UNIVERSITY OF EDINBURGH
Edinburgh Regional Computing Centre

User Note 27

(January 1986)

Title:

Some Utility routines available in the EMAS 2900 Subsystem

Author: John Wexler

Contact: Advisory service

Software Support Category: See Note 15

Synopsis

The routines and functions described in this Note are all part of the Edinburgh Subsystem, and are available to IMP and IMP80 programmers without any INSERT or OPTION SEARCHDIR=... commands. The appropriate %SPECs have only to be included in the source text of the program or routine which uses them.

This software has no formal support category, and can consequently (in principle) be changed or withdrawn without notice, and there is no guarantee that bugs would be corrected. However, it has in fact been in use unchanged for a considerable time, it has proved to be stable and reliable, and some of it is heavily used by production software. It is extremely unlikely to be altered during the life of the Edinburgh subsystem on EMAS 2900. Software which uses these routines could not, of course, be moved to other systems without alteration.

Routines which provide equivalent services to the ones described in this Note will be provided on EMAS-3. Their names and specifications will differ from those described here because they have been written to be callable from several programming languages. See User Note 80 for details.

Keywords

ANALYSE PARAMETERS, CAST OUT, CHOPLDR, data manipulation, data matching, data scanning, DFDFIN, DFDFOUT, EMAS 2900, ETOI, FILL, FILPS, HTOS, i/o control, IMP80, integer-character conversion, ITOE, ITOS, JOURNAL OFF, KWDSCAN, MATCH ADDR, MOVE, parameter analysis, PARMAP, PHEX, PSTOI, QENV, QUERY PROMPTS, RCL, REPORTON, SAMEBYTES, SETPAR, SIZE OF, SPAR, STARTSWITH, STOREMATCH, SUPPRESS RECALL, system environment, TRAIL SPACES, UCSTRING, UCTRANSLATE

Contents

The utilities may be divided into six groups:

parameter analysis:	see sections 2, 8;
i/o control:	see sections 3, 4, 1;
simple manipulation of data in store:	see sections 5-10, 12, 15, 19;
scanning and matching data in store:	see sections 7, 11-14;
integer-character conversion:	see sections 16, 17;
system environment:	see section 18.

Conventions

The IMP80 on EMAS 2900 form of statement is shown first with keywords in lower case:

%external %routine %spec MOVE %alias "S#MOVE"(%integer LENGTH, FROM, TO)

If the IMP on EMAS 2900 form is different it is given next with keywords in upper case,

{%SYSTEM %ROUTINE %SPEC MOVE (%INTEGER LENGTH, FROM, TO)}

When both forms are the same, only the IMP80 on EMAS 2900 form is given.

See User Note 80 for details of the equivalent routines on EMAS-3.

1 Reading a line of input into a %STRING: RCL

```
%external %routine %spec RCL %alias "S#RCL" %c
    (%string %name S, %integer BLANKS, %integer %name R)

{%SYSTEM %ROUTINE %SPEC RCL %C
    (%STRING %NAME S, %INTEGER BLANKS, %INTEGER %NAME R)}
```

This routine reads a line of text from the currently selected input stream and puts it in S. If you want to ignore blank lines, call it with BLANKS=0. Otherwise use BLANKS=1.

The permitted values of BLANKS are:

-1	initialise the routine.
0	read text, ignore blank lines.
1	read text, return blank lines if found.

The value of R after the call indicates the result:

-1	end-of-file detected, no data available: S will be a null string, even if BLANKS=0.
0	line successfully read.
1	line successfully read but end-of-file detected: no more data available after this.
2	routine successfully initialised.
3	invalid parameters supplied by caller (i.e. BLANKS<-1 or BLANKS>1).

The characters in the string -

do not include the final newline which terminated the line;
do not include the end-of-file character (decimal 25) when end-of-file is reported;
include no 'trailing spaces'.

Before you make calls on RCL to read input, you must call it at least once with BLANKS=-1. Once you have started reading text with RCL, you should not use BLANKS=-1 again except after SELECT INPUT for a different input stream.

If you set BLANKS=1, then every line will be returned, including blank lines.
If BLANKS=0 then RCL will return only non-blank lines (except when end-of-file is detected) and you will get no indication of any blank lines which have been skipped. Since RCL always removes trailing spaces from each line, a line which contains nothing but spaces will count as a blank line and will be 'suppressed' if BLANKS=0.

Just as the text supplied does not include a terminating NEWLINE symbol, so it does not include an end-of-file symbol (decimal 25) when end-of-file is detected. End-of-file normally produces S="" and R=-1, but it may give a non-null value for S with R=+1. If you use BLANKS=1, then you could get S="" with R=+1. If you use BLANKS=0, then S="" with R=-1 is the only way that you can get a null string returned by RCL.

The length of the string will be the only indication of the number of characters in the line, and it is not possible to detect whether any trailing spaces were deleted. If there were more than 255 characters on the line, then all the characters beyond the 255th will disappear without warning. The length will be 255 (or less, if any trailing spaces were removed from the truncated line). The lost characters cannot be read or recovered by any means.

Reading always starts at the beginning of a line. If RCL is called when the last character read was not a newline, then all characters up to and including the next newline will be skipped without warning. Exception: this rule does NOT cause the first line of input to be skipped, so long as RCL starts reading at the beginning of the line.

If you call other input routines as well as RCL, you need to know that on return from RCL the NEWLINE has in fact been read in, so that the next READ or READ SYMBOL (or any other input routine) will start at the first character of the next line. If you use BLANKS=0, this should cause no problems; you must simply remember that you cannot expect to find the NEWLINE with READ SYMBOL or READ CH. READ and READ STRING skip over newlines anyway, so it should make no difference to them. But if you use BLANKS#0, it is a bit more complicated than that: RCL reads as far as the NEWLINE which terminates a non-blank line, so after you have called RCL, even if you had BLANKS=1 and you got a blank line returned in S, then a READ would start at the beginning of the next line after a non-blank line. What this means is that you should not switch from RCL to other input routines unless the last call of RCL produced a non-blank line; or, if you want to make things even simpler, don't mix calls of RCL with other input routines.

