POP-10 USER'S MANUAL

Author: D. J. M. Davies

Revised for POP-10 Version 8(261)
Revised for POP-10 Version 11E(313)
Revised for POP-10 Version 12A(335)
Revised for POP-10 Version 12D(363)
Date: 29 May 1976

Department of Computer Science,
University of Western Ontario,
London, Ontario, Canada.

Refer to the book "Programming in POP-2" for a general introduction to and a detailed description of the language. This memorandum describes the particular features of this POP-10 implementation.

1.Ø INTRODUCTION

POP-1Ø is the version of POP-2 in use on this DECsystem-1Ø computer installation. The name has been changed to avoid confusion, since this implementation varies in some respects from the formal definition of POP-2 given by

R.M. Burstall, J.S. Collins & R.J. Popplestone;
"Programming in POP-2",
Edinburgh University Press, 1971.
(sometimes called the 'Silver Book').

1.1 Copyright Warning.

1.2 Using POP-1Ø

The changes introduced in POP-1Ø are designed to take advantage of the facilities of the DECsystem-1Ø operating environment, and also to avoid various problems that have arisen in normal use of 'standard' POP-2.

To use the POP-1Ø system, type:

```
R POP1Ø
```

The POP-1Ø system will be started; it will print a header line identifying the system version and then:

```
setpop
:
```

and the system is now waiting for you to type commands to it. You may type normal POP-2 commands, which will then be executed. You may also instruct the system to compile specified files of POP-2 program, or you may define functions by directly typing in suitable text. (The system can be made to automatically compile a file when starting

up, by using the INIT.POP facility - Section 3.2.)

To compile a file of POP-2 program, (say) FILE1.POP, type a command to POP-1Ø in this format:

```
COMPILE([FILE1.POP]);
```

Note that ".POP" is the recommended extension for files of POP-2 program text.
Using COMPILE in this way will work even if the file has line-numbers in it, according to standard DECsystem-1Ø conventions, provided that the POP-1Ø system has been assembled to include "SOS" code.

To leave POP-1Ø, type the control-character ↑C followed by ↑X. This will cause the POP-1Ø system to return control to the monitor, printing the messages

```
int:
Exit POP1Ø
```

The warning

```
%PEDBNE buffer not empty
```

will also be printed if the PED buffer is not empty.

If you leave the POP-1Ø system, you may subsequently continue by using either the .CONT or the .REENTER commands, provided the POP-1Ø core-image has not been destroyed. The .CONT command will make your POP-1Ø program continue as it was before the interruption; the .REENTER command will cause execution of the POP-1Ø program to be suspended, and

```
ready
Ø::
```

will be printed. The program is now in a "READY" break and you may type POP-1Ø commands to examine its state. To continue the suspended program and leave the break, type ↑Z.

1.3 Accuracy Of Numbers

There are two types of number representation in POP-1Ø, called Integers and Reals. Integers are in a fixed-point integer two's complement representation, with a sign bit and 33 magnitude bits. Numbers in the range -2↑33 to (2↑33 -1) caf be dirEctly represented, i.e. -8,589,934,592 to 8,589,934,591. Boolean truth values are represented with the integers Ø and 1, and positive integers may also be treated as 33-bit logical bit-strings.

Real numbers are in a base-2 system with 8-bit exponent, and 25-bit fraction (and sign), and can represent numbers in the range of magnitudes: approximately 1.7ØE+38 to 1.47E-39, with an accuracy of about 7 decimal places.

1.4 Errors In POP-2

The system now handles errors slightly differently. When an error is detected, the system first prints a message. Then it enters a break to permit more information about the error to be collected if needed. For example, you may type "?" to get an explanation of the error number, or you may type ":" to obtain a listing of the calling sequence of functions. You may also examine variables or perform other computations by typing normal POP-10 commands (see Section 3.4).

When the break is terminated (by typing ↑Z), the system restarts at the top level (or at the current superior break if there is one active). Now, however, control leaves any functions being abandoned with a normal jumpout action, so their local variables are unwound properly.

If ↑Z is typed at the top level (not inside a break) then the top level SETPOP or SETEDIT is simply restarted, instead of causing an error.

2.0 SUMMARY OF CHANGES FROM POP-2

The changes from strict POP-2 are mainly extensions which should not affect existing POP2 programs (provided the new standard identifiers do not clash). However, there are some changes which alter statements in the Reference Manual. Additionally, there are some facilities and features in this POP-10 system which are not strictly part of the language itself, such as the POPMESS facilities and the ↑C interrupt facility mentioned above. These will be described later.

The changes from Reference POP-2 are:

1. WORDs and identifiers may have up to 127 characters
2. The character-set is 7-bit ASCII
3. WORDS and CSTRIPs have 7-bit components
4. Upper and lower-case letters ARE distinguished
5. Exponents in real numbers are usually indicated by "E"
6. Operations of precedence 1 behave slightly differently
7. SECTIONS have different syntax and semantics

8. Conditionals may appear at the top level

9. AND and OR may conjoin expressions in arbitrary contexts - see Section 2.5

2.1 Long Identifiers

The ability to have identifiers longer than 8 characters may affect some POP-2 programs if such identifiers have been spelled in different ways after the 8th character. For example, "MACRESUL" is no longer equivalent to "MACRESULTS". Variant spellings of long identifiers will probably show up through 'comments' messages when the program is compiled, though unfortunately this cannot be relied upon.

2.2 ASCII Character Set

The change of character set to 7-bit ASCII is prompted by the fact that that is the standard character-set in the DECsystem-10. A few of the characters in the POP-2 Reference Language are not present in the standard ASCII set, particularly the 'subscript-ten' character. On the other hand, there are many more characters available in ASCII, including control characters, two cases of letters, and more punctuation marks. Abolition of the POP-2 character set has avoided many difficulties in programming I/O, both for the user and for the POP-10 system itself. However, all uses of character codes directly by number will have to be changed. Words, identifiers and character strips written directly in to the program will not be affected by this change, since the itemiser (INCHARITEM) will automatically insert the correct character codes.

Upper and lower case letters are now distinguished. ALL standard identifiers in POP-10 are in Upper case letters. There is a case-control facility available in CHARIN similar to that in TECO, and described later; the default state is to force all letters from the terminal to be treated as upper-case, so terminals which generate lower-case characters will not normally cause difficulties.

Because not all characters in the POP-2 reference language are actually present in standard ASCII, certain changes have been required in the syntax for individual text items.

An exponent in a real number is now normally signalled by the letter "E", but this choice of character can be changed (with the POPMESS-SUBTEN facility). This use of "E" gives the same format for reals as FORTRAN uses. E.g.: 1.0E6 is 1 million.

In the POP-2 reference language, the closing string quote character is a grave accent "`". However, this mark is only available on terminals with lower-case, so the two characters "!" and "@" are also treated by POP-10 as closing string quotes. E.g. 'A12 3! is equivalent to 'A12 3@ and to 'A12 3`. When a character string is printed by the function PR, a closing string-quote character has to be printed at the end. The system initially prints a grave accent, which will actually appear as "@" on hard-copy devices without lower-case capability, but the character printed can be changed (e.g. to "!") by the POPMESS-ENDSTRING facility described later.

The itemiser treats all control characters as terminators if encountered while reading a number or word; inside a character string any control characters will be inserted into the string along with the other characters of the string. Outside a string, the control character ↑A has special significance: it starts an 'end of line comment' and all characters from the ↑A up to the next following line-feed will be completely ignored. This facility will make it much easier work to comment POP-10 programs.

```
FUNCTION SUMSQ X Y;   ↑A A function to add sqs of numbers
  ↑A The two numbers are in X & Y
  X*X + Y*Y;   ↑A Multiplication is faster than ↑2
END;
```

Words and character-strips now have 7-bit components. This change permits them to hold ASCII characters, and should not cause any difficulties.

2.3 Operations

Operations of precedence 1 do not conform exactly to the definition given in "Programming in POP-2". They cannot be used as infix operators, and can generally only be written in the same contexts as ordinary non-operation variables. If such an operation identifier (e.g. "Q1") is preceded by a unary minus or followed by a 'dot-operator', then the operation is run first, and the value produced (if any) is then negated or operated on by the dot-operator. Thus

```
-Q1.PR;
```

will print the negative of the value produced by Q1, just as for an ordinary variable in the same context.

2.4 SECTIONS

The definition of SECTIONS given in the 'silver book' has not been followed in this implementation; all POP-2 implementations have conformed to another definition for this facility, which is described in Section 6.

2.5 CONDITIONALS

Conditional statements starting with IF, LOOPIF, UNLESS and UNTIL may be typed at the top level, and are no longer restricted to appearing inside function definitions.

The syntax words AND and OR may now be used to separate expressions in all normal contexts, and are not restricted to appearing in the conditions of conditional statements. This permits such constructions as

```
IF (L.ISLINK AND L.HD="VW")
        OR (L.ISFUNC AND L.FNPROPS/=NIL) THEN..
```

AND and OR take an item from the stack, from the preceding expression, and conditionally evaluate the following expression(s). The scope extends to the following "closing bracket", "THEN", or assignment arrow.

AND evaluates to FALSE if the previous expression produced false, otherwise evaluating the following expression.

OR returns the value of the first expression if that is not FALSE, otherwise evaluating the following expression.

3.Ø OPERATING SYSTEM ASPECTS OF POP-1Ø

3.1 ↑C Intercept Facilities

When control-C is typed on the terminal, the next time POP-1Ø requests input from the terminal the normal execution of the program is interrupted and a special routine takes over. This is called the 'Control-C Intercept'. The intercept routine types the message

```
int:
```

to show that it is waiting for you to type in a control character. When this happens, you must type in one of the following characters, and this will cause the effect indicated in the following table. If you type any other character to the intercept routine, it will be printed back at you and ignored, except that "H" (for "Help") will cause a short reminder to be printed.

<CR>	dismiss interrupt, continue program
↑C	exit to monitor level at once
↑X	exit to monitor level (prints Exit POP1Ø)
↑G	do a SETPOP - restart compiler
↑R	do a "READY" break - same as ↑C .REENTER
↑E	print current calling sequence and continue

↑F do a SETEDIT - restart compiler in edit-mode
↑F does a SETPOP if edit mode is not available

3.2 INIT.POP Facility

If you have a disc file DSK:INIT.POP in your default disc area when POP-10 is started or restarted, then that file will be compiled automatically by the system before SETPOP is called. This makes it easy for you to perform any regular initialisations you require in POP-10, perhaps in the form of compiling your favourite utility functions.

3.3 INPUT FROM THE TERMINAL

POP-10, like POP-2, possesses the standard function CHARIN, which is a character repeater for input from the terminal. CHARIN does not always return precisely the characters typed at the terminal; it responds specially to certain control characters, and you can also enter lower-case characters from an upper-case terminal using the facilities as described below.

