X, Y, 20, 20, 1)
```

In IMP, parameters called by name are completely determined by the actual values of all relevant quantities (including global variables) at the time of call. For example, it may happen that a routine with a parameter list containing say

```
.....(%REALNAME X, %INTEGERNAME I.....)
```

is called with the actual parameters

```
.....(A(J), J, .....)
```

where A is the name of a previously declared real array. If the value of J at the time of the call is, say, 10 then in the execution of the routine the formal parameter X is replaced everywhere by A(10) no matter how J varies during execution of the routine.

The reader is warned that the alternative convention whereby, in the above example, the array element replacing X would be determined by the current value of J during the execution of the routine is used in some other programming languages (e.g. Algol).

The following table is a complete list of formal parameters together with the permissible forms for the actual parameters:

Formal Parameter:				Corresponding Actual Parameter:
%BYTE	%INTEGER		%NAME	Name of a %BYTEINTEGER variable
%SHORT	%INTEGER		%NAME	Name of a %SHORTINTEGER variable
	%INTEGER		%NAME	Name of an %INTEGER variable
	%REAL		%NAME	Name of a %REAL variable
%LONG	%REAL		%NAME	Name of a %LONGREAL variable
	%STRING		%NAME	Name of a %STRING variable
%BYTE	%INTEGER	%ARRAY	%NAME	Name of a %BYTEINTEGERARRAY
%SHORT	%INTEGER	%ARRAY	%NAME	Name of a %SHORTINTEGERARRAY
	%INTEGER	%ARRAY	%NAME	Name of an %INTEGERARRAY
	%REAL	%ARRAY	%NAME	Name of a %REALARRAY
%LONG	%REAL	%ARRAY	%NAME	Name of a %LONGREALARRAY
	%STRING	%ARRAY	%NAME	Name of a %STRINGARRAY
%BYTE	%INTEGER			An integer expression
%SHORT	%INTEGER			
	%INTEGER			
	%REAL			A general expression (i.e. a real or integer expression)
%LONG	%REAL			
	%STRING (n)			A string expression
		%ROUTINE		Sometimes it is required to pass on the name of a routine, or function (see below) as a parameter. The actual parameter is the name of a routine or function which must correspond in type and specification with the formal parameter, the specification of which will be found in the routine body.
%BYTE	%INTEGER	%FN		
	%INTEGER	%FN		
	%INTEGER	%FN		
	%REAL	%FN		
%LONG	%REAL	%FN		
	%STRING	%FN		
%BYTE	%INTEGER	%MAP		The name of a map function (see Section 7). The name of a map function may be passed to a routine in the same context as a routine or function name.
%SHORT	%INTEGER	%MAP		
	%INTEGER	%MAP		
	%REAL	%MAP		
%LONG	%REAL	%MAP		
	%STRING	%MAP		
	%RECORD		%NAME	The name of a %RECORD
	%RECORD	%ARRAY	%NAME	The name of a %RECORDARRAY

FUNCTION ROUTINES

When a routine has a single output value it may be written as a function routine and then used in an arithmetic expression in the same way as the permanent functions (COS, SIN etc.).

The declaration, call and description of routine and functions are compared in the following table:

	Routine:	Function:
Declaration:	%ROUTINE %SPEC....	<type> %FN %SPEC....
Result of Call:	Execution of an instruction	A value of the appropriate type and length
Description:	%ROUTINE....	<type> %FN....

where the type of function may be any one of the allowed real or integer types, i.e. %SHORTINTEGER, %LONGREAL etc.,or of type %STRING. For example, the routine FACTORIAL described earlier may be rewritten as a function routine as follows:

```
%INTEGER SHRIEK
%INTEGERFNSPEC FACT 1 (%INTEGER N)
    ..........
    SHRIEK = FACT 1 (10)
    ..........
        %INTEGERFN FACT 1 (%INTEGER N)
        %INTEGER PROD,I
            %IF N=1 %THENRESULT = 1 ; %COMMENT NOTE 1. BELOW
            PROD = 1
            %CYCLE I = 2, 1, N ; %COMMENT NOTE 1. BELOW
                PROD = I * PROD
            %REPEAT
            %RESULT = PROD
        %END
```

NOTES

1. The reader should study carefully the two occurrences of the assignment of the value of the function to %RESULT . Depending on the value of N, either is a possible exit point. The two lines marked with a %COMMENT could be combined, as in the routine FACT, but the similarity to the example later in this Section on recursion would be lost.
2. Both the routine call FACTORIAL (SHRIEK, 10) and the assignment SHRIEK=FACT1(10) produce identical results.

FUNCTIONS AND ROUTINES AS PARAMETERS

This is illustrated by the following example involving an integration routine:

```
%ROUTINESPEC INTEGRATE(%REALNAME Y, %REAL A,B, %INTEGER N, %REALFN F)
```

which integrates a function F(X) over the range (A, B) by evaluating

$$Y = (F(0) + 4*F(1) + 2*F(2) + \ldots + 4*F(2N-1) + F(2N))*(B-A)/(6*N)$$

where $F(I) = F(A + I*(B-A)/(2*N))$

An auxiliary function is required to evaluate F(X) and details of it must be passed on to the integration routine. This is done by means of the routine type parameter and the body of the routine might then be:

```
%ROUTINE INTEGRATE (%REALNAME Y, %REAL A, B, %INTEGER N,%C
%REALFN F)
[%REALFN] %SPEC F(%REAL X)
%REAL H; %INTEGER I
    H = (B-A)/(N*2)
    Y = 0
    %CYCLE I = 0,2,2*N-2
        Y = Y+2*F(A+I*H)+4*F(A+(I+1)*H)
    %REPEAT
    Y = (Y-F(A)+F(B))*H/3
%END
```

To enable instructions such as:

```
Y = Y+2*F(A+I*H)+4*F(A+(I+1)*H)
```

to be translated, a specification of the formal parameter F is required. In this case the delimiter %REALFNSPEC can be abbreviated to %SPEC since the type of the function is given explicitly by the formal parameter itself. Now consider a program to evaluate

```
Z = EXP(-Y)*COS(B*Y)
```

for various values of B read from a data file, the last value being followed by 1000, using for N the integer nearest to 10B.

```
%BEGIN
%ROUTINESPEC INTEGRATE (%REALNAME Y,%REAL A,B,%INTEGER %C
    N,%REALFN F)
%REALFNSPEC AUX (%REAL Y)
%REAL Z, B
%COMMENT SIMPSON RULE INTEGRATION
    %CYCLE
        READ (B)
        %IF B = 1000 %THEN NEWLINES(10) %ANDSTOP
        INTEGRATE (Z, 0, 1, INT(10B), AUX)
        NEWLINE
        PRINT (B, 1, 2);SPACES(2);PRINT (Z, 1, 4)
    %REPEAT
        %REALFN AUX(%REAL Y)
            %RESULT = EXP(-Y)*COS(B*Y)
        %END
        %ROUTINE INTEGRATE (%REALNAME Y,%REAL A,B,%C
            %INTEGER N,%REALFN F)
            .
            .
        %END
%ENDOFPROGRAM
```

NOTES

1. The names given to the auxiliary routine and its parameters need not be the same in the integration routine as in the main program but they must correspond in type.

2. Since the result of the integration is a single quantity, the routine could be rewritten as a %REALFN:-

```
%REALFNSPEC INTEGRATE(%REAL A,B, %INTEGER N, %REALFN F)
```

and called by:

```
PRINT(INTEGRATE(0,1,INT(10B),AUX),1,6)
```

LANGUAGE LIBRARY

A complete list of the routines and functions in the IMP Language Library is given in Section 16. Note that certain of the 'routines', those described as intrinsic, for example READ and WRITE, are not strictly routines and their names cannot be substituted as actual parameters in place of formal parameters of routine type. They would first have to be re-defined as formal routines. For example the intrinsic routine write could be re-defined thus:

```
%ROUTINE MYWRITE (%INTEGER A,B)
    WRITE(A,B) ;!THIS IS INTRINSIC ROUTINE WRITE
%END

%ROUTINE WRITE (%INTEGER A,B)
    MYWRITE (A,B)
%END
```

This solution involving two routines MYWRITE and WRITE is needed when it is necessary to use intrinsic routines as parameters to other routines or functions, and to preserve their usual names or when one wishes to alter the effect of an intrinsic routine.

SCOPE OF NAMES

In general all names are declared at the head of a routine, or function, either in the routine heading or by the declarations %INTEGER, %REAL, %INTEGERARRAY etc., and the various routine specifications. They are local to that routine and independent of any names occurring in other routines. However, if a name appears in a routine which has not been declared in one of the above ways, then it is looked for outside i.e. in the routine or block in which it is embedded. If it is not declared there it is looked for in the routine or block outside that and so on until the main block is reached.

Now the main block is itself embedded in a hypothetical outer block, so that if a name is not found in the main block it is looked for here. This outer block effectively contains all the implicit and intrinsic library routines, functions and maps which have preassigned names. These preassigned names may in fact be redeclared locally at any level, but clearly it would be unwise to assign new meanings to such routines as LOG, PRINT etc. Very often, the only non-local names used in a routine will be the preassigned names.

Routines and functions themselves have the property of being global to any block interior to the one in which they have been declared and described.

USE OF %OWN VARIABLES

When a routine or block is left, any information stored in variable corresponding to local declarations in that routine is normally lost, and no further reference may be made to it. In some cases it may be desireable to retain some of this information and be able to refer to it on a subsequent entry to the routine. this may be accomplished by prefixing the relevant declaration by %own as described in Section 1.

RECURSIVE USE OF ROUTINES AND FUNCTIONS

Routines and functions have the property of being global to any block interior to the one in which they are declared. In particular, a routine or function can be used within the description of that routine or function itself. This process is called RECURSION. Such a routine may also call itself indirectly by invoking other routines which make use of it. On each activation of the routine a fresh copy of the local working space is set up in the stack, so that there will be no confusion between variables on successive calls. (This does not apply however to %OWN variables. See above). Some criterion within the body of the routine must eventually inhibit the calling statement and allow the process to unwind.

Example: A function RECFACT equivalent to the function FACT described earlier can be defined recursively as follows:

```
RECFACT(1) = 1
RECFACT(N) = N * RECFACT(N-1)
```

This is easily programmed:

```
%INTEGERFN RECFACT (%INTEGER N)
    %IF N = 1 %THENRESULT = 1
    %RESULT = N * RECFACT (N-1)
%END
```

Note, however, that in this example it would have been more efficient to use recurrence rather than recursive techniques.

The following example, however, cannot be easily rewritten as a cycle:

QUICKSORT: Quicksort is an elegant method of sorting numbers (or any other quantities) into order.

The basic routine,

1. Selects some member of the set to be sorted, and uses this as the 'partition bound'.

2. Partitions the remainder of the set into two groups, one containing members not greater than the partition bound, and the other containing members not less than it. These groups are positioned to the left and right of the bound.

3. Calls itself recursively to sort each of these two groups.

A possible description of this routine, in which the partitioning bound, D, has been arbitrarily chosen to be right-hand member, and in which the elements to be sorted are members of a string array, is:

```
          %ROUTINE STRINGSORT (%STRINGARRAYNAME X, %INTEGER A, B)
          ! SORTS ELEMENTS OF STRINGARRAY X FROM X(A) TO X(B)
          %INTEGER L, U
          %STRING(255) D
          %RETURNIF A >= B
          L = A; U = B                          ; !SET POINTERS
          D = X(U)                              ; !DUMP PARTITION BOUND
          -> FIND
UP:       L = L +1                              ; !THIS SECTION MOVES
          -> FOUND %IF L = U                    ; !L FORWARD UNTIL
FIND:     -> UP %UNLESS X(L) >= D               ; !FIND A MEMBER >= D
          X(U) = X(L)
DOWN:     U = U - 1                             ; !THIS SECTION MOVES
          -> FOUND %IF L = U                    ; !U BACK UNTIL WE
          -> DOWN %UNLESS X(U) <= D             ; !FIND A MEMBER <= D
          X(L) = X(U)
          -> UP
FOUND:    X(U) = D                              ; !PARTITIONING COMPLETE

          STRINGSORT (X, A, L - 1)              ; !SORT FROM X(A) TO X(L-1)
          STRINGSORT (X, U + 1, B)              ; !SORT FROM X(U+1) TO X(B)

          %END
```

INCLUSION OF ROUTINES IN LIBRARY FILES

A user who has developed and tested a set of routines may wish to save these, in their compiled state, for use by subsequent programs. The required commands or JCL statements to create such files are described in the appropriate User's Guide.

The Language requirement is as follows.

1. There must be no %BEGIN at the start of the text.

2. Each routine must be prefaced by %EXTERNAL.

3. The text must be closed by %ENDOFFILE rather than by %ENDOFPROGRAM.

4. Variables which are global to the set of routines must be declared as %OWN or %CONST variables.

5. External routines may call any other external routine in the same file provided it has been compiled first, or a routine spec has been given for it.

Example

```
%OWNINTEGER A
%OWNREAL X
%STRINGMAPSPEC THIRD(%STRING S)
%EXTERNALROUTINE FIRST(%INTEGER I)
    .
    .
    .
! These routines may reference A and X as global variables
%END
%EXTERNALREALFN SECOND(%INTEGER J,%REAL Y)
    .
    .
    .
! SECOND may call FIRST as FIRST has already been compiled
! SECOND may call THIRD as a routine spec has been given
%END
%EXTERNALSTRINGMAP THIRD(%STRING S)
    .
    .
    .
%END
%ENDOFFILE
```

A program which wishes to call these routines must include the appropriate %EXTERNALROUTINESPEC statement - as for system library routines. It is essential that the parameter list in the %EXTERNALROUTINESPEC statement is identical to that for the %EXTERNALROUTINE itself, except that the names of the parameters are not significant. If the number or type of parameters differ the program may still compile apparently successfully, but at run time obscure faults may occur.

%EXTERNAL VARIABLES

An alternative method of communicating between %EXTERNAL routines and the programs and other routines that call them involves the use of %EXTERNAL variables. These are declared as global variables in a program or file of %EXTERNAL routines and functions, as are %OWN variables. They have all the attributes and restrictions of %OWN variables i.e. they retain their values between calls and they can be initialised in the same way as %OWN variables. Additionally they can be accessed by the calling program or by other %EXTERNAL routines by declaring them as %EXTRINSIC variables. The %EXTRINSIC declaration does not result in any space being allocated, instead it generates a link to the %EXTERNAL variables of the same name. For example:

```
%EXTERNALINTEGER FLAG
%EXTERNALBYTEINTEGERARRAY LINE(1:72)=' '(72)
    %EXTERNALROUTINE INPUT
        .
        .
        .
    %END
%ENDOFFILE
```

```
%BEGIN
%EXTERNALROUTINESPEC INPUT
%EXTRINSICINTEGER FLAG
%EXTRINSICBYTEINTEGERARRAY LINE(1:72)
    .
    .
    INPUT %UNTIL FLAG=1
    %STOP %IF LINE(1)='*'
    .
    .
%ENDOFPROGRAM
```

It is important to ensure that the name and type of the %EXTRINSIC declaration is identical to that of the %EXTERNAL declaration, and that the bounds of arrays are the same.

LENGTH OF %EXTERNAL NAMES

The names given to %EXTERNAL routines, functions and variables can be up to 255 characters, as for names of other entities in IMP programs. However only the first 8 characters are used for linking separately compiled object files. Thus two %EXTERNAL routines called TWEEDLEDEE and TWEEDLEDUM for example, would give a fault at run time 'DUPLICATE ENTRIES'. This problem can be avoided by ensuring that names of %EXTERNAL entities differ in their first 8 characters.

SECTION 7 - STORE MAPPING FACILITIES

INTRODUCTION

Facilities are provided in IMP to allow the programmer to use alternative names for the same variables. This is useful for the following reasons:

1. to save space in core

2. to access a variable both as declared, and as a set of sub-variables, for example an %INTEGER can be accessed as four separate %BYTEINTEGERs

3. to access a particular array member as a scalar with consequent saving of machine time

4. to improve the clarity of a program

5. to access a %RECORD using different formats.

Store mapping facilities are very powerful. On the other hand by allowing the user to operate on addresses they increase the chance of causing program errors which can be very hard to diagnose.

STORE MAPPING FUNCTIONS

The store mapping function can facilitate the storage of large but partially redundant arrays. For example, if a two-dimensional array is symmetrical, X(i,j) = X(j,i), only the values of X(i,j) with i>=j need be stored. By keeping only these values in the one-dimensional array, A(p), and providing alternative location names, X(i,j) for the elements of A through a store map, we can have the most economical use of store without losing the symmetrical appearance of the array X.

The store mapping function is declared by:

```
<type> %MAP %SPEC
```

where type depends on the nature of the variable to be renamed and may be %BYTEINTEGER, %SHORTINTEGER, %INTEGER, %REAL, %LONGREAL or %STRING.

The general form of a store mapping function W is written:

```
<type> %MAP W(%INTEGER I,J,....)
%RESULT== A(expl(I,J,...),exp2(I,J,...)...)
%END
```

In this case the array A is to be given the alternative name W, and the suffices of A, expl, exp2 etc. are general expressions in terms of the suffices of W - I,J, etc. There is no restriction on the number of suffices that can be associated with W. It is assumed above that A is global to the description of the mapping function, but A could have been declared as a formal parameter in the function heading, thus:

```
%SHORTINTEGERMAP W(%INTEGER I,J, %SHORTINTEGERARRAYNAME A)
```

As an example, the map for the case of the symmetrical two-dimensional array X stored in A described above is:

```
%INTEGERMAP X(%SHORTINTEGER I,J)
    %RESULT== A(I*(I-1)/2 + J) %IF I>J
    %RESULT== A(J*(J-1)/2 + I)
    %END
```

Like functions, mapping functions can appear in arithmetic expressions but have the added property that they can appear on either the left or right-hand side of an assignment statement. On either side, the result of a mapping function is an address from which, or to which a value is fetched or stored according to context.

The saving in storage space achieved by using mapping function is obtained by sacrificing speed in the execution of the compiled program. For this reason mapping functions are not recommended in situations where they would be called frequently.

THE BUILT IN MAPPING FUNCTIONS

There are seven built in mapping functions available to the user, for simple variables:

%INTEGERMAPSPEC	INTEGER (%INTEGER N)
%SHORTINTEGERMAPSPEC	SHORT INTEGER (%INTEGER N)
%BYTEINTEGERMAPSPEC	BYTE INTEGER (%INTEGER N)
%REALMAPSPEC	REAL (%INTEGER N)
%LONGREALMAPSPEC	LONG REAL (%INTEGER N)
%STRING (255) %MAPSPEC	STRING (%INTEGER N)
%RECORDMAPSPEC	RECORD (%INTEGER N)

They all give locations of a particular byte having as its absolute address in the main store the value N. BYTE INTEGER picks up only the byte; SHORT INTEGER picks up 2 bytes; REAL and INTEGER pick up 4 bytes; LONG REAL picks up 8 bytes; and STRING picks up the number of bytes determined by the string length given in the first byte. An address error occurs if SHORT INTEGER attempts to pick up two bytes which are not correctly halfword aligned, if REAL or INTEGER attempts to pick up four bytes not correctly fullword aligned, or if LONG REAL attempts to pick up eight bytes not correctly double word aligned.

In contrast to user defined maps the above built in maps are very efficient in terms of speed and space. For example an %INTEGER can be unpacked into its four component %BYTEINTEGERs thus:

```
%INTEGER I,J
%BYTEINTEGERARRAY B(0:3)
    ..........
    I=........
    %CYCLE J=0,1,3
        B(J) = BYTEINTEGER (ADDR(I)+J)
    %REPEAT
    ..........
```

It is sometimes necessary to do the reverse of that shown above; say to reform a long real variable, X, from two components stored in %INTEGERS J1 and J2. The following example shows the use of a mapping function on the LHS of an expression.

```
%LONGREAL X
%INTEGER I,J1,J2
.....
    INTEGER(ADDR(X))= J1
    INTEGER(ADDR(X)+4)=J2
```

POINTER VARIABLES

Pointer variables provide an additional mapping facility. A pointer variable is declared in the same way as a normal scalar variable except that %NAME is added to the type.

Example:
```
%INTEGERNAME I,J,K
%LONGREALNAME P1
```

The declaration does not result in any space being allocated for the variables, it merely causes the compiler to record the names. Before being used the pointer variable has to be equivalenced to a declared variable using the == operator. From then on both the original name and the name of the pointer variable can be used to access the variable. For example if a three-dimensional array is being accessed in such a way that frequent reference is made to its first element it would improve the efficiency of the program to equivalence the first element to a pointer variable.

```
%INTEGERARRAY TABLE(1:10,1:10,1:10)
%INTEGERNAME BASE
    BASE==TABLE(1,1,1)
```

Additionally pointer variables can be used in conjunction with the built in maps to provide a more elegant solution in the situation where it is required to reference the same space in two different ways. For example if it is required to reference a %BYTEINTEGERARRAY as a %STRING the following code could be used:

```
%BYTEINTEGERARRAY IN(0:80)
%STRINGNAME LINE
LINE==STRING(ADDR(IN(0)))
```

From this point onward the array can be referenced either as an array or as the string, LINE. Obviously the length byte, IN(0), will have to be set to an appropriate value.

ARRAY MAPPING

Apart from mapping for individual variables it is possible to use the built in map ARRAY. This takes two parameters: an address and the name of an %ARRAYFORMAT. In the following example the two-dimensional array ATWO is mapped on to an array AONE which is declared as a one dimension array:

```
%INTEGERARRAY AONE(1:10000)
%INTEGERARRAYNAME ATWO
%INTEGERARRAYFORMAT AFORM(1:100,1:100)
ATWO==ARRAY(ADDR(AONE(1)),AFORM)
ATWO(27,27)=928
.
.
```

The %ARRAYFORMAT statement is used to describe the characteristics of the array ATWO - i.e. number of dimensions and bounds for each dimension. As an alternative to using the name of an %ARRAYFORMAT for the second parameter, the name of another %ARRAY can be used, if one with suitable characteristics has been defined in the program.

RECORD MAPPING

This is described in Section 9.

SECTION 8 - STRINGS

INTRODUCTION

A 'string' in the IMP language is a string of between 0 and 255 characters. Space and newline characters may be included.

Strings in IMP are declared and manipulated in ways largely analogous to those for the arithmetic entities in IMP. They can be declared singly or in arrays of one or more dimensions. They are declared at the head of blocks or routines. The space which they occupy can be allocated dynamically, or they can be declared as %OWN, %CONST, %EXTERNAL or %EXTRINSIC. The same scope rules apply as for the names of arithmetic types. Strings may appear as the results of functions, and may be written into programs as constants. They may be referenced as elements of records and through mapping functions and pointer variables. They may be passed as parameters to routines, functions and mapping functions by 'value' and by 'name', and these modes are analogous to those for the arithmetic types.

String expressions may be tested in conditions. The elaboration of a string condition is sometimes markedly different from that of an arithmetic condition, but a set of IMP statements can be made conditional by prefixing or suffixing a string condition in a manner similar to prefixing or suffixing an arithmetic condition, and the lexicographical forms are similar.

Finally, the IMP run-time diagnostic package treats strings in essentially the same way as other IMP variables and provides for detection of run-time faults such as 'unassigned variable' and the listing of string type variables in the diagnostic routine trace-back. String manipulation fault conditions may be trapped using the standard fault-trapping mechanism. (see Section 14)

This section describes features of string declaration and manipulation in so far as they differ from those of the arithmetic entities in IMP, and in particular describes string operations and conditions.

TERMINOLOGY

The 'value' of a string means the sequence of characters forming the string. In the context of this section, 'value' and 'string' are practically synonymous, except that a 'value' normally describes the result of evaluating a 'string expression'. The value of a string is denoted where necessary by the characters of the string enclosed in single quotes, except that each single quote within the string is denoted by two single quotes. The 'length' of a string is the number of characters forming the string. It may be zero, when the string is (has value) null.

Within this section except where otherwise stated,

'location'	means	'string location'
'constant'	means	'string constant'
'variable'	means	'string variable'
'function'	means	'string function'
'mapping function'	means	'string mapping function'
'record element'	means	'string-type element of a record'
'expression'	means	'string expression'
'value'	means	'value of a string or string expression'
'assignment'	means	'string assignment using the '=' or '<-' operator'

All these terms are defined in the text.

STRING LOCATIONS

A 'string location' is a portion of storage which a program accesses using string operators. Thus string declarations cause space to be allocated as a set of one or more locations. A location has an associated 'maximum length', which may be specified as part of a declaration, or in defining a function or mapping function. The number of bytes occupied by a location is one greater than this maximum length. A location has no special alignment.

When a location holds data (a string, or 'value'), the data format in the location is as follows. The first byte of the location holds the length of the string. Successive bytes (so far as necessary) hold the characters of the string as a sequence of ISO character values. Clearly the length of the value held cannot exceed the maximum length of the location.

A location may be referenced in the following ways:

1. using the name of a variable,

2. using a subscripted array name,

3. using a mapping function name (possibly subscripted with parameters), (see Section 7),
4. using a record name subscripted with a record element name, (Section 9), or

5. using a pointer variable name (Section 7).

A reference to a location implies using the address of (the first byte of) the location, and this is the address produced by the built-in function ADDR when applied to a location (Section 7).

STRING CONSTANTS

A 'string constant' is denoted by its value enclosed in single quotes, except that each single quote contained within the string is denoted by two single quotes. Examples are:

```
'IT''S MINE'
'TESTING
'
''
```

The second example denotes a constant whose value has eight characters (whose length is eight), the eighth being newline. The third example (two adjacent single quotes) denotes a null constant.

Constants may appear in 'expressions', and in declarations where initialization is permitted.

STRING VARIABLES

A 'string variable' is an unsubscripted name used to reference a location. A declaration of a variable causes (either static or dynamic) allocation of space for the location which it references. The declaration must specify the maximum length of the location by enclosing this in parentheses after the delimiter '%STRING' e.g.

```
%STRING(255) U
%STRING (20) LH,RH
```

%OWN, %CONST and %EXTERNAL variables may be initialized explicitly by specifying an initializing constant, thus:

```
%OWNSTRING(19) FILENAME='ERCC00.TEST'
```

If the explicit initialization is omitted, the variable is given an initial value of null.

STRING ARRAYS

A 'string array' is an array of one or more (maximum seven) dimensions of locations which may be referenced using the array name and one or more subscripts. All locations of a given array must have a common maximum length, which should be stated in the array declaration or array format statement.
Examples of array declarations are:

```
%STRING(63) %ARRAY FIELDS(1:5)
%STRING(63) %ARRAY NAMES1,TAGS(0:9,-1:0)
```

%OWN, %CONST and %EXTERNAL arrays may be explicitly initialized by giving a list of constants, possibly with repetition factors, as for comparable arithmetic type arrays.
For example:

```
%OWNSTRING(6) %ARRAY F(0:4)= 'FRED','A',''(3)
```

If the explicit initializations are omitted, each location is initialized to null.

STRING FUNCTIONS

Analogously with arithmetic functions (Section 6), a 'string function' may appear in a 'string expression' (see below). A string function is declared in exactly the same way as an arithmetic function, except that the maximum length of the value which the function can yield is stated in the declaration. Thus:

```
%STRING(20) %FN FIELD (%INTEGER I)
```

Execution of a string function terminated at a %RESULT statement must assign a 'string expression', described below, to %RESULT, which causes this value to be used in the 'string expression' at the place the function was called.

Assignment to %RESULT does not at present cause the 'CAPACITY EXCEEDED' run-time fault, though later compilers may take note of the maximum length included in the function declaration.

STRING MAPPING FUNCTIONS

String mapping functions provide a means of referencing an area of storage as a string location. A map name (possibly subscripted with parameters) is a synonym for a location whose address is the %RESULT of the mapping function. For example, if the mapping function:

```
%STRING(3)%MAP XA (%INTEGER I)
%RESULT = ADDR(A(I))
%END
```

is declared, where A is a (0:5) %INTEGER array, then XA(I) is synonymous with the string location which has the same address as the Ith element of A.

The declaration of the mapping function includes the maximum length of the locations which it is to reference, but assignments to such locations cannot cause the 'CAPACITY EXCEEDED' run-time fault (see below). Thus, as always with mapping functions, care is required to ensure that a program does not unintentionally overwrite data not directly being referenced.

The intrinsic mapping function STRING may be of particular use. Its effect is that of the following:

```
%STRING(255) %MAP STRING (%INTEGER ADDRESS)
%RESULT = ADDRESS
%END
```

STRING EXPRESSIONS : CONCATENATION

A 'simple operand' is a constant, a string function or one of the denotations for referencing a location (listed above in the section defining locations).

A 'string expression' is a simple operand or a denotation composition of one or more operations on simple operands. It has a value, namely that of the simple operand or that which results from performing the operations on the simple operands.

Only one kind of operation is permitted in string expressions: concatenation, denoted by dot (.). This is a binary operator, and its two operands are written on either side of it, thus:

```
A.B
```

The result of concatenating two operands is the string comprising the value of the first operand followed by the value of the second operand. Thus the expression:

```
'HASTINGS'.'1066'
```

has value denoted:

```
'HASTINGS1066'
```

It is not commutative (A.B#B.A).

Unlike arithmetic expressions, string expressions may not contain sub-expressions. Thus a string expression must always be written as a sequence of say N simple operands separated by N-1 dots (N>0). For example, the following is an expression:

```
'CONST'.VAR.TOSTRING(I+J)
```

where VAR is a variable and TOSTRING is a function.

When an expression comprises more than one concatenation of simple operands, the concatenations are performed starting from the left of the expression. If a concatenation results in a value (intermediate or final) whose length exceeds 255, the 'CAPACITY EXCEEDED' run-time fault may occur (see Section 13).

STRING ASSIGNMENTS

The value of an expression may be assigned to a location (referenced in one of the ways listed in the section above defining string locations) or, in a function, to %RESULT.

There are two types of assignment, analogous to the arithmetic assignments, denoted by '=' and '<-'. The former is called simply '(ordinary) assignment' and the latter is called 'jam transfer'. Examples are:

```
SARR(J)=S
S<-A.B.C
```

where A,B,C,S are string variables, J is an integer variable and SARR is an array.

For the '=' assignment, the value of the expression on the right-hand side is assigned to the location denoted by the left-hand side. The 'CAPACITY EXCEEDED' run-time fault may occur (see Section 13).

The '<-' assignment operates exactly as the '=' assignment except that the 'CAPACITY EXCEEDED' run-time fault cannot occur: if the length of the value denoted by the right-hand side exceeds the maximum length N of the left-hand side location, assignment only of the N left-most characters of the right-hand side value occurs. This assignment may be used intentionally to truncate the value being assigned.

STRING RESOLUTIONS

A further type of operation provides a powerful tool for analysing strings and assigning 'substrings'. Explicitly, a string S is a 'substring' or a string T if it can be concatenated with two other (possibly null) strings to form the string T.

A 'string resolution' is a left-to-right operation denoted by '->'. Its left-hand operand must be a location (referenced in one of the ways listed above defining string locations). Its right-hand operand must be a sequence alternately of locations and expressions. The locations and expressions must each be separated by a dot (.) separator, and the expressions must further be enclosed in parentheses.For example:

```
L->M.(E).N.(F).P
```

where L,M,N,P are locations and E,F are string expressions.

To describe the effects of resolution, we take the simple case of the following resolution:

```
L->M.(E).N
```

When executed, the resolution may 'succeed' or 'fail'. It 'succeeds' if the value of the expression E is a substring of the value at the location L. This value is now considered as three substrings: that part which precedes the left-most occurrence of E in it; that part which is the left-most occurrence of E; and that part which follows the left-most occurrence of E. The location L remains unchanged, but the first substring above is assigned to M and the value of the third substring above is assigned to N. (M or N may be assigned null values in the resolution). The 'CAPACITY EXCEEDED' run-time fault may occur during these assignments (see below). Otherwise the resolution always succeeds if the value of E is null.

If the value of E is not a substring of the value at location L, the resolution 'fails' and no assignments take place. In this case (unless the resolution forms part of a 'string condition', described below) the 'resolution fails' run-time fault (Fault number 26) occurs. This fault may be trapped, see Section 14.

The following is an example of the simple resolution so far described. If location L contains 'HASTINGS1066' then

```
L->M.('10').N
```

succeeds and causes 'HASTINGS' to be assigned to M and '66' to N.

Consider now the more complex resolution:

```
L->M.(E).N.(F).P
```

This is executed exactly as though a private location PRIV, of maximum length 255, existed and the resolutions:

```
L->M.(E).PRIV
PRIV->N.(F).P
```

were performed. The resolution succeeds if both of these resolutions would succeed; otherwise it fails.

The general resolution is executed in an analogous way, hypothetically using further 'private' locations to split the resolution into simple resolutions. Note that in the general resolution some of the intermediate assignments or other consequent actions (such as the execution of functions or mapping functions) may have been effected before the resolution fails or before a possible 'CAPACITY EXCEEDED' run-time fault occurs.

The following is an example of a more complex resolution. If location L contains value 'ERCCOOINDEX2' and E has value 'INDEX' then

```
L->M.('ERCC').N.(E).P
```

succeeds and causes M to be assigned a null value, '00' to be assigned to N and '2' to be assigned to P.

Finally, in a resolution the initial or final locations, or both, with their associated separator operators ('.'), may be omitted, provided that the right-hand side of the resolution still contains at least one expression in parentheses and at least one location. Examples are:

```
L->(E).M
L->M.(E)
```

The first example succeeds if the value of E is a substring of L and there is no non-null substring of L to the left of the occurrence of E in L. The second succeeds if the value of E is a substring of L. Note that there is asymmetry between the cases of these examples: the second is equivalent to

```
L->M.(E).SINK
```

where SINK is an unwanted location of length 255. However, the first example is not equivalent to

```
L->SINK.(E).M
```

Explicitly, if location L contains 'HASTINGS' then

```
L->('HA').M          succeeds
L->('TING').M        fails
L->M.('TING')        succeeds
L->M.('TINGS')       succeeds.
```

STRING CONDITIONS

'String conditions' are analogous to arithmetic conditions in that %IF, %UNLESS, %WHILE and %UNTIL may cause tests to be made on locations and cause the sequence of execution of statements to depend on the outcome of the tests. A set of IMP statements may be made conditional by adding string conditions and the lexicographical forms are essentially the same.

The tests performed may be of two kinds. The first kind is a relational test, analogous to an arithmetic test, in which two (or three) expressions are compared, specifying one (or two) of the relational operators <,<=,=,>=,>,#. (Three expressions with two relational operators form a double-sided condition, whose effect is analogous to that of a double-sided arithmetic condition). The second kind is a test of a resolution, as described above.

Examples of the two kinds are:

```
%IF S<'WATER' %THEN ->LAB
%IF L->M.('INGS').N %THEN ->LAB
```

RELATIONAL STRING CONDITIONS

The expressions to be tested are first evaluated (the order of evaluation is not defined). A test of a relationship between the two values commences with up to M character comparisons, where M is the minimum of the lengths of the two values. The comparisons are based on the internal codes for the characters of the strings, namely the ISO character codes. The test may continue with a comparison of the lengths of the two values.

If the relational operator is '=', the relationship is TRUE if and only if the values are identical, that is:

1. the lengths L of the two values are equal, and

2. if L>0, the i-th characters of each are equal for 0<i<=L.

Otherwise the relationship is FALSE. These truth values are reversed for the relational operator '#'.

If the relational operator is '<', the relationship is TRUE if and only if the left-hand value precedes the right-hand value in a 'dictionary ordering' of the two values, based on the internal code for the characters. Thus if M is the minimum of the lengths of the left-hand and right-hand values, the relationship is TRUE if and only if:

1. if M>0, the i-th character of the left-hand value has internal code less than that of the i-th character of the right-hand value for 0<i<=M, and

2. if M=0 or if the i-th characters are equal for 0<i<=M, the length of the left-hand value is less than the length of the right-hand value.

Otherwise the relationship is FALSE. These truth values are reversed for the relational operator '>='. Analogous comparisons are made for the relational operators '>' and '<='.

Thus:

```
'AB'<'C'        is TRUE
'AB'<'ABC'      is TRUE.
'IMP'<'FORTRAN' is FALSE
```

'Double-sided' conditions are permitted. An example where S and T are string expressions is:

```
%IF 'AB' < S < T %THEN -> LAB
```

RESOLUTION STRING CONDITIONS

The resolution which is the subject of the condition is elaborated exactly as described in the section concerned with 'string resolutions', except that the 'resolution fails' run-time fault cannot occur. If the resolution succeeds the condition is TRUE, and if it fails the condition is FALSE.

The statements of the following example remove all 'leading' space characters from the string S.