2 Command parameter analysis and prompting

The routines which analyse parameter strings, handling keywords, supplying default values and prompting for missing values, are available for use in user-written commands. Their effect, as observed by the interactive EMAS user, is described in the 1982 edition of the EMAS 2900 User's Guide. A specification for the IMP programmer is given below, with an outline of the console user's interface for the parameter prompting mechanism.

The routines and functions described here are SETPAR, FILPS, SPAR, PARMAP, QUERY PROMPTS and ANALYSE PARAMETERS. You can choose to use SETPAR or FILPS or ANALYSE PARAMETERS to do the parameter analysis.

ANALYSE PARAMETERS is the most complicated to use, but also the most versatile. If you choose it, then none of the other routines and functions will be required.

If you use SETPAR or FILPS, you will also need to use SPAR. You may also want to use PARMAP and (with FILPS) QUERY PROMPTS. SETPAR does a very simple analysis. FILPS allows for the use of keywords, and default values, and it prompts the interactive user if a parameter is missing.

2.1 SETPAR, FILPS, SPAR etc.

If a parameter string has been analysed by SETPAR or FILPS, then PARMAP can be used to determine which parameters are non-null, and SPAR can be used to find the values of the individual parameters.

Parameters are numbered from 1 upwards, from left to right, in the parameter string for SETPAR or in the template string for FILPS.

2.1.1 %external %integer %fn %spec PARMAP %alias "S#PARMAP"

{%SYSTEM %INTEGER %FN %SPEC PARMAP}

This returns a 32-bit value in which the right-hand, least significant, bit (value 1) represents parameter number 1, and the next bit (value 2) represents parameter number 2, and so on. A bit is zero if the corresponding parameter value is null, or 1 otherwise. Thus, if parameter values 1, 3 and 4 are non-null, then PARMAP will return B'00000000000000000000000000001101' (i.e., 13).

2.1.2 %external %string %fn %spec SPAR %alias "S#SPAR" (%integer N)

{%SYSTEM %STRING %FN %SPEC SPAR (%INTEGER N)}

If N>0, SPAR(N) produces the Nth parameter, which will be null if and only if the corresponding bit in the PARMAP value is zero.

SPAR(0) yields a non-null parameter value. If SPAR(0) is the first call of SPAR after SETPAR or FILPS, then it will yield the first non-null parameter value. Otherwise it will produce the next non-null parameter value in order after the parameter value yielded by the previous call of SPAR. SPAR(0) will return a null string if and only if there are no more non-null parameter values.

The effect of calling SPAR(N) with N<0 is not defined.

If the original parameter string was analysed by SETPAR, then the value returned by SPAR will have been translated to upper-case. This does not happen with FILPS. If you use SETPAR and you get at the parameter values by any other route than through SPAR, then the translation may not be done either. Since most parameters supplied to user-written commands will have been translated to upper case (by CAST OUT, q.v.) before they are passed to the command, the translation in SPAR is usually redundant, and for most purposes it does not matter whether it is done or not.

2.1.3 %external %routine %spec SETPAR %alias "S#SETPAR" (%string (255) S)

{%SYSTEM %ROUTINE %SPEC SETPAR (%STRING (255) S)}

This simply resolves S into substrings separated by single commas.
Examples:

S	SPAR(1)	SPAR(2)	SPAR(3)	SPAR(4)	SPAR(5)	Successive calls of SPAR(0)
MYFILE	MYFILE	null	null	null	null	MYFILE,null
V,L,W=80	V	L	W=80	null	null	V,L,W=80,null
FISH,,,FOWL	FISH	null	null	FOWL	null	FISH,FOWL,null
,,,,FIFTH	null	null	null	null	FIFTH	FIFTH,null

SETPAR does NOT invoke the 'prompting for missing parameters' mechanism.

If you use SETPAR, then subsequent calls of SPAR will translate the parameter values to upper case. (This does not happen if you use FILPS.)

2.1.4 FILPS and QUERY PROMPTS

2.1.4.1 For the programmer:

%external %routine %spec FILPS %alias "S#FILPS" (%string %name DPF, S)

{%SYSTEM %ROUTINE %SPEC FILPS (%STRING %NAME DPF, S)}

To use this routine, you must first decide on the order of the parameters, their keywords, and their default values (if any).
You use these to construct a string like this:

"FILENAME,OPTIONS=,OUT=T#LIST"

The string consists of fields separated by commas. Each field is a keyword optionally followed by an "=" sign and a default value. You can see from the example how a default value which is an empty string is indicated, and how you can distinguish that case from a parameter which has no default at all. This string is called the 'template'. Your command will also need declarations for -

%external %routine %spec FILPS %alias "S#FILPS" (%string %name DPF, S)
%external %integer %fn %spec PARMAP %alias "S#PARMAP"
%external %string %fn %spec SPAR %alias "S#SPAR" (%integer N)

{%SYSTEM %ROUTINE %SPEC FILPS (%STRING %NAME DPF, S)
%SYSTEM %INTEGER %FN %SPEC PARMAP
%SYSTEM %STRING %FN %SPEC SPAR (%INTEGER N)}

If S is the parameter string you want analysed, simply call FILPS (TEMPLATE,S) (where TEMPLATE is your template!), after which SPAR(n) will yield the nth parameter (nth in the order which you specified in your template). Some of these parameters may be empty strings, of course. If you want a quick check on which parameters are non-empty, use PARMAP, which returns an integer in which each bit corresponds to one parameter - the least significant bit (value 1) for the first parameter, the next bit (value 2) for the next, and so on. Thus a value of 13 indicates that the first, third and fourth parameters are non-null. There is no way (at present) to detect which parameters were actually supplied by the user and which have taken their default values.

It is FILPS which prompts the caller for 'missing parameters' if he supplies no value and there is no default given in the template. The console user can optionally request prompts for all parameters (see below).

