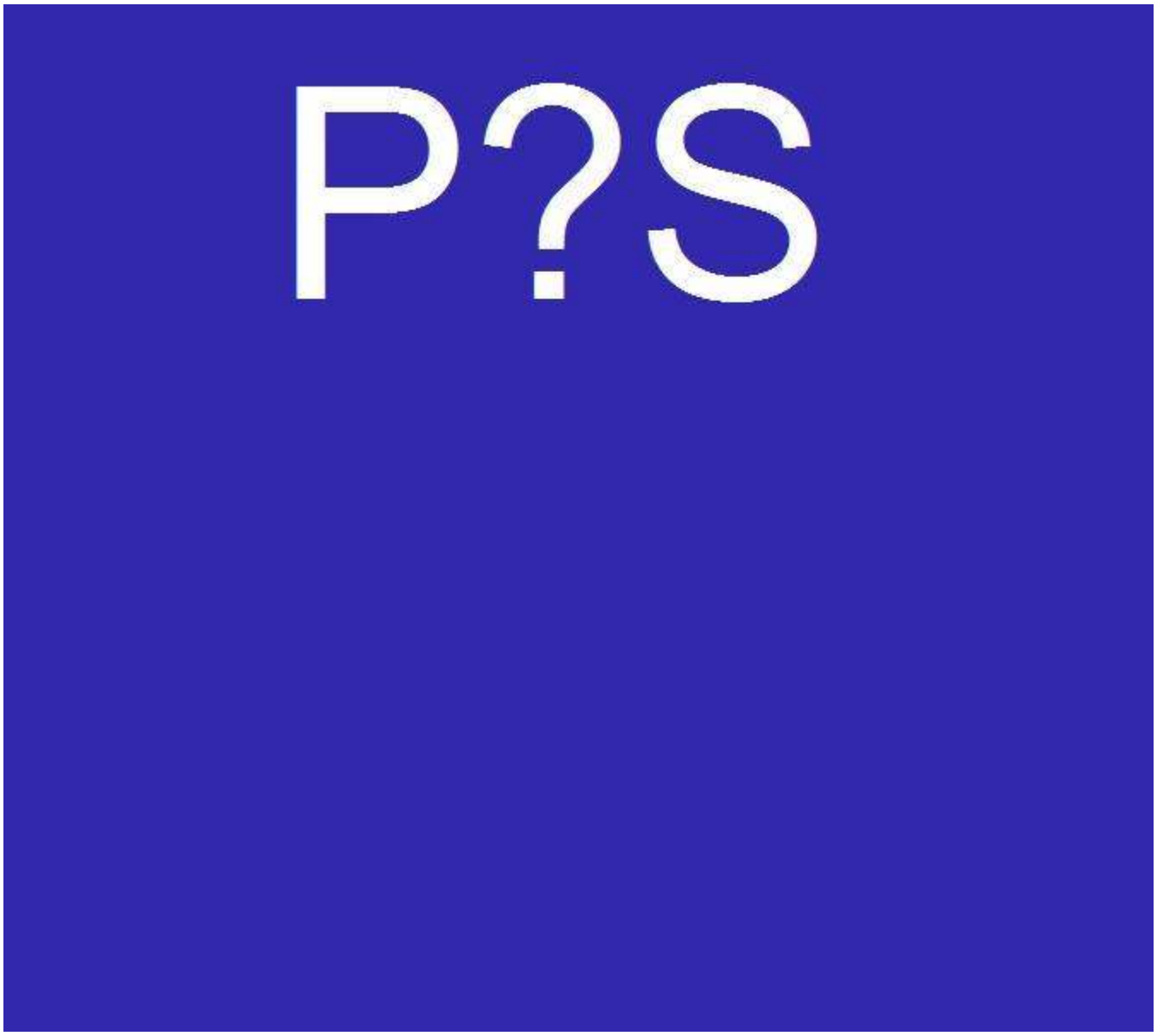
P?S/8 PAL

Technical Guide

Prerelease



Name:	P?S/8 PAL Technical Guide
Document:	PQS8 PAL Technical Guide.rtf
File[s]:	PQS8 PAL Technical Guide.PDF
P?S Index:	O>R,O>R.70037O@K0037SDBGMHB@K0037FTHCD
Security Level:	2
Location:	NY
Date:	14-Oct-2019
Medium:	PDF

1st Private Development Prerelease	June 2015
2nd Private Development Prerelease	August 2015
3rd Private Development Prerelease	January 2016
1st Semi-Public Prerelease	January 2019
2nd Semi-Public Prerelease	October 2019

P?S/8 PAL Version 8S

Technical Guide

Contact:
CLASystems@gmail.com

CONTENTS

1.00 Introduction

1.10 P?S/8 keyboard monitor commands to invoke P?S/8 PAL

1.11 PAL input files

1.12 PAL output files

1.13 Command-line option switches

1.14 Passed numerical value

2.00 Command-line option descriptions

2.01 Output file options

2.02 Chaining options

2.03 Pass control options

2.04 Symbol table options

2.05 Listing options

2.06 Literal and link options

2.07 Limitations of generated links

2.08 Literal statement examples

2.09 Literal extent issues

2.10 Dual assembly mode options

2.11 Dual mode assembly tips

2.12 LINCtape information

2.13 Memory management options

2.20 P?S/8 PAL assembler performance

2.30 Additional binary chaining options

2.40 Text management options

2.41 Text directive issues

CONTENTS
(continued)

2.42 Other text-related issues

2.50 Additional command-line option switches and issues

2.60 P?S/8 PAL directives

2.61 Issues regarding the DTORG directive

2.62 PAL8 implementation of DTORG

2.63 Proposed remedies to the DTORG problem

2.70 Additional information

2.80 Issues with regard to using FLIP.EXE to convert line conventions

Appendix A - P?S/8 PAL error messages

Appendix B - P?S/8 PAL directives

Appendix C - P?S/8 PAL conditional literals and related topics

Appendix D - P?S/8 PAL command-line option switch ordered summary

Glossary of Terms

This page intentionally left blank.

1.00 Introduction

P?S/8 supports a general-purpose assembler for *PDP-8 Assembly Language* (*PAL*) known as *P?S/8 PAL*. This document describes all technical aspects of P?S/8 PAL in terms of both language features and constructive usage; this includes certain tips and techniques for PAL programming helpful to all users regardless of assembly language skill level.

Note: The current release of P?S/8 PAL is Version 8S which is provided as part of P?S/8 Version 8Z; this includes a small number of documented binary patches made since the original 06-Oct-1987 release. Specific patch details can be found within the *P?S/8 System Change Log* maintained as a separate document.

LINC mode assembly (as used on LINC, LINC-8 and PDP-12 computers) is enabled by the use of certain command-line options; the implementation is generally compatible with the embedded assembler utility of the *LAP6-DIAL*/*DIAL-MS* operating system for the PDP-12. When assembling LINC code (possibly interspersed with PDP-8 code), P?S/8 PAL can be considered to be a cross-assembler.

While execution of binary LINC mode code requires LINC computer hardware, P?S/8 PAL does not require any additional hardware beyond the basic P?S/8 system for any PDP-8 (or DECmate) model to assemble LINC source code into binary code. Depending on the specifics of the application, it may be necessary to devise utilities to write the developed program on the intended hardware. (The LINC is not software compatible with the PDP-8; however, it is possible to write arbitrary data on LINCtape blocks formatted for use on the LINC. No version of P?S/8 (or *OS/8*) directly supports this media; as such, utilities must be written to transfer data as required to write data for the LINC. There are similar requirements for use with analogous systems that run on the LINC-8 and the PDP-12.)

