EMAS 2900 IMP NOTE

No: 1
Date: 10/11/78

IMP Call Sequences

The pre-call sequence is first to odd-align the stack front and then to store the current LNB at SF (this SF to be the new LNB, i.e. in the called routine). The stack front is further raised by 4 words for the link descriptor (placed in old LNB+1,2 by the CALL) and for the called routine's PLT descriptor, to be placed in LNB+3,4 by the post-call sequence. The purpose of odd-aligning the stack front is so that the link and PLT descriptors are double word aligned. This is all done by the single instruction

```
          PRCL    4
```

Next the parameters are stacked (if required) and LNB is raised by 5 plus the number of parameter words to point to the new display (starting at the stack front as it was after the odd-aligning but before the storing of LNB).

Programmers should note the effect of the pre-call alignment when designing routine headings. For example

routine ONE(integer A, integername B)

results in the descriptor to B being across a double-word boundary and requires 2 core cycles to fetch it.

routine TWO(integername B, integer A)

is more efficient.

XNB is set to the called routine's textually surrounding routine for internal calls.

Finally the CALL is executed, with operand either

(i) for an internal routine: N	where N is offset in halfwords from current PC
(ii) for an external routine: @(XNB+N)	where XNB+N has a descriptor descriptor which points to the code descriptor in the called routine's PLT.

The PLT is conceptually an array of code descriptors for the routines in the module. An external reference is a descriptor descriptor to one of these (i.e. a descriptor which points to one of these). The loader fills in the addresses appropriately.

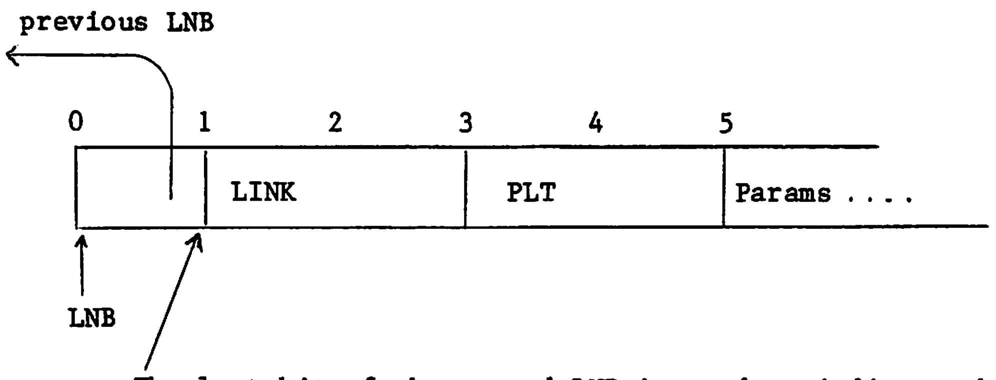
To summarise, the CALL instruction loads DR with the operand descriptor (if the operand is indirect). It also places a link descriptor at old LNB+1,2. LNB has been set up by the pre-call sequence to point to the new display.

For external routines, the post-call sequence (in the called routine) is to store DR, which has a descriptor to the descriptor in the called routine's PLT, at LNB+3,4, having used INCA to set the address in DR back to the start of PLT. That is, LNB+3,4 have a descriptor to this routine's PLT.

So far,

INCA	-N	where N is the offset of this routine's descriptor from the start of PLT (omitted if N=0).
LDB	0	set bound to zero now, but later set to offset from CST of ON and diagnostic block patched (ORed) in at end of routine.
STD	(LNB+3)	and 4.

LNB+4 has the address therefore of PLT (GLA), and this is loaded to XNB subsequently as required (e.g. for referencing <u>own</u> variables and for making further copies).



The last bit of the stored LNB is used to indicate that SF was increased by 1 in the pre-call sequence - it should be ignored when chaining back in dumps, etc.

P.D. Stephens

EMAS 2900 IMP NOTE

No: 2
Date: 28/8/78

%NAME Parameters in IMP

The general parameter %NAME allows any entity to be passed to the called routine. This facility is used in the sequential and direct access routines but is not a user facility. It is possible to find out quite a lot about the actual parameter by examining the descriptor.