In earlier versions of the subsystem, the
%external %routine SETPAR %alias S#SETPAR (%string (255) S)
{%SYSTEM %ROUTINE SETPAR (%STRING (255) S)} was available. It performed a much simpler analysis of the parameter string S, and set up tables ready for use by PARMAP and SPAR. This routine is still available, and still works in the same way. PARMAP and SPAR work in exactly the same way, no matter whether SETPAR or FILPS is used to do the parameter analysis. Thus commands which used to call SETPAR can easily be modified to use FILPS: a suitable template must be set up, and the call on SETPAR must be replaced by a call on FILPS. Strictly speaking, the call of FILPS should be preceded by a call on
%external %string %fn UCSTRING (%string (255) S) or on
%external %routine UCTRANSLATE %alias "S#UCTRANSLATE" (%integer ADDRESS, LENGTH)
{%SYSTEM %ROUTINE UCTRANSLATE (%INTEGER ADDRESS, LENGTH)} if you want complete compatibility with SETPAR, since SETPAR has the effect of translating all parameters into upper case but FILPS does not.

Examples -

```
SETPAR (S)
```

could be replaced by

```
S = UCSTRING (S)
FILPS (TEMPLATE, S)
```

or by

```
UCTRANSLATE (ADDR(CHARNO(S,1)),LENGTH(S))
FILPS (TEMPLATE, S).
```

However, if you are writing a new command, you should consider carefully whether you want this translation done. If your command is called from a console or from a batch job, any parameters supplied will be translated to upper case by the command interpreter before they are passed in to your command - unless the caller explicitly bypasses the translation by using double quotes around the parameters that he types. The translation will not be done if your command is called from another program. For most ordinary purposes it is therefore redundant to do a further translation before calling FILPS.

The **%external %routine QUERY PROMPTS %alias "S#QUERYPROMPTS" (%integer I)** {%SYSTEM %ROUTINE QUERY PROMPTS (%INTEGER I)} controls parameter prompting in commands called from programs. A command which uses the parameter prompting mechanism will always prompt for those parameters which are missing and for which no default values are known. However, if the command is called from an interactive console, and if it is preceded by a single "?" character, then it will prompt for ALL its parameters (indicating the default values as part of the prompts). If the command is called from a program instead of from an interactive console, then its prompting will normally be controlled by the presence or absence of a "?" at the start of the command which invoked the calling program. Normally, a user-written command or program will supply a full set of parameters when it calls a subsystem command, so that no prompting will be needed. However, if the user-written command had been invoked by the console user with a request for 'full prompting', then any subsystem commands which it might call would also prompt for all their parameters to be supplied from the console. This can be avoided by preceding the calls on the subsystem commands by

```
QUERY PROMPTS (0).
```

On the other hand, one might want to write a program so that it reads the parameters for a call on a subsystem command immediately before calling the command. A simple way to get the same effect would be to call

```
QUERY PROMPTS (n)      {where n#0}
```

before calling the command. The command will then prompt for all its parameters. Any parameter values supplied by the program when it calls the command will be offered to the console user as default values when the command prompts for parameters.

2.1.4.2 For the console user:

The 'old' form of command should still work: you simply type the parameters separated by commas. Missing parameters are indicated by consecutive commas. Example -

Command: FILES TA*,,TAFLIST

Alternatively, if you know the keywords for a command, you can give each parameter preceded by its keyword and an "=" sign -

Command: IMP80 SOURCE=COLS,OBJECT=COLY,ERROR=.OUT

Here there is no need for extra commas to indicate missing parameters. Furthermore, the order of the parameters is no longer important. The last command is entirely equivalent to -

Command: IMP80 ERROR=.OUT,SOURCE=COLS,OBJECT=COLY

You can also save typing by abbreviating long keywords to (at least) three characters -

Command: IMP80 ERR=.OUT,OBJ=COLY,SOUR=COLS

You do not need to specify keywords for every parameter you want to supply. If you are using a command with parameter keywords INPUT, CONTROL, OPTIONS, LINES, START, FINISH, OUTPUT, PHASE, ORDER in that order, then you could give

Command: TXSET HU8L2,,A3,OUT=TXSETUP,8,ASC

to supply parameter values as follows -

INPUT	HU8L2
CONTROL	default (if any)
OPTIONS	A3
LINES	default (if any)
START	default (if any)
FINISH	default (if any)
OUTPUT	TXSETUP
PHASE	8
ORDER	ASC

The trick is that, if you specify a parameter by keyword, followed by one or more without keywords, those are taken to be the ones which follow the 'keyed' parameter in the parameter sequence defined by the command. If you leave the command to take the default value of a parameter, and it does not have a defined default value, then you will be prompted with the keyword, like this:

Command: ANALYSE OPT=H
FILE:

If you do not know the parameters required by the command, you may get some help by typing the command name preceded by a "?", and with no parameters.

Command: ?FORTE
SOURCE: LA608
OBJECT: LA608J
LIST(T#LIST):
ERROR(): .OUT

You can see here how the keywords are used as prompts, and where there is a

default it is included after the keyword in brackets. In such cases, you can choose to accept the default by simply responding with a carriage return. There is also an example of a parameter whose default is a null string.

2.2 ANALYSE PARAMETERS

ANALYSE PARAMETERS is the routine which is used by FILPS. It is more complicated to use than FILPS, but at the same time it is more versatile. If you use ANALYSE PARAMETERS instead of FILPS or SETPAR, then you cannot use SPAR and PARMAP to get the parameter values. Instead, ANALYSE PARAMETERS itself puts the parameters in an %array which you supply yourself. ANALYSE PARAMETERS does NOT invoke the 'parameter prompting' mechanism. For some purposes, there may be advantages in using ANALYSE PARAMETERS rather than FILPS. For instance, it allows you (as author of a command) to recognize whether a parameter value was supplied in the parameter string or whether the default value was taken; it allows you to recognize the difference between a 'null parameter' and a 'missing parameter'; and you do not even need to know the keywords!

```
%record %format DRF (%integer LENGTH, AD)
%external %routine %spec ANALYSE PARAMETERS %alias "S#ANALYSEPARAMETERS" %c
      (%string %name TEMPLATE, CALL PARMS, %integer MAX PARMS,
       %string %array %name KEYS, %integer MAX KEY SIZE,
       %record (DRF) %array %name ACTUAL,
       %integer %name TOTAL KEYS, RESPONSE)

{%RECORD %FORMAT DRF (%INTEGER LENGTH, AD)
%SYSTEM %ROUTINE %SPEC ANALYSE PARAMETERS %C
      (%STRING %NAME TEMPLATE, CALL PARMS, %INTEGER MAX PARMS, %C
       %STRING %ARRAY %NAME KEYS, %INTEGER MAX KEY SIZE, %C
       %RECORD (DRF) %ARRAY %NAME ACTUAL, %C
       %INTEGER %NAME TOTAL KEYS, RESPONSE)}
```