Certain characters are treated specially by the monitor when they are typed on a terminal, and the POP-10 program will never be able to receive them directly from the terminal. Those characters are:

↑C which sends control to the intercept routine (see above)
<rubout> and ↑U which control editing of the input line
↑O which controls suppression of output to the terminal
↑R causes retyping of the current input line
↑T causes a USESTAT message to be printed by monitor

The following control characters have a special effect inside CHARIN, as described:

↑Z is translated to <termin> [break char.]
↑F forces SETEDIT (or SETPOP) [not break char]
↑G forces SETPOP [break char]
↑Q 'quotes' the following character
↑V controls lower-case conversion of letters
↑W controls upper-case conversion of letters
↑↑ makes lower-case punctuation mark

The character ↑Z is translated to <termin> by CHARIN, and many POP-10 programs which expect input from the terminal will have their input terminated by typing this control character. It is a break character, and will be received by CHARIN as soon as it is typed.

The characters ↑F and ↑G have the same effect typed in to CHARIN as they do when typed to the control-C intercept routine. However, note that ↑F is not a 'break' character, and therefore has to be followed by (e.g.) <esc> to take effect. Note also that these

characters do not cause erasure of any characters typed previously on the same line; therefore it may often be more appropriate to type ↑C and then the required control character.

The control character ↑Q may be used to 'quote' a control character from the list above; it causes CHARIN to return the following character unchanged. However, this cannot be used to type in the characters ↑C, ↑R, ↑T, ↑O or ↑U directly since the monitor handles these specially. The ↑Q quotes any other character; i.e. it prevents case-conversion of letters.

The control characters ↑V and ↑W control alphabetic case conversion much as in TECO. Initially POP-1Ø does not change the case of any alphabetic characters typed in. Typing ↑W↑W puts CHARIN in Upper-case mode, where it automatically turns all alphabetic characters into upper case. This is the default state. Similarly, typing ↑V↑V will put CHARIN in lower case mode, where all alphabetic characters are turned into lower case.

Typing either ↑W↑V or ↑V↑W will turn off automatic case translation, passing letters to POP-1Ø exactly as typed. Also, irrespective of whether there is automatic case translation in force, a single letter at a time can be turned into upper case by prefixing it with a single ↑W, or into lower case by prefixing it with a single ↑V.

The character ↑↑ (ctrl shift O) is ignored unless the following character is one of "@", "[", "\", "]", "↑", or "←". In that case, the character is converted to its lower-case range equivalent, namely: grave, open brace, vertical bar, close brace, 'tilde', or 'rubout'. (Thus ↑↑← will come through as <rubout>, but this feature will not often be useful.) ↑↑\ is converted to vertical bar "¦".

The function INASCII is similar to CHARIN, but none of these special case conversion facilities are in operation. INASCII returns all characters exactly as typed, except for ↑C, etc.

3.4 The Error/Break Package

3.4.1 Introduction -

The POP-1Ø system has a new package for handling errors and breaks. It is based on the old system and is superficially similar, but much the code has been rewritten, and further changes are planned. The general intention is to approach the convenience of the UCI-LISP system with respect to error-handling and breaks.

The break package provides a facility for suspending execution of a program to examine its state, retaining the ability to continue execution when desired. A Break can be entered in one of four ways:

1. By typing ↑R to the ↑C intercept

2. By giving a REENTER command to the monitor

3. By calling the function POPREADY; or

4. Automatically, when an error occurs.

When an error is detected by the system, the function ERRFUN is called to print an error message, and then (when ERRFUN returns) a BREAK is entered. When the break exits, SETPOP or SETEDIT is called to restart the system at the top level.

If an error is detected during a break, then the system routine POPERROR will call first ERRFUN, then a nested break, and finally control is returned to the superior break, rather than back to the top level. In this case, when control returns to the superior break, the message

readyset

is printed, and the compiler is then restarted in that level of break.

Since breaks may become nested as a result of errors or for other reasons, the prompt string is set in a break to a string of the form "n:: ", where <n> is "Ø" for the topmost level of break, and increases for nested levels.

See Section 8.3 for more advice about changing the treatment of errors in special systems.

3.4.2 Errors. -

When an error occurs, the standard function POPERROR is applied to items giving information about the error. POPERROR can be applied by the user to cause an error. It first calls the function ERRFUN to print an error report. Then, when ERRFUN returns, it may call an automatic break and it then returns control to the top level or to a superior break.

POPERROR is protected, and is used to cause errors. ERRFUN is unprotected, and may be changed in order to obtain non-standard treatment of errors.
The standard value of ERRFUN is the function SYSERR, which is also available in the standard protected variable SYSERR. See Section 8.3 for advice on changing ERRFUN. POPERROR, ERRFUN and SYSERR all take identical arguments, as described in the details of SYSERR in Section 7.

ERRFUN does not normally print a great deal of information: it gives the error number, the culprit(s) and the function(s) being run and compiled, if any, and also the section name when appropriate. If FULLERR is set, it also prints the contents of the user stack, but without disturbing the items in it. In some systems, a standard unprotected variable NICERR is provided. In this case, an explanation of the error number is also printed automatically if NICERR is nonzero.

Then a break is called in most cases. The exceptions are:

1. If a stack overflowed, then no break is called, but control is returned to "readyset" or SETPOP or SETEDIT as appropriate, after printing the calling sequence.

2. If the error is 'no more memory', then no break is entered, and any breaks in existence at the time of the error are also ignored. After ERRFUN returns, the calling sequence is printed and then control is sent back to SETPOP or SETEDIT.

3. If the error is a PED editor error, and PEDSHTERR is true, then no subsidiary break is entered, and control returns to the current break or to the top level.

After the break, control returns to the top level or to a superior break. When this happens, the local variables of all functions being left are unwound. This differs from previous versions of POP-2 and POP-1Ø, where after an error the local variables could still be examined in the variables of their names (it was the global versions of these variables which were smashed).

It will be noted that SYSERR prints neither an explanation of the error (unless NICERR is on), nor the calling sequence at the time of the error. These items of information, and the values of any local variables of interest, may be obtained during the break. Indeed, the calling sequence and local variables cannot be examined once the break has been left (which is why the calling sequence is printed automatically on stack overflow or store exhaustion).

If a stack overflows, the stack concerned is extended slightly to permit the error function to run, but no break is entered. (The stack concerned is restored to its normal length when ERRFUN returns.) If the same stack overflows again during execution of ERRFUN (this cannot normally happen unless ERRFUN has been redefined by the user) then the message

?repeated stack overflow

is printed, and control returns direct to SETPOP or SETEDIT at the top level. However, if it is the auxiliary stack which overflowed catastrophically, then local variables of functions being abandoned are NOT unwound normally.

If an error did not enter a break for some reason, or if you have left the break, you can still get an explanation of the error by calling POPXPLNER.

3.4.3 Breaks -

The break routine POPREADY has now been changed slightly to make it more useful at errors. The changes have actually been obtained mainly by altering the standard value of POPRDYFN.

When the break is first entered, POPRDYFN scans PROGLIST (which has been redefined to come from the terminal) looking for either of the special words "?" or ":".

If you type "?" at the start of a break, an explanation of the most recent error is printed (with POPXPLNER). If you type ":" at the start of a break, the current calling sequence of functions is printed (with POPTRACE).

POPRDYFN stops scanning PROGLIST as soon as it sees any text item other than these, and the rest of the proglist (with any initial question marks and colons removed) is compiled in the break. This makes it easy to examine variables or perform other computations in the break, by typing normal POP-2 commands.

You can leave the break by typing ↑Z or GOON.

A break can be entered by any of the routes listed earlier, and the special treatment of "?" and ":" at the start of the break apply in all of these cases, although this was motivated mainly by the needs of an error break.

If the top level of the system is in edit mode (i.e. SETEDIT) then all breaks compile in edit mode (once POPRDYFN is done). In this case, their prompts are of the form "n!!←" to indicate that.

If a user wishes to change POPRDYFN, it is desirable to make the replacement perform the same job on initial question marks and colons. The current definition is, in effect:

```
FUNCTION POPRDYFN; VARS X;
 LOOP: ITEMREAD()->X;
   IF X="?" THEN .POPXPLNER; 1.NL;
   ELSEIF X=":" THEN .POPTRACE; 1.NL;
    ELSE X::PROGLIST->PROGLIST; 1.NL EXIT
 GOTO LOOP;
END;
```

There may be reason, in some programs, to replace POPRDYFN by another function which performs the same job, but also recognises other code-words as requests for other problem-specific pieces of information.

Note that POPRDYFN uses ITEMREAD to read items. This could cause trouble if a macro is typed at the start of a break, which expands into NONMAC <some macro>, because the NONMAC will be stripped off by ITEMREAD the first time round, and then the following macro will be expanded first thing inside the compiler in POPREADY. The solution is to type a semicolon or some other harmless text item before such a macro, at the start of a break.

On the other hand, a macro (say) "??" could be defined to expand into ? and : and perhaps some other standard information printing call. Then ?? could be typed at the start of an error break, to get all of the desired information.

3.5 System Crashes

If POP1Ø crashes with a message

```
?GCFAIL at user PC nnnnnn
```

or

```
?ILL MEM REF at user PC nnnnnn
```

the job is stopped with the terminal in monitor mode. At this point, it will not be possible to continue or restart; a new R POP1Ø command will have to be given. However, first save the core image with a .SAVE monitor command. The resulting .SAV file can be used by your POP1Ø implementation consultant to determine the problem.

4.Ø POPMESS FACILITIES

The standard function POPMESS is used to communicate with the operating system for various purposes. This includes obtaining all facilities for data and program input and output (except through CHARIN and CHAROUT), and also permits a variety of other facilities. The function POPMESS always takes a list as argument, and the head of the list must be a word - one of a set of control-words recognised by POPMESS. The control-word is actually converted to SIXBIT form inside POPMESS, so it does not matter whether the characters are in upper or lower case, and only the first six characters are significant.

The control words recognised are listed below in turn, with brief notes on their uses. A few of these words are marked with an asterisk (*), which means that that word controls an optional facility in POP-1Ø, and may not be accepted by every implementation.

Some of the POPMESS facilities involve access to files on disk or in other media. in these cases, the sign '-filespec-' means a sequence of list elements comprising one or more of the following:

dev: filenm.ext <prot> proj,prog

e.g.: DSK: RUNIT.POP <8:155> 8:4Ø76,8:352

In this file specification, 'dev', 'filenm' and 'ext' will be either words or character strips containing the requisite characters, and they are converted to SIXBIT format so it does not matter what case the letters are in. Only the first six characters of 'dev' and 'filenm', and only the first three characters of 'ext', are significant.