Note: A list of supported P?S/8 hardware configurations can be found within the *P?S/8 Keyboard Monitor Command Guide* for P?S/8 Version 8Z (or newer); P?S/8 PAL is a standard component of all systems.

For more information concerning dual-mode PDP-8 and LINC assembly, see Sections 2.10 and 2.11 below which document dual-mode assembly considerations. As stated above: Depending on the ultimate target system for the binary output of the assembly, additional utilities may be necessary to load the binary data onto the intended environment.

Unlike competing assemblers (*TOPS-10 PAL10* and OS/8 *PAL8*), there are numerous command-line option switches supported by P?S/8 PAL; this is necessary to allow the modular sections of P?S/8 PAL to be invoked as required. Various portions of this document will explain these command-line switches including information on how best to deploy them for any given assembly situation. Only certain features of P?S/8 PAL are needed in most typical assembly scenarios; as such, it is recommended that only relevant command-line option switches be set during any particular assembly operation.

Note: Official releases of TOPS-10 PAL10 are of limited use for serious PDP-8 program development as compared to either P?S/8 PAL or OS/8 PAL8. This is due to a small number of important features that are either entirely lacking or are misimplemented; as such, PAL10 is unusable for a significant amount of PDP-8 program development.

This is ironic since, at one point in time, most PDP-8 program development (outside of the *R-L Monitor System* and all but the earliest implementations of P?S/8) was performed on TOPS-10 systems to take advantage of the then-considered-rich feature set of PAL10 and the advanced system features of TOPS-10 for program developers using (nearly) any language. Unfortunately, as new features were added to other assemblers, PAL10 maintenance programmers made little (if any) changes; thus, PAL10 failed to keep pace with the emerging needs of PDP-8 system programmers who used either PAL8 or P?S/8 PAL (or both) including some of the newer features required for proper program development.

It is hoped that proficient PDP-10 developers can produce an updated PAL10 release generally compatible with at least the important features that distinguish P?S/8 PAL (and OS/8 PAL8) from the present PAL10 release. The developers of P?S/8 will gladly discuss these issues (and recommend that P?S/8 PAL features beyond compatibility with PAL8 be included).

An important feature of the PAL language is the availability of several forms of literal statements; sections 2.06 through 2.09 below cover various aspects of literal usage and recommended syntax.

Fortunately, the current PAL10 literal implementation is largely compatible with PAL8 (which is in turn a subset of the P?S/8 PAL literal implementation).

The primary difference between the P?S/8 PAL literal implementation and that of PAL10 (and PAL8) is the conditional (or *dependent*) literal, a feature conceptually unique to P?S/8 PAL. Appendix C below discusses numerous issues related to conditional literals.

1.10 P?S/8 keyboard monitor commands to invoke P?S/8 PAL

Note: Command element descriptions in this document follow the conventions used in the P?S/8 Keyboard Monitor Command Guide.

Command-line options are passed within P?S/8 in the same manner in which OS/8 *Concise Command Language* (*CCL*) commands are formed (with certain extension features added).

Note: P?S/8 supports several command prompt variations. The prompt used throughout this document is identical to TOPS-10 and OS/8 to facilitate comparisons between the PAL implementations in the various operating systems. <SP>, , and <HT> are freely allowed between command elements to enhance command readability and are not required. For a discussion of all P?S/8 command prompt options, consult P?S/8 *SET* command documentation available elsewhere.

Typical command examples (in slightly simplified form) are structured as follows:

.PAL output files < input files option switches

or

.PAL input files > output files option switches

Note: As shown above, the P?S/8 keyboard monitor allows command formation in either file specification order; this allows the user to work in a familiar environment regardless of prior experience with various other DEC operating systems (or equivalent). The relevant commands are generally similar to various *concise command* implementations found in other DEC operating systems.

The option switches can be placed anywhere in the command line after the program name (in this instance *PAL*) and may be stated redundantly.

For more information about P?S/8 command formation and command-line variations, consult the P?S/8 Keyboard Monitor Command Guide available as a separate document.

1.11 PAL input files

PAL program input data is generally the concatenated contents of a group of *Tiny File System* (*TFS*) text files passed in a specified order in the input section of the command. Each file contains 2048 12-bit words, which will tend to consist of up to 3/4 of this capacity as six-bit bytes in the P?S/8 ASCII text (subset) character set (which includes the <HT> character). Typically this results in about 3000 characters per TFS file (depending on average line width and attendant line number requirements). Effective line width is greatly influenced by the use of <HT> characters, which generally occupy several printed spaces, yet are stored as a single six-bit byte each. The recommended PAL syntax encourages the free use of <HT> characters as required.

Note: P?S/8 TFS text files include line numbers which are ignored by P?S/8 PAL. Outside of the edit buffer of the keyboard monitor (which can modify the text contents and line numbers), the line numbers are permanent (once saved as a TFS file). Most P?S/8 system programs ignore line number content within input files; exceptions include several utilities specifically designed to display or otherwise interact with line numbers (present only in TFS text files and not in other files, such as extended-length text files).

The maximum number of files that can be passed to any P?S/8 system program is 17. Since several file slots may be needed for output files, the effective maximum number of input files tends to be somewhat smaller (typically 14 input files to allow three output files).

Note: P?S/8 also supports extended-length text files, any one of which can typically contain considerably more text characters than the entirety of 17 TFS files. For more information regarding extended-length file considerations, consult documentation of the P?S/8 *OS8CON* program (or other similar file conversion utility programs) available as separate documents.

1.12 PAL output files

As stated above, all P?S/8 TFS files are fixed size (2048 12-bit words). TFS binary files are created in the versatile *Slurp* binary loader format, which is significantly more efficient than binary files based on paper-tape frame images (such as the standard DEC *BIN* loader format). A typical TFS binary file contains nearly 6/7 as much binary data as a block-format core-image of identical size, without implementing any form of internal block structure.

Binary loaders for TFS Slurp format binary files are available on all standard P?S/8 configurations; a generic virtual Slurp format binary loader is implemented in all systems. Most systems also support a device-specific Slurp format binary loader (which will be the default loader if available). Command-line option switches control which loader will be used in specific situations initiated by P?S/8 PAL command variations (*assemble then load and go* or *assemble then create bitmap then load and go*).

At the end of the assembly, the count of created binary files is displayed (along with other statistics); this allows any excess passed output files to be repurposed as desired.

Note: If insufficient output files are passed, a BO (Binary Overflow) error message will be displayed on the system console; the assembly process will be aborted.

1.13 Command-line option switches

The command-line option switches are passed in a manner consistent with CCL as implemented in TOPS-10 (and OS/8). Two variant forms are supported as follows:

1) One or more individual option switches each preceded by / such as:

   /M /N /L

2) Grouped switches preceded by (such as:

   (MNL)

   Note: The trailing) is optional unless needed to separate the option group from another command-line section.

Both forms of passing command-line option switches can be used together in the same command; all command-line option switches may be given redundantly without error.