```
%WHILE S->(' ').S %CYCLE; %REPEAT
```

THE 'CAPACITY EXCEEDED' RUN-TIME FAULT

The 'CAPACITY EXCEEDED' run-time fault (Fault number 30) is a 'trappable' fault which is normally 'enabled'. For assignment operations only it may be 'disabled' (that is, execution will be allowed to proceed without diagnostic message), in a program by specifying the compiler option NOARRAY when the program is compiled. If the fault is disabled it cannot be 'trapped'. Disabling the fault enables shorter and faster object code to be produced by the compiler, but should be used with discretion.

The fault occurs during expression evaluation when the length of an intermediate or final result exceeds 255. It also occurs, when enabled, during assignment (by the '=' operator or during resolution but not by the '<-' operator) when the length of the value being assigned exceeds the maximum length of the location being assigned to.

Note that when this fault occurs during the evaluation of a complex expression or during the elaboration of a resolution, some of the intermediate assignments or other consequences (such as execution of functions or mapping functions) may already have been effected.

If the 'CAPACITY EXCEEDED' condition arises during assignment when the fault is disabled, the results will be unpredictable (for example through consequent over-writing of locations not intentionally referenced). It is therefore very undesirable that this fault be disabled before a program is well-proved to execute without the condition arising, or that adequate cognisance has been taken of the consequences.

Note that the '<-' assignment (described above) can be used to circumvent the 'capacity exceeded' condition in a controlled way.

STRING MANIPULATION FUNCTIONS

The following functions are either in the intrinsic or implicit category and hence can be called without being specified in the program (see Section 16).

```
%INTEGERFN CHARNO (%STRINGNAME S,%INTEGER N)
```
This returns the internal code value of the Nth character of string S. If N is greater than the current length of S then the result is undefined.

```
%STRING(255)%FN FROMSTRING(%STRINGNAME S,%INTEGER I,J)
```
The result is the sub-string of S comprising the Ith to Jth (inclusive) characters of S. The fault 'STRING INSIDE OUT' will occur unless 1<=I<=J and J is not greater than the current length of S.

```
%INTEGERFN LENGTH (%STRINGNAME S)
```
The result is the current length of the string, for example if S currently contains the string 'FIRE' then the result of a call of LENGTH(S) would be four.

```
%STRING(1)%FN TO STRING(%INTEGER I)
```
The result is a string of length 1 whose value is the character defined by the least significant 7 bits of the integer I.

STRING INPUT/OUTPUT ROUTINES

The routine provided for the input and output of strings are all in the implicit or intrinsic category, so do not need to be specified. They operate on the currently selected input and output streams. (See Section 10).

%ROUTINE READSTRING(%STRINGNAME S)
This routine which takes the name of a %STRING variable as its parameter and is used to read a string into the variable. The string should be written as described under the heading 'STRING CONSTANTS', above. Any spaces and newlines are ignored before the first quote character of the string. The trappable faults 'CAPACITY EXCEEDED' will occur if an attempt is made to read a string into a variable which has not been defined to be sufficiently long. Also 'INPUT ENDED' and 'SUBSTITUTE CHARACTER IN DATA' faults can occur. (See Section 10).

%ROUTINE READITEM(%STRINGNAME S)
This routine takes the name of a string variable as its parameter. It is used to read the next symbol from the current input stream and to put it into the variable as a string of length 1. Faults appropriate to READSYMBOL can occur.

%STRINGFN NEXTITEM
This is a string function which takes no parameter. It returns as its result a string of length 1 whose value is that of the next symbol on the current input stream. As with the function NEXTSYMBOL the pointer to the input stream is not moved by this function. Faults appropriate to NEXTSYMBOL can occur.

%ROUTINE PRINTSTRING (%STRING(255)S)
This routine takes a string expression as its parameter, of maximum length 255 characters. The expression is evaluated and the resulting string of characters is printed on the current selected output stream. PRINTSTRING effectively uses PRINTSYMBOL to output characters so all the characteristics of PRINTSYMBOL apply.

SECTION 9 - RECORDS

INTRODUCTION

'Records' in the IMP language provide a means of handling collections of data types as single entities. Like an array, a 'record' has an identifier which can be used to refer to the whole collection of data within it. Unlike an array, however, the sub-fields (elements) of it may have different types. Whereas an array element is referenced using the array identifier subscripted with an expression which evaluates a numerical index, an element of a record is referenced by subscripting the record identifier with the required sub-field identifier. The syntactic form of the subscript also is different for a record element, as will be made clear below.

> The content of this section of the manual may be subject to change in matters of detail with later versions of the IMP compiler.

RECORD FORMATS

A 'record format' defines the collection of objects which are to form a record. An IMP record format statement comprises a format identifier followed by a list of sub-field identifiers enclosed within parentheses.

Each sub-field identifier must have a 'type' which is one of the types of entities in IMP, namely

%BYTE or %SHORT %INTEGER, %INTEGER, %REAL, %LONGREAL, or %STRING,

Also arrays (but with single dimension and constant bounds) of the above types, %NAMEs and %ARRAYNAMEs ('pointers') of the above types may be used.

Additionally, a sub-field type may be %RECORD, %RECORDARRAY or %RECORDNAME and these are discussed below.

An example of a record format statement is:

```
%RECORDFORMAT F(%BYTEINTEGER A, %STRING(8) S, %C
    %INTEGERARRAY M,F(1:100), %REAL Y)
```

Record format statements do not cause allocation of storage (Section 5). Sub-field identifiers need not be distinct from identifiers of other entities in the program, block or routine, , since identifiers of sub-fields always occur in conjunction with the name of a record, as described below. Record format statements are placed at the head of a block or routine along with the storage-allocating declarations, and the same scope rules apply for these identifiers as for those of other entities in IMP.

Sub-fields named in record formats have 'lengths' and 'alignments' equal to those of the corresponding types of IMP entities; the record format likewise has a 'length' which is implied by the lengths and alignments of its sub-fields. This is discussed more fully below.

RECORDS

Records are declared in the same way as the arithmetic and string types of entities in IMP. They are declared at the heads of blocks or routines. The space which they occupy can be allocated dynamically, or they can be declared as %OWN, %CONST, %EXTERNAL or %EXTRINSIC. However, the space occupied by these latter types may not be initialized explicitly, and will be initially all zeros (all bits will be zero). The same scope rules apply for record identifiers as for arithmetic and string type identifiers. Space is allocated, and subsequently referenced, according to the record format whose identifier forms part of the record declaration. The required record format identifier, which must be previously declared and in scope, is written in parentheses following the record identifier. Several records of a given format may be declared in a single statement, as in the following examples:

```
%RECORDFORMAT F(%INTEGER A, %STRING(8) S)
%RECORD R(F)
%RECORD PP, QQ, RR(F)
```

The amount of space allocated is equal to the length of the record format. The first sub-field of the record is double-word aligned. A fuller discussion of lengths and alignments is given later in this section.

Each sub-field of a record can be referenced as an IMP location of appropriate type by subscripting the record identifier with the sub-field identifier: the record identifier is followed by the underline '_' character followed by the sub-field identifier. Thus, following the above format and record declarations, one may write:

```
%IF R_S='INC' %THEN R_A=R_A + 1
```

A further example follows:

```
%RECORDFORMAT PE(%INTEGER I, %REALARRAY X(0:10))
%RECORD P(PE)
%INTEGER J
P_X(J+1) = P_X(J) * 2
```

RECORD ARRAYS

'Record arrays' are entirely analogous to the arithmetic and string types of arrays. Each element of a record array is a record of format specified in the record array declaration. The identifier of the format which is to be applied to each element is written in parentheses following the record array identifier and bounds. Thus:

```
%RECORDFORMAT F(%INTEGER A, %REALARRAY X(1:5))
%RECORDARRAY RA(1:100) (F)
```

Then the fifth element of sub-field X in the 100th record array element may be referenced:

```
RA(100)_X(5)
```

As with other types of array, several record arrays having the same bounds and format may be declared in a single statement. Thus:

```
%RECORD %ARRAY RR1, RR2 (1:100) (F)
```

The space for a record array may be allocated dynamically, or the array may be %OWN, %CONST, %EXTERNAL or %EXTRINSIC. These latter types may not be initialized explicitly, but all bits will be initially zero. The first sub-field of the first element of the array is double-word aligned, but subsequent elements are given an alignment which provides the closest packing of the elements of the array consistent with the format of each element.

RECORD 'POINTER' VARIABLES

Analogously with the %NAME ('pointer') variables for the arithmetic and string entities, (see Section 7) 'record names' and 'record array names' may be declared; they must be given an associated format by writing a record format identifier (previously declared and in scope) in parentheses following the pointer variable being declared. For example:

```
%RECORDFORMAT F(%INTEGER A, %REALARRAY X(0:5))
%RECORDNAME F1(F)
%RECORDARRAYNAME Z,W(F)
```

These pointer variables are assigned to using the '==' operator, the right-hand operand of which must be a reference to a record location having the same format as that specified for the pointer variable which is the left-hand operand. Following the assignment, the pointer variable identifier is synonymous with that of the location reference which was assigned to it.

Thus, taking the declarations of the previous example, one may write

```
%RECORD Q,R(F)
%RECORDARRAY A(1:10) (F)
```

Then

```
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 synonymous with A.
```

A reference to a record location which is of particular value is the built-in special record mapping function RECORD, whose single parameter is a suitably-aligned address. This function may appear as the right-hand operand of an '==' assignment to a %RECORDNAME variable, which then provides a means of accessing sub-fields of the area starting at the address given as parameter (even though that area may not previously have had a format applied to it, or if it had previously been referenced as a location of different format). The following example illustrates both these points.

```
%INTEGER J
%INTEGERARRAY II(1:100)
%RECORDFORMAT A(%BYTEINTEGER I,J,K,L)
%RECORDFORMAT B(%SHORTINTEGER P,Q)
%RECORDNAME X(A)
%RECORDNAME Y(B)
X==RECORD(ADDR(II(J)))
Y==RECORD(ADDR(II(J)))
```

Now for example

X_I is a reference to the left-most byte of II(J)

Y_P is a reference to the left-most half-word of II(J).

SUB-FIELDS OF TYPE '%RECORD'

A sub-field in a record format statement may itself be of type %RECORD. The format for the sub-field identifier is written after it in parentheses as in the following examples:

```
%RECORDFORMAT P(%INTEGERARRAY X(0:4), %INTEGER I)
%RECORDFORMAT F1(%INTEGER A,B, %RECORD D(P))
%RECORDFORMAT F2(%RECORD J,K(P))
%RECORD ENT(F1)
%RECORD JAK(F2)
```

An arbitrary 'depth' of subscription can thus obtain, depending only on the scope rules for the declarations. With the above declarations, the following are valid references to record elements:

ENT_D_X(1)

JAK_J_I

SUB-FIELDS OF TYPE '%RECORDNAME': '%RECORDSPEC'

Additionally a sub-field of a format may be of type %RECORDNAME but in this case a format is given to the sub-field identifier not in the format statement itself, but in a separate and subsequent %RECORDSPEC statement, which is analogous to the %SPEC statement requirement for a %ROUTINE or %FN parameter (Section 6). For example:

```
%RECORDFORMAT F(%INTEGER I, %RECORDNAME J)
%RECORDFORMAT K(%REAL X,Y)
%RECORDSPEC F_J (K)
```

The following example is interesting in that the recursive nature of the format and sub-field format definitions facilitates the creation of a list structure:

```
%RECORDFORMAT F(%INTEGER DATA, %RECORDNAME LINK)

%RECORDSPEC F_LINK(F)

%RECORDARRAY P(1:1000)(F)
```

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

```
%INTEGER J

J=1

%WHILE J<1000 %CYCLE

P(J)_LINK==P(J+1)

J=J+1

%REPEAT
```

The 'link' field of the last element may be set zero using the built-in special record mapping function RECORD with a parameter of 0. Thus:

```
P(1000)_LINK==RECORD(0).
```

RECORDS AS PARAMETERS TO ROUTINES: '%RECORDSPEC'

Records may be passed as parameters to routines, functions and mapping functions only by 'name'. Where a routine is declared having a %RECORDNAME parameter, the identifier in the formal parameter list as usual has scope which is the textual extent of the routine but excluding contained blocks in which the identifier is re-declared. But the record location to be referenced by that identifier has no format implicitly associated with it within that routine. Before the formal parameter identifier can be used, it is necessary to include within the routine a '%RECORDSPEC' statement, which associates a record format identifier, in scope at that textual position, with the %RECORDNAME formal parameter identifier. The format thus associated with the parameter need not be identical with any format used outside the routine to reference locations outside the routine, although great care should be used in the use of different formats for the same record.

The record format identifier is written in parentheses following the identifier with which it is to be associated. Thus:

```
%ROUTINE R(%RECORDNAME P)
%RECORDFORMAT F(%BYTEINTEGER A,B,C,D)
%RECORDSPEC P(F)
```

RECORD ASSIGNMENTS

Whilst sub-fields of records may be assigned and manipulated exactly as if they were IMP entities of corresponding type, it is possible additionally to assign a whole record from one location to another. As with arithmetic and string assignment two assignment operators are permitted, namely '=' and '<-'. Both require that the left and right-hand operands are references to record locations, except that in the case of '=' a right-hand operand of zero is permitted, which has the effect of setting all bits in the left-hand location to zero. The '=' operator further requires that the formats associated with the left and right-hand operands have the same length; the '<-' effects a transfer of the number of bytes which is the smaller of the lengths of the two record formats. In both cases the transfer is without regard to considerations of format within either record.

The following example shows uses of the '=' assignment:

```
%RECORDFORMAT F(%INTEGER X,Y,Z,A)
%RECORDFORMAT Q(%BYTEINTEGERARRAY B(0:15))
%RECORD J(F)
%RECORDARRAY K(1:100) (Q)
J=K(1)
K(1)=0
```

LENGTH AND ALIGNMENT

The length of a record format, and the amount of space occupied by a record of given format, is the total number of bytes occupied by all the sub-fields, with the sub-fields, with the first sub-field double-word aligned and a minimum number of 'spacing' bytes inserted between adjacent sub-fields where necessary to achieve alignments appropriate to their types.

For example, the record format

```
%RECORDFORMAT F(%BYTEINTEGER B, %INTEGER I)
```

has length 8. Three bytes (not referenced explicitly through this format statement) must follow the sub-field B when the format is applied to an area of storage having double-word alignment in order that sub-field I is properly aligned. The requirement to double-word align a record may also result in up to 7 non-referenceable bytes being assigned.

The alignments and lengths of possible sub-fields of records formats are shown in the following table.

Type		Alignment	Length
%BYTEINTEGER	(and array)	byte	1 (and N*1)
%SHORTINTEGER	(and array)	half-word	2 (and N*2)
%INTEGER	(and array)	word	4 (and N*4)
%REAL	(and array)	word	4 (and N*4)
%LONGREAL	(and array)	double-word	8 (and N*8)
%STRING	(and array)	byte	L+1 (and N*(L+1))
%RECORD		double-word	R
%ARRAY %NAME		word	8
other %NAME		word	4

where N is the number of elements in the array,
L is the length of the string,
R is the length of the record.

In the case of %RECORDARRAYS the first record in the array is double word aligned. Any subsequent records are aligned as necessary for the elements therein.

SECTION 10 - INPUT/OUTPUT FACILITIES

INTRODUCTION

Input/Output (I/O) facilities are provided to enable programs to read data from input devices and files and to output data to output devices and files. This section describes the routines and functions provided by the IMP System in terms of their program characteristics. All the routines refer in some way to logical I/O CHANNELS. These channels are assigned numbers in the range 0 - 99, of which 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 there must be no conflict between channel numbers used for different types of file. Each implementation of the language provides facilities for linking these logical channels to particular files or devices and information on this subject is contained in the User Manual for the appropriate computer.

CHARACTER AND BINARY INFORMATION

The primary I/O facilities in IMP use character information, that is information which can be represented in sequences of printable characters. A variety of routines is provided to handle individual characters, or to interpret sequences of characters as numeric values, or string values.

Additionally I/O routines are provided to handle binary information, as a direct copy of the internal representation of values held in the computer's core store. Two types of binary I/O are provided - Sequential for use when data is accessed in the order in which it is held in the file, and Direct Access for use when data is accessed randomly from all parts of the file.

All the routines associated with character I/O are intrinsic or implicit - that is no declaration of them is needed before they are called. All the binary I/O routines are explicit, that is they must be declared in each program in which they are used.

CHARACTER CODES

All the character handling routines in IMP use an Internal Character Code based on the ISO Code for the Interchange of Data, see Section 16 of this manual. Some implementations of IMP may use other codes e.g. EBCDIC to represent characters on storage devices but this need not concern the IMP programmer since the necessary translation to and from the Internal Code will be carried out by the operating system.

CHARACTER STREAMS

All character handling routines and functions operate with respect to either the currently selected INPUT STREAM or the currently selected OUTPUT STREAM. On entry to a program the system selects default streams for input and output. At any point in the program it is possible to redirect character input or output by a call of

```
        %ROUTINE SELECTINPUT(%INTEGER I)
or      %ROUTINE SELECTOUTPUT(%INTEGER I)
```

Each of these routines takes one integer parameter which should have the value of the logical channel number required.