Of these items, only 'filenm' must be present, and this may be omitted if a device is specified. A 'dev' is signalled by a following colon word, an 'ext' is signalled by a period, a protection code is signalled by a "<" word. The protection code, if present, must be a positive integer between Ø and 8:777; it is usual to use the 8:xxx format for octal integers, since the three octal digits have separate significance. The project and programmer numbers, if present, are separated by a comma, and are normally given as 8: octal integers.

Usually only a filename and extension are given, e.g. RUNIT.POP . In this case, the 'dev' defaults to DSK:, the protection defaults to the system default <Ø55> or <Ø57> (the protection is ignored except in OUT, OUTBAK, OUTCCT, PROTECT and RENAME), and the PPN defaults to the default search path (normally the user's logged-in PPN).

If a comma is included for the PPN, it is possible to omit either or both of the project and programmer numbers, in which case they default to the project or programmer number under which the job is logged in. For example, a job logged in under [4Ø5Ø,352] may access the file TEST.POP[4Ø76,352] with the filespec

[TEST.POP 8:4Ø76,].

Sub-File Directories (SFD'S) may be accessed if the appropriate code is included in POP-1Ø (this is an assembly-time option). The SFD path is specified in the normal way. For example, the file PROG.POP in the SFD BLURBS of the UFD [4Ø5Ø,352] may be accessed with the filespec

PROG.POP 8:4Ø5Ø, 8:352, BLURBS

If the job is logged in under [4Ø5Ø,352], this can be shortened to

PROG.POP,,BLURBS

The special device identifier LIB: may be used ij POP-1Ø file specifications. This provides access to the POP-1Ø program library. For example, the following two expressions both provide input character repeater functions for the same library program:
POPMESS([IN LIB: ALLSORT.LIB]) and LIBRARY([ALLSORT])
This facility, however, also permits such constructions as
POPMESS([RESTORE LIB:POPLER.SVP]);
where access to the library is not for an ordinary character repeater.

The various POPMESS control words follow, grouped according to function.

4.1 FILE CONTROL

IN
POPMESS([IN -filespec-]) returns an input character repeater function for the specified file. Note that the repeater will supply 7-bit ASCII characters, terminated by <termin>.

INSOS *
POPMESS([INSOS -filespec-]) returns an input character repeater function for the specified file, like POPMESS in. However, if the input file has line numbers as created by LINED or SOS, the line numbers are omitted by an INSOS repeater, whereas they are included in the stream of characters from an IN repeater.

OUT
POPMESS([OUT -filespec-]) returns an output character consumer function for the specified new file. The consumer function should be applied to 7-bit ASCII characters, and the new file will be created when the output stream is closed (usually by applying the consumer to <termin>). When the output stream is closed, any old file of the same name will be lost.
If a protection is specified, it is used for the new

---

* means an optional facility.

file. Otherwise the file keeps the old protection if it supersedes a file of the same name, or else it gets the system default protection.

OUTBAK

POPMESS([OUTBAK -filespec-]) is like POPMESS-OUT in creating a character consumer function. In this case, however, when the output stream is closed, any old file of the same name will be renamed to have extension .BAK. (The protection code for the .BAK file will be that specified for the new file, if any, or else the protection the old version had, except that the owner's protection field is reduced to 0.)

If the output device is not the default (DSK:), or if a PPN is specified, or if the extension is .BAK or .TMP, then POPMESS-OUTBAK is treated as POPMESS-OUT, and no attempt is made to preserve an older version.

OUTCCT

POPMESS([OUTCCT -filespec-]) will return an output repeater for the specified file, like POPMESS out. However, this output repeater will translate control characters to Uparrow format, in the same way as CHAROUT does (q.v.). This facility is intended for use when sending files to the line printer, etc., and there is no 'backup' version of OUTCCT.

CLOSE

POPMESS([CLOSE <I/O-function>]) will close the specified input or output stream. The second item in the list may be either an I/O function (i.e. a character repeater or consumer, or a BLOCKIO doublet), or it may be a <word> whose VALOF is such a function. (This option is available in all POPMESS calls which expect an I/O function as an argument.) If the function was a character consumer, this will have the same effect as applying it to <termin>.

BLOCKIO

popmess([blockio -filespec-]) returns a doublet for performing binary block-data transfers between the named file and POP-2 data structures. This facility is available for files on disc, DEC-tape or magnetic tape, but is only encouraged for disc files. It is described in more detail below in Section 4.5.

EXISTS

POPMESS([EXISTS -filespec-]) returns TRUE if the specified file exists and can be looked-up, otherwise it returns FALSE.

LENGTH

POPMESS([LENGTH -filespec-]) returns the number of PDP-10 words written in the specified file (each word 36 bits). If the file is on a DEC-tape, the length result will be rounded up to the next multiple of 127 decimal (giving, by division, the length of the file in blocks written). UNDEF is returned if the file does not exists or cannot be accessed.

LOOKUP *

POPMESS([LOOKUP -filespec-]) returns UNDEF if the file

cannot be accessed or if it does not exists, but otherwise p a list of information about the file. It gives the same information as the "/S" switch in DIRECT. The list is of the form

[<length> <prot> <acc-date> <cr-time> <cr-date> <mode>]

These items are all positive integers: the length in blocks, the protection code (0-8:777), the date of last access, the creation time and date, and the mode in which the file was written. The date integers are in the form

((year-1964)*12 + month-1)*31 + day-1

and the <cr-time> is in minutes after midnight.

RENAME

POPMESS([RENAME -newfilespec- = -oldfilespec-]) looks up the file -oldfilespec- and renames it according to -newfilespec-. This can be used to change protection, or more generally to rename the file, very much like the .RENAME monitor command.

Note that if you wish to specify the 'dev:' explicitly, this must be done in -oldfilespec-, which differs from the .RENAME command. A file cannot be renamed into a different file-structure, though on DSK: it can be renamed into a different PPN if you have the necessary privileges.

If a protection is specified, it is used, but otherwise the file's protection remains unchanged.

PROTECT

POPMESS([PROTECT <filespec>] looks up the specified file, and then changes its protection to that specified in the filespec. If no protection code is specified, then the protection is set to zero.

E.g.: POPMESS([PROTECT INIT.POP <8:155>]);

DELETE

POPMESS([DELETE -filespec-]) will delete the specified file from its file-structure.

COMPILE

POPMESS([COMPILE -filespec-])

is equivalent to

COMPILE([FILESPEC])

and causes the text of the specified file to be compiled as a POP-2 program. Note that the input character repeater argument of the function COMPILE is available as the standard variable POPCREP, and may be found there during the break after a compiling error.

EDIT *

POPMESS([EDIT -filespec-]) is usually omitted, but if present calls the SEQED editor to edit the file. This editor is essentially the old LIB POPEDIT slightly improved. The PED-editor is also available in POP-10.

WIDTH *

POPMESS([WIDTH <i/o-function> <width?>]) allows you to read or specify the page-width for an output stream. Normally an output stream will include exactly those ASCII characters sent to the consumer function. However, if a width is specified, then extra new-lines will

automatically be inserted by the output consumer if necessary. If this facility is included, then CHAROUT has the same width in POP-1Ø and the Monitor. The width of CHAROUT may be changed with either the POPMESS-WIDTH facility or with POPMESS-TTY. All other consumers have a width limit of Ø initially, which means 'no limit'.

If a width limit is specified in POPMESS width, it must be an integer between Ø and 8:77777 inclusive; POPMESS sets the limit and returns with no result.

If no limit count is included in the argument list, then POPMESS returns the current limit, e.g.

```
POPMESS([WIDTH CHAROUT])=>
** 72
```

LNK *

POPMESS([LNK TTYnnn:]) returns a doublet for asynchronous inter-computer communications with another computer using the specified asynchronous line. See Section 4.6 for details.

4.2 INFORMATION AND CONTROL

DATE

POPMESS([DATE]) returns three integers: day, month, year.

DAY

is a synonym for DATE.

TIME

POPMESS([TIME]) returns an integer, the number of milliseconds (real time) since midnight last.

RUNTIME

POPMESS([RUNTIME]) returns two integers: <runtime>, <gctime>. Both times are counted in milliseconds. The <runtime> is the total run-time for the job (CPU time), excluding time spent in garbage collection and storage compaction; the <gctime> is the CPU time spent in garbage collection and storage compaction, i.e. the overhead time for the dynamic storage allocation scheme.

CORE

POPMESS([CORE]) returns two integers: <maxcore>, <core>. Both figures are core-sizes in K-words for the low segment. <core> is the currently allocated space available for the user's data structures and programs, and <maxcore> is the maximum limit to which this can rise (and depends on the computer administration). (The figures are actually 2 or 3K less than the true low segment size, since the system's own area and the stacks are omitted.) POP-1Ø will dynamically alter the amount of space allocated to suit the situation, subject to the upper limit. The user can change the amount allocated directly with:

POPMESS([CORE <size>])
which attempts to change the allocation to the amount requested (in K-words), and returns a truth-value: TRUE if the request was successful.

After such a call POP-10 will not automatically reduce the coresize below the limit specified amount, however much core is subsequently reclaimed by the garbage collector. To reduce the core size again, you must use POPMESS-CORE, or restart.

PPN

popmess([ppn]) returns two integers: <project>, <programmer>. They are the user's logged-in project and programmer numbers.

PPNO

is a synonym for PPN.

PJOB

POPMESS([PJOB]) returns the number of the job currently running.

MTAPE *

POPMESS([MTAPE <I/O-function> <integer>]) performs an MTAPE operation as for the MTAPE MUUO (cf. "DECsystem-10 Monitor Calls"). The I/O-function must have been opened for input or output with a DEC-tape or a magnetic tape. Only certain integers from 0 to 8:17 and 8:100 or 8:101 are valid - see documentation for monitor.

LOCATE *

POPMESS([LOCATE <station>]) is equivalent to the LOCATE monitor command. The specified remote station is selected for further spooled input and output when not over-ridden by explicit device specifications.

DEVCHR

POPMESS([DEVCHR <I/O-function>]) uses the DEVCHR MUUO to get the 'device character' bits for the device involved in the specified I/O function. The left-hand half-word of DEVCHR'S result is returned as an (18-bit) integer on the stack.

GETSTS

POPMESS([GETSTS <I/O-function>]) uses the GETSTS MUUO to get the status bits for the device involved in the given input/output channel. The 18-bit result is returned as an integer.

ENDSTR

POPMESS([ENDSTR <ASCII code>]) changes the character printed by PR at the end of printing a string constant argument. Initially it is 8:140 which is lower case grave, but with some teletypes 8:41 = "!" may be preferred.

SUBTEN

POPMESS([SUBTEN <ASCII code>]) changes the character used as <subten> for real-number exponents. Initially this is 8:105 ="E", but 8:077 = "?" may be required for reading in data files created by POP-2 systems.