1.14 Passed numerical value

P?S/8 PAL can optionally chain to P?S/8 *BIN*, the binary file loading and binary paper-tape utility program, for the purpose of loading (or punching to binary paper-tape) the binary files created during the assembly. If desired, the assembled program can be automatically started at an address expressed as =*xxxx* where *xxxx* is up to four octal digits in the range of 0000 through 7776; for programs starting in extended memory, the /1 through /7 switches provide the starting field. See Sections 2.02 and 2.30 below for more information.

The P?S/8 keyboard monitor internally passes the value 7777 when an explicit numerical value is not provided. When binary file loading is in effect, P?S/8 BIN uses this value to effect program start at a system-specific location somewhere within the system kernel area (07600-07777); a safe halt instruction is loaded at the selected location. Pressing Continue on the front panel will effectively loop in place preventing potential problems; the user is expected to constructively start the program manually (at a user-determined address) using the computer front panel switch register.

Note: By use of the P?S/8 *GET* command (or adding the equivalent options to the P?S/8 PAL command line), a core-image of the program can be created at the end of the assembly process to allow further program development. Debugging and related binary program options are described in a separate document.

On systems lacking front panels (such as the PDP-8/m or PDP-8/a without the optional programmer's panel, or any DECmate model) the GET command should be used to allow program execution control from P?S/8 *ODT* or other utilities.

2.00 Command-line option descriptions

The following sections of this document describe command-line options grouped by common function. Command-line option lists and additional information are provided elsewhere in this document. Appendix D below contains a list of all command-line options in sorted order.

2.01 Output file options

If explicit binary output files are not specified in the command line, the following command-line option switches can be used:

/B The *%* file on the P?S/8 system device bootup logical unit (see description of the /U command-line option switch below) is used as the (first) binary output file.

/D If the /B command-line option switch is also set, the *$* file on the P?S/8 system device bootup logical unit (see description of the /U command-line option switch below) is used as the second binary output file (if required).

/U Change the logical unit that applies (when the /B and/or /D command-line option switches are used as described above), from the system device bootup logical unit to the value determined by the following calculation:

(System device bootup logical unit XOR 1); this selects the *opposite* logical unit of the primary pair of logical drive units.

Note: On certain P?S/8 systems the system device bootup logical unit can be non-zero (perhaps logical drive 1 through 7); as such, this method will generally achieve a useful drive pairing.

Due to the physical proximity to the beginning of storage devices, specifying the % and $ files can noticeably speed up various system commands. Additionally, these files are not subject to the overhead of TFS directory functions on the same logical device unit; as such, many operations have significantly lower overhead when these files are used strategically. On tape-based systems, throughput can be further improved by using a logical drive unit different from the system bootup drive.

Note: The /B, /D and /U command-line option switches are provided to simplify the specification of output files, as most assemblies require one or two output files. All explicit file specifications can designate an optional device unit; as such, any combination of files can state associated logical device units as necessary. The use of these command-line option switches merely allows specifying commonly used output file combinations in a more succinct form; however, if used in BATCH files, the actual files used could be determined by the particular system device bootup logical unit in effect.

If more than two binary output files are required, or the created binary files should be written to permanently named TFS files, the command line must include explicitly stated output files.

Note: If the second binary output file is not required, the /D command-line option switch will be ignored (effectively reducing the command-line to have been given with only the /B command-line option switch set regarding output file selection); the report given at the end of the assembly indicates the total count of binary output files actually created.

2.02 Chaining options

Note: Only binary output files created during the assembly process are used during chaining operations; excess output files (if any) are deleted from the passed file list.

Automatic chaining options can be invoked to further process binary files created during the assembly as follows:

/G The files created during the assembly are passed as input files to the P?S/8 BIN utility for loading and execution (or punching binary paper-tapes as described below in Section 2.30); additional loading (or punching) options may be applied as necessary.

Note: The chain to P?S/8 BIN is prevented if there were any assembly errors to avoid execution of unknown instructions.

With regard to chaining to P?S/8 BIN for the purpose of loading binary files, see Section 2.30 below for additional details; binary chaining options include the /H, /I, /V and /Z command-line option switches and the =xxxx passed numerical value (possibly in conjunction with the /1 through /7 command-line option switches).

Note: The /W command-line option switch must be clear to allow binary loading (as opposed to punching binary paper tapes).

With regard to chaining to P?S/8 BIN for the purpose of punching binary paper-tape output, additional binary chaining options include the /F and /R command-line option switches. See Section 2.30 below for more information.

Note: The /W option switch must be set when punching binary paper-tape output (as opposed to binary file loading).

/M The files created during the assembly are passed as input files to the P?S/8 *MAP* utility to create a bitmap of the assembled program indicating actual memory usage.

Note: The format of bitmap output created by P?S/8 MAP is generally consistent with the output produced by PAL10 after any FIELD directive is used (or at the end of assembly) and is a superset of the output of OS/8 *BITMAP*. The title format and pagination specifics are the same as those used for the assembler listing output (if any) and assembly statistics pages (which in part depend on the usage of the /N option switch as described in Section 2.05 below).

It is possible to specify chaining options to P?S/8 MAP and P?S/8 BIN in a single command. The chain to P?S/8 MAP will always be performed first; the additional chain to P?S/8 BIN will be prevented if errors occur during the assembly.

Note: When the P?S/8 MAP utility is used without chaining, certain command-line option switches can be used to limit which memory fields are eligible for mapping; during the chaining process this feature is inhibited. As such, all memory fields are eligible for bitmap output following the assembly; the output is limited to actually processed fields.

For more information about mapping field choices, consult documentation of P?S/8 MAP available in a separate document.

2.03 Pass control options

/K Perform a two-pass assembly; the default is a single assembly pass.

Note: The /K command-line option switch is not required if any binary or listing options are specified, as these functions intrinsically require two pass operation.

Unlike PAL8 which always performs two (or three) passes, P?S/8 PAL can perform a useful single-pass assembly (which is consistent with the paper-tape system assembly program known as *PAL III*) as follows:

At the end of pass one, all symbols still not defined are shown on the system console terminal in sorted order; the value indicated is the address within the assembly where each undefined symbol was first encountered. This facilitates early program development when it is likely that (many) portions of the program have not yet been completed. Alternatively, variables and temporaries may not yet be defined (or perhaps are the object of typographical errors which can cause problems when certain symbols have names similar to that of other symbols); most problems of this nature can be quickly resolved by referencing the relatively short list of undefined symbols.

Contrast this functionality with that provided by PAL10 or PAL8, neither of which support a mechanism to highlight undefined symbols. The only facility provided in these assembly programs is to create a second (or third) pass listing file of the entire assembly; searching through this far more voluminous output can be tedious as there are usually several references to each undefined symbol as well as many lines of output unrelated to errors.

Note: In perverse instances, large quantities of erroneous output could be triggered by the failure to correctly define key symbols during early program development; these circumstances could make the manual process of eliminating errors far worse due to cascading dependencies.

As such, using PAL10 or PAL8 can easily become a time-consuming process. In such circumstances, it would be prudent to limit the assembly to a single pass using P?S/8 PAL until key problems are solved. Using only the undefined symbol report (available at the end of pass one of the P?S/8 PAL assembly), most problematic issues can generally be eliminated.

Using a second pass listing provided by P?S/8 PAL would be as frustrating to use as that obtained from PAL8; however, only P?S/8 PAL provides the one pass method that is recommended for use during the early stages of program development.