```
Examples:   SELECTOUTPUT (3)
            SELECTINPUT (I + 17)
```

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

Both input and output characters are buffered by the operating system into lines. This has the following effects on the use of SELECTINPUT and SELECTOUTPUT. If an input stream is re-selected later in a program the first character read after re-selection will be the first character in the line following that last accessed. Thus it is possible that part of a line will be lost. When an output stream is re-selected output will continue after the output that has already been put out. Additionally when SELECT OUTPUT is called a newline character is output on the current output stream if there is anything in the line buffer, before selecting the new stream.

CHARACTER INPUT ROUTINES

The following routines and functions operate on the currently selected input s tream.

```
%ROUTINE READSYMBOL(%NAME I)
```
This routine is used to transfer the internal value of the next symbol from the current input stream into an %INTEGER, %SHORTINTEGER or %BYTEINTEGER variable.

```
Example:    READSYMBOL (IN)
```

```
%INTEGERFN NEXTSYMBOL
```
This integer function which takes no parameter, 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 NEXTSYMBOL='A' %THEN ......
```

%ROUTINE SKIPSYMBOL
This routine, which takes no parameter, moves the pointer to the current input stream along one symbol without transferring any information to store. In the following example SKIPSYMBOL and NEXTSYMBOL are used together to skip over a series of space characters.

```
%WHILE NEXTSYMBOL=' ' %THEN SKIPSYMBOL
```

%ROUTINE READ(%NAME I)
This routine is used to read numeric data into arithmetic variables. The single parameter should be a variable of type %INTEGER, %SHORTINTEGER or %BYTEINTEGER if the number being read is known to be integral or of type %REAL or %LONGREAL if the number being read is likely to include a fractional part. The numbers being read should be written as described under the heading 'DECIMAL CONSTANTS', in Section 1. Note the following points:

1. READ will skip over any SPACE or NEWLINE characters which precede the number. Thus space or newline characters can readily be used as separators between numbers.

2. On returning from READ the input pointer will be left pointing to the character immediately following the number.

3. The fault 'REAL INSTEAD OF INTEGER IN DATA' will occur if an attempt is made to read a real number, or a number with an exponent, into an integer variable.

4. The fault 'SYMBOL IN DATA' will occur if the first character of a number is neither a sign, a decimal point or a decimal digit.

%ROUTINE READSTRING(%STRINGNAME S)
This routine, which is described in Section 8 is used to read a value into a %STRING variable.

%ROUTINE ISOCARD(%BYTEINTEGERARRAYNAME B)
This routine, which is intended for use with fixed format data, is used to read a card image into a %BYTEINTEGERARRAY. The parameter should be the name of the array, and the card image will be read into elements 1-80 of it.

Example:
```
%BYTEINTEGERARRAY INCARD(1:80)
    ISO CARD (INCARD)
```

LINE RECONSTRUCTION

All the above routines except ISOCARD operate on data which has undergone a process known as line reconstruction. The main features of this are:

1. All characters before the left hand margin and after the right hand margin are suppressed. (See SET MARGINS below).

2. All space characters at the right hand end of the line are suppressed.

3. Double quote deletion is performed - that is each double quote (") character and the character preceeding it is suppressed, as far back as the beginning of the line.

4. All marked characters (see Section 16) are suppressed.

5. All illegal characters are translated to SUB (decimal value 26). In addition, when an attempt is made to read the first character of a line containing a SUB character the fault 'SUBSTITUTE CHARACTER IN DATA' occurs. This fault may be trapped (See FAULT TRAPPING) in which case the next attempt to read the first character of the line will be successful although the user must be prepared for it, or at least one character in the line, to be a SUB.

AVOIDING LINE RECONSTRUCTION

It is occasionally useful to be able to read all characters from the input stream before line reconstruction has been performed. The routine READCH is provided for this purpose.

```
%ROUTINE READCH(%NAME I)
```
This routine, like READSYMBOL requires one parameter which must be a variable of type %INTEGER, %SHORTINTEGER or %BYTEINTEGER. A call of READCH will transfer the next character from the input stream to store in internal code, without regard for Margin Settings, double quote deletion, trailing space, or Marked Character suppression or illegal character conversion.

INPUT ENDED

All of the above input routines cause the fault 'INPUT ENDED' if an attempt is made to read beyond the end of a file. This fault can be trapped (See FAULT TRAPPING).

CHARACTER OUTPUT ROUTINES

A number of routines are provided to write individual characters or sequences of characters to the output stream. Although most routines only output part of a line it is important to appreciate that the characters are initially put in a line buffer which is only output on the file or device when a NEWLINE or NEWPAGE character is output. This fact is particularly relevant when using a TELETYPE or other terminal as an output device. The basic character output routine is PRINTSYMBOL; most of the other output routines operate via this routine.

```
%ROUTINE PRINTSYMBOL(%INTEGER I)
```

This routine takes one parameter which should be an integer expression. The expression is evaluated and the least significant 7 bits only are used by the routine. If the value of this corresponds to a symbol in the IMP extended character set (see Section 16) it is sent to the output stream. If not, the character SUB (decimal 26) is sent.

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

TEXT OUTPUT ROUTINES

The following routines are provided to simplify the output of text:

SPACE	-	no parameter - outputs one space
SPACES(n)	-	where n is an integer expression - outputs n spaces
NEWLINE	-	no parameter - outputs one newline character
NEWLINES(n)	-	where n is an integer expression - outputs n spaces
PRINTSTRING(s)	-	where s is a string expression - outputs the string. See Section 8.
%PRINTTEXT'TEXT'	-	This is not a routine call, it is a built in phrase known to the compiler. Any characters may be included in the quotes following %PRINTTEXT. Note that all characters included within the quotes will be printed including spaces and newlines. Note that the routine PRINTSTRING has a similar effect and is a preferred alternative to %PRINTTEXT. However since a PRINTSTRING is a routine its parameter must be contained within brackets. Hence, for example the two following lines would have the same effect:

```
PRINTSTRING('MY PROGRAM')
%PRINTTEXT'MY PROGRAM'
```

OUTPUT OF NUMBERS

Three routines are provided for the output of numeric information. WRITE is used to output the value of integer expressions, PRINT and PRINTFL are used to output the value of real expressions, the latter in floating point form.

%ROUTINE WRITE(%INTEGER I,J)
This routine takes two parameters, both should be integer expressions. The first is the value to be output, the second is the number of positions to be used. To simplify the alignment of positive and negative integers an additional position is allowed by the routine for a sign, which is only printed for negative numbers. If the value being output is too large for the number of positions specified more positions are taken.

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

%ROUTINE PRINT(%LONGREAL X,%INTEGER I,J)
This routine takes three parameters. The first should be the real expression whose value is to be printed, the second and third should be integer expressions specifying the number of places to be allowed before and after the decimal point. The same arrangements apply for the sign and for insufficient space for the integer part as for WRITE. If necessary the fractional part will be rounded.

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

%ROUTINE PRINTFL(%LONGREAL X,%INTEGER I)
This routine takes two parameters; the first is a real expression whose value is to be printed, the second is an integer expression specifying the number of digits to be printed after the decimal point. The printed number takes up the specified number of places plus 7 additional places.

```
Example:    PRINTFL(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, and as for WRITE a space is allowed for a sign for both the mantissa and the exponent.

PRINTING NON-STANDARD CHARACTERS

Occasionally it is useful to print characters that do not appear in the extended IMP character set. The routine PRINTCH has the same effect as PRINTSYMBOL except that any character whose internal code value is in the range 0-127 can be output.

OUTPUT EXCEEDED

If an attempt is made to output more data than can be accommodated on the current output stream then the fault 'OUTPUT EXCEEDED' will occur. This cannot be trapped.

CLOSING STREAMS

All input and output streams are closed automatically at the end of a job. If it is necessary to close a stream during a job the routine CLOSE STREAM can be used. It takes one parameter which should be an integer expression whose value is the number of the stream to be closed. Note the following points.

1. An attempt to close the currently selected input or output stream will fail.

2. If a stream is closed and selected later using SELECTINPUT the pointer to the stream will be positioned at the start of the file. This makes it possible to re-read a file, or to read a file which has been written earlier in the program.

3. If a stream is closed and selected later for output, the output will be written from the start of the file. Any other output in the file will be lost.

SET MARGINS

Associated with each input and output stream are settings for the left hand and right hand margins. Some of these settings are defined by the IMP system and this information will be found in the relevant User Manual. All other streams use default settings of 1 and 80. Margin settings can be altered by a call of the routine SET MARGINS. This routine takes three parameters, all being integer expressions. The first is the stream number, in the range 1 - 80, which must be that of either the currently selected output stream or the currently selected input stream. The second and third should be the left hand margin setting and right hand margin setting required.

For input streams the margins must be in the following range:

```
1<=LHM<=RHM<=160
```

The characters on a line to the left of the left hand margin and those to the right of the right hand margin will be ignored.

For output streams the margins must be in the following range:

```
1<=LHM<=RHM<=132
```

The effect of altering the left hand margin will be to tabulate the output, for example if the left hand margin is set to 10 all output lines will be preceeded by 9 spaces. The right hand margin is used to limit printing on a line. If an attempt is made to print beyond the right hand margin then a newline character is inserted and printing continues on the next line.

If a stream is reselected its margins should be set again using SET MARGINS.

BINARY FILES

Binary files can be used for storing intermediate results of computations, either during the execution of one program, or for use by subsequent programs. The data stored is a direct copy of the contents of the core store. The file handling routine calls all make explicit references to the logical channel being used, there is no equivalent to the currently selected stream used for character I/O. The variables accessed by binary I/O calls are normally held in arrays and the read and write routines each require a specification of the area of core store to be accessed. For example the WRITESQ routine uses its second and third parameters to specify the area of the store to be output.

Example:
```
%INTEGERARRAY IN(1:1000)
    WRITESQ(CHAN,IN(1),IN(500))
```

In this example the first 500 elements of the array IN will be written out to the sequential file defined for channel CHAN.

The following notes refer to all the binary file handling routines.

1. Each routine call should reference only one array.

2. The type of an array used in a call of a binary input routine should be the same as that used for the call of binary output routine that created the file.

3. When using multi-dimensional arrays the whole array should be written and read, unless the full implications of the layout in store of the array are understood.

SEQUENTIAL BINARY FILES

Sequential binary files (SQFILES) are accessed by the routines OPENSQ, WRITESQ, READSQ and CLOSESQ. All these routines are in the explicit category, that is they must be declared at the head of the program or routine in which they are used. The specifications of the routines are given below. The precise characteristics of the file handling routines may vary from one implementation to another, and further information should be sought from the User Manual for the appropriate computer.

```
%EXTERNALROUTINESPEC OPENSQ(%INTEGER I)
```
This routine takes one parameter, the channel number of the file to be opened. It must be called before READSQ or WRITESQ is called for that channel. If a file is closed in a program and then re-opened it will be reset to the start. By this means it can be reread or information that had been written earlier in the program can be read. In the case of writing after re-opening, since the file is positioned at the start all information in it will be lost.

```
%EXTERNALROUTINESPEC CLOSESQ(%INTEGER I)
```
All files are closed automatically at the end of a job. CLOSESQ is only required if it is necessary to re-open a file.

%EXTERNALROUTINESPEC WRITESQ(%INTEGER I,%NAME A,B)
Each call of the WRITESQ outputs one logical record. The operating system may store records in a variety of ways on its storage device but this need not concern the programmer. The routine takes three parameters; the channel number and two %NAME parameters to define the area to be written out (see above).

Example: WRITESQ(2,A(27),A(426))

%EXTERNALROUTINESPEC READSQ(%INTEGER I,%NAME A,B)
This routine reads in one record, normally written out by a call of WRITESQ. The first parameter defines the channel to be used and the second and third the area of store into which the record is to be read.

Example: READSQ(2,A(1),A(400))

%EXTERNALINTEGERFNSPEC LENGTHSQ
This function returns as its result the length, in bytes of the last record read by a call of READSQ. It is useful in situations where the length of records vary. Note that the length is that of the user data read into the program variables. It does not include any system information which is sometimes appended to the record.

SEQUENTIAL FILE FAULTS

The fault 'INPUT ENDED' will occur if an attempt is made to read beyond the end of a file. This fault can be trapped. None of the following faults resulting from use of sequential file handling can be trapped.

1. Attempting to use READSQ, WRITESQ or CLOSESQ for a file which is not open.

2. Attempting to OPEN a file which is already open.

3. Attempting to write or read an area such that the address of the first element is higher than that of the last element, for example:
   WRITESQ(1,IN(100),IN(1))

4. Attempting to mix READSQ and WRITESQ calls for the same channel, without the use of an intermediate CLOSESQ and OPENSQ.

DIRECT ACCESS BINARY FILES

Direct Access Files (DAFILES) are used for similar purposes to SQFILES. There are two main differences in the method of use.

1. Records can be accessed in a random order.

2. Calls of both WRITEDA and READDA are legitimate on an open channel.

All the DAFILE routines are in the explicit category, that is they must be declared before they are used. The specifications of the routines are given below. Each record of a DAFILE contains a maximum of 1024 bytes. Records are referenced by their position in the file, starting at 1.

```
%EXTERNALROUTINESPEC OPENDA(%INTEGER I)
```
This routine takes one parameter, the channel number of the file being opened. It must be called before a call of READDA or WRITEDA for the file. When OPENDA is used for the first time with a particular file it initialises the file by writing the unassigned pattern to all the records. Thus, if an attempt is made to read from a record to which nothing has been written an 'UNASSIGNED VARIABLE' failure will occur when the data which has been read is accessed.

```
%EXTERNALROUTINESPEC CLOSEDA(%INTEGER I)
```
All DAFILES are closed automatically at the end of a job so this routine is rarely needed. It may be required to minimise the number of files which are concurrently open.

```
%EXTERNALROUTINESPEC WRITEDA(%INTEGER I,%INTEGERNAME J,%NAME A,B)
```
This routine is used to write data from an area of core store to a file. A call of WRITEDA will result in one or more records being written, depending on the number of bytes in the area defined. For example if 256 elements of an %INTEGER array are written out then only one record will be used, whereas if 1024 %INTEGERs are written four records will be used. If the area written is not an exact multiple of 1024 bytes then the end of the last record written will be filled with rubbish. The first parameter is an integer expression which defines the channel number, the second is the name of a %INTEGER variable. On entry this should contain the number of the first record to be written. On exit it will contain the number of the last record written by this call, which may be used for checking purposes. The third and fourth parameters define the area of store to be written, as for SQFILES.