For variables as actual parameters the descriptor is a scaled vector descriptor with the address field pointing at the leftmost byte of the actual parameter. The size of the item can be obtained from the size code of the descriptor, as follows:

3 = byte
4 = half (care needed with this - not supported by hardware)
5 = integer or real (32-bit)
6 = long integer or long real (64-bit)
7 = long long real (128-bit)

The bound field gives the type as follows:

1 = integer
2 = real
3 = Boolean (ALGOL(E) only)

For structures (i.e. strings and records) an unscaled byte descriptor is passed with the bound field containing the size of the entity in bytes.

Having decoded the descriptor, the program can store items back into the variable using IMP mapping functions; i.e.

```
BYTEINTEGER(ADDR(name)) = ...

STRING(ADDR(name)) = ...
```

as appropriate.

P.D. Stephens

No: 4
Date: 20/3/79

Interactive Debugging in IMP

DEBUG is an interactive debugging aid for IMP programs on EMAS 2900. It includes commands for examining and modifying data and hence changing the course of a computation, for setting breakpoints, and for the conditional execution of DEBUG commands. The breakpoint scheme makes it possible for the user to gain control during program execution and then specify any further action he wishes. The conditional execution facility allows DEBUG commands specified by the user to be stored, and executed only when certain conditions related to the running program are satisfied, e.g. when a given line number in the program has been reached.

In general, the intention has been to allow the user to intervene in the execution of the running program in a way consistent with the effect that would be produced by additional source statements in the program. Hence it is only possible to examine or change variables which are in scope when the DEBUG command is executed. However conditional commands may be issued at any time regardless of whether any variables they refer to are currently in scope.

Establish access to DEBUG by first typing:

```
Command:OPTION(SEARCHDIR=CONLIB.GENERAL)
```

Then compile the program with PARM(DEBUG) set. Additional parms may be set with the exception of NOTRACE, NODIAG and OPT.

When the program is run, DEBUG will print the current line number and then prompt for a command:

```
DEBUG n: (where n is the index number of the command - see Kill)
```

The user can then input one or more DEBUG commands. Commands take two forms, simple and conditional. Simple commands, which consist of a letter followed, if appropriate, by a parameter, are executed immediately and then discarded. Conditional commands are stored, and only executed when the specified condition is satisfied. The program will only resume execution when the command R (see below) is given.

Conditional commands take the form:

```
<simple command> .IF <conditional expr>
```

or, for typing convenience,

```
<simple command> @ <conditional expr>
```

The following simple commands are available:

A <variable>=<value>	- Assign the value to the variable; it is not possible to assign a value to an unequivalenced name type variable.

B	-	Set a breakpoint; when executed, accept a DEBUG command from the console. B is only used as part of a conditional command.
D <addr>,<length>,<format>	-	Dump an area of the virtual memory of <length> bytes, starting at <addr> to the console. <format> may be C for a character dump or H for a hexadecimal dump. The user must have determined the relevant virtual memory addresses prior to the execution of the program.
F	-	Print a map of the object file, indicating the starting line numbers of the routines and blocks it contains.
H	-	Halt execution and return immediately to command level.
I	-	Ignore all stored conditional commands and continue execution of the program in non-debug mode.
K <number>	-	Kill the specified stored commands. The parameter can include several command index numbers or ranges of command index numbers separated by commas, e.g. K 2,4,6-8 Note that the command index number n is given in the prompt "DEBUG n:" current when the command was input.
M	-	Execute %MONITOR.
P <variable>	-	Print the value of the named variable.
R	-	Resume execution of the program. DEBUG will force a break at the next line unless at least one conditional command is currently stored.
S <line no>,<count>	-	Print <count> lines of the source listing starting at line <line no>. A prompt is issued for the name of the compiler generated listing file.