This routine takes two %string parameters, TEMPLATE and CALL PARMS.

It produces in the %string %array KEYS all the keywords declared in TEMPLATE, in the correct order, and in %record %array ACTUAL, pointers to the corresponding actual parameter texts. The actual parameter texts are found in CALL PARMS if they are given there; otherwise the default values are taken from TEMPLATE (and if no value is given in TEMPLATE either, that is reported to the calling program).

The pointers will be %records of the format DRF. In each pointer, the LENGTH field gives the number of characters in the text, and the AD field gives the address of the text. In the LENGTH word, the length is an unsigned positive integer held in the least significant 24 bits of the word, and the most significant 8 bits are set to X'18'. But where no actual parameter text is found either in TEMPLATE or in CALL PARMS, the LENGTH word will be -1 (X'FFFFFFFF'). Thus for most purposes you should clear the top 8 bits of the LENGTH before using it:

```
e.g., L = ACTUAL(I)_LENGTH & X'00FFFFFF'.
```

The text is NOT in the form of an IMP %string. It does NOT start with a length byte. It is simply a number of consecutive bytes containing characters. The pointers will be to areas within the %string CALL PARMS or (where a default is used) within TEMPLATE. ANALYSE PARAMETERS does not overwrite any part of CALL PARMS or TEMPLATE, so the text will not be translated to upper-case.

It is possible to have a length of zero characters (represented by a LENGTH word of X'18000000') if the value was specified as a null string by ...,KEY=,... in either CALL PARMS or TEMPLATE.

The values of MAX PARMS and MAX KEY SIZE must be set on entry to indicate the maximum number of parameters and the maximum length of the keyword strings which can be accepted. The %arrays KEYS and ACTUAL must be declared with upper bounds not less than MAX PARMS and lower bounds of 1. The %strings in %array KEYS must have maximum length not less than MAX KEY SIZE.

On exit, RESPONSE will be =0 for success, >0 for warnings and <0 for failure. As well as the sign bit, other bits may be set to indicate specific warning or error conditions.

bit 24 (value 128) - keywords indistinct: two keywords have the same first characters, so that their abbreviations could not be distinguished in a call.
bit 25 (value 64) - 'wrap-around': first parameter has been specified by position, but not in first position in the call.
bit 26 (value 32) - some parameter specified more than once: latest value accepted.
bit 27 (value 16) - unrecognized keyword in call: field ignored.
bit 28 (value 8) - keyword too long in call: extra characters ignored.
bit 29 (value 4) - too many fields in declaration.
bit 30 (value 2) - keyword too long in declaration: extra characters ignored.
bit 31 (value 1) - field with no keyword in declaration.

If RESPONSE>=0, then TOTAL KEYS will also be set to indicate how many parameters there are.

There is an interesting possibility with ANALYSE PARAMETERS: if you are writing a command which accepts a parameter X, you can call ANALYSE PARAMETERS with X for the TEMPLATE and a null string for CALL PARMS. The effect is best demonstrated by an example: if the user of the command typed
PRTY=LOW,DEST=BATCH,OPT,NOCHECK then

```
PRTY
DEST
OPT
NOCHECK
```

would be recognized as KEYS, and LOW and BATCH would appear in ACTUAL as parameter values corresponding to PRTY and DEST.

3 Redefining default i/o streams: REPORTON, DFDFOUT, DFDFIN

```
%external %routine %spec DFDFOUT %alias "S#DFDFOUT" %c
    (%string (31) FILE, %integer CHAN, %integer %name FLAG)

%external %routine %spec DFDFIN %alias "S#DFDFIN" %c
    (%string (31) FILE, %integer CHAN, %integer %name FLAG)

{%SYSTEM %ROUTINE %SPEC DFDFOUT %C
    (%STRING (31) FILE, %INTEGER CHAN, %INTEGER %NAME FLAG)

%SYSTEM %ROUTINE %SPEC DFDFIN %C
    (%STRING (31) FILE, %INTEGER CHAN, %INTEGER %NAME FLAG)}
```

These allow a program or command to redefine temporarily the destination (or source) for output (input) on stream 0. The preferred user interface is REPORTON (for output only), but the specifications of DFDFOUT and DFDFIN are given here for completeness.

Both DFDFOUT and DFDFIN have the effect of DEFINE(CHAN,FILE), and they also ensure that when i/o is requested on channel 0, channel CHAN will actually be used.

Either input or output may be redirected, or both may be redirected provided that the same value of CHAN is not used for input and for output. FILE may be a file name or a device name or .NULL, and for DFDFOUT it may also be filename-MOD to append the output to the existing contents of the file. If FILE is an empty string, then CHAN is ignored, and channel 0 input or output reverts to using whatever route was active before redirection - that is, the interactive console, or the command file or journal for a batch job. The original DEFINE for the redirected channel is not cleared.

%external %routine %spec REPORTON (%integer CHAN) - allows users to reroute stream 0 output to another channel. Particularly for subsystem error messages ("FRED is a copy of GEORGE"), and for diagnostics following program failure. Stream CHAN must already be DEFINEd, possibly as .NULL. REPORTON(0) reverts to 'normal' channel 0.