4.3 JOB CONTROL FACILITIES

SAVE

POPMESS([SAVE -filespec-]) creates a new file to the given specification, and writes in it a complete copy of the user's current program and data-structures. It then returns with the result FALSE to show that the program has just been saved. A subsequent call of POPMESS RESTORE will cause the saved core image to be reloaded from the file, and execution will continue exactly as before, exiting from the call of POPMESS SAVE, except that POPMESS now returns TRUE to show that the program has just been restored. After a program has been restored, there will be no input or output streams active, except for CHARIN and CHAROUT. Any I/O functions saved in the SAVE-FILE will appear to have been closed after the file is restored (though they will remain active through the original SAVE-ING of the program).

The preferred extension for files made by SAVE is .SVP (SaVe-Pop). The call will normally resemble this example:

```
IF POPMESS([SAVE thispr.SVP]) THEN restartfn() CLOSE;
```

where "restartfn" stands for some function to restart the saved program.

RESTORE

POPMESS([RESTORE -filespec-]) restores a core-image of a POP-10 program previously made with POPMESS SAVE, as outlined above. In general, a SAVE file cannot be RESTORED in any POP-10 system except the one in which it was created.

RESTART

POPMESS([RESTART]) will cause the POP-10 system to be re-initialised exactly as when it is first started. The file INIT.POP will be compiled if it exists, and then SETPOP will be called.

EXIT

POPMESS([EXIT]) will return the terminal to monitor mode, after printing the message:

```
Exit POP10
```

Control will return from POPMESS with no result if the CONT command is given to the monitor. This call of POPMESS is made automatically if ↑X is typed to the ↑C intercept routine (see earlier).

HIBER *

POPMESS([HIBER <sleep time> <wake bits>]) will cause the job to hibernate for the specified length of time or until a Wake condition is satisfied. The sleep time is given in milliseconds, and is subject to a maximum of about 68 seconds. A sleep time of zero means indefinite sleep.

The wake bits are encoded as follows (see the Monitor Calls Manual for more information):

bit 8:10 = wake on character ready (HB.RTC)
8:20 = wake on line of input ready (HB.RTL)

8:40 = wake on PTY activity (HB.RPT)
8:400000 causes job to be swapped out at once (HB.SWP)
No other wake bits are allowed.
Any hibernate will be stopped if interrupted by a ↑C intercept. POPMESS([HIBER 0 0]); causes an indefinite sleep.

4.4 TERMINAL CONTROL

CLRBFI
POPMESS([CLRBFI]) will clear the terminal input buffer of all characters typed ahead. This is useful sometimes in recovering from error situations, and is done automatically by SYSERR.

POPTTON
POPMESS([POPTTON]) turns off ↑O print suppression. That is, anything printed by the program just after this call of POPMESS will actually get printed even if the user types ↑O. This call of POPMESS does not exit until the terminal output buffer is empty, in case the user types ↑O after POPMESS was entered but while the output buffer still has output queued for printing. This call of POPMESS is made at the start of SYSERR to ensure that the user gets the error message.

PROMPT
POPMESS([PROMPT <cstrip>]) changes the prompt printed by CHARIN at the start of each line of input. The normal prompt is ': @, but this facility permits that to be changed. POPMESS returns the <cstrip> of the old prompt as result, in case the original prompt is to be restored later. It is possible to get the same effect by applying POPMESS([PROMPT]); with the required cstrip on the user stack. This is equivalent to the above call of POPMESS, and no longer causes an error. The old prompt string is still returned on the stack.

TTYIN
POPMESS([TTYIN]) does a SKPINL UUO, and returns TRUE if there is a complete line already typed ready for reading from the terminal by CHARIN. Otherwise it returns FALSE. This call of POPMESS also has the side-effect of restoring printing if ↑O has been typed, but this is subject to a timing problem if the output buffer is not empty.

TTY *
POPMESS([TTY <codeword> <integer?>]) provides the facilities of the SET TTY monitor command. If the integer is omitted, it defaults to 1, which means TRUE if it is a boolean switch. The codewords recognised are:

```
ALT     as SET TTY ALTMODE
BLANKS  as SET TTY BLANKS
CRLF    as SET TTY CRLF
ECHO    as SET TTY ECHO
```

FILL as SET TTY FILL - takes a count 0 to 3
FORM as SET TTY FORM
GAG as SET TTY GAG
LC as SET TTY LC
UC as SET TTY UC or NO LC
PAGE as SET TTY PAGE
TAB as SET TTY TAB
WIDTH as SET TTY WIDTH - takes a count 16 to 200

All these, apart from FILL and WIDTH, take a boolean argument. Specify the integer zero to turn the facility off.
POPMESS([TTY LC]); is equivalent to .SET TTY LC
POPMESS([TTY LC 0]); is equivalent to .SET TTY NO LC
and similarly for the others.

SETTTY *
is a synonym for TTY.

PEDMARK *
POPMESS([PEDMARK char-code]) is provided if the PED editor is present. Verify commands print a marker character at the position of the buffer pointer, and this POPMESS facility changes the character printed. Initially the character is "↑", but 8:12, for instance, will cause a line-feed to be printed. Zero will stop printing of a marker.

TTYTAPE *
POPMESS([TTYTAPE <truthvalue>]) is optional, and controls the use of the auxiliary paper-tape reader provided on some terminals. The effect is similar to that of the .SET TTY (NO) TAPE monitor command. If the truth-value is TRUE, then paper-tape mode is enabled, and reading of a tape will start when ↑Q is next typed on the terminal, and stops when ↑S is encountered on the tape. If ↑S is punched repeatedly throughout the tape, you will have to keep typing ↑Q each time to restart reading, but this does no harm otherwise. If the truth-value is FALSE, then the reader is stopped, the paper tape mode is disabled, and typing ↑Q should no longer start the reader. If the TTYTAPE facility is included, the paper tape mode is automatically turned off by SYSERR and SETPOP.

4.5 POPMESS-BLOCKIO

POPMESS can be used to create a 'BLOCKIO' doublet for binary transfers between a file and POP-2 data structures. The file must be on disk, DEC-tape or magnetic tape. The doublet is created by a call of the form:

POPMESS([BLOCKIO -filespec-]) => <biofn>

That expression returns a doublet function <biofn>. As a selector, <biofn> will read part or all of the disc-file into a data structure (a record, strip or array), starting at a specified block.

Its updater will write the contents of a data structure into the disc file. If the -filespec- given to POPMESS identifies a file which already exists, then the doublet's updater will write into that file in update mode; or will give a suitable error message if it only has read access to that file. If there is no such file, then a new file will be created, initially empty, and then that file will be accessed in update mode. An error is caused if the file does not exist and cannot be created.

The blockio-function is used thus:

<biofn>(<structure>, <block no.>) => <block no.>

The block number supplied is the logical block number at which reading is to start. It must be a positive integer between 0 and 8:777777. Block 1 is the first block of the file. The block number 0 should not be used on the first read, but subsequently causes reading to continue at the next block after the previous read. On disc, any other block number causes a USETI UUO to that block, which enables random access to blocks within a disc file. On DEC-tape, a block number greater than 1 similarly causes a USETI, but note that this accesses a physical block on the DEC-tape, not a Logical block, and this facility is not suitable for random access to blocks of a DEC-tape file. On magnetic tape, the block number is always ignored, permitting only sequential access to records written on the tape. There is a qualification to this remark: use of the POPMESS MTAPE facility may permit a more flexible use of magnetic tapes, but this facility must be treated with caution.

The <structure> may be any record, strip or array, provided that it does not contain any compound items after filling from the disc. This is always the case if all components are simple packed items of size 1 to 34, but there will be an error if there is a COMPND component. The system checks that there are no compound items if the structure contains full-item components. The structure may not be a <word>.

The transfer from the file into the structure will read 1 or more blocks, the last perhaps only partially. If the file is on disc, then the <block number> returned will be the logical number of the first block following those read. If the file is on DEC-tape or magnetic tape, then the result 0 is returned. Note, however, that in any case, the result <termin> is returned if the transfer attempted to read beyond the end of the file. In this last case, any complete blocks from the starting block to the end of the file will have been read into the <structure> (if it is more than 1 block long), but the remainder of the structure's contents will not have been changed.

On output, the updater of <biofn> works similarly:

<structure> -> <biofn>(<block no.>) => <block no.>

The structure and block number supplied are interpreted exactly as for input, except that the structure is checked for compound items

before the transfer takes place, instead of afterwards. A nonzero block-number to DSK: or DTA: will be applied with USETO before output takes place, to select the block at which writing starts. The transfer will lengthen the file if it writes beyond the previous end, so <termin> will never be returned as the "new block number". The block number 8:777777 for output to a disc file will cause new blocks to be appended to the end of the file (see description of the USETO UUO).

The usual method of reading or writing a file containing more than one structure is as follows. The first structure is written or read specifying Block 1. The <biofn> returns a block number when used, and this is used for the next transfer; the block number returned then is used next, and so on. This reads or writes the file sequentially, and the technique works for any BLOCKIO file.

4.6 POPMESS-LNK

The call

POPMESS([LNK -filespec-]) => <link doublet>

returns a doublet for communicating with another processor over an asynchronous teletype line. The teletype line must already have been assigned by the monitor ASSIGN command, and the other computer must be connected and ready. The '-filespec-' should specify a device TTYnnn: which will be the line to use. There are no facilities in POP10 for automatic dialling.

The selector function of the doublet will be for reading data from the other computer - it is a 'byte repeater'. The updater function of the doublet will be for sending data to the other computer. It is a 'byte consumer function'. All communications take place by transmitting 'Logical Files' from one computer to another. A logical file comprises a stream of 0 or more arbitrary 8-bit binary bytes (expressed as integers in the range 0-255) terminated by TERMIN. ASCII text may be transmitted by sending one character in each successive byte. In this case, the 'parity' bit of each byte is normally kept to zero, and the byte repeater and consumer functions then act just like ordinary character repeater and consumer functions.

A 'link doublet' is only capable of transmitting one 'logical file' at a time over a link. At the end of each logical file (TERMIN received or sent), a new logical file may be started, and the new one may be in either direction. In applications with cooperating computers, it is probably best if successive logical files are transmitted in alternating directions, but this is not essential. At the start of a new logical file, the direction of transmission is decided by whether the repeater or the consumer is called next.

Certain warning messages may be printed on the terminal to give information about the state of the link. These messages may be suppressed by setting FALSE into the variable POPLNKWARNINGS.

A logical file is split up into a sequence of "messages" for transmission. Each message carries up to 71 data bytes of the logical file, and the last message of the file also carries a bit to indicate EOF. Each message includes CRC-16 checksums, and must be acknowledged by the receiving station before the sender goes on to the next message of the file.

Details of the protocol are given in the file LINKS.DOC.

The warning messages controlled by POPLNKWARNINGS are:

```
[waiting for message]
[waiting for ACK/NAK]
<bell>   still waiting for one of the above
[stx missing]
[etx missing]
[checksum error]
[line error]
```

In addition, there are various errors that may occur, but these usually indicate that the line has failed after several attempts to transmit data. See Appendix A for the error messages.