```
%EXTERNALROUTINESPEC READDA(%INTEGER I,%INTEGERNAME J,%NAME A,B)
```
This routine is used to read data written by WRITEDA. The parameters are used in the same way and if an attempt is made to read into an area which is not an exact multiple of 1024 bytes, the area is filled and the rest of the last record is ignored.

DIRECT ACCESS FILE FAULTS

None of the following faults associated with the use of DAFILES can be trapped:

1. Attempting to use READDA or WRITEDA before opening a file.

2. Attempting to access a record with a number that is less than 1 or greater than the number of the last record in the file.

3. Attempting to access an area such that the address of the first element is higher than that of the last.

SECTION 11 - AIDS TO PROGRAM DEVELOPMENT

INTRODUCTION

The IMP compilers normally operate in diagnostic mode. This means that when compiling a source program, additional code is planted to enable the program to give useful diagnostics in the event of an error occurring at run time. The type of checks which are made are listed below, under their standard compiler options. The User Manual for the computer being used will provide information about selecting suitable compiler options.

CHECK (Unassigned checking)

Checks are made that all variables, except byte and short integers, are assigned to, before being used. Parameters passed to routines by value are checked at time of passing but those passed by name are not checked at all at present. Unassigned checking can be suppressed by use of the compile time option 'NOCHECK'.

ARRAY (Array bound checking)

Checks are made to ensure that the declared bounds of arrays are not exceeded. Checks are also made to prevent either too large an entity being assigned to a byte or short integer variable, or too many characters being assigned to a string variable. Additionaly checks are made for attempts to jump to %SWITCH labels that are not set, or are outside the bounds declared for the %SWITCH. These checks can be suppressed by using the compile time option 'NOARRAY'. Note that the current release of the IMP compiler suppresses array bound checking on all arrays when 'NOARRAY' is used. This was not the case with earlier releases.

TRACE (Routine traceback)

Code is planted to allow a routine trace back to be given. This facility can be suppressed by 'NOTRACE'.

DIAG (Line number updating and local variable values)

Code is planted to save the line numbers of IMP statements and thereby link any diagnostics to a specific line of IMP code. Code is also planted to enable the local scalar variables to be printed during a routine traceback, unless 'NOTRACE' has been selected. These diagnostics can be suppressed by use of 'NODIAG'.

Should a program fail, the diagnostics package is entered after the cause of the failure has been reported. It is also entered on execution of the instructions, %MONITOR and %MONITORSTOP. The function of the diagnostics package is as follows:

1. To identify (subject to TRACE being operative) the logical block i.e. routine, function, map or block in which the failure occurred by printing out its name, in the case of a routine, function or map, and the number of its first line if in a block. For example:

```
ROUTINE/FN/MAP FRED STARTING AT LINE 31
BLOCK STARTING AT LINE 6
```

2. To print out (subject to DIAG being operative) the values at the time of failure of all the scalar variables (up to a maximum of 100) declared in the logical block. If no value has been assigned to a variable, this is indicated. Records and arrays are not printed.

```
LOCAL SCALAR VARIABLES
J= 10
TABLEOFC= 4001
SHOLD='THIS IS'
K= NOT ASSIGNED
```

   Note that the names of scalar variables are truncated at eight symbols, and that string values are enclosed in quotes.

3. If the logical block is a routine, function or map, to print out the number of the line from which it was called. This may be in the current routine, function or map if the call was recursive.

4. To repeat the above three functions for the logical block from which the previous logical block was entered. Thus routines, functions or maps, which are used recursively, will have the values assigned to the variables in each occurrence printed out.

   There is one overall restriction on the amount of diagnostics given. After 400 lines of diagnostics the output is terminated and the message

```
DIAGS OUTPUT EXCEEDED
```

   is given.

Certain source statements which are indicated below may also be included in a program under development to provide further information for de-bugging purposes.

PROGRAM LISTING

A line by line transcript of all statements in an IMP program is automatically given for each compilation unless the compiler option 'NOLIST' has been specified in the job control when a program is compiled. Then a short listing in which only the start and finish of each 'begin', 'routine', 'function' or 'map' block is produced.

Selective listing of statements in particular sections of a program may be obtained by enclosing each required section between the statements %LIST and %ENDOFLIST and giving the compiler option 'NOLIST' in the job control or command language. Note that apart from saving paper the 'NOLIST' option will result in faster compilation, and hence should be used unless a full listing is really required.

EFFICIENCY

The following suggestions are given to enable a programmer to program in a way which will make more efficient use of machine time.

1. The parts of a program in which efficiency is particularly important are those which are executed many times.

```
%CYCLE I=1,1,100
    %CYCLE J=1,1,100
        <code>
    %REPEAT
%REPEAT
```

In the above example, any minor improvement made in the code in the inner cycle will result in a total saving in time of 10,000 times that small amount.

2. Whenever some part of an expression which is a constant factor, is required many times, it should be evaluated once and stored as shown in the right hand version of the example below.

```
%CYCLE I=1,1,100
    A(I)=A(I)*K*22/7
%REPEAT
```

```
K=K*22/7
%CYCLE I=1,1,100
    A(I)=A(I)*K
%REPEAT
```

3. It takes longer to move numbers in and out of array elements than to access simple variables. The right hand version of the example given below is the more efficient.

```
A(I,J)=0
%CYCLE K=1,1,15
    A(I,J)=A(I,J)+B(K)
%REPEAT
```

```
X=0
%CYCLE K=1,1,15
    X=X+B(K)
%REPEAT
A(I,J)=X
```

4. Integer to real conversion is costly and it is often worthwhile to use brackets to separate out integer sub-expressions, as shown below on the right.

```
%REAL X
%INTEGER I,J,K
    ......
    X=X+I+J-K
```

```
%REAL X
%INTEGER I,J,K
    ......
    X=X+(I+J-K)
```

5. If an arithmetic expression contains a mixture of real and long real variables then it is evaluated double length. It is slow to stretch real variables to long real variables and brackets can often be used to minimise the cost of conversion, as shown below on the right.

```
%REAL X,Y,Z                          %REAL X,Y,Z
%LONGREAL A,P                        %LONGREAL A,P
    .......                              .......
    P=X*A/COS(Z)*Y/Z                     P=(X*Y/Z)*A/COS(Z)
```

It is particularly important to note this effect when making a call on library routines such as SIN, COS etc., which are all double length, from a program having mostly single length real variables.

6. When array bound checking has been suppressed, array suffices of the form (I+constant) are more efficient than (I-constant). The example given below left may be reprogrammed more efficiently as shown on the right.

```
%CYCLE I=1,1,N                       %CYCLE I=-1,1,N-2
    B(I)=X+C(I-2)*C(I-3)                 B(I+3)=X+C(I+2)*C(I)
%REPEAT                              %REPEAT
```

7. Output should be reduced to the minimum which is really useful to the programmer in the interest of saving machine time and money. To this same end, answers should be printed across the line printer page where possible as the cost is proportional to the number of lines printed.

8. Once a program has been thoroughly tested, its efficiency can be greatly increased by turning off certain or all of the diagnostic checks made by the compiler. This results in a great saving in space as well as time. All programs which are past the initial testing stage should be compiled without the unassigned checking option. As more production experience with the program is obtained it may be possible to compile without some of the other options, array bound checking for example. Programs should be made as efficient as possible by other means, however, before removing any of the checks.

9. There is a compile time option 'OPT' which has the effect of switching off array bound checking, unassigned checking and diagnostics for line numbers and values of local variables. Additionaly some optimisation of the program is carried out and other checking is suppressed. Although the result is a program that executes more efficiently this facility should be used with caution. It should not be used on programs which have Compile Time Faults since the error messages that are produced may be misleading. Further it should only be used for programs or %EXTERNAL routines which have been thoroughly tested. Much time may be wasted trying to diagnose faults which result from the use of 'OPT' with incorrect programs.

SECTION 12 - COMPILE TIME FAULTS

INTRODUCTION

As the compiler processes a source program and produces a map or listing, it may encounter errors in the syntax or the semantics of certain statements. If this occurs, the compiler issues a message to describe the situation of the error and as far as possible, its nature.

Two main types of error are defined; syntactic errors and semantic errors.

SYNTACTIC ERRORS

These are errors which cause the form of the statement to be unrecognisable, since the strict rules of syntax have not been obeyed. Mistakes such as omission of statement separators or misspelling of key words are examples. This type of error is indicated by the message

```
* 111   SYNTAX
```

where 111 is the line number of the incorrect statement. A copy of the offending source statement is output on the next line of the program map or listing.

Where possible, the compiler indicates the exact position of the SYNTAX error by outputting a marker (!) under the character where the analysis failed. Sometimes, however, this marker can only be approximate as, for example, in the following case:

If the statement

```
%ROUTINE SPECIFY ORBITALS
```

is mistyped as

```
%ROUTINESPECIFY ORBITALS
```

the failure message would be

```
* 111 SYNTAX    ROUTINESPECIFY ORBITALS
                          !
```

The marker is misplaced to the right as the erroneous line more nearly corresponds to a '%ROUTINESPEC' statement than the intended '%ROUTINE' statement.

The compiler will continue to process the remainder of a program after detecting such an error, but will not allow the program to be executed.

SEMANTIC ERRORS

These occur when a statement is syntactically correct, but causes ambiguity in meaning or is totally meaningless. Examples are the declaration of a name for two uses in the same context, or the use of a name which has not been declared at all.

The following list describes the semantic errors detected by the compiler and diagnosed by messages of the type