4 Temporary SUPPRESS RECALL or JOURNAL OFF

The **%external %routine JOURNAL OFF %alias "S#JOURNALOFF"** (no parameters) {%SYSTEM %ROUTINE JOURNAL OFF} allows a program or command to stop its output from being saved in the journal (RECALL file). This is for software which generates large quantities of uninteresting output - e.g., transfers of binary data to microcomputers. Once JOURNAL OFF has been called, RECALLing is temporarily turned off until the next *Command:* prompt. The preferred interface is SUPPRESS RECALL (see below).

%external %routine %spec SUPPRESS RECALL
This routine is used to suppress RECALL until return to command level. It takes no parameters. It allows a program or command to stop its output from being saved in the journal (RECALL file). This is for software which generates large quantities of output - e.g., transfers of binary data to microcomputers. Once SUPPRESS RECALL has been called, RECALLing is temporarily turned off until the next *Command:* prompt. SUPPRESS RECALL has exactly the same effect as JOURNAL OFF.

If either JOURNAL OFF or SUPPRESS RECALL has been used, 'RECALLing' can be re-enabled by a call CONSOLE(14,DUMMY,DUMMY) where CONSOLE has been specified as

```
%external %routine %spec CONSOLE %alias "S#CONSOLE" %c
      (%integer EP,%integer %name I,J)
{%SYSTEM %ROUTINE %SPEC CONSOLE(%INTEGER EP, %INTEGER %NAME I,J)}
```

5 MOVE data from one area of virtual store to another

%external %routine %spec MOVE %alias "S#MOVE"(%integer LENGTH, FROM, TO)

{%SYSTEM %ROUTINE %SPEC MOVE (%INTEGER LENGTH, FROM, TO)}

The routine copies LENGTH bytes from address FROM to address TO. The two areas may overlap in any way, and even if they do overlap, the final contents of the TO area will be the same as the original contents of the FROM area. If they do not overlap, then the contents of the FROM area will not be changed. MOVE has no effect if LENGTH is not greater than zero. It does not check FROM and TO, and invalid addresses will cause 'address errors' to be detected by hardware and reported as program failures. MOVE will not enable the user to violate the hardware protection mechanisms, but it cannot prevent overwriting of the contents of the user's own virtual memory, and that includes those files which are connected for writing.

6 FILL an area of virtual store with copies of a byte

%external %routine %spec FILL %alias "S#FILL" (%integer LENGTH, AT, FILLER)

{%SYSTEM %ROUTINE %SPEC FILL (%INTEGER LENGTH, AT, FILLER)}

This routine overwrites each of the LENGTH bytes starting at address AT with a copy of FILLER. Only the least significant eight bits of FILLER are used, and the rest are ignored. FILL has no effect if LENGTH is not greater than zero. It does not check AT, and the consequences of using an incorrect address can be the same as with MOVE.

7 Translate to upper-case: UCTRANSLATE and UCSTRING

%external %routine %spec UCTRANSLATE %alias "S#UCTRANSLATE" %c
(%integer ADDRESS, LENGTH)

{%SYSTEM %ROUTINE %SPEC UCTRANSLATE (%INTEGER ADDRESS, LENGTH)}

LENGTH bytes starting at ADDRESS are inspected. Each one whose value is the ISO representation of a lower-case letter is overwritten by the value representing the corresponding upper-case letter. All other bytes in the range are unchanged. In short, the area is translated to upper- case. If LENGTH is not greater than zero, the routine has no effect at all. ADDRESS is not checked, and an invalid ADDRESS may have the same consequences as for MOVE.

%external %string %fn %spec UCSTRING (%string (255) S)

This function returns a string which is the same as S except for having been translated to upper-case as described for UCTRANSLATE.

8 CAST OUT spaces from a string and translate to upper-case

%external %routine %spec CAST OUT %alias "S#CASTOUT" (%string %name PSTR)

{%SYSTEM %ROUTINE %SPEC CAST OUT (%STRING %NAME PSTR)}

This routine removes spaces from PSTR and translates any lower-case letters to upper case. Other symbols (including newline symbols) are not affected.
Double-quotes are an exception: they are treated specially so that suitable strings (or parts of strings) can survive unchanged. Any parts of PSTR which are enclosed between double-quotes are not affected - spaces are not removed and lower-case letters are not changed. The enclosing double-quotes are removed. If there are an odd number of double-quotes, the last (or only) one is assumed to be matched by a double-quote immediately after the last character of the string. If two consecutive double-quotes are themselves enclosed between double-quotes, then one of them is discarded and one is retained. This is the only way in which a double-quote can survive the action of CAST OUT; it is also the only change which is made to text enclosed between double-quotes.
Examples:
N.B. The enclosing < and > are not part of the strings; they are provided only to show leading and trailing spaces.

```
  BEFORE                          AFTER
<ABCD1234>                      <ABCD1234>
<    Ab cDEf G >                <ABCDEFG>
<ab 12 " cd 34" >               <AB12 cd 34>
<ab 12 " cd 34 >                <AB12 cd 34 >
<ab 12 " cd ""34""" e fG>       <AB12 cd "34"EFG>
```

9 Remove leading characters from a string: CHOPLDR

%external %routine %spec CHOPLDR %alias "S#CHOPLDR" (%string %name A, %integer I)

{%SYSTEM %ROUTINE %SPEC CHOPLDR (%STRING %NAME A, %INTEGER I)}

This routine discards I bytes from the start of the string A, so that the length of A is reduced by I. I must be <= LENGTH(A) on entry: this is not checked. CHOPLDR has the same effect but is more efficient than the IMP80 statement A=SUBSTRING(A,I+1,LENGTH(A))

10 Count trailing spaces in a string: TRAIL SPACES

%external %integer %fn %spec TRAIL SPACES %alias "S#TRAILSPACES" %c (%integer LINE END, LINE START, TRANS)

{%SYSTEM %INTEGER %FN %SPEC TRAIL SPACES %C
(%INTEGER LINE END, LINE START, TRANS)}

This function yields a count of the 'trailing spaces' in the text whose first byte is at address LINE START and whose last byte is at LINE END. That is, if the N consecutive bytes at addresses LINE END-N+1 to LINE END inclusive are all spaces, and the byte at LINE END-N is not a space, then the function will return N. If all the bytes from LINE START to LINE END are spaces, it will return LINE END-LINE START+1. If the byte at LINE END is not a space, it will return zero. Bytes at addresses below LINE START are never inspected and cannot affect the result. If TRANS is zero, it counts ISO space characters (the EMAS standard); if TRANS is non-zero, it counts EBCDIC spaces. It returns zero if LINE START is greater than or equal to LINE END. The addresses are not checked otherwise, and invalid addresses may lead to address errors (but not to the overwriting of store or of files).