See the description of the /L and /N command-line option switches for the alternate (and unrelated) usage of the /K command-line option switch.

Second pass assembly considerations.

A second assembly pass is performed if binary output files are created and/or assembly listing output is produced. During pass two, any additional error messages will be displayed on the prevailing output device (system console terminal or lineprinter) in a format similar to that of PAL10 and PAL8 (including the nearest symbol plus any numerical offset) to facilitate source code development; in certain instances additional information may be displayed.

Programmers may embellish the displayed output of the assembly process when critical sections are being processed as follows:

a) P?S/8 PAL supports the ERROR directive; this causes a deliberate assembly error. By using the ERROR directive, the programmer can design internal safeguards to prevent logical inconsistencies that might occur during program maintenance.

   The design of safeguarded code sections generally includes the use of conditional assembly statements to calculate problematic scenarios; as long as the specific requirements are met, there will be no output associated with the use of the ERROR directive.

   Note: The output of the ERROR directive is consistent with that of all standard errors during each assembly pass and can include the display of an optional evaluated argument. Automatic binary loading by chaining to P?S/8 BIN will be prevented if the ERROR directive was assembled.

b) P?S/8 PAL supports the PAUSE directive both as originally implemented in PAL III (to ignore the rest of the contents of the latest input file) and also as an extended feature which can embellish displayed output during the assembly process.

   When the PAUSE directive is used with an argument, the argument is evaluated as a 12-bit unsigned integer. Displayed output is consistent with the format of standard error messages; however, usage of the PAUSE directive does not report as an error.

The programmer can utilize the PAUSE directive for a variety of purposes (such as announcing the assembly pass or start of assembly of a critical section). When a logical inconsistency is detected, the PAUSE directive can be used to output additional information about the error to assist maintenance programmers to obtain an understanding of the nature of the problem.

The techniques described above and other features unique to P?S/8 PAL can be qualified to prevent use with incompatible assemblers.

The *PQS* directive is defined only in P?S/8 PAL; this allows the programmer to create conditional assembly statements to restrict usage of these features to P?S/8 PAL and not generate erroneous errors caused by the failure to support the features in other assemblers. An example of this usage is as follows:

```
        IFNDEF    PQS  <PQS=  0> /ZERO ONLY IF NOT P?S/8 PAL.

        IFNZRO    PQS  <PAUSE .> /DISPLAY CURRENT ORIGIN.
```

Note: The XLIST directive, as implemented in P?S/8 PAL, is consistent with PAL10 and PAL8; XLIST statements can be used to prevent output of portions of an assembly as necessary, such as assembler-specific statements as described above. When deployed judiciously, all unassembled conditional assembly can be hidden from view when producing a second-pass listing; the only indication of this technique will be a corresponding change in the statement number field when normal listing output is restored.

2.04 Symbol table options

/S The symbol table will be output at the end of the assembly.

If the /N command-line option switch is also set, the symbol table format will be consistent with that of an enhanced listing pass (if enabled).

Note: The exact output may be subject to the number of assembly passes performed; two pass assembly is used if binary output files are created and/or listing output is produced, or the /K option switch is set.

/A If the /S command-line option switch is set, the entire symbol table will be output at the end of the assembly; this includes all permanent symbols including assembly directives.

Note: If dual-mode (PDP-8 and LINC) assembly is in effect, the symbol table will be set to that which applies at the end of the assembly; certain symbols with dual definitions will display values consistent with the final assembly mode.

Certain permanent symbols may have been redefined during the assembly by source code statements; this is especially important in LINC mode assembly where there may be multiple definitions of certain instructions that are model-dependent within the various LINC hardware implementations.

While all default LINC mode definitions within P?S/8 PAL are for the PDP-12 LINC variant, source code changes can be made to redefine affected symbols for compatibility with the LINC-8 or classic LINC as required. Permanent change of certain *SKP* class (or other LINC instructions) can only be made at the source-code level of P?S/8 PAL itself; as such, it is recommended that all affected symbols be redefined in user source code statements as required.

2.05 Listing options

/L Assembly listing output will be produced.

Unlike PAL10 and PAL8, P?S/8 PAL performs (at most) two passes. P?S/8 PAL is capable of simultaneously creating binary file and listing output during pass two; as such, the following situation can be avoided:

Obscure PAL language quirks can be problematic when literals are in effect. Novice programmers must avoid certain scenarios where the definition of certain symbols and values is dependent on the number of passes performed as well as specific addresses of generated literals.

While dependence on specific literal addresses is very poor programming technique (and usually avoided by experienced programmers), it can be confusing to debug mystifying results obtained from PAL8; PAL8 binary output is created using symbolic definitions obtained at the end of pass one, possibly updated during pass two. However, PAL8 listing output may be created using symbolic definitions obtained at the end of pass two and possibly updated during pass three.

Unless the programmer is careful to avoid pass-dependent definitions, the binary output may differ from the values indicated in the listing output (which may appear correct). Since P?S/8 PAL always creates all output during pass two, there cannot be pass-dependent differences between binary output and listing output. Relevant programming blunders should be discovered sooner using P?S/8 PAL (along with additional experience when the improper techniques are recognized and avoided).

/P Enable wide-printing listing output. Note: This option primarily affects symbol table output allowing additional columns of printed symbols per page. (The expected output format changes from 72/80 columns to 132 columns as often implemented on lineprinters that use 11" x 17" ledger/tabloid paper stock.)

Note: PAL8 only creates output for a minimal line width to (nearly) guarantee compatibility with narrow-carriage devices such as the LP01 lineprinter, despite the fact that most printers support 132 column mode; outputting to an LP01 may cause excessive truncation.

P?S/8 PAL supports source statements significantly wider than PAL8. Where relevant, the listing output of P?S/8 PAL will avoid truncation (unlike PAL8) when assembling the identical source code.

Note: PAL8 users must exercise caution when creating (somewhat) wide program statements. Due to quirks in the PAL8 implementation, the precise number of characters leading to truncation problems varies with statement type and other unexpected issues.

Depending on specific circumstances, a portion of the intended source statement may be truncated. This can unexpectedly change logical program flow.

Warning: This is not merely a matter of truncating the listing output; rather, part of the source code statement may be ignored. Debugging unexpectedly truncated assembly statements may be difficult (especially when conditional assembly techniques are used in conjunction with the XLIST directive).

/T Prevent output to the system lineprinter. All output is directed to the system console terminal. See the description of the /N command-line option switch below.

/N Output *niceties* are applied to the listing output including titles on each printed page. If output is directed to the system console (instead of the system lineprinter), the output will produce alternatives to form-feeds by the use of tear-off lines instead of <FF> characters; this will ensure complete compatibility with system console devices that use roll paper such as the Teletype Model 33.

Note: For systems lacking a lineprinter that support hard copy console terminals such as the LA36 (which can properly support <FF> characters), the *Logical Console Overlay* can be configured to redirect lineprinter output to the same physical device as that used for console output. For more information, consult documentation of the P?S/8 Logical Console Overlay available separately.

When the /N command-line option switch is set, one additional page of memory is allocated. See Section 2.13 below regarding memory management options for more information.

/K Enable assembly title text from each input file.

When the /L command-line option switch is used in conjunction with the /N command-line option switch, the title field text for the entire assembly is taken from the first line of the first input file.