See the "INTERDATA 7/32 Users Guide" for details of programming the Interdata 7/32 end of the link. Note that if the Interdata is receiving in ASCII mode, it will receive only complete lines of alphanumeric text. Lines are terminated by carriage returns (not line feeds) in the Interdata system, and linefeeds are ignored. Therefore, 2.NL; to the link will have only the same effect as 1.NL; since the NL function only sends one carriage return character. Also, a carriage return must be transmitted at the end of the last line of text transmitted, before the TERMIN. Note also that when the Interdata is receiving in ASCII mode, it will read a line into a buffer preallocated in user address space, and that very long lines of text may be truncated. In general, 80 to 120 characters per line is the most that can be relied upon to be received without truncation. No error warning is given if a line is truncated for this reason.

5.Ø INCHARITEM

INCHARITEM is the function which produces an input item repeater from an input character repeater.

INCHARITEM E <character repeater> => <item repeater>

Because POP-1Ø uses 7-bit ASCII with 128 characters possible, compared with only 64 characters in POP-2, some changes have been needed. The syntax definitions for the various types of character groups: <integer>, <real>, <unquoted word> and <string constant>, are changed only minimally.

1. The syntax for integers is unchanged.

2. The syntax for reals is unchanged except for the <subten> character. This character is initially "E", but may be changed to any other character by using POPMESS-SUBTEN, giving the ASCII code of the desired <subten>.

3. The syntax for unquoted words is almost unchanged.

    1. In alphanumeric words, upper and lower case letters are distinguished, and any character is treated as a <letter> if it is preceded by the <vertical bar> "¦" = lower-case "\" = code 8:174. This character may be typed in, in CHARIN, by the combination "↑↑\". This feature permits unusually spelled identifiers to be typed into programs. The character "#" may also be used in alphanumeric identifiers; it is treated as a <letter>, not as a sign. E.g. "NOUN#PHRASE".

    2. The list of <sign> characters has been extended and adjusted:
       <sign> ::= + - * / $ & = < > : ? \ ↑ ← ~

    3. The lower-case braces { and } are also <brackets>, and may be decorated: {% and %}.

    4. The closing string-quote characters are "!", "@", and grave "`".

4. All control characters, and <space> and <rubout>, act as character-group separators when encountered outside string constants, but are preserved unchanged inside string constants (and of course act as <letter>s if preceded by code 8:174 outside of string constants).

5. The control-character ↑A is special in that it introduces an 'end of line comment' if encountered outside a string constant (and not immediately preceded by code 8:174). The previous character group is terminated by ↑A, and then the ↑A and all characters up to the next <line feed> are

ignored.

Note that this behaviour of ↑A is a feature of the itemiser, and not a feature of CHARIN. ↑A is not treated specially by CHARIN, but conversely, this end-of-line comment feature is available while compiling files from any source.

6. Inside a string, the string quote characters may be treated as normal characters, by prefixing them with a "¦" "quote" character. The "quote" character causes the following character to be included in the string; e.g. '¦'@.

6.Ø SECTIONS

The definition of Sections has been changed from that given in the 'Silver Book', to conform with other implementations of POP-2 available.

The syntax for SECTION is changed slightly:

```
<exporteds> ::=  => <declaration list element*?>
<importeds> ::=  <declaration list element*?>
<section name> ::=  <identifier>
<section header> ::=  SECTION ;  ¦
        SECTION <section name> <importeds?> <exporteds?> ;
<section> ::=  <section header> <program?> ENDSECTION
```

If "SECTION" is followed immediately by ";", then the section is said to be 'anonymous', and it has no 'imported' or 'exported' variables. Otherwise, the identifier immediately following "SECTION" is declared as a macro, as explained later, and becomes an 'exported' variable in addition to any other exporteds specified. All standard variables are automatically accessible inside and outside the section.

The effect of SECTION is as follows. All variables whose identifiers are 'imported' or 'exported' have a scope which extends both inside and outside the section, and those variables may be freely accessed everywhere. Variables declared outside the section and whose identifiers are neither 'imported' nor 'exported' are not accessible inside the section, and the same identifier may be used for completely different variables inside and outside. Similarly, any variable declared inside the section whose identifier is not 'imported' nor 'exported' cannot be accessed outside the section. If an undeclared variable is created automatically inside a section, then that variable only has scope within that section. Mention of an identifier as 'imported' or 'exported' automatically amounts to a declaration of a variable with the specified properties (if any). if

the variable has not already been declared.

The Section Name (if any) is declared as a macro which expands to a sequence of all the Exporteds, any exporteds which are macros being prefixed by the word "NONMAC". The only difference between the 'importeds' and 'exporteds' is that the former are not included in the expansion of the section name. Note that the statement CANCEL <section name>; will cause every exported of the section (including the section name itself) to be cancelled.

No error will be caused if an identifier is included in the importeds or exporteds lists which is already declared as an operation or macro but is included without any property specification in the list. This permits <section name> to be included in the importeds or exporteds list of another section, to make all externals of this section importeds or exporteds of the other. Similarly, no error is caused if an identifier is included in the importeds or exporteds lists of a section which has already been declared, and which has also been 'protected' with POPROTECT.

This change to the definition of Sections is consistent with the examples given in the reference manual, which still work as described.

The implementation of sections has been radically changed, and the following features are now available.

If a <saved-state> is created inside a section, it saves the complete section dictionary for identifiers. When the saved-state is subsequently reinstated, the section is re-entered.

A section may be left by compiling "ENDSECTION" or by jumping out of the POPVAL with a jumpout function (or by calling SETPOP). The section is not cancelled when this is done, but only when a subsequent garbage collection collects the dictionary after any pointers to it (in savedstates) have been lost. When the section is cancelled, every internal identifier of the section is cancelled. This has two possible side-effects which can be detected subsequently.

If an identifier is cancelled and at that time its VALOF is the pair [WD . UNDEF] for that particular word, then the VALOF is changed to UNDEF.

If an identifier is cancelled and at that time its VALOF is a <function> whose FNPROPS is the list [WD] for that particular word, then the FNPROPS is changed to NIL.

If a pointer to a <word> is held by the program and that word was currently associated with any variable at the time it was created (by CONSWORD or the itemiser), then VALOF will always subsequently access that particular variable irrespective of sections and cancellations. That is, if (through use of SECTION or CANCEL) there are two different variables whose identifiers have the same characters, then those variables have different word-cells associated with themselves in core. The different word-cells with the same characters are "=", but not "EQ".

7.Ø STANDARD IDENTIFIERS

7.1 POP-2 Standard Identifiers

In this section, the new and changed standard identifiers are described. All of the standard identifiers defined in the POP-2 Reference Manual are present, and are unchanged except for the following (which are altered as described).

AND	see "conditionals"
CHARIN	changed to use ASCII
CHAROUT	changed to use ASCII
CHARWORD	changed to 7-bit characters, and longer words
COMPILE	will accept filename or character repeater
CONSWORD	changed to 7-bit characters, and longer words
CUCHAROUT	changed to use ASCII
DATALENGTH	also works on arrays
DATALIST	also works on arrays (see APPDATA below)
DESTWORD	changed to 7-bit characters, and longer words
ERRFUN	see Section 3.4
IF	can appear at top level
INCHARITEM	changed as described above
INITC	changed to 7-bit components
LOOPIF	can appear at top level
OPERATION	operations of precedence 1 slightly changed
POPVAL	null PROGLIST accepted as [GOON]
OR	see "conditionals"
SECTION	definition changed
SETPOP	does a jumpout
SUBSCRC	changed to 7-bit components

7.2 "Optional Functions"

The identifiers given in Appendix 2 to the Reference Manual "Optional Functions" are almost all included, with a few alterations in detail. The following are included:

/= ARCTAN APPLIST APPLY APPDATA COPYLIST COREUSED COS EQUAL EXP FNCOMP LENGTH LIBRARY LISTREAD LOG NUMBERREAD PRBIN PROCT REV SIN SQRT TAN VALOF

However, there are a few changes:

COPYLIST copies a list at the top level only. COPYLIST(L) gives the same result as L<>NIL .

COREUSED is an operation of precedence 1 instead of an ordinary variable.

VALOF is not identical to the definition given in the Silver Book.
When VALOF is applied to a word item, it accesses the

variable currently associated with that wordcell at the time that the word-cell was created, even if that variable has subsequently been cancelled. This ensures that cancellation of variables (for instance by ENDSECTION) will not affect the behaviour of programs already compiled which use VALOF. (Similar remarks apply to IDENTPROPS and POPIDENTYPE.)

FNCOMP now returns a doublet if its second argument is a doublet. The updater of F1 FNCOMP D2 will apply the function F1 and the updater of D2. Otherwise, FNCOMP is unchanged.

APPDATA is implemented in a more efficient way than that suggested in the "Silver Book", and it applies the function to all components of the data structure without first using any memory to build a list of the values. This not only reduces the load on the storage allocation system, but it also means that this function works for arbitarily long data structures: there is no risk of the stack overflowing during construction of the datalist.

APPDATA can be used in conjunction with DATALENGTH and a new function DATASIZE (see below). All of these functions, when given an array, will operate on the strip used internally in the array to hold the elements. APPDATA and DATALENGTH, when applied to a closure function, will operate on the frozen values, since the DATALIST of a closure is defined to be a list of the frozen values. The DATALIST of any other function (than arrays and closures) is just its FNPROPS, so APPDATA when given a function item will APPLIST its second argument to the FNPROPS of that function.

Aside from such ordinary functions, the DATALIST of an item X is actually calculated essentially as [% APPDATA(X,IDENTFN) %]. Thus stack overflow can occur if you try to form the DATALIST of a long data structure or array.

The following variables are not provided:

CARRYON, POPTIME

POPTIME is replaced by POPMESS-RUNTIME.

The facilities of the library program LIB DEBUG are provided as standard in POP-10. There are therefore the following variables:

Macros: BUG, UNBUG

Variables: DEBPR, DEBSP, DEBUG

DEBSP is reset to 0 by SETPOP and SETEDIT, and DEBUG is initially TRUE.

The DEBUG facility has been modified by inclusion of a variable DEBCHAROUT which should hold an output character consumer function. All debug printing is sent to the consumer in this variable, instead of to CUCHAROUT as previously. DEBCHAROUT is initialised to contain CHAROUT, but it can be assigned to, and it is not reset to CHAROUT by SETPOP and SETEDIT. You may, of course, assign the same new consumer function to both of CUCHAROUT and DEBCHAROUT.