11 Compare areas of virtual store: SAMEBYTES and STOREMATCH

%external %integer %fn %spec SAMEBYTES %alias "S#SAMEBYTES" %c (%integer L, A1, A2)

{%SYSTEM %INTEGER %FN %SPEC SAMEBYTES (%INTEGER L, A1, A2)}

This function compares L bytes at A1 with L bytes at A2. It returns a count of bytes which match at the start of the two areas: zero if the first bytes are not the same, and L if the two areas are exactly the same. It returns zero if L is zero. Only the least significant 24 bits of L are significant, and they are taken as an unsigned (i.e. positive) integer. Consequently, if a negative value is supplied for L, the effect will not be sensible, and an address error may well occur. A1 and A2 are not checked, and invalid addresses may also produce address errors. However, SAMEBYTES will not overwrite anything even if it is called with invalid parameters.

%external %integer %fn %spec STOREMATCH %alias "S#STOREMATCH" %c
(%integer L, A1, A2)

{%SYSTEM %INTEGER %FN %SPEC STOREMATCH (%INTEGER L, A1, A2)}

This function compares L bytes at A1 with L bytes at A2. It returns non-zero if the two areas are the same, or zero if they differ. If L is zero, STOREMATCH will return a non-zero value: i.e. any two areas of zero length are taken to be equal. In other respects its behaviour is exactly like that of SAMEBYTES, and all the same remarks apply.

12 Check whether one string starts with another: STARTSWITH

%external %integer %fn %spec STARTSWITH %alias "S#STARTSWITH" %c
(%string %name A, %string (255) B, %integer CHOP)

{%SYSTEM %INTEGER %FN %SPEC STARTSWITH %C
(%STRING %NAME A, %STRING (255) B, %INTEGER CHOP)}

This function returns zero if string A does not start with a copy of string B, and returns a non-zero value if string B is the same as the first characters of string A. If CHOP is zero, then that is the only effect of STARTSWITH. If CHOP is non-zero, then STARTSWITH also has the side effect of discarding the copy of B from the beginning of A, so that A is 'shortened' by LENGTH (B) bytes.

This function is provided to replace the IMP resolutions

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

which could be used to test whether A 'started with' B, and in the former case to remove B from the beginning of A. The revised definition of string resolution in IMP80 means that those two resolutions now test whether A includes a copy of B, even if that copy is not at the start of A. To achieve the intended result in IMP80 one may write

```
    A -> DUMMY.(B).C; %if DUMMY="" %then A = C
or  %if A -> DUMMY.(B) %and DUMMY="" %then ....
```

but STARTSWITH provides an alternative method.

13 Scanning an area of store for a given pattern of bytes: MATCH ADDR

%external %integer %fn %spec MATCH ADDR %c
(%integer PATTERN ADDRESS, PATTERN LENGTH, RANGE ADDRESS,
RANGE LENGTH)

{%EXTERNAL %INTEGER %FN %SPEC MATCH ADDR %C
(%INTEGER PATTERN ADDRESS, PATTERN LENGTH, RANGE ADDRESS, %C
RANGE LENGTH)}

This function scans a row of RANGE LENGTH bytes at RANGE ADDRESS to find the first slice which matches the pattern of PATTERN LENGTH bytes at PATTERN ADDRESS. It returns the address of the first byte of the slice, if found: otherwise it returns zero. The pattern may overlap the range at either end.

A zero length pattern will produce a match and return a copy of RANGE ADDRESS as the result. Otherwise zero or negative lengths, or lengths not less than 2**24, will

produce a result of zero. If the pattern is longer than the range then a zero result will also be returned.

MATCH ADDR can be used for some of the purposes which IMP string resolutions can fulfil. It is not restricted to areas of less than 256 bytes long. If it is being used for tests on IMP strings, the length bytes of the strings should not be included in the areas being compared.

14 Keyword index facilities: KWDSCAN

For some purposes, it is useful to have an 'index' of names of things with a collection of 'keywords' for each name, so that if you are interested in a particular keyword, you can find all the 'things' which have that keyword. For instance, the 'things' might be files of geographical data, and the keywords for each file might be the names of the regions described in the file; and you would want to be able to find all the files which contain data about whichever region you choose. Or the 'things' might be the names of books, and the keywords might be the names of authors, and you would like to be able to locate the books written by any particular author.

This section describes a simple format for a 'keyword index', and a routine which will search any index in that format. The routine is extremely efficient as implemented on EMAS 2900. It allows you to nominate one or more keywords for a search, and if you provide several it will locate those 'things' which have ALL the keywords. (If you want to find those things which have ANY of the keywords, you can simply do a separate search for each keyword.) You can specify complete keywords, or you can ask for keywords which start with, or include, or end with, some pattern.

```
%external %integer %fn %spec KWDSCAN %c
      (%integer %name XLENGTH,XADDR, %integer KWDCT,
       %string %array %name KEYS)

{%EXTERNAL %INTEGER %FN %SPEC KWDSCAN %C
      (%INTEGER %NAME XLENGTH,XADDR, %INTEGER KWDCT, %C
       %STRING %ARRAY %NAME KEYS)}
```

This function searches a 'keyword index' for a section which includes all of a set of keywords nominated by the caller. Successive calls will discover all such sections in the index.