Typically, assemblies are processed by concatenating the contents of multiple TFS text files. In most cases there is no particular connection between program sections in different files and intended assembly title changes; file boundaries are merely an independent file contents consideration.

To maintain compatibility with PAL10 and PAL8, taking the title field characters from the first line of every input file can be enabled; the /K command-line option switch must be given along with the /L and /N command-line option switches.

As long as each logically independent section of a program is contained within either an extended-length file (or a single TFS file such as a small parameter setup file), meaningful title changes can be properly managed.

As a practical matter, intentional use of assembly title field updates via file breaks is used infrequently. For example, the source code of *FOCAL, 1969* updates the title field only once during the entire assembly; as such, two extended-length text files (for the two main modules known as *FOCAL* and *FLOAT* respectively) support the intended title field changes during the assembly.

Additionally, the title on the assembly can also be changed using other methods that are independent of file boundaries. See the description of the *EJECT* and *TITLE* directives in Appendix B of this document for further information.

Note: A two-pass assembly will occur because the /L command-line option switch is set to create listing output. As such, the /K command-line option switch should only be used to enable title field text obtained from the first line of every input file. The alternate usage of the /K command-line option switch (to force a two-pass assembly) is moot in this usage.

/X Enable the creation of cross-reference output by loading the cross-reference module; an output element is created for each symbolic (and literal reference if any) as well as the associated statement number. Cross-reference data is created in statement order during the assembly; at the end of the assembly process the data is sorted and included as part of the overall listing output.

Enabling cross-reference output causes all listing output to be several columns wider due to the inclusion of the lowest statement number on the line at the left margin. (The PAL language allows multiple statements on a line separated by ;.) Statement numbers are unique in the range of 1 through 99999.

Note: It is strongly recommended that cross-reference listing output with statement numbering be printed on 132 column lineprinters, especially if the /P command-line option switch is also enabled.

If the /A command-line option switch is enabled, all symbolic references will generate cross-reference output; the default action is to reference only user symbols (and literals, if any).

When the /X command-line option switch is enabled, two additional pages of memory are allocated. See Section 2.13 of this document regarding memory management options for more information.

Implementation Restrictions.

Due to changes planned for release with P?S/8 Version 9A (which is not yet available as of this writing), the cross-reference output is currently written to the system device logical unit 7 starting at block 0000; all prior contents of this device will be overwritten as necessary . No additional output of cross-reference data is available; no final data sorting or processing is provided in the current release. The cross-reference output is limited to the storage available on logical device unit 7 (since only one device is used).

Once the cross-reference output programming is upgraded to allow allocation of extended-length files via a user-settable configuration table, this restriction will be eliminated. Significantly larger programs will be handled than presently possible on appropriate configurations (such as RK8E/RK05-based systems) due to the availability of multiple extended-length files to store reference data during the assembly.

As of this writing, the processing of the cross-reference output is not fully implemented pending the changes described above with regard to cross-reference output data storage allocation issues; however, both statement numbering and creation of reference data are currently functional (although the cross-reference information is not currently accessible). Logical unit 7 must be available as described above when enabling this feature in the current release. (The availability of logical Unit 7 will become optional once this restriction is eliminated; extended-length files will be configurable on all available logical drive units when new options of the P?S/8 *SET* command are made available.)

Additional required changes will be made to the System Programs Directory (expected to be implemented in the next full release) in conjunction with other performance improvements.

Note: PAL8 does not directly support cross-reference output. As an alternative, OS/8 listing files are reparsed by OS/8 *CREF* (if enabled) as a post-processing utility. There are several disadvantages to this method:

a) The listing output of PAL8 is not a statement-by-statement listing file; statements hidden by the use of the XLIST directive (and related techniques) do not appear in the listing file. As such, the cross-referenced listing output is potentially incomplete; statement numbers are inherently inaccurate and untrustworthy.

   By way of contrast, P?S/8 PAL statement numbering is compatible with PAL10, which correctly shows processed statement numbers, revealing the presence of any hidden statements.

b) A valid technique for including documentation within PAL source code is to use deliberately failing conditional assembly on a section of commentary as large as an entire printable page of the listing. Content examples include details regarding program operation and *banner pages* of large character outlines created from a series of smaller characters; the nature of this form of output is limited only by the programmer's creativity.

   Unfortunately, CREF has no ability to discern this aspect of a PAL8 listing file as commentary; as such, the commentary is cross-referenced as if part of assembled code-generating statements. When banner pages are used, there can be numerous erroneous junk references to symbols that merely correspond to the individual characters used to build up the larger character layouts; as such, the cross-reference output can be cluttered with many superfluous references. Since CREF has limitations on overall symbol table size, this can have a deleterious effect on the overall capability of CREF. See c) and d) below.

c) CREF internally reckons statement numbers as 13-bit integers in the range of 1 through 8191 with an odd quirk: additional statements wrap back to 4096 instead of 1. As such, there is inherent anomaly in the cross-reference output regarding precise statement numbering of individual references.

   By way of contrast, P?S/8 PAL and PAL10 allow unique statement numbering in the range of 1 through 99999.

d) CREF has limited overall capability. If the entire cross-reference cannot be accomplished in one pass, the program aborts unless the /M command-line option switch is set. If the /M command-line option switch is enabled, CREF operates as a rigidly-defined two-pass operation which may (eventually) allow the process to complete. However, each *half* of the overall operation must complete to obtain a successful overall result.

   Note: A quirk of the two-pass method used is the insertion of an extraneous <FF> in the output stream when the second pass begins. (This is documented but not explained.)

   The *split-point* for this so-called *mammoth* mode is set to the theoretical symbol *LGNNNN*. References to all symbols that sort lower than this value are scanned during the first pass while ignoring references to symbols that sort higher. The second pass processes all symbols that sort as LGNNNN or higher, including all references to current-page and page zero literals; the references processed in the first pass are ignored. Each pass must scan the entire listing file without symbol overflow.

   Because of the arbitrary nature of the split-point, it is possible for the overall cross-reference to fail because the program symbols are not *balanced* sufficiently to conform to this arbitrary scheme (for which there are no options).

e) Because of the inherently redundant nature of rescanning text files attempting to obtain information about the assembly without having access to the original symbol table and parsing techniques, CREF operates very slowly.

   By way of contrast, assembler-based cross-reference implementation has the advantage of only processing legitimate references to symbols and literals; commentary is ignored. Instead of rescanning entire listing files, the essential cross-reference data is scanned as many times as necessary to process all references; there is no arbitrary static split-point nor requirement of at most two passes on the far smaller set of data.

   Note: Many large PAL programs cannot be fully processed by CREF. This includes (ironically) the source code of P?S/8 PAL (and many larger programs).