7.3 New Standard Identifiers

The following new standard identifiers are provided in POP-10, in addition to those referred to above:

```
<*, *>          macros for iterations
<? ?>           macros for iterations
COREFREE        Operation 1: returns no. of words of store free
DATASIZE        Function: gives size of components of data structure
EQ              function: fast "="
ERASE2          function: remove two items from stack
EXPPR           function: EXPPR(N,DIGITS) prints n in exponent form
FBACK           function: fast non-checking BACK
FDESTPAIR       function: fast non-checking DESTPAIR
FFRONT          function: fast non-checking FRONT
FULLERR         default TRUE: when set, SYSERR prints user stack
INASCII         function: CHARIN but CTRL chars not translated
INTPR           function: INTPR(N,DIGITS) prints n as decimal integer
ISARRAY         function: TRUE if applied to <array>, else FALSE
ISNUMBER        function: TRUE if applied to number, else FALSE
LOGXOR          function: forms XOR of 2 bitstrings, cf. LOGOR
NEGATE          function: unary negation
OUTASCII        function: CHAROUT but ctrl chars not translated
POPARRPR        default PR: item-printer for print-arrow
POPCOMMENTS     default TRUE: if false, no 'comments' messages
POPCREP         holds repeater argument of COMPILE
POPCTIDY        function: causes a garbage collection and compaction
POPCTRACE       default FALSE: if set, messages at garbages, etc.
POPERRNUM       general: holds number of most recent error
POPERROR        function: calls standard error-handling routines
POPEXECUTE      protected boolean: TRUE when at execute level
POPGETITEM      function: get item from PROGLIST, no expansions
POPIDENTYPE     function: return 18-bit type-code for <word>
POPLNKWARNINGS  default TRUE; set for POPMESS-LNK messages
POPPIP          function: copy from input to output streams
POPRDYFN        function: see description of POPREADY
POPREADY        function: called by by ↑R break
POPROTECT       syntax: set identifiers as 'protected'
POPSETFN        default IDENTFN: called during SETPOP
POPTRACE        function: prints current function-calling sequence
POPXPLNER       function: prints explanation of most recent error
SYSERR          function: standard value of ERRFUN
UNLESS          syntax: like   IF (NOT ..)
UNTIL           syntax: like   LOOPIF NOT( ..)
```

Certain other standard variables exist if the PED Editor option is included in the POP-10 system; they are documented in the PED.MEM file. Most of them start off with the letters "PED", to avoid difficulties with identifiers used by users' programs. A few start "POP..", and the one other identifier is SETEDIT.

Further notes follow on some of the standard variables listed above.

Iteration Macros

The text: <expression> <* -code- *> will cause the program -code- to be performed <expression> times. E.g.: 4<*pr(1)*> will cause printing of ' 1 1 1 1'.

The text: <? <condition> THEN -code- ?> will cause the program -code- to be repeated so long as <condition> evaluates to a non-false item. E.g: <?L.ISLINK THEN FUN(L.DEST->L) ?> will cause FUN to be applied to every member of the list L.

These macros are useful because they permit iterations to be concisely typed at execute level - i.e. outside functions. They expand to fixed text as follows:

"<*" to " ;LAMBDA POPLIMIT; VARS POPCOUNT;
FORALL POPCOUNT 1 1 POPLIMIT "
"<?" to " ;LAMBDA; LOOPIF "
"*>" and "?>" to " CLOSE END.APPLY; "

COMPILE may be applied either to an input character repeater function, or to a filename (a list). In the latter case, POPMESS-INSOS is used to make an input repeater. The repeater obtained will be kept in POPCREP, which is local to COMPILE. Similar remarks apply to PEDITFROM if it is present. (POPMESS-IN is used if the INSOS facility was omitted from the POP-1Ø system.)

COREFREE and COREUSED are both operations of precedence 1. They each perform a garbage collection, and then return the number of PDP-1Ø words of core free or in use.

DATALIST and DATALENGTH have been fixed to work sensibly when applied to array functions (those made by NEWARRAY or NEWANYARRAY). They return a list of, and the number of, all components in the array. See description of APPDATA above.

DATASIZE is a function which may be applied to any item, and which returns information about the sizes of its components (if any). If applied to a strip, it returns the size as specified to STRIPFNS for the strip class. This size is Ø for full strips made by INIT, and is 7 for character strips. DATASIZE also returns 7 for a wordcell, since its character components are of size 7.

DATASIZE when applied to a record returns a copy of the specification list originally supplied to RECORDFNS for the record class. Thus the datasize of a list cell is [Ø Ø] since it has two full-size components.

If DATASIZE is applied to an array, it returns the datasize for the strip class being used in that array (see NEWANYARRAY). A NEWARRAY array has DATASIZE Ø.

The DATASIZE of anything else is UNDEF. In general, an item has a DATALENGTH and DATALIST, and can be used in APPDATA, if and

only if either its DATASIZE is not UNDEF or it is a closure function. (To recognise closure functions, you can use SAMEDATA, since SAMEDATA regards closure functions as the same data class and as different from other functions.)

EQ is nearly equivalent to "=". The difference is that words with the same characters will sometimes not be "EQ", though they will always be "=". EQ is faster than "=" for comparing numbers and non-words.

FBACK, FFRONT and FDESTPAIR
These functions are equivalent to BACK, FRONT and DESTPAIR respectively when applied to <pair>s. However, they do not check their arguments, and strange results may be produced if they are applied to non-pairs. They are somewhat faster than BACK, etc., but must be treated with care. If they are applied to numbers, then an ILL MEM REF crash may result. This is the consequence of attempting to access memory outside the permitted address space.

FULLERR is a boolean variable which controls printing of information by SYSERR. It is initially TRUE, and causes SYSERR to print the contents of the user stack. If set false, SYSERR no longer prints this information. In any case, SYSERR does not remove any items from the user stack, except for the error number and culprits.

OUTASCII and CHAROUT are both functions to send an ASCII character to the user's terminal. They perform identically to each other except when applied to a control character (a code less than 8:40). OUTASCII sends all control codes to the terminal exactly as supplied. CHAROUT, on the other hand, translates most control codes into 'arrow' format. For example, CHAROUT(1); will print '↑A'. CHAROUT prints all control codes in arrow format except for the following:
<bell> <tab> <linefeed> <vt> <form> and <CR>.
Note, in particular, that CHAROUT(0); prints '↑@'. CHAROUT's behaviour with control characters is the same as that of POPMESS OUTCCT character consumers. In contrast, consumers made by POPMESS-OUT and POPMESS-OUTBAK behave like OUTASCII and send all characters as given (except 0, which they ignore).

POPARRPR is an unprotected function which initially contains PR. It should contain an item-printing function which sends characters to CUCHAROUT. That item printer will be used by "=>" and by SYSERR when printing items.

POPCREP is the formal parameter of the function COMPILE, and it will contain the input repeater being compiled after a compiling error in function COMPILE. To examine the rest of the file after the point where the error was detected, type:

```
POPPIP(POPCREP,CHAROUT);
```

You have to do this during the break which follows the error report. (However, if you are getting compile-time errors, it may be more convenient to compile the program from the editor buffer

with PCOMP.)

POPCTRACE is a boolean variable, initially FALSE. It may be assigned to. When it is nonzero, messages are printed at garbage collections and when the store allocation is changed. '[garbage nnnn]' is printed at a garbage collection, where 'nnnn' is the number of words of store free, in decimal. '[compacted]' is printed at a storage compaction. '[core nn]' is printed when the store allocation is changed, <nn> being the allocation in Kwords, in decimal.

POPERRNUM normally contains the number of the most recent error (if any). It is set by SYSERR, but can be assigned to by the user. Its value is used by POPXPLNER to select the explanation to be printed.

POPERROR is a protected function to call an error and dispatch properly afterwards. It calls ERRFUN (which usually contains SYSERR), and then enters a break if this is appropriate. Then it returns control to the top level or to a higher level break, according to the rules given in Section 3.4. The calling sequence is the same as for SYSERR (q.v.). If POPERROR is called with zero error number, then it does not call errfun but immediately goes on to enter a break (if needed) and then to restart the compiler. This can be useful when building special programs for handling errors.

POPEXECUTE is a protected boolean variable, but is only useful inside macros at compile time. It enables the macro to detect whether it has been encountered inside or outside a function body. It returns TRUE if the macro is being called at "Execute Level" -- i.e. outside any function bodies. It returns FALSE if the macro is inside a function.

POPGETITEM superficially resembles the function ITEMREAD, but it returns a text item from PROGLIST without Macro expansion. It may be defined by:

```
FUNCTION POPGETITEM; PROGLIST.DEST->PROGLIST; END;
```

but is implemented more efficiently. It is actually the function used by ITEMREAD and the compiler to read items from PROGLIST.

There is one situation in which POPGETITEM does not return the item from PROGLIST unchanged. If the compiling is in Edit-mode (see section on PED-editor), then POPGETITEM will return words already prefixed by "PED" when appropriate.

If PROGLIST is null, then POPGETITEM returns "GOON" as a convenience when constructing argument lists for POPVAL; "GOON" need not be provided explicitly. However, in this case, POPGETITEM also assigns zero to PROGLIST, and it gives an error if called while PROGLIST is zero. This error is provoked, for example by an unterminated list constant.

POPIDENTYPE is a system-oriented version of IDENTPROPS. It returns the 18-bit type-code for a word. E.g.: 8:1 is set in a general variable; 8:2 in a variable restricted to functions, 8:40000 is

set in protected identifiers, and 8:100000 in cancelled identifiers.

POPPIP is useful for copying from input streams to output streams. It takes two arguments, an input specification and an output specification. Each specification may consist of a repeater function, a file title, or a list of repeaters and file-titles. If the input specification is a list of files, then they are concatenated in order given to form a single joint input stream. If the output specification is a list of files, then the input stream is copied to all of them in parallel. On input, a file title has "IN" CONSed on front, and then POPMESS is applied. On output, a file title has "OUT" CONSed on front and POPMESS is applied. So

```
POPPIP([A.POP],[B.POP])
```

is equivalent to

```
POPPIP(POPMESS([IN A.POP], POPMESS([OUT B.POP])); .
```

and copies the contents of A.POP into a new file B.POP. The function DTYPE

```
FUNCTION DTYPE FILE; POPPIP(FILE,CHAROUT); END;
```

will print the contents of a file on the terminal.

POPREADY is the function called by ↑R to produce a 'ready' break. It may also be called directly by the user. It first saves the state of the program being suspended (with a jumpout function), and sets up PROGLIST to come from the terminal. Then it runs the function POPRDYFN and finally applies POPVAL to PROGLIST.

POPRDYFN is initially a function which scans the first items on PROGLIST. If it sees a "?", then it applies POPXPLNER to print an explanation of the most recent error.
If it sees a ":", then it applies POPTRACE to print the calling sequence of POP-10 functions.
See Section 3.4.3 for more information.