The index must be laid out as follows:

Each section consists of a section name (any sequence of bytes not including NL, space or colon - they need not all be printable characters - there are no rules about the first character being alphabetic - the name may, but need not, be an IMP string starting with a 'length byte'), and the section name is immediately followed by a colon and a space. That is followed by one or more keywords, separated by single spaces. Keywords should consist of printable characters, i.e., format effectors and length bytes should NOT be included, and the three characters NL, space and colon are not permitted in keywords. There are no other restrictions on keywords. The last keyword of a section must be followed by a space and NL. The next section, if any, follows immediately, with no intervening characters. The first section in an index may, but need not, be preceded by a NL character. The index should contain no characters after the NL terminating the last section.

The caller supplies the number of bytes in the index as the parameter XLENGTH, and the address of the first byte of the index as the parameter XADDR. The number of keywords to be sought is given as KWDCT, and the keywords themselves are supplied in IMP strings as KEYS(1:KWDCT).

If a section is found whose keywords include all those nominated by the caller, then KWDSCAN will return the address of the first byte of the title of that section. If no such section is found, the result will be zero. In either case, XLENGTH and XADDR will be updated to specify that part of the index which should be scanned at the next call of KWDSCAN. That is, if KWDSCAN returns the address of a section title, then XADDR will point to the next section (if any) and XLENGTH will be correspondingly reduced. Thus a further call of KWDSCAN, using the new values of XLENGTH and XADDR, will find another section which satisfies the caller's keyword specification (if there is another such section). If there are no more sections, or if KWDSCAN returned zero after an unsuccessful search, then XLENGTH will be zero and XADDR will point to the first byte after the end of the index.

The KEYS may use the same characters as are permitted for the keywords in the index. It is also permissible for the first and/or last character of any of the KEYS to be a space. If one of the KEYS starts with a space, then it can only be matched by a keyword in the index which starts with the non-space characters of the KEY. If a KEY ends with a space, then it is matched only by keywords which end with the non-space characters of the KEY. If it begins and ends with a space, then only keywords which exactly match all the non-space characters of the KEY will be acceptable. A KEY which contains no spaces can be matched by any keyword which includes the KEY.

None of KEYS(1:KWDCT) may be a null string. KWDCT must be >0.

15 ISO/EBCDIC translations: ITOE and ETOI

%external %routine %spec ITOE %alias "S#ITOE" (%integer AD, L)

{%SYSTEM %ROUTINE %SPEC ITOE (%INTEGER AD, L)}

This routine translates L bytes starting at address AD from ISO to EBCDIC. In all other respects, its behaviour is like that of UCTRANSLATE, and all the same remarks apply.

I	E	I	E	I	E	I	E	I	E	I	E	I	E	I	E
0	0	32	64	64	124	96	121	128	32	160	65	192	118	224	184
1	1	33	79	65	193	97	129	129	33	161	66	193	119	225	185
2	2	34	127	66	194	98	130	130	34	162	67	194	120	226	186
3	3	35	123	67	195	99	131	131	35	163	68	195	128	227	187
4	55	36	91	68	196	100	132	132	36	164	69	196	138	228	188
5	45	37	108	69	197	101	133	133	21	165	70	197	139	229	189
6	46	38	80	70	198	102	134	134	6	166	71	198	140	230	190
7	47	39	125	71	199	103	135	135	23	167	72	199	141	231	191
8	22	40	77	72	200	104	136	136	40	168	73	200	142	232	202
9	5	41	93	73	201	105	137	137	41	169	81	201	143	233	203
10	37	42	92	74	209	106	145	138	42	170	82	202	144	234	204
11	11	43	78	75	210	107	146	139	43	171	83	203	154	235	205
12	12	44	107	76	211	108	147	140	44	172	84	204	155	236	206
13	13	45	96	77	212	109	148	141	9	173	85	205	156	237	207
14	14	46	75	78	213	110	149	142	10	174	86	206	157	238	218
15	15	47	97	79	214	111	150	143	27	175	87	207	158	239	219
16	16	48	240	80	215	112	151	144	48	176	88	208	159	240	220
17	17	49	241	81	216	113	152	145	49	177	89	209	160	241	221
18	18	50	242	82	217	114	153	146	26	178	98	210	170	242	222
19	19	51	243	83	226	115	162	147	51	179	99	211	171	243	223
20	60	52	244	84	227	116	163	148	52	180	100	212	172	244	234
21	61	53	245	85	228	117	164	149	53	181	101	213	173	245	235
22	50	54	246	86	229	118	165	150	54	182	102	214	174	246	236
23	38	55	247	87	230	119	166	151	8	183	103	215	175	247	237
24	24	56	248	88	231	120	167	152	56	184	104	216	176	248	238
25	25	57	249	89	232	121	168	153	57	185	105	217	177	249	239
26	63	58	122	90	233	122	169	154	58	186	112	218	178	250	250
27	39	59	94	91	74	123	192	155	59	187	113	219	179	251	251
28	28	60	76	92	224	124	106	156	4	188	114	220	180	252	252
29	29	61	126	93	90	125	208	157	20	189	115	221	181	253	253
30	30	62	110	94	95	126	161	158	62	190	116	222	182	254	254
31	31	63	111	95	109	127	7	159	225	191	117	223	183	255	255

%external %routine %spec ETOI %alias "S#ETOI" (%integer AD, L)

{%SYSTEM %ROUTINE %SPEC ETOI (%INTEGER AD, L)}

This routine translates L bytes starting at address AD from EBCDIC to ISO. In all other respects, its behaviour is like that of UCTRANSLATE, and all the same remarks apply.