```
* 125 FAULT n
```

where 125 is the number of the faulty line and n the number of the fault. The fault number will be followed by an abbreviated description of the fault which will usually be sufficient for the programmer to identify his mistake. Fuller descriptions of current Compile time faults are given below. If a program produces any fault not listed below then the user should contact the Advisory Service for further information. All will cause rejection of the offending program at the end of compilation.

FAULT 1 (Too Many %REPEATS)

A %REPEAT instruction is encountered for which no %CYCLE statement earlier in the same block can be matched uniquely.

FAULT 2 <label> (Label Set Twice)

The current instruction bears a label which has already been used to identify a statement in the same block. This will clearly cause ambiguity. Also occurs if a numeric label is not within the permitted range of 1<=label<=16383.

FAULT 3 (%SPEC Faulty)

The offending statement is a %SPEC in the short form, within a routine, function or map, which specified a name not appearing in the formal parameter list of the current block, or which appears in a %BEGIN block context.

FAULT 4 (%SWITCH Vector Not Declared)

A name used in the context of a switch label has not been declared as a %SWITCH name in the current block or routine.

FAULT 5 (%SWITCH Label Error)

The subscript appended to the name used as a %SWITCH label is not a single integer constant or is outside the range defined by the declaration of the switch. (See also FAULT 18).

FAULT 6 (Switch Label Set Twice)

The current instruction is identified by a switch label which has already been used to label a statement in the same block.

Fault 7 <name> (Name Set Twice)

A declaration statement declares 'name' which is already set in the current block, except when the name is that of a %ROUTINE, %FN or %MAP description which has been previously specified but not described within that block.

If the statement declared a number of names, any of these not already set is set in the normal fashion. The diagnostic will appear once for each name which is already set.

FAULT 8 (Too Many Parameters in Routine Type Description)

A routine type description is encountered for which a declaration already exists, and the number of parameters declared in the description exceeds that in the declaration.

FAULT 9 <name> (Parameter Fault in Routine Type Description)

The type of a formal parameter name appearing in a routine type description differs from the corresponding parameter appearing in an earlier declaration.

FAULT 10 (Too Few Parameters in Routine Type Description)

A routine type description is found to declare fewer parameters than the corresponding specification.

FAULT 11 <label> (Label Not Set)

On encountering an %END statement, it is found that the label, <label>, has been referenced in a jump instruction in the current block, but has not been identified with any statement in that block. This message appears, therefore, immediately before the 'END OF BLOCK' message. Note that %ROUTINE, %FN or %MAP descriptions are treated as separate blocks. (On 'END OF PROGRAM' this also refers to labels appearing in %FAULT statements that are unset).

FAULT 12 (Type General Parameter Misused)

An attempt has been made to store or fetch into a type general parameter (e.g. %NAME). These are only used by the input/output routines and this fault should not normally be encountered.

FAULT 13 (%REPEAT Missing)

This diagnostic appears at the end of a block, and indicates that a %CYCLE has been opened in that block but has not been matched uniquely with a %REPEAT instruction in the same block.

FAULT 14 (Too Many %ENDS)

An %END statement is encountered which matches logically with the opening %BEGIN of the program. It should rightfully be an %ENDOFPROGRAM. Compilation ceases and the job terminates. Any subsequent text is not scanned.

FAULT 15 (Too Few %ENDs)

An %ENDOFPROGRAM statement is found to correspond to the opening of a block internal to the main program level, showing a lack of block ends.

FAULT 16 <name> (Name Not Set)

A name employed in the offending instruction has not been declared. The name quoted is artificially declared as an %INTEGER so that this diagnostic is suppressed on later appearances of the name. However, other faults may occur later if the 'name' is used subsequently, in any other context e.g. as a %REAL name which will cause FAULT 24.

FAULT 17 (Not a %ROUTINE Name)

A statement having the form of a routine call is encountered. The name quoted in this statement is that of an item declared as something other than a routine. (FAULT 16 will indicate the case where the name has not been declared at all).

FAULT 18 (%SWITCH Vector Error)

In a %SWITCH declaration, the upper or lower subscript bound quoted for any switch named is outside the range -32767 to +32767. Both bounds are set to zero. The offending switchname(s) will be valid in the present block, but any reference may generate a diagnostic. (See FAULT 5).

FAULT 19 <name> (Wrong Number of Parameters or Subscripts)

A reference to the name of an array is made, but the number of subscripts appended to it does not agree with the dimensionality declared for that array.

This diagnostic also occurs when the number of actual parameters attributed to a routine call is not equal to the declared complement of formal parameters.

FAULT 20 (%SWITCH Vector or %RECORDFORMAT name in Expression)

A name appearing in an arithmetic expression has been found to identify a %SWITCH in which circumstances it is patently illegal. Also occurs if a %RECORDFORMAT name is found in an expression.

FAULT 21 (Routine Type or %RECORD Name Not Yet Specified)

A %ROUTINE, %FN or %MAP named as a formal parameter of another routine type block is referenced in that block before the parameter has been specified (by a %SPEC statement). Hence its own parameter list is unknown. Also occurs if a %RECORDNAME variable is used before being specified.

FAULT 22 (Actual Parameter Fault)

In a reference to a routine type name, an actual parameter is of incorrect type as defined in the declaration of that routine.

FAULT 23 <name> (%ROUTINE Name in Expression)

A name appearing in an arithmetic expression has been found to identify a %ROUTINE, in which circumstances it is patently illegal.

FAULT 24 (Real Quantity in Integer Expression)

In any expression assigned to an %INTEGER variable or otherwise expected to have an %INTEGER value, a %REAL constant, variable or function name has been employed illegally. If the expression occurs in an array declaration, as a dimension bound, the array name remains set. Also occurs if a logical operator is found in a real expression.

FAULT 25 (%CYCLE Variable Not Integer Type)

The control variable named on the left-hand side of a %CYCLE assignment is not an %INTEGER variable. N.B. %BYTEINTEGER and %SHORTINTEGER variables are not permitted as control variables.

FAULT 26 (Fault Statement Not at Basic Textual Level)

The %FAULT statement is allowable only in the outermost block of the program, and must refer to labels existing in that block only. Appearance of a %FAULT statement in any internal block will result in this message.

FAULT 28 <name> (%ROUTINE Body Not Described)

Occurs at the end of a block when a specification appears in that block for a routine type name which is not described, whether referred to or not. The name given is that quoted in the offending %ROUTINESPEC.

FAULT 29 (LHS Not a Destination or Name is Not an Address)

In an assignment statement, the name appearing on the left-hand side of the assignment is not a variable name.

FAULT 30 (%RETURN Out of Context)

A %RETURN statement occurs in a block other than a %ROUTINE in which circumstances it is meaningless.

FAULT 31 (Result Out of Context)

A %RESULT statement occurs in a block other than a %FN or %MAP body in which circumstances it is meaningless.

FAULT 34 (Textual Level > 8)

This occurs immediately after the opening statement of a new begin block or routine type description which causes nesting of blocks to 8 levels (the main program being at level 1, allowing for the compiler level). The compiler is unable to monitor this depth of nesting during execution, and hence subsequent object code produced is lost. However, the compiler contrives to scan subsequent text in the normal way for further syntactic or semantic errors.

FAULT 35 (Routine Level > 5)

The routine level (initially one) is increased every time a %ROUTINE %FN and %MAP statement is encountered and decreased when the corresponding %END is found. Insufficient addressing registers are available on Systems 4 and 370 to allow more than five routine levels. The compiler will continue to check subsequent text for errors.

FAULT 36 (Attempt to Trap an Untrappable Fault)

This diagnostic occurs if an untrappable fault number appears in a %FAULT statement.

FAULT 37 (Array Has Too Many Dimensions)

In this implementation, arrays are restricted to a maximum of seven dimensions.

FAULT 38 (Overflow)

Overflow has occurred while compiling the statement. This is caused by using a constant that is too large for the type of variable involved.

FAULT 39 (Real Quantity as Exponent)

A %REAL constant, variable or parenthesised expression appears immediately to the right of an exponentiation symbol, in which position it is illegal. This diagnostic replaces the diagnostic 'FAULT 24' in these circumstances.

FAULT 40 (Declarations Misplaced)

Declarations must be placed at the head of the block in which they occur. In particular they must come before any labels, %FAULT statements, jumps, conditional statements or cycles in the same block.

FAULT 42 (%STRING Variable in Arithmetic Expression)

In any expression deemed by context to be arithmetic, a %STRING variable or the concatenation operator ,(.), has been found.

FAULT 43 (Bound pair inside out)

In a declaration, a bound pair consisting of two constants has the lower bound greater than the upper bound. Both bounds are set to the lower bound and the declaration is accepted, in order to reduce the number of subsequent error messages.

FAULT 44 (Const Error)

A constant of incorrect type has been used to initialise an %OWN variable. May be a %REAL constant for an %INTEGER variable or too large a constant for the variable.

FAULT 45 (%OWN Array Error)

An own array has been declared where the number of constants does not correspond with the bounds.

FAULT 46 (%EXTRINSICS)

An attempt has been made to initialise an %EXTRINSIC variable. Extrinsic variables exist in a separately compiled module and thus cannot be initialised.

FAULT 47 (Dangling %ELSE)

An %ELSE has been found after a %FINISH which is not associated with a condition.

FAULT 48 (Substitute Character in Program)

The substitute character has been found in a line of program which is listed. No attempt is made to analyse the offending line as this would merely result in a SYNTAX fault.

FAULT 51 (Spurious %FINISH)

A %FINISH has been found for which no %START exists.

FAULT 52 (Missing %REPEAT Inside %START/FINISH)

This diagnostic occurs at a %FINISH and indicates that a %CYCLE has been started within a %START- %FINISH block but the corresponding %REPEAT has not been found.

FAULT 53 (%FINISH Missing)

Within a block or routine there exist more %START statements than %FINISH statements.

FAULT 54 (%EXIT out of context)

An %EXIT statement has been found which is not within a %CYCLE-%REPEAT context.

FAULT 55 (%EXTERNALROUTINE in Program)

An %EXTERNALROUTINE has been found in a program. %EXTERNALROUTINEs are only allowed in library files (see Section 6).

FAULT 56 (%ENDOFFILE Out of Context)

An %ENDOFFILE statement has been found in a program. %ENDOFFILE is only allowed when compiling library files. This fault also occurs if %ENDOFPROGRAM occurs when compiling a library file (see Section 6).

FAULT 57 (Level 0 Used)

Statements have been found at level 0. The first %BEGIN has probably been omitted. When compiling library routines, %OWN, %CONST, %EXTERNAL and %EXTRINSIC variables may be declared at level 0 and used to communicate between routines.

FAULT 62 (Wrong Format)

An attempt has been made to declare an entity of type %RECORD by reference to a name which is not currently declared as a %RECORDFORMAT.

FAULT 63 (%RECORDSPEC in Error)

An attempt has been made to assign a format to an entity not of type %RECORD or to a %RECORD whose format is already known.

FAULT 64 (Subname Omitted)

A reference to a %RECORD element does not specify a subname.

FAULT 65 <name> (Wrong Subname)

The indicated subname cannot be found in the %RECORDFORMAT statement referenced by the %RECORD declaration.

FAULT 66 (Record assignment)

An attempt to assign one record to another cannot be compiled as the records are of different sizes.

FAULT 69 (Subname Out of Context)

A subname has been attached to an entity which is not of type %RECORD.

FAULT 70 (Invalid Length in String Declaration)

The maximum length of the string being declared has either been omitted or lies outside the range 1 to 255.

FAULT 71 (String Expression Contains a Variable)

A variable of type other than %STRING has been found in a string expression.

FAULT 72 (String Expression Contains Invalid Operator)

An operator other than '.' has been found in a string expression.

FAULT 73 (Resolution Comparator Out of Context)

The resolution comparator, ->, has been used with a variable which is not a string or else in a double sided condition.

FAULT 74 (Resolution Format Incorrect)

The bracket expression is not correct. This is usually caused by the omission of the brackets themselves.

FAULT 75 (String Expression Contains Subexpression)

Bracketed subexpressions are neither required nor permitted in string expressions. Brackets may only occur in string expressions as described under resolution, (see Section 8).

FAULT 81 (Item == <exprn>)

The address assignment operation '==' has been used to equivalence a variable to an expression, which is patently absurd.

FAULT 82 (Not an Address)

The address assignment operator has been used to assign an address to a variable that is not of %NAME type.

FAULT 83 (Non Equivalence)

The '==' operator has been used to equivalence operands that are not of identical type.

FAULT 84 (RECORD Misused)

The special mapping function RECORD has been misused.

FAULT 98 (Addressability)

The program or one of its data areas exceeds the limit of addressability (at present 1/4 megabyte).

CATASTROPHIC COMPILE TIME FAULTS

Some errors which exceed the physical limits imposed on the compiler by its particular operating environment are catastrophic and result in compilation ceasing at the point the error occurred. These faults have numbers greater than 100.

FAULT 101 (Source Line Too Long)

The line of source text to be analysed is larger than the input buffer (currently 300 characters). The statement should be broken down into several simpler statements.

FAULT 102 (Long Analysis Record)

The current statement requires too many compiler recursions in its analysis. The statement should be broken down into several simpler statements.

FAULT 103 (Dictionary Overflow)

FAULT 104 (Too Many Names)

Too many or too long names are currently declared. The remedy is either

1. To use the block structure so that names are undeclared when not required.
or

2. Use shorter names and replace name labels by integer labels.

FAULT 105 (Too Many Levels)

Textual level 10 has been reached.

FAULT 106 (String Constant > 255 Symbols)

This fault will occur if a string constant contains more than 255 symbols. Note that the text in a %PRINTTEXT statement is treated as a string constant.

FAULT 107 (ASL Empty)

The compiler tables are full - this is unlikely to occur and the User should contact the Advisory Service.

FAULT 108 (End Message Character in Program)

The end message character was found in the IMP program. This could be caused by the omission of the %ENDOFPROGRAM statement.

Faults greater than 200 indicate compiler errors which should be reported to the Advisory Service.

SECTION 13 - RUN TIME FAULTS

INTRODUCTION

Various faults can occur during the execution of an IMP program. Some of these are described in the relavent sections of the manual, they are all described below. Some Run Time faults can be trapped. (See Section 14).

TRAPPABLE FAULTS

INTEGER OVERFLOW
REAL OVERFLOW

On the evaluation of an expression a number has been generated which is too large to be contained in a register of the appropriate type.

INVALID %CYCLE

A %CYCLE instruction has been executed where it is impossible to reach the final value of the control variable.

NOT ENOUGH STORE

On executing a declaration, insufficient space is available in the store to accommodate the variables requested.

SQRT NEGATIVE
LOG NEGATIVE

A negative argument has been passed to the appropriate routine.

%SWITCH VARIABLE NOT SET

An instruction of the form -> SW(<EXPRN>) has been executed and the required label cannot be found.

INPUT FILE ENDED

The current file of input data has been exhausted.

NON-INTEGER QUOTIENT

A quotient of two %INTEGER quantities when evaluated in an integer context is found to have a non-integral value.

%RESULT NOT SPECIFIED

An attempt has been made to exit from a %FN or %MAP via the %END instruction. Control can only be returned from these blocks via a %RESULT= statement.

SYMBOL IN DATA S

In executing a read instruction, the non-numeric symbol S has been found.

REAL INSTEAD OF INTEGER IN DATA

A real value has been found on execution of a read instruction which expects an %INTEGER value. This will also occur if an integer of modulus >(2**31-1) is read, or if an exponent is given which is not an integer.

DIVIDE ERROR

A division has been attempted that would cause overflow (usually division by zero).

SUBSTITUTE CHARACTER IN DATA

If a line contains the substitute character (placed in the character stream if an invalid code is read), the above trappable fault occurs on attempting to read the first symbol of the line. (See Section 10).

(GENERAL GRAPH PLOTTER FAULT)

The details are given in the 'ERCC Graph Plotting Reference Manual'.

ILLEGAL EXPONENT

An exponent greater than 255 (or 63 in an integer context) has been found. This restriction is to avoid excessive looping required to complete the evaluation with a probable occurrence of overflow.

TRIG FN INACCURATE

The argument of a SIN, COS or TAN function is so large (>1@7) that the results are no longer guaranteed.

TAN TOO LARGE

The argument passed to the TAN function is so near a multiple of $\pi/2$ that an overflow condition would exist.

EXP TOO LARGE

The argument passed to the EXP function is so large that overflow would be caused by evaluation.

LIBRARY FN FAULT n

This is a general library function fault which may arise from any one of several different function calls.

n=1	The parameter passed to ARCSIN is not in the range -1 to +1
n=2	The parameter passed to ARCCOS is not in the range -1 to +1
n=3	The parameters passed to ARCTAN are both zero.
n=4	The modulus of the parameter passed to HYPSIN is >=172.694.
n=5	The modulus of the parameter passed to HYPCOS is >=172.694
n=10	The value of X**2 or the value of (X**2+Y**2) is greater than the largest real number allowed.

RESOLUTION FAILS

An unconditional resolution cannot be completed as the string to be resolved does not contain the symbols being searched for.

INTPT TOO LARGE

The modulus of the required integer is too large for a fixed point register (2**31-1). Also caused by INT function as this is interpreted as INTPT (<exprn> + 0.5).

ARRAY INSIDEOUT

A dynamic array declaration has been executed in which the lower bound for any dimension exceeds the corresponding upper bound.

CAPACITY EXCEEDED

Too large an entity has been assigned to a %BYTE or %SHORT integer by the = operator. (The <- operator can be used to avoid this test and assign the least significant bits only).

Also caused by %STRING assignments if a string >255 characters is produced or if the LHS of a string assignment is too small.

UNASSIGNED VARIABLE

An attempt is made to use a variable to which no value has been assigned. This can also occur on a %REPEAT if the corresponding %CYCLE has not been executed - in this case, the compiler's copy of increment and final values are not assigned.

ARRAY BOUND FAULT n

An array suffix (n) has been found to be outside the declared bounds.

NON-TRAPPABLE FAULTS

TIME EXCEEDED

The time for this job as given in the Job Control statements, or foreground Command (or by default) has been exhausted. The program may be stuck in an unproductive loop.

OPERATOR TERMINATION

The operator has prematurely terminated the job. A memorandum explaining the reason should be received, unless the 'operator' is the user running the program from an interactive terminal.

OUTPUT EXCEEDED

The available amount of output as specified by the Job Control statements (or by default) has been exceeded.

CORRUPT DOPEVECTOR n (formerly WRONG DIMENSION OF ARRAY)

The dope vector for an array, which is an area of store containing dimension information about the array, is not as expected. It may have been overwritten, for example by incorrect use of mapping functions, or more commonly occurs if the array passed as an %INTEGERARRAYNAME or %REALARRAYNAME has the dimension, n, which is not the dimension expected by the called routine.

ADDRESS ERROR

The address error interrupt has occurred. Likely causes include running a faulty program which has been compiled with the option 'NOARRAY', wrong use of mapping functions, or incorrect declaration of %EXTERNAL routines or functions. The user is advised to check these suggested faults before seeking the assistance of the Advisory Service.

ILLEGAL OPCODE

An illegal instruction has been obeyed. This can be caused by a -> SW(I) instruction when NOARRAY has suppressed the bound checking.

UNEXPLAINED INTERRUPT

Some other interrupt has occurred - contact the Advisory Service.

SECTION 14 - FAULT TRAPPING

INTRODUCTION

In certain cases when a run time fault occurs, it is not desirable for the run to be terminated. For example, if a particular program processes a series of data sets, it may be preferable to continue to the processing of the next data set rather than terminate the whole job in the event of, say, FAULT 5(SQRT NEGATIVE) occurring.

An instruction is provided which enables certain faults to be trapped causing control to be transferred to a preassigned point in the program:

```
%FAULT N -> <LABEL>
```

where N is the number of a trappable fault, see list below, and <LABEL> is a simple label (a %SWITCH label cannot be used). More than one fault can be directed to the same label, and groups of faults directed to separate labels can be combined in one %FAULT statement. for example:

```
%FAULT 1,2,6 ->99, 3,5 -> FAIL
```

means, 'if a fault of type 1,2 or 6 subsequently occurs then go to label 99: and if a fault of type 3 or 5 occurs then go to label FAIL'.

The effect is to preserve all the necessary control data to enable control to revert to this point in the program (and then jump to label 99 or label FAIL) should one of the specified faults occur at some lower (or the same) level.

NOTES

1. %FAULT statements can only appear in the outer block of a program; they cannot appear in %EXTERNAL routines.

2. The label to which the fault refers must also be in the outer block.

3. %FAULT statements can appear at any point in the block after all declarations within the block.

4. More than one %FAULT statement can appear in a program for the same fault. In the following example the fault INPUT ENDED is trapped twice:

```
%BEGIN
    %INTEGER I,J,K
    SELECTINPUT(1)
    %FAULT 9 -> INEND1
    .
    .
INEND1: %FAULT 9 ->INEND2
    SELECTINPUT(2)
    .
    .
INEND2:
    .
```

LIST OF TRAPPABLE FAULTS

Number	Description
1	Integer Overflow
2	Real Overflow
3	Invalid %CYCLE
4	Not Enough Store
5	SQRT Negative
6	LOG Negative
7	%SWITCH Variable Not Set
9	Input Ended
10	Non-integer Quotient
11	%RESULT Not Specified
14	Symbol In Data
16	Real Instead Of Integer In Data
17	Divide Error
18	Substitute Character In Data
19	Graph Plotter Fault
21	Illegal Exponent
22	Trig Function Inaccurate
23	TAN Too Large
24	EXP Too Large
25	Library Function Fault
26	Resolution Fails
27	INTPT Too Large
28	Array Inside Out
30	Capacity Exceeded
31	Unassigned Variable
32	Array Bound Fault

Further information about these faults is given in Section 13.

SECTION 15 - INTERNAL CHARACTER CODE

INTRODUCTION

The IMP internal character code is based on the code for data interchange defined by the International Standards Organisation (ISO). This code currently assigns graphical or control characters to code values 0 - 127. Most of the control characters will only concern users of special hardware such as graphical display devices, but they are listed overleaf for convenience.

NOTES

1. In the case of some characters there is a discrepancy between the graphical representation on different peripherals. For example the ISO character 33 is printed as either exclamation mark or vertical bar. The most common alternatives are given below.

2. Characters followed by an asterisk in the table below are 'marked' during LINE RECONSTRUCTION and hence are ignored by all character input routines other than READ CH. (See Section 10).

3. IMP programmers should rarely need to refer to the code value of a particular character. If a comparison is required the preferred method is to use a character constant. The required character is enclosed in single quotes and the result is a constant having the internal code value of that character. For example if it is required to skip a sequence of letters the following statement could be used:

```
SKIP SYMBOL %WHILE 'A'<=NEXT SYMBOL<='Z'
```

   The built in function NL can be used to avoid writing a NEWLINE character within quotes:

```
%IF I=NL %THEN .....
```

4. It should be noted that in this code upper case letters appear in 26 consecutive locations, as do lower case letters. This fact simplifies sorting and text manipulation. Note also that the numeric characters have values lower than letters. In these and other respects this code differs from some other data exchange codes, e.g. EBCDIC, and programmers involved in converting programs to IMP from other languages should bear these points in mind.

INTERNAL CHARACTER CODE

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	$(π)	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		*

SECTION 16 - ROUTINES, FUNCTIONS AND MAPS IN THE IMP LIBRARY

INTRODUCTION

The routines, functions and maps provided for the IMP programmer are listed in the table below. They are divided into three classes INTRINSIC, IMPLICIT and EXPLICIT. Items in the first two classes can be used without explicit declaration since their names and characteristics are known to the compiler. Items in the EXPLICIT class, however, must be specified before they are used. The specification provides information to the compiler as to the name and parameter list of the routine, function or map. It also causes the compiler to generate an entry in a table to ensure that the necessary module is loaded when the program is run. It is most important to type the specification accurately, in particular the number and types of parameters must be correct (the names of parameters used are not significant). The specification of an item in the EXPLICIT class is preceded by the delimiter %EXTERNAL, hence for the routine WRITE SQ the specification would be:

```
%EXTERNALROUTINESPEC WRITE SQ(%INTEGER I,%NAME A,B)
```

There is one restriction in the use of IMPLICIT routines and functions. Because they are compiled (for efficiency reasons) as part of the program rather than being proper calls to routines compiled seperately, they cannot themselves be passed as parameters to routines. This limitation can be overcome (see Section 6).

The routines etc. listed below are either described in this manual, in which case an appropriate section number is given, or in the Edinburgh IMP/FORTRAN System Library Manual, in which case the reference 'LM' is used.

NAME	TYPE	CLASS	PARAMETERS	REF
ADD MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B,C, %INTEGER I,J	LM
ADDR	%INTEGERFN	INTRINSIC	%INTEGER I	7
ARCCOS	%LONGREALFN	IMPLICIT	%LONGREAL A	
ARCSIN	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
ARCTAN	%LONGREALFN	IMPLICIT	%LONGREAL A,B	LM
ARRAY	%ARRAY MAP	INTRINSIC	%INTEGER I,%ARRAYNAME J	7
BITS	%INTEGERFN	EXPLICIT	%INTEGER I	3
BYTE INTEGER	%BYTEINTEGERMAP	INTRINSIC	%INTEGER I	7
CHARNO	%INTEGERFN	INTRINSIC	%STRINGNAME S,%INTEGER I	8
CLOSE DA	%ROUTINE	EXPLICIT	%INTEGER I	10
CLOSE SQ	%ROUTINE	EXPLICIT	%INTEGER I	10
CLOSE STREAM	%ROUTINE	IMPLICIT	%INTEGER I	10
COPY MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B, %INTEGER I,J	LM
COS	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
CPUTIME	%LONGREALFN	EXPLICIT	NONE	LM
DATE	%STRINGFN	EXPLICIT	NONE	LM
DET	%LONGREALFN	EXPLICIT	%LONGREALARRAYNAME A, %INTEGER I	LM
DIV MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B, %INTEGER I,J, %LONGREALNAME C	LM
ERFN	%LONGREALFN	EXPLICIT	%LONGREAL A,B	

NAME	TYPE	CLASS	PARAMETERS	REF
ERFNC	%LONGREALFN	EXPLICIT	%LONGREAL A,B	LM
EXP	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
EXP TEN	%LONGREALFN	EXPLICIT	%LONGREAL A	LM
FRAC PT	%LONGREALFN	INTRINSIC	%LONGREAL A	2
FROM STRING	%STRINGFN	IMPLICIT	%STRINGNAME S,%INTEGER I,J	8
HYPCOS	%LONGREALFN	EXPLICIT	%LONGREAL A,B	LM
HYPSIN	%LONGREALFN	EXPLICIT	%LONGREAL A,B	LM
HYPTAN	%LONGREALFN	EXPLICIT	%LONGREAL A,B	LM
IFD BINARY	%INTEGERFN	EXPLICIT	%SHORTINTEGERARRAYNAME I, %INTEGER J,K,%INTEGERNAME L	LM
IFD ISO	%INTEGERFN	EXPLICIT	%BYTEINTEGERARRAYNAME I, %INTEGER J,K,%INTEGERNAME L	LM
INT	%INTEGERFN	INTRINSIC	%LONGREAL A	2
INT PT	%INTEGERFN	INTRINSIC	%LONGREAL A	2
INTEGER	%INTEGERMAP	INTRINSIC	%INTEGER I	7
INVERT	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B, %INTEGER I,%LONGREALNAME J	LM
ISO CARD	%ROUTINE	EXPLICIT	%BYTEINTEGERARRAYNAME K	10
LENGTH	%INTEGERFN	INTRINSIC	%STRINGNAME S	8
LOG	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
LOGTEN	%LONGREALFN	EXPLICIT	%LONGREAL A	LM
LONG REAL	%LONGREALMAP	INTRINSIC	%INTEGER I	7
MOD	%LONGREALFN	INTRINSIC	%LONGREAL A	2
MULT MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B,C, %INTEGER I,J,K	LM
MULT TR MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B,C, %INTEGER I,J,K	

NAME	TYPE	CLASS	PARAMETERS	REF
NEWLINE	%ROUTINE	INTRINSIC	NONE	10
NEWLINES	%ROUTINE	INTRINSIC	%INTEGER I	10
NEWPAGE	%ROUTINE	INTRINSIC	NONE	10
NEXT ITEM	%STRINGFN	INTRINSIC	NONE	8
NEXT SYMBOL	%INTEGERFN	INTRINSIC	NONE	10
NL	%INTEGERFN	INTRINSIC	NONE	15
NULL	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A, %INTEGER I,J	LM
OPEN DA	%ROUTINE	EXPLICIT	%INTEGER I	10
OPEN SQ	%ROUTINE	EXPLICIT	%INTEGER I	10
PRINT	%ROUTINE	IMPLICIT	%LONGREAL A,%INTEGER I,J	10
PRINT CH	%ROUTINE	INTRINSIC	%INTEGER I	10
PRINT FL	%ROUTINE	IMPLICIT	%LONGREAL A,%INTEGER I	10
PRINT STRING	%ROUTINE	INTRINSIC	%STRING S	8
PRINT SYMBOL	%ROUTINE	INTRINSIC	%INTEGER I	10
RADIUS	%LONGREALFN	IMPLICIT	%LONGREAL A,B	LM
RANDOM	%REALFN	EXPLICIT	%INTEGERNAME I,%INTEGER K	LM
READ	%ROUTINE	IMPLICIT	%NAME A	10
READ CH	%ROUTINE	INTRINSIC	%NAME I	10
READ DA	%ROUTINE	EXPLICIT	%INTEGER I,%INTEGERNAME J, %NAME K,L	10
READ ITEM	%ROUTINE	INTRINSIC	%STRINGNAME S	8
READ SQ	%ROUTINE	EXPLICIT	%INTEGER I,%NAME J,K	10
READ STRING	%ROUTINE	IMPLICIT	%STRINGNAME S	8
READ SYMBOL	%ROUTINE	INTRINSIC	%INTEGER I	10
REAL	%REALMAP	INTRINSIC	%INTEGER I	7

NAME	TYPE	CLASS	PARAMETERS	REF
RECORD	%RECORDMAP	INTRINSIC	%INTEGER I	9
RFD BINARY	%LONGREALFN	EXPLICIT	%SHORTINTEGERARRAYNAME I, %INTEGER J,K,%INTEGERNAME L	LM
RFD ISO	%LONGREALFN	EXPLICIT	%BYTEINTEGERARRAYNAME I, %INTEGER J,K,%INTEGERNAME L	LM
SELECT INPUT	%ROUTINE	INTRINSIC	%INTEGER I	10
SELECT OUTPUT	%ROUTINE	INTRINSIC	%INTEGER I	10
SET MARGINS	%ROUTINE	IMPLICIT	%INTEGER I,J,K	10
SHIFT C	%INTEGERFN	EXPLICIT	%INTEGER I,J	3
SHORT INTEGER	%SHORTINTEGERMAP	INTRINSIC	%INTEGER I	7
SIN	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
SKIP SYMBOL	%ROUTINE	INTRINSIC	NONE	10
SOLVE LN EQ	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B, %INTEGER I,%LONGREALNAME C	LM
SPACE	%ROUTINE	INTRINSIC	NONE	10
SPACES	%ROUTINE	INTRINSIC	%INTEGER I	10
SQRT	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
STRING	%STRINGMAP	INTRINSIC	%INTEGER I	8
SUB MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B,C, %INTEGER I,J	LM
TAN	%LONGREALFN	IMPLICIT	%LONGREAL A	LM
TIME	%STRINGFN	EXPLICIT	NONE	LM
TOSTRING	%STRINGFN	INTRINSIC	%INTEGER I	8
TRANS MATRIX	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A,B, %INTEGER I,J	LM

NAME	TYPE	CLASS	PARAMETERS	REF
UNIT	%ROUTINE	EXPLICIT	%LONGREALARRAYNAME A, %INTEGER I	LM
WRITE	%ROUTINE	INTRINSIC	%INTEGER I,J	10
WRITE DA	%ROUTINE	EXPLICIT	%INTEGER I,%INTEGERNAME J, %NAME K,L	10
WRITE SQ	%ROUTINE	EXPLICIT	%INTEGER I,%NAME J,K	10

APPENDIX: DIFFERENCES IN 2900 IMP

The following is a list of the differences between the IMP language implemented on ICL 2900 series computers and the language described in the body of this Manual. It should be noted that some of these changes have been incorporated in the System 4 EMAS implementation and, to a lesser extent, in the NUMAC OS implementation. They are expressed here as changes to the Manual description.

1. Abolition of the modulus operator '!...!' coupled with the introduction of a standard integer function IMOD, corresponding to the existing real function MOD, to yield the absolute value of an integer.

   The change removes a source of ambiguity (e.g. meaning of A!!B!!C).

   Examples:
   ```
   I = IMOD(IVALUE)
   ```
   where I and IVALUE are integer variables.
   ```
   R = MOD(VALUE)
   ```
   where R is a real variable and VALUE is real or integer.

2. Abolition of the use of a special symbol for π in favour of a standard long real function PI.

   The special symbol is not found in standard character sets.

   Example:
   ```
   CIR = PI*DIAM
   ```
   where CIR is a real variable and DIAM is real or integer.

3. Re-specification of the standard procedures LENGTH and CHARNO as maps rather than functions.

   The extension permits a number of string-manipulation operations to be effected more neatly.

   Example:
   ```
   %BEGIN
   %STRING (255) S, T
   %INTEGER I
      S = "FIRST LINE
   SECOND LINE";        ! Note the use of "..." (see item 9 below).
      T = S
   !
   ! Below, S is truncated at the first newline.
   !
      %CYCLE I = 1,1,LENGTH(S)
         LENGTH(S) = I-1 %AND %EXIT %IF CHARNO(S,I) = NL
      %REPEAT
   !
   ! Below, each newline character in T is replaced by a space.
   !
      %CYCLE I = 1,1,LENGTH(T)
         CHARNO(T,I) = ' ' %IF CHARNO(T,I) = NL
      %REPEAT
      PRINTSTRING(S);  NEWLINE
      PRINTSTRING(T);  NEWLINE
   %ENDOFPROGRAM
   ```

When this program is executed, the following output is produced by it:

```
FIRST LINE
FIRST LINE SECOND LINE
```

4. Abolition of numeric labels.

Most current programming languages provide only alphabetic labels, in order that labels may be meaningful in the same way as other identifiers. The need to use labels at all in IMP, as compared with Atlas Autocode, is greatly reduced by the availability of such program-structuring facilities as compound conditional statements and loop control clauses.

Example:

```
233: D=B*B-4*A*C; ! Not valid in 2900 IMP.

L233: D=B*B-4*A*C; ! Valid.

LAST PART: D=B*B-4*A*C; ! Valid and more meaningful.
```

5. Restriction of the division operator '/' to yield a result of type real in all cases.

The integer-division operator '//' becomes obligatory where an integer result is intended. The effect of this and items 6, 7 and 9 is to remove all remaining cases of type ambiguity from the language.

Example: `K = I/J` where I, J and K are integer variables, would fail since the right hand side is always a real expression (even when J divides exactly into I).

6. Restriction of the exponentiation operator '**' to yield a result of type real in all cases, coupled with the introduction of an integer-exponentiation operator '****'.

The case of exponentiation becomes exactly parallel to the case of division.

The reverse slant, \ for real and \\ for integer, is an alternative notation for these operators.

Examples: `K = I****J` or `K = I\\J` where I, J and K are integers, is valid.

`K = I**J` where I, J and K are integers, would fail, since the right hand side is a real expression.

7. Restriction of the form of integer constants to disallow the inclusion of decimal points and exponent symbols.

A constant form like 0.1@1 is technically an integer in System 4 EMAS and NUMAC OS IMP, because it has an integral value. Such forms are not allowed in 2900 IMP.

8. Abolition of double quote deletion (page 1.1, item 8).

9. Introduction of a distinctive quote symbol for strings (double-quote in place of single-quote).

   The main effect is to make it possible to distinguish a single character string from a character constant.

   Examples:

   ```
   ASTRING = "THIS IS A STRING"    where ASTRING is a string variable.
   ASYMBOL = 'A'                   where ASYMBOL is an integer variable.
   ```

10. Extension of the set of ranges of integer to include %LONGINTEGER (64 bits).

    Long integers were implemented to allow the easy handling of a systems software feature of 2900 machines, rather than for manipulating very large numbers. However, most long integer arithmetic will give the expected results.

    The following characteristics of long integers should be noted:

    * A long integer can be assigned a decimal value only in the range ± 214783647 ($\pm 2^{31} - 1$). If an attempt is made to assign to a long integer a value outside this range, the fault

      ```
      REAL IN INTEGER EXPRESSION
      ```

      will occur. A larger number may however be stored in a long integer by using an explicit hexadecimal or binary pattern to represent the number. For example, if L is a long integer variable,

      ```
          L = X'1FFFFFFFFFFFFFFF'      (16 hex digits) will succeed,
      but L = 999999999999999          (15 decimal digits) will fail.
      ```

    * Input/Output: long integer values (i.e. in the range $\pm 2^{63} - 1$) may be read into long integer variables using the input routine READ. They may be printed out using the output routine PRINT, with the third parameter set to 0. However, numbers outside the range $\pm 10^{15}$ are floated and printed in exponential form.

    * A variable of type %LONGINTEGER can be shifted by up to 63 bits left or right, using the << and >> operators. However, an attempt to shift either way 64 bits will have no effect (it might be expected to clear a long integer to 0).

    * In general, integer arithmetic is done with a precision of 32 bits if all the operands are of 32-bit length. This has some implications when long integer variables are used. For example:

      ```
      %INTEGER I,J
      %LONGINTEGER K
      K = (I<<32)!J; ! Effect is K = J, since 32-bit working is used.
      K = I; K =(K<<32)!J; ! This assigns K as intended.
      ```

* If a long integer is set to a negative integer (32-bit) value, the sign bit is propagated through the long integer, resulting in a negative long integer. Thus

```
L = X'FFFFFFFF'
```

where L is a long integer, sets X'FFFFFFFFFFFFFFFF' (the value -1) in L and not X'00000000FFFFFFFF'. The latter value may be set by including the leading zeros in the assignment statement.

11. Changes to the set of ranges of reals.

The 2900 implementation offers three precisions for reals: 32-bit (somewhat inefficient), 64-bit and 128-bit. These are specified as %REAL, %LONGREAL and %LONGLONGREAL, respectively. Working is either in 64 bits or in 128 bits. Thus type %REAL is only effective in storing values in scalars and arrays: a variable of type %REAL will have its value lengthened when used in any calculation.

12. Additions to the built-in mapping functions:

```
%LONGINTEGERMAP LONGINTEGER(%INTEGER ADDRESS)

%LONGLONGREALMAP LONGLONGREAL(%INTEGER ADDRESS)
```

13. Abolition of type %SHORTINTEGER.

14. Abolition of the functions IFD BINARY and RFD BINARY.

15. Provision of functions for changing lengths of reals and of integers:

LENGTHENI(K)	Parameter of type %INTEGER, result of type %LONGINTEGER.
SHORTENI(K)	Parameter of type %LONGINTEGER, result of type %INTEGER.
LENGTHENR(S)	Parameter of type %LONGREAL, result of type %LONGLONGREAL.
SHORTENR(S)	Parameter of type %LONGLONGREAL, result of type %LONGREAL.

These functions need only be used when it is desired to change the length of an operand in an expression.

16. Change in the interpretation of grave sign (`).

The grave sign stands for itself; i.e. it is not mapped onto @ on input.

17. Abolition of the %MONITORSTOP statement.

It is now necessary to give the two statements %MONITOR; %STOP.

18. Abolition of the %PRINTTEXT statement.

19. Abolition of implied multiplication.

In System 4 EMAS and NUMAC OS IMP, constructions such as 23A(6) are allowed, i.e. constant followed, without an asterisk, by a variable or array element. This is not allowed in 2900 IMP, and would have to be written as 23*A(6).

20. Abolition of fault trapping in favour of an event mechanism.

It has long been agreed that the structure of fault trapping in System 4 EMAS and NUMAC OS IMP is inadequate, especially in respect of the forced return to the main program and the inability to trap faults in external routines. The architecture of the 2900 Series has enabled a more structured approach to be adopted and the following notes describe it.

The term %EVENT is introduced to describe the class of conditions which may be detected or signalled. It is broader than the %FAULT concept, in that it may include user-defined events as well as the conventional 'faults', such as division by 0.

The following list groups certain faults to provide a set of events, of which 1-10 are predefined and events 11-14 may be user defined.

Event Classes

No.	Name	Fault Numbers
1	Arithmetic overflow	1,2,17,22,23,24,27
2	Excess store (or other resource)	4
3	Substitute character in data	18
4	Invalid data	14
5	Invalid arguments	3,5,6,7,21,28
6	Out of range	30,32
7	Resolution failure	26
8	Unassigned variable	31
9	Input ended	9
10	Library subroutine error	25
*11-14	GENERAL PURPOSE	

*11 is also used by the Graph Package (EMAS/NUMAC fault 19)

Within an event class, the individual faults are assigned different sub-event numbers. The faults are thus categorised as follows:

Fault Number	Description	Event/Sub-event no.
1	Integer Overflow	1/1
2	Real Overflow	1/2
3	Invalid %CYCLE	5/1
4	Not Enough Store	2/1
5	SQRT Negative	5/2
6	LOG Negative	5/3
7	%SWITCH Variable Not Set	5/4
9	Input Ended	9/1

Fault Number	Description	Event/Sub-event no.
10	Non-integer Quotient	*
11	%RESULT Not Specified	*
14	Symbol in Data	4/1
16	Real Instead of Integer in Data	*
17	Divide Error	1/3
18	Substitute Character in Data	3/1
19	Graph Plotter Fault	11/n
21	Illegal Exponent	5/5
22	Trig Function Inaccurate	1/4
23	TAN Too Large	1/5
24	EXP Too Large	1/6
25	Library Function Fault	10/n
26	Resolution Fails	7/1
27	INTPT Too Large	1/7
28	Array Inside Out	5/6
30	Capacity Exceeded	6/1
31	Unassigned Variable	8/1
32	Array Bound Fault	6/2

The following System 4 EMAS/NUMAC OS IMP faults are omitted from the event classes:

* Non-integer quotient (10) - redundant in 2900 IMP.

* Result not specified (11) - redundant because of a compile-time check.

* Real instead of integer in data (16) - in 2900 IMP an integer 'read' terminates on a '.' sign or '@' sign, in addition to its EMAS/NUMAC definition.

The syntactic structure is:

```
%ON %EVENT nlist %START      e.g. %ON %EVENT 1,7,12 %START
   .                                 .
   .                                 .
   .                                 .
%FINISH                          %FINISH
```

This structure must follow the declarations at the head of a block (routine) and may be regarded as the last declaration of the block. The code within the %START .. %FINISH is not executed on entry through the head of the block but is jumped to should an event which is contained within the list occur. The flow of control then depends on the contents of the %START .. %FINISH.

An event may be forced by the unconditional instruction:

```
%SIGNAL %EVENT n, exprn
```

where n is the event required and 'exprn' is an optional integer expression which may be used to specify sub-event information. n must be given as a constant, and 'exprn' must yield an integer in the range 0-255.

%SIGNAL %EVENT statements are the only way of causing user-defined events to occur, although they can also be used with the predefined events (1-10).

If an event is forced by a %SIGNAL %EVENT statement in an %ON %EVENT %START/%FINISH block which includes the occurring event in its event list, a branch is not made to the head of that block, since such a branch would probably cause looping. Instead the event is traced up the stack through each superior block until either a suitable %ON %EVENT statement is found or the user environment is left.

In parallel with these language statements two implicit integer functions are introduced which enable the programmer to determine further information when an event occurs. They may only be meaningfully called in a block which has an %ON %EVENT statement within it:

%INTEGERFN EVENT INF

returns (event no<<8)!sub-event no for the last event which has occurred. Fault 16 occurs at compile time if the function is called in a block with no %ON %EVENT statement, and an unassigned variable will result at run time if no event has in fact occurred when the function is called.

%INTEGERFN EVENT LINE

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 unassigned variable will result.

If an event is not trapped 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, the diagnostic package being entered in the latter case. When a suitable %ON %EVENT statement is encountered in a superior block, program control is transferred to its %START/%FINISH block.

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.

Two examples of the use of the event mechanism are given on the next page.

System defined events

```
%INTEGER SUBCLASS, EVENTNO
%CONSTSTRING(21)%ARRAY MESSAGE(1:2)="CAPACITY EXCEEDED",
                                    "ARRAY BOUNDS EXCEEDED"
%ON %EVENT 6 %START
    SUBCLASS = EVENT INF&X'FF'
    EVENTNO = EVENT INF>>8 & 15
    %IF 1<=SUBCLASS<=2 %THEN PRINTSTRING(MESSAGE(SUBCLASS)) %ELSE %C
                             PRINTSTRING("INVALID SUBCLASS")
    NEWLINE
    ->ERROR EXIT
%FINISH
  .
  .
ERROR EXIT: .....
```

User defined events

```
%INTEGER EVENT, SUBEVENT
%ON %EVENT 12 %START
    PRINTSTRING("EVENT 12 HAS BEEN TRAPPED"); NEWLINE
    ->EVENT 12
%FINISH
  .
  .
SUBEVENT = 2
  .
  .
%SIGNAL %EVENT 12, SUBEVENT
  .
  .
EVENT 12: .....
```

COMPATIBILITY OF SYSTEM 4 EMAS IMP WITH 2900 IMP

The IMP compiler on System 4 EMAS will issue a warning for each of the first 30 statements in any compilation that are acceptable to System 4 IMP but not to 2900 IMP.

The IMP compiler on System 4 EMAS will accept an %ON %EVENT statement as described above, provided it occurs in a main program block only. The statement is not compiled but is transposed into %FAULT statements which are compiled. This arrangement enables the majority of programs that use fault trapping to be written in a way that is acceptable to both System 4 IMP and 2900 IMP.

The System 4 compiler, however, does not accept the %SIGNAL %EVENT statement, or calls on EVENT LINE or EVENT INF.

ADDRESSING ON 2900 SERIES MACHINES

On 2900 series machines, all the 32 bits of the address field are significant. Thus a 'negative' address is not necessarily invalid.

WRITING PORTABLE IMP PROGRAMS FOR EMAS AND ICL 2900 SERIES MACHINES

For several years many users have found it useful to transfer IMP programs between EMAS and IBM 360 series machines. By this means they have been able to exploit the convenient program development environment provided by EMAS and the powerful batch job environment provided on our local 370 and more recently at NUMAC.

While only batch facilities are available on the ICL 2980 at the Bush Estate, it is likely that some users will have a requirement to write IMP programs which can be run without change at NUMAC, on EMAS and on the 2980.

This note indicates areas of difference between the IMP compilers on the three machines. These areas must be avoided if compatible IMP programs are to be written for running unchanged on more than one machine. Note that, in general, the NUMAC and EMAS compilers are the same (the two exceptions are noted in the text).

Constants

* Integer constants

  Integer constants which contain decimal points or exponents must be avoided.

* Use of π

  The read-only variable PI should be used. The use of the ISO currency symbol for this purpose must be avoided.

* Strings

  Both single quote (') and double quote (") are permissible string constant delimiters. At some date, to be announced, only double quote string delimiters will be accepted.

  N.B. ONLY single quotes are allowed in NUMAC IMP.

  %PRINTTEXT must not be used.

Arithmetic operators

* Division operators '/' and '//'

  The use of '/' must be restricted to expressions which yield a real result. '//' should be used in all cases where the result must be of type %INTEGER, e.g. where the result is being assigned to an integer variable.

* Multiplication

  Implied multiplication must be avoided.

* Exponentiation operator '**'

The '**' operator gives a result of type %REAL. If an integer result is required then INT can be used; for example, K = INT(I**J).

* Modulus functions

The intrinsic functions MOD (for reals) and IMOD (for integers) must always be used.

Variable types

The following variable types can be used:

%BYTEINTEGER	%LONGREAL
%INTEGER	%STRING
%REAL	%RECORD

Labels

Labels must be standard IMP names, i.e. the first character must be a letter; e.g. L100:

Control statements

* %MONITORSTOP

This must be coded as two separate statements: %MONITOR; %STOP

* Fault trapping

In implementing IMP on the 2900 series the opportunity has been taken to redesign the fault trapping mechanism, in the light of frequent complaints of inadequacy of the %FAULT facility. In order to be completely portable, programs must not use fault trapping.

N.B. The EMAS compiler does allow limited use of the 2900 %ON %EVENT statement by translating it into a standard EMAS IMP %FAULT statement. The following restrictions apply:

a. The %ON %EVENT statement may only appear in the outermost block of a main program.
b. Only a simple jump statement may be placed within the
%ON %EVENT ..... %START; ....; %FINISH
block.

For example:

```
%ON %EVENT 9 %START
    -> INPUT END
%FINISH
```

Conclusion

It is hoped that users will keep these suggestions in mind when writing programs which are likely to have a lifetime of more than a few months. By so doing they will minimise the work needed to transfer them to a 2900 series machine should this become necessary or desirable.

M.D. Brown
R.R. McLeod

September 1977

accuracy
  of real arithmetic 2.3
actual parameters 6.3
addition 2.1
ADDRESS ERROR 13.4
alignment 7.3
  of record sub-fields 9.1
%AND 4.2
arithmetic
  assignment 2.4
  division 2.2
  expressions 2.2
  operations 2.1
  operators 2.1
  variables 1.3
array bound checking 11.1
array bound fault 1.6
ARRAY BOUND FAULT 13.3
ARRAY INSIDEOUT 13.3
array mapping 7.3
ARRAY 7.4,11.1
%ARRAYFORMAT 7.4
arrays
  declaration of 1.6
  dynamic 1.6
  %RECORD 9.2
  %STRING 8.3
  with variable bounds 1.6
assignment
  of arithmetic expressions 2.4
  of logical expressions 3.1
  of records 9.6
  of strings 8.6
  of symbols 2.5
  to pointer variables 7.3

%BEGIN 5.4
binary files 10.8
bit patterns 3.2
BITS 3.5
block structure 5.2,5.4
built in maps 7.2
%BYTEINTEGER 1.3
BYTEINTEGER 7.2
%BYTEINTEGERMAP 7.1

calling routines 6.1
CAPACITY EXCEEDED 13.3
  in arithmetic operations 2.4
  in logical operations 3.2
  in string assignment 8.5
  in string expressions 8.5
  in string functions 8.4
  using strings 8.10
character
  code 10.1,15.1
  input routines 10.2
  marked 10.4,15.1
  output routines 10.5
  streams 10.2
  SUB 10.4
CHARNO 8.10
CHECK 11.1
CLOSE STREAM 10.7
CLOSEDA 10.10
CLOSESQ 10.8
comments 1.2
compiler options 11.1
concatenation of strings 8.5
condition
  string 8.8
conditional repetition 4.4
conditional routine calls 6.2
conditional string resolution 8.9
conditions 4.1
%CONST
  records 9.2
  string variables 8.3
%CONST variables
  declaration of 1.7
constants 1.4
  binary 1.5
  decimal 1.4
  hexadecimal 1.5
  quotes in 1.4
  spaces and newlines in 1.4
  string 8.3
  symbol 1.4
CORRUPT DOPE VECTOR 13.4
cycles 4.3
  conditional 4.4
  with control variables 4.5

declaration
  location of 5.3
  of arrays 1.6
  of routines and functions 6.1
  of variables 1.6
delimiters 1.2
  rules for typing 1.1
DIAG 11.1
diagnostics 11.1
DIAGS OUTPUT EXCEEDED 11.2
direct access files 10.10
DIVIDE ERROR 13.2
division
  in integer expressions 2.2

in real expressions 2.2
double quote deletion 1.1
double sided conditions 4.2
dynamic arrays 5.4

EBCDIC 15.1
efficiency 11.3
%ELSE 4.3
%END 5.4,6.1
%ENDOFFILE 6.13
%ENDOFLIST 11.2
%ENDOFPROGRAM 5.3
exclusive or 3.4
%EXIT 4.6
EXP TOO LARGE 13.2
exponentiation 2.1
in integer expressions 2.2
in real expressions 2.3
%EXTERNAL
records 9.2
string variables 8.3
%EXTERNAL routines 6.13
%EXTERNAL variables 6.14
declaration of 1.7
%EXTERNALROUTINESPEC 6.13,16.1
%EXTRINSIC
records 9.2
%EXTRINSIC variables 6.14

%FAULT 14.1
faults
compile time 12.1
run time 13.1
trappable 14.2
trapping 13.1,14.1
files
character 10.1
direct access 10.10
library 6.13
sequential 10.8
%FINISH 4.2
formal parameters 6.3
FRACPT 2.4
FROMSTRING 8.10
functions 6.1
in system library 16.1
string 8.4

global variables 6.1
GRAPH PLOTTER FAULT 13.2

%IF 4.2
ILLEGAL EXPONENT 13.2
ILLEGAL OPCODE 13.4
implicit routines 6.10
implied multiplication 2.1
inclusive or 3.4
indefinite cycle 4.6
initialisation
of const% variables 1.7
of %OWN variables 1.7
of strings 8.3
INPUT ENDED 10.4,10.9,13.1
input/output 10.1
INT 2.3
integer division 2.2
integer expressions 2.2
INTEGER OVERFLOW 13.1
%INTEGER 1.3
INTEGER 7.2
%INTEGERMAP 7.1
INTPT TOO LARGE 13.3
INTPT 2.3
intrinsic routines 6.10
INVALID CYCLE 4.5,13.1
ISO 15.1
ISOCARD 10.3

jumps 4.7

labels 4.7
length
of record sub-fields 9.7
of variables 1.3
LENGTH 8.10
LENGTHSQ 10.9
library 6.10
files 6.13
list of contents 15.1
LIBRARY FN FAULT 13.2
line number 11.1
line reconstruction 10.4
%LIST 11.2
LIST 11.2
listing of programs 11.2
LOG NEGATIVE 13.1
logical
and 3.1,3.4
assignments 3.1
channels 10.1
not 3.1,3.3
operations 3.1
operators 3.1
or 3.1,3.4
%LONGREAL 1.3
%LONGREALMAP 7.1

mapping
of arithmetic variables 7.1
of records 9.4
of strings 7.1,8.4
maps 7.1
in system library 16.1

margins
input 10.4,10.7
output 10.7
marked character 10.4
MOD 2.5
modulus 2.5
modulus signs 2.5
%MONITOR 11.1
%MONITORSTOP 4.8,11.1
multiplication 2.1

names 1.2
of %EXTERNAL entities 6.14
scope of 5.3
used for labels 4.7
nesting
of blocks 5.3
of conditions 4.6
of cycles 4.6
NEWLINE 10.5
NEWPAGE 10.5
NEXT ITEM 8.11
NEXTSYMBOL 10.2
NOARRAY 11.1
NOCHECK 11.1
NOLIST 11.2
NON INTEGER QUOTIENT 13.1
NOT ENOUGH STORE 13.1
NOTRACE 11.1

OPENDA 10.10
OPENSQ 10.8
OPERATOR TERMINATION 13.4
OPT 11.4
%OR 4.2
OUTPUT EXCEEDED 10.6,13.4
%OWN
records 9.2
string variables 8.3
%OWN arrays 1.7
%OWN variables
declaration of 1.7
in routines 6.11
initialisation of 1.7

parameters 6.3
actual 6.3
formal 6.3
of type %RECORDNAME 9.6
to maps 7.2
pointer variables 7.3
of type %RECORD 9.3
precedence
of arithmetic operators 2.1
of logical operators 3.4
PRINT 10.6
PRINTCH 10.6
PRINTFL 10.6
PRINTSTRING 8.11,10.5
PRINTSYMBOL 10.5
%PRINTTEXT 10.5
programs
continuation of statements 1.1
efficiency of 11.3
listing of 11.3
testing 11.4
typing conventions 1.1

READ ITEM 8.11
READ STRING 8.11
READ 10.3
READCH 10.4
READDA 10.10
READSQ 10.9
READSTRING 10.3
READSYMBOL 10.2
real
expressions 2.3
precision of variables 1.3,2.3
REAL INSTEAD OF INTEGER IN DATA 10.3,13.2
REAL OVERFLOW 13.1
%REAL 1.3
REAL 7.2
%REALMAP 7.1
%REALSLONG 1.7
%REALSNORMAL 1.7
%RECORD 9.1
arrays 9.2
%RECORDFORMAT 9.1
%RECORDMAP 7.1
%RECORDNAME 9.5
records
sub-fields of 9.1
%RECORDSPEC 9.5
recursive routines 6.11
%REPEAT 4.3
resolution
conditional 8.9
of strings 8.6
RESOLUTION FAILS 13.2
RESULT NOT SPECIFIED 13.1
%RESULT 6.8
of maps 7.2
%RETURN 6.2
routines 6.1
explicit 16.1
%EXTERNAL 6.13
implicit 6.10,16.1
in system library 16.1
intrinsic 6.10,16.1
recursive 6.11
with parameters 6.3
%ROUTINESPEC 6.1

scope of names 5.3

SELECTINPUT 10.2
SELECTOUTPUT 10.2
semantic errors 12.2
sequential files 10.8
SET MARGINS 10.7
shift operators 3.1,3.2
SHIFTC 3.3
%SHORTINTEGER 1.3
SHORTINTEGER 7.2
%SHORTINTEGERMAP 7.1
SKIPSYMBOL 10.3
SPACE 10.5
%SPEC 6.1
SQRT NEGATIVE 13.1
stack 5.1
stack pointer 5.1
%START 4.2
%STOP 4.8
storage allocation 5.1
store map functions 7.1
store mapping 7.1
string resolution 8.6
%STRING
  conditional resolution 8.9
  constants 8.3
STRING 7.2,8.4
%STRINGMAP 7.1
strings 8.1
sub-expressions 2.4
sub-fields 9.1
  of type %RECORD 9.4
  of type %RECORDNAME 9.5
SUB 10.4
SUBSTITUTE CHARACTER IN DATA 10.4,13.2
subtraction 2.1
SWITCH VECTOR NOT SET 13.1
switch vectors 4.7
%SWITCH 4.7
SYMBOL IN DATA 10.3,13.1
syntactic errors 12.1

TAN TOO LARGE 13.2
%THEN 4.2
TIME EXCEEDED 13.4
TO STRING 8.10
TRACE 11.1
trappable faults 14.2
TRIG FN INACCURATE 13.2

unassigned checking 11.1
UNASSIGNED VARIABLE 13.3
UNEXPLAINED INTERRUPT 13.4
%UNLESS 4.3
%UNTIL 4.3

variables 1.3
  arithmetic 1.3
  %CONST 1.7
  declaration of 1.6
  %EXTERNAL 6.14
  %EXTRINSIC 6.14
  %OWN 1.7
  pointer 7.3
  string 8.3

%WHILE 4.3
WRITE 10.6
WRITEDA 10.10
WRITESQ 10.9

UNIVERSITY OF EDINBURGH

Edinburgh Regional Computing Centre

IMP Language Manual Update 1

Originator	Contact	Issue date
Mrs L. Brewin	MrJ. Murison	Feburary, 1978

The second edition of the IMP Language Manual was published in May 1974 and until now there have been no updates. It is still largely accurate for the System 4 EMAS and NUMAC OS implementations of IMP, but the implementation on ICL 2900 series machines contains a number of important departures from what is described in the Manual. It has therefore been decided to issue an Appendix detailing these differences (attached); it should be inserted between Section 16 and the Index.

A sheet dated September 1977 is also attached: it gives advice on writing IMP programs to be compatible with NUMAC OS IMP, System 4 EMAS IMP and 2900 IMP.

It is recommended that this cover note be filed at the end of the Manual, as a record that it has been updated.