POPROTECT is a syntax word which is followed by identifiers terminated by a semicolon. The identifiers will be 'protected'. A protected identifier may not be assigned to by VALOF, nor may any assignments to it be compiled. However, the main advantage of protecting an identifier comes when it is the name of a function, operation or macro. If a function call is made from a protected variable, then a direct jump to the function is compiled, without indirection through the variable. This makes for faster function entry. This feature is retrospective in effect: any previously compiled calls of functions with protected identifiers will be changed at the next garbage collection following the use of POPROTECT.

POPSETFN is an unprotected variable. It initially contains IDENTFN, and it is applied by SETPOP after the stacks and system work-space have been cleared, but before the compiler is entered. POPSETFN may be changed to perform any extra re-initialisation required in SETPOP.

POPVAL is changed slightly as a result of modifications to POPGETITEM. It is no longer necessary to provide a "GOON" at the end of the argument list. Thus COMPILE can be defined by:

```
FUNCTION COMPILE POPCREP;
  POPVAL(FNTOLIST(INCHARITEM(POPCREP)));
END;
```

POPXPLNER is a standard function to print an explanation of an error on the terminal. It takes the error number from POPERRNUM, which is set by SYSERR and can also be assigned to by the user.

SETPOP is the standard function which restarts the compiler at the toplevel. It proceeds as follows:
First it jumps out from any currently active functions. That is, instead of simply abandoning them, as previously, it now unwinds their local variables.
Then it stops paper tape mode (if it is on), resets the teletype prompt, and resets DEBSP, ERRFUN, CUCHAROUT and APPSTATE to their standard values.
Next it applies the function POPSETFN, which is initially IDENTFN.
Finally, if POPSETFN returns, it restarts the compiler at the top level, and prints the message "setpop".

SYSERR is the standard value of ERRFUN - the error function. Normally it is called from ERRFUN by the POPERROR routine. It takes 1 or more arguments as follows. The top item on the stack is the error code item. If the error code is a positive integer, then the error number printed will be the remainder modulo 1ØØØ, and the quotient <top item>/1ØØØ will be taken as the number of culprits to print. The culprits will be taken from the stack after the error code item, being printed in the usual "=>" order. The error number is assigned to POPERRNUM.

If the error code item is not a positive integer, then it is printed as the error description, and SYSERR assumes that one culprit is also present on the stack.

To call an error, number (say) 6Ø1, with two culprits c1 and c2, type:

```
POPERROR(C1, C2, 26Ø1);
```

See Section 3.4 for more information on error-handling in the system.

UNLESS is a syntax word to start conditionals in place of "IF". It reverses the tests applied to the conditional expressions. For example,

```
UNLESS A OR B THEN ..
```

is equivalent to

```
IF NOT(A OR B) THEN ..
```

UNTIL is a syntax word to start iterative conditionals in place or LOOPIF. It reverses the sense of the truth-tests, and loops until the condition goes true.
For example:

```
        UNTIL NULL(L) THEN F(L.DEST->L) CLOSE;
```

will apply F to every element of the list L.

8.0 OTHER FEATURES

8.1

1. The functions associated with certain syntax words have been named: this makes it easier to locate compiling errors. The words are:
   SECTION, FUNCTION, MACRO, OPERATION, LAMBDA, (, (%, [, [%, IF, LOOPIF, UNTIL and UNLESS.

2. The function POPTRACE, which prints the function calling sequence, always ignores whichever functions are currently in the standard variables ERRFUN and POPRDYFN. Additionally, it ignores any function with FNPROPS = UNDEF. This reduces unnecessary printing at errors and breaks. See Section 3.4.

3. The debugging system macro BUG replaces the value of each BUG-ed variable with a closure function. BUG now copies the FNPROPS of the original function to the new value, thus preserving its name if printed, and allowing programs which use the FNPROPS to work.

4. The functions SP and NL apply INTOF to their argument, and use the integer result for the count. A real argument will not cause an error. However, if the argument is negative or very large (1000 or more), then nothing is printed. NL(i); actually prints one carriage return followed by i line-feeds.

5. If PR is applied to an invalid pointer, it prints "%" followed by the 12-digit octal number of the 'pointer'.

6. In the POP-2 compiler, a "." is followed by a non-operation identifier to form a <dot operator>. In POP-10, this has been generalised: a period may be followed by a <function> item, still forming a dot operator. In this case, a call of the function is compiled in the usual way. This facility will be of use where macros expand to function calls. The POP-10 compiler, like POP-2, will not (yet) accept a <function> item followed by a parenthesised expression, though this also is an obvious extension.

7. In POPMESS, a file-title and extension may be given as <word>s or character strips, as described in Part 4. Alternatively, a filename or extension may be given as a positive integer, e.g. 8:4076000352. In this case, the integer must be at least 8:1000000; i.e. its left-hand half-word must be nonzero.

This facility is provided so that UFDs can be accessed, as in:

```
POPMESS([BLOCKIO 8:4Ø76ØØØ352 .UFD]);
```

which accesses the UFD for [4Ø76,352]. This facility is further aided by the provision (assumed in that example) that if a file-extension .UFD is specified then the directory [1,1] is assumed for the look-up.

8. In POP-1Ø, if the itemiser finds a string too long to handle, then the partial string read thus far is returned as a culprit if possible. This error commonly arises from a missing closing string quote, and the partial string culprit makes it easier to find the error. If the itemiser reads <termin> while reading a string, the partial string is similarly returned.

9. This system does not create a garbage-stack overflow file unless and until it is needed. If an overflow file is used, the POPCTRACE messages are changed from '[garbage nnnn]' and '[compacted]' to '[garb ov age nnnn]' and '[comp ov acted]'. The garbage file, if created, is called nnnPOP.TMP, where nnn is the job number. This disc file is always deleted by POP-1Ø when you next return to monitor level, excepting only if you return to monitor level by ↑C↑C in the middle of a garbage collection or store compaction. The file is opened on channel Ø, and POP-1Ø never uses channel 17 (since future extensions may require it), and user I/O is restricted to channels 1 to 16 (octal) inclusive.

1Ø. In this system, the compiler reads all text items from PROGLIST with POPGETITEM. This carries a speed advantage when compiling files with COMPILE, for the system does not then go through the full dynamic list solidification process to read a text item, and the load on the storage allocation system is much reduced because of the lower turn-over in list links. There is another advantage in using COMPILE, because POPGETITEM ensures that all word items it returns are in the current dictionary: this is automatically the case inside COMPILE, but direct use of POPVAL on an arbitrary list will cause POPGETITEM to look every word up again in the current dictionary. Incidentally, the user can do this looking up himself by the instructions:

```
.DESTWORD.CONSWORD;
```

11. 'Phase' code is executed from the accumulators, at times, in this system, particularly in JUMPOUT, REINSTATE and APPSTATE. This technique has made these system operations faster. The garbage collector also derives minor benefit from this technique.

8.2 The Optimising Compiler

The POP-10 compiler optimises the compiled code in certain straight forward ways. The intention has been to reduce the amount of core used, but in some cases the speed has been increased as well.

1. A unary minus followed directly by a number compiles simply as the negative of that number - a normal constant. This saves time performing the negation.

2. The code for "FORALL I J K L" is optimised when both J and K are numbers rather than identifiers.

3. Structure constants do not use a separate reference, but instead a UUO to load the pointer on the stack. This saves store at the expense of speed. If speed is important keep the constant in a separate variable.

4. Positive integer constants under 8:1000000 ("short integers") are not kept in a separate reference, but are loaded onto the stack with a UUO. This is much like (3) above.

5. A <condition> in a POP-10 conditional is optimised if it consists of a single variable access, as in
   'IF x then ' or 'IF a or b then '
   This makes a smaller faster program by avoiding a stack-load and unload each time.

6. A sequence of stack-load-unload will be optimised to transfer the item via accumulator 0. This covers variable to variable, X->Y; and short integer to variable 34->Z; . The special case of loading zero is contracted into a single instruction, 0->X; or FALSE->X; . Repetitive transfers from the same source, as in NIL->X; NIL->Y; or 3->Z1;3->Z2; are improved by omission of the second and subsequent load instructions. These optimisations only occur so long as the sequence of instructions is not broken by a label.

7. A compiled call of ERASE or ERASE2 is optimised to a single machine instruction to adjust the stack pointer appropriately; this makes these function calls very fast, but they no longer check for stack underflow.

8. Various other optimisations are now made, especially in for jump instructions and push/pop pairs, as implemented by Arnold Smith. Backward jumps now perform their stack checking by executing an in-line conditional skip instruction instead of by using a UUO (Extra-code).

8.3 Errfun

See also Section 3.4.

8.3.1 Changing ERRFUN -

There are several different ways in which the error system may need to be changed for a special program.

1. Some programs change ERRFUN locally to be a jumpout function (or to call a jumpout function), so that control can be retained if an error occurs in a special context. If ERRFUN exits by a jumpout function, then the break call, etc. are completely bypassed. This can be a useful technique for testing conditions for which there is no direct predicate available (for example - whether a particular file can be written).

2. In some programs, it is desired to have extra printing of information when errors occur. It may be easiest to handle this by redefining POPRDYFN as described in Section 3.4.3, for two reasons:

    1. the value of ERRFUN is reset to SYSERR by SETPOP and SETEDIT, so special action needs to be taken to restore the special ERRFUN after each error;

    2. the less printing that is performed automatically the better (in general) now that there is an easy way to get extra information in those cases where it is needed.

3. In a few programs (e.g. POPLER), a whole new top-level has to be defined separately from the normal SETPOP or SETEDIT, and this special top-level must be restarted after errors. Maybe errors also need to print different information from the normal SYSERR.

    The best way to establish a top level is to put into POPSETFN a function which starts it, and then call SETPOP. SETPOP clears the stacks, etc., and then calls POPSETFN before starting the compiler. It is not necessary that POPSETFN should return - it may be a function which calls the compiler in its own way, or which calls an interpreter for a different language. There are two possibilities:

    1. If POPSETFN is left permanently set to its new value, then after an error, SETPOP will be called by the error routines after ERRFUN and the break, and the system will restart without any special further action being necessary.
        Note that SYSERR is assigned to ERRFUN by SETPOP

each time before POPSETFN is called, so if a non-standard errfun is required then it must be reset by POPSETFN each time.

2. If it is desired to keep both SETPOP and SETEDIT performing their normal functions, then a variation is possible. To start the top level, first temporarily save the normal value of POPSETFN, and assign to POPSETFN the new function to establish the new top level. The first action of the replacement POPSETFN should be to reset the variable POPSETFN itself to its usual value (probably IDENTFN). Then it must change ERRFUN as well. Then SETPOP is called.

ERRFUN may be defined in the general format:

```
FUNCTION ERRFUN;
 <print some information, e.g. by  .SYSERR; >
  .POPREADY;
  <restart the system>
END;
```

The system may be restarted from inside ERRFUN either by a special jumpout function created when the top level routine is established, or it may be restarted by the same trick again with POPSETFN and SETPOP.