2.06 Literal and link options

Properly utilized, PDP-8 literal statements allow rapid implementation of working programs. That said, certain compromises might arise:

A PDP-8 programmer may be concerned with producing the smallest possible coding for a specific program segment. Certain portions of operating systems have stringent code space requirements with regard to an extremely limited program structure such as a device handler. Often the expediency of using literals must be abandoned in favor of using a variety of tricks for which there is no analogue in other computer architectures:

a) Intrepid programmers can create equivalent literal values in various ways. The techniques generally use either the contrived location of instructions to force generation of needed constants or the use of seemingly superfluous OPR statement conditions that can create fortuitous constants regardless of code position. In rare cases, multiple contrivances are used in concert to achieve even smaller binary code. The ability to perform extreme optimization techniques requires intimate knowledge of the PDP-8 instruction set in terms of operation code values and bit wise specifics.

b) When the contrived literal values are achieved, the use of generated literal statements must generally be abandoned. As such, this extreme form of optimization contraindicates the convenience of using generated literals.

c) Purely for documentation purposes, some programmers use source code statements that include the specifics of the equivalent literal; however, that portion of the statement is part of an extended comment. This usage explains the intentions of the programmer prior to the optimization.

d) In general, the use of literals discourages code space optimization; as such, literals should be used if as and when appropriate. Many operating system components contain once-only initialization sections which are later repurposed as buffers or table/list space. It is totally appropriate to include literal statements within such code; generally, there is little to be gained using optimization techniques within these memory areas.

e) Beginning programmers are encouraged to learn the proper use of literals just after mastering the general instruction set. Code optimization is not a factor until the programmer is quite experienced. As required, programs can be (partially) optimized by removing literal statements as appropriate once the complexity of the project clearly demands optimization (after the basic structure has been realized).

Since PDP-8 code is generally efficient for most applications, a common technique is the use of literals during early development, perhaps in conjunction with conditional literals as described below. Only when necessary, optimization techniques can be applied (while simultaneously removing literal statements).

Many of the system components of P?S/8 do not use literals while other sections use literals in at least one of several contexts usually related to program initialization. This includes the implementation of P?S/8 PAL itself. Most conforming user-written programs use identical techniques during development.

In P?S/8 PAL, literals and/or generated links are enabled using a combination of command-line option switches as follows:

/Q Enable generation of current page and page zero literals. Link generation is disabled; attempts at off-page references will be treated as errors.

/O Enable generation of current page and page zero literals. Link generation is enabled; however, links will be flagged as errors.

/E Reset the current page literal extent when leaving the latest page.

Note: The use of the /E command-line option switch requires some combination of the /Q and /O command-line option switches to be meaningful.

Enabling literal generation and/or link generation requires three additional pages of allocated memory. See Section 2.13 of this document for more information regarding memory allocation and related issues.

Using the /Q command-line option switch in conjunction with the /O command-line option switch will suppress error messages on generated links; however, the count of links generated is always included in the statistics page produced at the end of the assembly.

Assembly listing lines containing generated links will be flagged with a ' character placed to the right of the generated instruction value. This feature cannot be disabled in P?S/8 PAL, PAL8 or PAL10.

Note: PAL8 and PAL10 are compatible with P?S/8 PAL with regard to all aspects of the disposition of generated link statements; however, all features regarding literals in PAL8 cannot be disabled. See section 2.07 below for detailed warnings regarding generated links that apply to all major PDP-8 assemblers.

2.07 Limitations of generated links

In general, generated links should be completely avoided; the shortest version: **Generated links do not work!**

While this simple caveat ought to be sufficient, a study of the PDP-8 architecture reveals specific problems as follows:

1) Link generation is an attempt to partially abstract and obfuscate the PDP-8 architecture. Unfortunately, this leads to sloppy programming techniques that (might) function while violating the hard-wired rules of the PDP-8 instruction set.

   A generated link is an indirect memory reference used to modify a malformed attempt at a direct memory reference that cannot be achieved; this is accomplished by using an address literal created in a manner analogous to that used within literal statements.

   The program specifies a direct memory reference that is impossible due to the violation of memory addressing rules; instead an indirect reference is automatically created by the assembler (assuming this feature is enabled). Since this process has much in common with the creation of literals, all other aspects of literal generation applies.

   The following assembled code fragment illustrates an attempt to violate the rules of the PDP-8 instruction set; the flawed statement is flagged as an error because link generation was not enabled.

```
       *0200          *200            /TYPICAL START ON PAGE 1.

IR  0577
000200  1377          TAD     577     /THIS REALLY CAN'T WORK
```

   The attempted usage cannot be achieved since the operand is outside of the allowed address range (0200-0377 in this instance); location 0577 is part of the following page (0400-0577). For an example such as this, locations on any page from 2 (0400-0577) through 37 (7600-7777) are equally inaccessible. The instruction is flagged with the IR error message (Illegal Reference).

   The next example is the identical code fragment assembled with link generation enabled. No errors are indicated, but the statement is flagged with ' indicating link generation.

```
       *0200          *200            /TYPICAL START ON PAGE 1.

000200  1777'         TAD     577     /THIS REALLY CAN'T WORK.
       *0377
000377  0577
```

   The generated code is an indirect instruction in the range of 1600-1777 as a consequence of the modification; (a direct instruction would be in the range of 1200-1377). The code also creates an address constant; the table of literal/link addresses is dumped at the end of the assembly (unless instructions exist past this section to cause this page's literals to dump sooner).

2) If link generation is enabled, the same source code produces flagged output. The instruction at 0200 is changed to *TAD I 377*. Additionally, location 0377 is generated containing 0577. Ignoring for the moment the instruction is now an indirect reference to 0577 using the pointer address 0377 as an intermediary, to a naive PDP-8 programmer (unaware of the potential negative consequences), the intended logic is satisfied: the contents of location 0577 are added to the accumulator.

   Note: In these coding examples, symbolic references are deliberately not used to avoid clouding the issues. Actual attempts at link generation are usually expressed in symbolic form. Had the code been located in an accessible location, no link would be required. Links often come about when code overflows the intended page into the next page. By using purely numerical statements, the results are easily revealed.

3) To the inexperienced programmer, the program fragment *works per se*. Unfortunately, in real PDP-8 programming, considerations outside of this example tend to crop up; programs that *had worked* suddenly *no longer work* and might also cause unavoidable assembly errors of a kind that merely enabling link generation cannot remedy.

   This programmer might be tempted to use a location already inaccessible (due to being located past the current page) as a pointer. The assembler will appropriately mark this with the error message II (Illegal Indirect). Since it takes a pointer to access indirectly the address of the pointer, it is not possible to use it indirectly. Had the programmer obeyed PDP-8 instruction rules, this would have come about without unexpected surprise.

   Before investigating the insidious nature of link generation failing to work when no error messages occur (just generated link flagging), it will be shown that *certain* forms of generated links actually work as intended, as long as the link generation mechanism is not pushed to the point of impossibility.

   In all cases, links involving *Direct JMP* or *Direct JMS* instructions properly function in the sense described above. (Statements attempting link generation of the form *JMS I* or *JMP I* cannot function as explained above.) If this were the only issue, lazy programmers might have a valid excuse to enable link generation.

   Unfortunately, all of the other forms of link generation may fail depending on the exact nature of the running program; the assembler cannot have knowledge of the programmer's intentions in the dynamic sense.

Repeating the generated link example from above:

```
       *0200          *200            /TYPICAL START ON PAGE 1.

000200  1777'         TAD     577     /THIS REALLY CAN'T WORK.
       *0377
000377  0577
```

A possible use of the example program includes manual start from the computer front panel; this allows setting the data field (DF) to a non-zero value in the range of 1 through 7.

When started this way, the program does not accomplish the intended function; the contents of location 0577 in the data field are added to the accumulator.

In general, PDP-8 programs are written for the 32K memory address space. The data field is often used as an internal convention of the program logic. When the program uses direct references (as the nominal statements attempt to do in the example), the data field plays no role in the logic of the program. Programmers who never enable link generation are certain their programs function as expected (as long as no other statements are flagged with errors).