Three types of conditional expression are available; they may be used singly or in combination:

```
     <simple command> @ <conditional expr>

 or  <simple command> @ <conditional expr>&<conditional expr>

 or  <simple command> @ <conditional expr>&<conditional expr>&<conditional
                                                                     expr>
```

Only one of each type can be specified in a single command. The three types are as follows:

C=<variable>	- This is satisfied whenever the specified variable has its value changed. If the variable is reassigned its current value, this is not detected as a change.
R=<routine/block>	- This is satisfied when the current line being executed lies within the specified routine or block. For blocks, the starting line number should be given; for routines, functions or maps, the name should be given. Note that this condition is not satisfied when the current line lies within a routine or block which itself lies within the specified routine or block.
L=<line spec>	- This is satisfied when the current line being executed lies within the line spec. The line spec is defined below.

```
<line spec>      :    <line>[,<line spec>]

<line>           :    <line expr> ! <line range>

<line range>     :    <line no> - <line no>

<line expr>      :    +<line no> ! -<line no> ! *<line no> ! #<line no>
```

The use of these is explained below:

L=n	- Satisfied when the current line number is n.
L=n-m	- Satisfied whenever the current line number lies between n and m.
L=+n	- Satisfied when the line number equals the current line number plus n (i.e. the line number current when the command was given + n)
L=-n	- Satisfied when the line number equals the current line number minus n.

Commands involving these expressions are permanently stored. However two temporary forms are provided which cancel themselves once satisfied:

L=*n	- Satisfied when the current line is encountered for the n'th time. This is intended for skipping through an iterative sequence without stopping at every iteration.
L=#n	- Satisfied when the n'th executable line after the current line is encountered. This allows control to be returned to the user after a test and branch operation, the result of which is not known in advance.

Examples

Several examples of command lines follow:

DEBUG 3:P VAL @ C=VAL & R=COMBINE	- Print variable VAL, whenever its value changes within routine COMBINE.
DEBUG 1:A S="HEADING"	- Assign "HEADING" to string variable S.
DEBUG 4:B @ L=10-20 & C=JIM	- Break whenever the value of JIM changes and the current line lies between 10 and 20.
DEBUG 8:R	- Resume execution of the program.

Notes

1. A newline terminates a command line unless it is part of a string (i.e. preceded by a quote). Spaces are not significant unless enclosed in quotes.

2. The index number n printed in the prompt DEBUG n: is incremented only if a storable (i.e. conditional) command is specified. When commands are Killed, the index number is unchanged except for the case in which all commands are Killed, when its value returns to 1.

3. If more than one C=<variable> commands are given for the same variable, then the same R=<routine> expression (if used at all) must be used in every case. This ensures that a single copy of the variable's value is held and hence avoids inconsistencies.

4. All the simple commands, with the exception of K (kill) and F (file map), may be used with a conditional expression to form a conditional command.

5. Commands B (break) and R (resume) operate by setting a single switch. Hence if a B conditional command succeeds but is followed by an R conditional command which also succeeds, the effect of the B is lost and no break occurs.

6. Except in line specification, a number may either be expressed in decimal or hexadecimal form, e.g. 32 or X20.

7. At present, users should only DEBUG one object file at a time. Thus a program and an external routine which it calls must not both be compiled with PARM DEBUG.

8. When a command such as B @ C=JIM is given, the condition is evaluated every time a line of the program is executed. In this case the evaluation involves a search for a declaration of the variable JIM; if it is not found, this is treated as a fault. In addition if there is more than one declaration of JIM in a program (or in the line range specified), the first accessed is used subsequently and the second, whenever encountered, is treated as a fault. Hence to avoid numerous error messages it is best to give a qualifying L= or R= expression with any C= expression.

9. Variables of all types except records may be referred to in Assign and Print commands or C= conditions. However, array elements must be specified with constant subscripts.

10. For every line of a program compiled with PARM DEBUG, a call is planted to the DEBUG routine which, consequently, is called many times in the execution of the program. To minimise overheads it is advisable to reach that part of the user program which is of interest before setting up several stored commands. In particular, repetitive operations like the initialisation of arrays become very slow if several stored commands have their conditional expressions evaluated on every line executed. However if only a single command of the form B @ L=n is stored, this will incur minimum overheads.

Sandy Shaw

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

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          )  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
        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.

.6. 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