E	I	E	I	E	I	E	I	E	I	E	I	E	I	E	I
0	0	32	128	64	32	96	45	128	195	160	209	192	123	224	92
1	1	33	129	65	160	97	47	129	97	161	126	193	65	225	159
2	2	34	130	66	161	98	178	130	98	162	115	194	66	226	83
3	3	35	131	67	162	99	179	131	99	163	116	195	67	227	84
4	156	36	132	68	163	100	180	132	100	164	117	196	68	228	85
5	9	37	10	69	164	101	181	133	101	165	118	197	69	229	86
6	134	38	23	70	165	102	182	134	102	166	119	198	70	230	87
7	127	39	27	71	166	103	183	135	103	167	120	199	71	231	88
8	151	40	136	72	167	104	184	136	104	168	121	200	72	232	89
9	141	41	137	73	168	105	185	137	105	169	122	201	73	233	90
10	142	42	138	74	91	106	124	138	196	170	210	202	232	234	244
11	11	43	139	75	46	107	44	139	197	171	211	203	233	235	245
12	12	44	140	76	60	108	37	140	198	172	212	204	234	236	246
13	13	45	5	77	40	109	95	141	199	173	213	205	235	237	247
14	14	46	6	78	43	110	62	142	200	174	214	206	236	238	248
15	15	47	7	79	33	111	63	143	201	175	215	207	237	239	249
16	16	48	144	80	38	112	186	144	202	176	216	208	125	240	48
17	17	49	145	81	169	113	187	145	106	177	217	209	74	241	49
18	18	50	22	82	170	114	188	146	107	178	218	210	75	242	50
19	19	51	147	83	171	115	189	147	108	179	219	211	76	243	51
20	157	52	148	84	172	116	190	148	109	180	220	212	77	244	52
21	133	53	149	85	173	117	191	149	110	181	221	213	78	245	53
22	8	54	150	86	174	118	192	150	111	182	222	214	79	246	54
23	135	55	4	87	175	119	193	151	112	183	223	215	80	247	50
24	24	56	152	88	176	120	194	152	113	184	224	216	81	248	56
25	25	57	153	89	177	121	96	153	114	185	225	217	82	249	57
26	146	58	154	90	93	122	58	154	203	186	226	218	238	250	250
27	143	59	155	91	36	123	35	155	204	187	227	219	239	251	251
28	28	60	20	92	42	124	64	156	205	188	228	220	240	252	252
29	29	61	21	93	41	125	39	157	206	189	229	221	241	253	253
30	30	62	158	94	59	126	61	158	207	190	230	222	242	254	254
31	31	63	26	95	94	127	34	159	208	191	231	223	243	255	255

N.B. Translations between different character codes are not straightforward. There are variants of EBCDIC and of ISO; the EBCDIC code has 256 values and ISO has only 128; and there are many criteria, not all mutually compatible, for 'sensible' translations. The EMAS standard translations are adequate for most purposes, but a more comprehensive translation scheme will be available among the magnetic tape handling utilities (since magnetic tape is the origin of most of the EBCDIC text handled on EMAS).

16 Conversions between integers and strings of digits: ITOS and PSTOI

%external %string %fn %spec ITOS %alias "S#ITOS" (%integer N)

{%SYSTEM %STRING %FN %SPEC ITOS (%INTEGER N)}

This function produces a string which gives the decimal representation of the value of N. The length of the string is the minimum possible. It will contain no spaces, nor indeed any characters except decimal digits and (if N is negative) a "-" sign as the first character. If N is zero, ITOS returns a single character "0".

%external %integer %fn %spec PSTOI %alias "S#PSTOI" (%string (63) S)

{%SYSTEM %INTEGER %FN %SPEC PSTOI (%STRING (63) S)}

If S is a non-null string containing no characters other than decimal digits, PSTOI will return a value which is the binary representation of the positive integer represented by S. Overflow may occur if S represents too large a value. If S is a null string, or if it contains any characters other than digits, PSTOI will return the value -1, and that is the only negative value which it can return. In particular, spaces ANYWHERE in S, + or - signs, and any sort of punctuation will cause PSTOI to return -1.

17 Convert integer to string of hexadecimal digits: HTOS and PHEX

%external %string (8) %fn %spec HTOS %alias "S#HTOS" %c
(%integer VALUE, PLACES)

{%SYSTEM %STRING (8) %FN %SPEC HTOS (%INTEGER VALUE, PLACES)}

This function supplies a string of PLACES characters, which is the hexadecimal representation of the least significant PLACES quartets (half-bytes) of VALUE. The string contains no spaces or punctuation, nor indeed any characters other than hexadecimal digits. PLACES must be in the range 1 to 8 inclusive; other values are not checked but may cause unpredictable results (but if PLACES is zero, HTOS will return an empty string).

%external %routine %spec PHEX %alias "S#PHEX" (%integer I)

{%SYSTEM %ROUTINE %SPEC PHEX (%INTEGER I)}

This routine simply outputs the string generated by HTOS(I,8) on the currently selected output stream. It outputs no other characters.

18 Command QENV: list system environment

QENV or QENV ,output

This command prints some text identifying the environment in which it was called - the processor, the versions of the supervisor, director and subsystem, the user, batch job or interactive session, and so on. The second parameter to QENV, output, might be .LPnn or the name of a file. User Note 35 contains a fairly comprehensive description of the information provided by QENV.

One use of QENV is to get a report on the status of the system if things are not behaving as you expect. For instance, if you get some incomprehensible diagnostics or an inexplicable failure, it may be helpful to have a QENV listing when you bring your problem to Advisory. For this purpose, use the command QENV *,file or QENV *,.LPnn. If you use * as the first parameter to QENV you may get a lot of output, so

it is best to direct the output to a file or device. QENV * is a possible command but it sends the output to your terminal where it is not very useful.

If you want to get this sort of information in a program, you can use UINFI and UINFS which are described in the EMAS 2900 User's Guide.

19 SIZE OF a variable

%external %integer %fn %spec SIZE OF %alias "S#SIZEOF" (%name X)

{%SYSTEM %INTEGER %FN %SPEC SIZE OF (%NAME X)}

This function returns the number of bytes occupied by the variable whose name is passed as the parameter X. It works for simple variables such as %byte %integers and %long %reals, and also for %records and %strings (for which it returns the declared maximum length PLUS ONE to allow for the 'length byte' – the length of the current contents of the string is ignored, and if it is required it can be obtained by using LENGTH).