Note: Lack of generated links is no guarantee there are no logic errors in the program; however, at least this area of bad design is avoided.

4) As soon as extended memory factors creep into the logic of a program dependent on link generation, insidious failures start to occur. The programmer who never learned the discipline required to avoid this programming pitfall is now faced with a nearly insoluble dilemma. The only remedy to this situation is to *bite the bullet* so to speak, eliminating all generated links as a first step towards debugging the program; by preventing the attendant error messages much of the problem will be eliminated.

   Note: Cleaning up after link generation might involve a large expenditure of effort all at once; had links never been used, the work necessary to eliminate assembly errors might have been accomplished in small increments. (And perhaps in the proper time frame to remember the constraints on the code as intended; all programmers are incapable of remembering everything, some sooner than others.)

In short, link generation correctly functions only if *Direct JMP* or *Direct JMS* instructions are involved. Link generation fails to work if any form of *AND, TAD, ISZ or DCA* instructions are involved because the logic is at the mercy of the very real possibility the data field (sometimes) doesn't match the instruction field when the program executes. (Or the prevailing data field might change in a future variation of the overall program.)

Positing a stubborn and lazy PDP-8 programmer (the author has met people who fall into this category; their careers as PDP-8 programmers who use generated links are always short), one of those who insist on using link generation despite all the warnings; their excuses are invariably within the list as follows:

a) Use only *JMP* and *JMS* links.

b) Never use the cases that cannot work and count how many links are generated in the assembler statistics page every time to ensure nothing important has changed.

   There are inherent logical fallacies in a) and b) above.

c) One can only assume a) is true; it is necessary to check the entire program to ensure this is the case. Confirming this is far more work than merely avoiding link generation.

d) The count of links generated may be the same as in a previous assembly. However, this is still no guarantee all of the generated links avoid the impossible cases; if the code is moved around sufficiently, a *JMP* or *JMS* link could drop out and be replaced with one of the impossible cases in terms of the assembler's generated links tally. It is still necessary to carefully check the entire program as in c) above; the unfortunate need for additional work still applies.

e) In general, people this stubborn are hypocritical. They will tend to use the potentially impossible form of generated links claiming certain knowledge of the data field throughout the entire program. This makes them vulnerable to the various insidious problems described earlier.

f) When cross-field calling conventions are established in meaningful programs, it is likely belatedly understood just how problematic the burden of link generation actually is. The work involved in elimination of the problems can be profound.

g) In the case of P?S/8 PAL, PAL10 and PAL8, additional option switches must be set to suppress error messages associated with link generation. This is by design, as the developers of successful PDP-8 operating systems virtually never use link generation; they are trying to guide programmers to success in PDP-8 programming in the sense of leading by example.

2.08 Literal statement examples

P?S/8 PAL supports the two general forms of literal generation along with a unique third form that can shorten program development time:

a) Current page literals - The operand address is located on the current page of the program; the address is assigned by the assembler starting at relative location 177. Additional literals are created in reverse order; a second current page literal is placed at relative location 176.

   The maximum number of current page literals allowed is 63 (decimal) in the relative range of 101-177. When a statement contains a current page literal, the current page literal table is searched for a matching value. If successful, the existing literal address is used. New values create additional current page literals. When either too many current page literals are attempted or statements on the current page overflow into the literal table, a PE (Page Exceeded) error message is displayed.

   A typical current page literal usage follows:

```
          TAD    (3)       /ADD 0003 TO THE ACCUMULATOR.
```

   Note: The trailing right parenthesis character is optional unless nested expressions are used.

b) Page Zero literals - The operand address is located on Page Zero; the address is assigned by the assembler starting at location 0177. Additional literals are created in reverse order; a second Page Zero literal is placed at location 0176.

   The maximum number of Page Zero literals is 112 (decimal) in the range of 0020-0177. (Locations lower than 0020 are disallowed due to the special auto-index characteristics of locations 0010-0017 when indirectly referenced.)

   When a statement contains a Page Zero literal, the Page Zero literal table is searched for a matching value. If successful, the existing literal address is used. New values create additional Page Zero literals. When either too many Page Zero literals are attempted or statements on Page Zero overflow into the literal table, a ZE (Page Zero Exceeded) error message is displayed.

   A typical Page Zero literal usage follows:

```
          TAD    [3]       /ADD 0003 TO THE ACCUMULATOR.
```

   Note: The trailing right square bracket character is optional unless nested expressions are used.

c) Conditional literals - The operand address is either on the current page or on Page Zero depending upon specific circumstances. If a matching Page Zero literal value exists, it will be used. Failing a Page Zero match, If a matching current page literal exists, it will be used. Failing both a Page Zero match and a current page match, a new current page literal is created (subject to the possibility of a PE error if the current page overflows).

A typical conditional literal usage follows:

```
          TAD    #3        /ADD 0003 TO THE ACCUMULATOR.
```

The actual code generated is dependent on the literal pools in effect at the point in the assembly the usage is encountered.

Note: Since there is no corresponding closing character, it is impossible to create nested conditional literal expressions.

A common technique employed in well-developed programs is the use of source code libraries of commonly used subroutines. It is desirable to use Page Zero literals where appropriate. However, Page Zero literals should not be used if there are only singular references. By using conditional literals, more effective management of memory is achieved without the need to hand-optimize the library source code statements containing literals.

Additional information about conditional literals and related subjects can be found in Appendix C of this document.

Note: This feature is not supported by PAL10 or PAL8.

2.09 Literal extent issues

In general, all literals associated with any current page should be defined in proximate order to the code statements referencing them. This facilitates locating any potential matches to avoid creating additional literals (if possible).

Leaving the present page, the existing literals are dumped to finish the intent and requirements of all source code statements defined so far.

If this page is ever revisited during the assembly, it is possible to define additional literals because the literal extent value formerly in effect was saved and will be restored. However, since the actual former literals have been dumped, every new literal statements will only be able to cause a search for a match within the newly-created literal pool subset.

Note: This feature, while enabled by default, is hardly used. It is supported primarily to maintain compatibility with the literal generation features of PAL10 and PAL8.

If the /E command-line option switch is used, the literal extent when leaving a current page is reset; no history of previous literals on any current page is retained.

Note: The use of the /E command-line option switch does not impact page zero literals (which are completely reset upon either the end of the assembly or the use of the FIELD directive). In certain PDP-8 documents, this is referred to as *forgetting* current page literals.

Resetting literal extents is useful when defining overlays. The overlay code must be created in the context of another section of the assembly which presumably will be replaced in memory by the overlay code; however, the overlay code must maintain its own current page literals as required.

while there are many techniques associated with the deployment of overlays, the /E command-line option switch is always required to allow overlays to maintain their own current page literal pools within the overlay section.

Note: Future implementations of P?S/8 PAL (or a new assembler for the P?S/8 *SHELL* based on P?S/8 PAL tentatively to be known as *PALX* or perhaps *XPAL*) will likely invert the default condition regarding literal extent retention due to disuse of the current default; as a practical matter, sections of program code that revisit formerly used pages are virtually always part of an overlay structure. There are no justifiable reasons to deliberately separate small program sections that load on the same code page.