It is possible, of course, to use SETEDIT and POPEDFN instead, in the same way. The only difference would be that breaks will be in edit mode.

8.3.2 Other Remarks -

POPTRACE ignores functions with FNPROPS = UNDEF, and this enables "private" functions of a special system to become 'invisible' to users. This facility can be used to reduce the amount of relatively unintelligible output produced at an error.

Sometimes a user-defined ERRFUN may call JUMPOUT to make a jumpout function. For instance, POPREADY calls JUMPOUT. However, JUMPOUT may NOT be applied during ERRFUN if a stack has been extended (errors 31, 32 and 33).

SETPOP and SETEDIT and "readyset" unwind local variables of functions from which they exit, just as a jumpout function does. The only time this does not work is after repeated stack overflow on the auxiliary stack.

However, there is a slight "gap" in the definition of POP-2 - it is also possible to leave functions without unwinding their local variables, by using saved-states. If reinstating a saved-state causes control to leave a particular function, then the locals of that function are not unwound properly unless they are also locals of the functions into which control is being transferred. In POPLER 1.5 it has actually been necessary always to perform a jumpout to the level of the barrier before reinstating any saved-state. If POP-1Ø is reimplemented with a different control structure, then this "gap" in the definition of saved-states will be corrected.

9.Ø STORE USED AND SPEED

POP-1Ø runs with a shared high segment of 14 K-words, and a low segment for each user - minimum of 3 K-words. (These figures apply to a system including the PED editor and most of the other options, but excluding the LNK code which adds another 1K.) The low segment can be reduced in size by making the stacks smaller.

9.1 Sizes Of Data-structures

1. List Cells - 3 words each

2. References - 2 words each

3. Word cells - 3+intof((c+1)/5) where c=no. of characters plus core for the valof if any (1 list cell initially).

4. Full strips - (1+n) words for n components

5. Compnd strips - 2+n/2 words, rounded up

6. Packed strips - 2+n/m words rounded up, where m is the number of components which can fit into one word , i.e. 36/size.
   m components are packed per word, left justified, as this is the standard DEC-1Ø byte-packing convention.

7. Records - 1+m words where the components require m words. Components are packed from left to right. A full component requires a whole word. A compnd component requires a half-word (right or left). A packed component (size 1-35) requires that many consequtive bits in one word and cannot cross a word boundary.
   For example, records with the specification [Ø COMPND Ø 15 2 COMPND] need 5 words each in all, and leave unused a halfword after the first compnd component,

and a bit in the last word.

8. A new strip class - 21 words if a packed strip, else 13 words

9. A new record class - 15+15*n words where n = no. of components

10. a function - add up the number of variable references, function and operation names, and gotos; then add 2 for each AND, OR, then; and 1 for each ELSE ELSEIF LOOPIF and UNTIL; add 2 for each backward goto, and 2 for each real constant, and an extra 1 for each formal parameter and output local (these should already have been counted once) and add 4.

11. a doublet - making a doublet from two existing functions uses no store.

12. A closure function - 3+2*n where n=no. of frozen formals.

13. An Array - 6+2D+strip where D=no. of dimensions, and strip= size of strip required to hold all components of the array.

APPENDIX A

ERROR NUMBERS

1	Non-char. given to text itemiser
2	Number too large on input
3	Number contains 2nd decimal point
4	Impermissible use of sub-ten
5	Non-octal digit in 8: integer
6	Non-binary digit in 2: integer
7	Impermissible use of %
8	Unexpected closing string quote
9	NUMBERREAD or LISTREAD: bad item read
10	Impermissible separator
11	Impermissible or missing closing "bracket"
12	Missing separator
13	Non-operation after "NONOP", or operation after "."
14	Non-WORD text item: WORD expected
15	Impermissible use of text item
16	Unexpected ":"
17	Compiling call of protected non-function identifier
18	Operation precedence not in range (1 - 9)
19	Non-INTEGER text item: INTEGER expected
20	Unexpected label
21	Two labels of same name in function
22	Mis-use of syntax word or protected identifier
23	Impermissible assignment statement
24	assigning non-FUNCTION to restricted variable (operation?)
25	Type clash in variable declaration
26	Jump to undefined label
27	Missing closing word quote
28	Impermissible statement outside function body
29	CANCEL in function body, or not an identifier
30	User stack underflow
31	User stack overflow
32	Link stack overflow
33	Auxiliary stack overflow (infinite recursion?)
34	Too long a string on input (missing closing quote?)
35	PROGLIST exhausted in compiler
36	Not enough store
37	Request for oversize cell
38	Too few results for JUMPOUT fn
39	Integer arithmetic overflow
40	Floating point overflow

41 Integer overflow in INTOF
42 Attempt to take the LOG of a non-positive number
43 Attempt to take the SQRT of a negative number
44 Mis-use of
45 Non-number in arithmetic function
46 Floating numbers in //
47 Non-integer for integer function
48 Multiple = in SECTION spec.
49 Macro has formal parameters or changes user stack
50 APPLYing a non-function
51 Using an undefined operation
52 Using UPDATER of a non-function
53 Calling an undefined UPDATER, or compiling call of updater of a protected function with undefined UPDATER
54 Non-function given when function expected
55 PARTAPPLY - not a list
56 Attempt to change UPDATER or FNPROPS of system function
57 Attempt to assign non-function to an FNPART or UPDATER
58 Obeying a JUMPOUT function outside its scope
59 Using an undefined MACRO
60 Item is not pair or list as it should be
61 DESTREF or CONT: not a REF
62 WORD-function: not a WORD
63 Using updater of VALOF on protected identifier
64 FNPROPS or UPDATER: not a function
65 FROZVAL or FNPART: not a closure function
66 FROZVAL - incorrect subscript
67 JUMPOUT - incorrect result-count argument
68 DATALIST, DATALENGTH or COPY: impermissible argument
69 Attempt to update the DATAWORD of standard item-class
70 RECORDFNS: impermissible record-specification
71 Wrong item for a record destructor
72 Wrong item for a record doublet
73 Impermissible item for record component
74 Assigning simple item to an FNPROPS or UPDATER
75 Applying LENGTH to a circular list
76 MAPLIST or APPLIST: not a function arg.
77 Dynamic list function: not 1 result
78 applied to non-list
79 HD, TL, etc applied to NIL
80 STRIPFNS: illegal component size
81 Strip constructor - illegal length
82 Wrong item for strip selector
83 Wrong item for strip updater
84 strip selector - wrong subscript
85 strip updater - wrong subscript
86 illegal component for strip
87 CONSWORD - illegal word length
88 CONSWORD - illegal character
89 CHARWORD - illegal subscript
90 NEWARRAY or NEWANYARRAY - illegal boundslist
91 NEWARRAY or NEWANYARRAY: illegal bounds
92 NEWANYARRAY: strip function is non-function
93 Illegal array subscript

94 BOUNDSLIST: illegal argument
95 JUMPOUT function created while a stack overflowed
96 PRSTRING: not a CSTRIP
97 SP or NL: not a number
98 INTPR, PRREAL or EXPPR: bad format arg
99 INTPR, PRREAL or EXPPR: not a number
100 POPMESS or LIBRARY: not a list
101 POPMESS: Unknown control word
102 POPMESS: Bad list format
103 POPMESS or LIBRARY: Bad list element
104 POPMESS-CLOSE - not an I/O function
105 Non-char. given to consumer
106 Using file - already CLOSEd
107 Hardware output error
108 Hardware input error
109 POPMESS-WIDTH - not a char, repeater, or valid width
110 POPMESS-CORE illegal arg
111 POPPIP - bad arg
112
113 I/O saturation (14 channels already open)
114
115 DEV: unknown, or wrong mode
116 Bad protection value
117 ERIPP% Incorrect PPN
118 ERFNF% File not found
119 ERPRT% File protection violation
120 ERFBM% File being modified
121 ERNRM% Quota exceeded, or file structure full
122 ERWLK% File structure write-locked
123 Cannot rename old file as .BAK, new version is .TMP
124 POPMESS-EDIT not available
125 Attempt to edit a .BAK file
126 Hardware error during editing
127 Editor: a .TMP file of the same name is open elsewhere
128 Editor: cannot rename as .BAK, new version is .TMP
129 ERAEF% RENAME: file already exists
130 LOOKUP, RENAME or ENTER UUO: "ersatz" error
131 BLOCKIO - illegal structure
132 BLOCKIO - illegal block number
133 POPMESS-GETSTS: bad argument
134 BLOCKIO - compnd item in structure
135 BLOCKIO - hardware output error
136 BLOCKIO - hardware input error
137 POPMESS-MTAPE: not a tape I/O function
138 POPMESS-MTAPE: bad integer
139 POPMESS-RESTORE: not enough core, or bad file
140 Old file renamed .BAK, but cannot find new version
141 POPMESS-DEVCHR: not an I/O function
142 POPMESS-LENGTH: not a DSK: or DTA: file
143 File protection violation - BLOCKIO output
144 File being modified - BLOCKIO output
145 Quota exceeded, or file-structure full - BLOCKIO output
146 File-structure write-locked - BLOCKIO output
147 POPMESS: PPN illegal or bad format

148 POPMESS-RENAME format - no "="
149 POPMESS-PROMPT: not a CSTRIP
150 BARRIERAPPLY: incorrect integer
151 APPSTATE applied outside its barrier
152 REINSTATE not given a SAVED-STATE
153 Reinstating a SAVED-STATE outside its barrier
154 POPMESS-LOCATE: bad argument or stn. not in contact
155 POPMESS-SETTTY error
156 POPMESS: not a char. code
157 POPMESS-HIBER - improper argument
158 Facility not implemented
159 Attempt to access illegal or non-existent memory (may be system bug)
160 PEDWNF% Window bound Not Found
161 PEDIWA% Illegal Window bound Argument
162 PEDIIA% Illegal Insert Arg:
not string, word, number, file or grabbed object
163 PEDNDF% Not a Disk File in PEDIN
164 PEDBNE% Buffer Not Empty in PEDIT
165 PEDRGO% Reinserting Grabbed Object, already inserted or undone
166 PEDIMA% Illegal Move Argument; not an integer
167 PEDSFL% Search Fail; item not found, position unchanged
168 PEDISA% Illegal Search Arg:
not string, word, number or function object
169 PEDFBB% Filing Bad Buffer: buffer empty, or no NAME
170 PEDUNF% UNDO Fail; nothing to undo
171 PEDMMF% MM Fail; text not properly nested
172 PEDIBA% Illegal BVAL Arg: assigning non-pos.integer
173 PEDUAC% buffer contents using unassigned channel
174
175
176
177
178
179
180 POPMESS-LNK: not an assigned TTY
181 LNK: communications mixed up
182 LNK: Using selector during write
183 LNK: using updater during read
184 LNK: non-byte given to consumer
185 LNK: both stations trying to receive
186 LNK: too many line failures

INDEX