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

October 1981

IMP80 on EMAS 2900: Differences from IMP9

Contents

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 identifier Item 1a is not distinguished from identifier 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 onto a new line by terminating the first part 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.)

   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 or special symbols).

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

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

8. own initialisation

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 second statement above should have been given as

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

which is unambiguous, whatever version of IMP is used.

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.

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=' '
```

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

```
statement until condition
```

d) for cycles: the cycle variable must be of type integer; it should not be changed explicitly within the cycle body; the cycle body is executed (final-init)//inc + 1 times or zero times, whichever is the greater; if the cycle is not executed the cycle variable is not set; (final-init) must be exactly divisible by inc; the simple form 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 constant 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.

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 translations are recommended:

```
        if S -> (B).C then ...                          in IMP9
becomes if S -> A.(B).C and A="" then ...               in IMP80
     [A is a new string (255) variable]

and     if S -> (B).S then ...                          in IMP9
becomes if S -> A.(B).C and A="" then S = C and ... in IMP80
     [A and C are new string (255) variables]
```

14. Records

a) The syntax of declarations in IMP80 differ from that 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 or is 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.

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

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

```
routinespec NAME1(record name NAME2, ...)
```

must now be converted to

```
routinespec 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. Note that redundant brackets round alternatives are not allowed.

c) Records can contain records. The format of such a record must have already been defined, or be explicit.

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

d) 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
```

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