2.10 Dual assembly mode options

P?S/8 PAL is the only known PDP-8 assembler that fully supports LINC mode assembly including conditionals and one's complement arithmetic, literals, large assemblies and the full complement of PDP-8 assembler features (including conditional assembly features generally associated with the LAP6-DIAL/DIAL-MS embedded assembler).

Note: While LAP6-DIAL/DIAL-MS only runs on the PDP-12 computer, P?S/8 PAL does not require LINC hardware to assemble LINC mode programs. Additionally, the LAP6-DIAL/DIAL-MS embedded assembler does not support literals and many other standard PDP-8 assembler features.

Dual assembly is enabled in P?S/8 PAL as follows:

/8 Enable LINC mode assembly; the default operating mode is PDP-8 mode (*PMODE*). While not strictly necessary, it is good practice when enabling dual assembly to state the desired initial operating mode; this is equivalent to using the PMODE directive. The current location counter is set to the usual PDP-8 starting value of 0200.

/9 Enable LINC mode assembly; the default operating mode is LINC mode (*LMODE*). While not strictly necessary, it is good practice when enabling dual assembly mode to state the desired initial operating mode; this is equivalent to using the LMODE directive. The current location counter is set to the usual LINC starting value of 4020.

Enabling the dual mode assembly module requires the allocation of three additional pages of memory. Consult Section 2.13 of this document for more information regarding memory allocation and related issues.

2.11 Dual mode assembly tips

While it is good practice to start dual-mode coding with the appropriate mode directive, it is possible (using conditional assembly techniques) to determine the current assembly mode. References to the current definition of the *HLT* instruction (which will be 7402 in PMODE and 0000 in LMODE) can determine whether LMODE or PMODE is in effect. Critical programs can abort the assembly if it is determined the assembly process is not in keeping with source code requirements (which could include failure to enable LINC mode assembly).

While the use of literals generally applies to PDP-8 code, certain language features can be adapted to LINC mode programming using appropriate addressing restrictions. Advanced users can contrive viable cases (minimally) worth pursuing.

In general, LINC programming does not require literals due to significantly different addressing modes. Immediate mode addressing conceptually replaces literals because the immediate mode operand cannot be (efficiently) used in other instructions. Immediate mode can specify the address of common variables; this can be likened to the use of literals. If the addresses can be contrived to conform to PDP-8 page addressing, actual PDP-8 literals can be used obtaining the intended results.

Note: Any form of arithmetic expression used to calculate LINC mode addresses is performed using one's complement arithmetic.

All LINC instructions on the PDP-12 and the classic LINC are implemented in hardware. On the LINC-8, certain instructions are trapped and emulated using a support program known as *PROGOFOP* (*PROG*ram *OF OP*eration). PROGOFOP is required for 100% emulation support; however, short sections of LINC programming can be initiated by PDP-8 programming with interrupt-handling optional.

Note: The PDP-12 also supports an alternate form of trap simulator to allow programs written for the classic LINC and/or LINC-8 to be run as intended, perhaps with alternate interpretation of the LINCtape class instructions. For more information, consult DEC documentation of the PDP-12 trap simulator.

2.12 LINCtape information

P?S/8 and OS/8 configured for PDP-12 TC12 LINCtape operation use 128 words/block LINCtapes; LAP6-DIAL/DIAL-MS uses 256 words/block LINCtapes formatted significantly longer than the maximum length handled by the classic LINC (512 blocks maximum). However, certain DIAL-MS configurations have limitations when accessing blocks larger than the classic LINC limit (0777 octal) such as when logical storage units are implemented on the RK8E/RK05.

This is apparently caused by some internal confusion about the size of a logical unit sometimes allowed to be exceeded by hardware limitations; the RK05 is arbitrarily divided into logical units exactly the size of the classic LINC tape storage as opposed to the typical amount of storage available on the DIAL-MS extended-length LINCtapes. Perhaps further information can be obtained from maintenance programmers, since this was a later add-on to this system and not part of the original system as created primarily by Jack Burness; the rest of DIAL-MS predates the existence of the available hardware (the RK8F added to a DW8E-P).

Utilities to access 256 words/block LINCtapes require custom programming specific to each machine. To transfer assembled programs to the classic LINC requires a custom image utility that runs under the current operating system; conventions for LINC operating systems vary widely, other than the same underlying LINCtape format. LINC systems cannot access 128-word LINCtapes associated with P?S/8 and OS/8; using custom programming, the LINC-8 can handle all formats because the underlying hardware is actually a PDP-8 storage peripheral. The PDP-12 TC12 LINCtape subsystem supports extended operations to work with any viable LINCtape format. P?S/8 PAL can be instrumental in creating appropriate utilities as needed.

P?S/8 PAL includes all common LINCtape instruction definitions as used on the LINC, LINC-8 and the PDP-12. These are generally double-word instructions with block information in the second word immediately following the LINCtape instruction.

Note: No useful purpose is served creating 129 words/block LINCtapes for use with P?S/8 or OS/8 on the PDP-12. There are only downsides to doing so as follows:

a) The only reason 129 words/block LINCtapes exist is because the most common LINCtape formatting programs on the PDP-12 were created as part of the LAP6-DIAL/DIAL-MS operating system. Ignorant decisions were made to only provide 129 word format (when choosing other than 256 words/block options). There are no circumstances where this option is ever constructively used. (Contrast this with LINC-8 MARK program (*MARKL8*) options that generally include standard 256 words/block LINCtapes and 128 words/block LINCtapes only.)

Note: It is understandable how this blunder came about. Only those familiar with PDP-8 DECtape software understand the relevant reasons (and apparently the software staff of the LAP6-DIAL/DIAL-MS project were not aware of any of this and likely guessing). Only the Disk Monitor on DECtape actually uses the 129th word on DECtape blocks for data purposes. All other supported systems use the first 128 words of each block. Thus, the 129th word is ignored and merely present because all DECtapes must be an integer multiple of 18-bit words for compatibility issues that do not pertain to LINCtapes.

The P?S/8 version of the *MARK12* formatting program is modified to create 128 words/block LINCtapes as used on the LINC-8 and PDP-12; this applies equally to both P?S/8 and OS/8 on both hardware configurations.

Note: The source code for the P?S/8 version of MARK12 (as well as the companion *TC12F* program for DECtape and LINCtape conversion) can be assembled with P?S/8 PAL using the /9 command-line option switch to enable dual-mode assembly starting in LMODE.

b) Both P?S/8 and OS/8 for PDP-12 LINCtape support a nuisance feature to allow reduced functionality on 129 words/block LINCtapes. When 129 words/block format is used, the memory location following the read/write buffer can be corrupted during a read operation. As such, all relevant handlers for both systems preserve the potentially-destroyed word, then restore it after the read has completed. This also means that system device handlers for both operating systems must be able to function while potentially the contents of location 07600 are at risk. As such, there is a small chance that if the computer is manually restarted at an improper moment in time, the system device handler might become corrupted.

   The circumstances that arise for this potential problem are actually quite common; any core-image program that loads into 07400-07577 is potentially at risk for destruction of the system kernel that starts at 07600.

c) The LINC-8 system (and non-system) device handlers for P?S/8 and OS/8 cannot handle the 129 words/block format LINCtapes. In fact, no software exists to read or write such tapes on the LINC-8 presently. (Specific conversion software would be necessary to manage this situation which is totally avoidable if the proper tape format is used. For example, LINCtapes can be formatted on the LINC-8 with 128 words/block and copies of the LINCtapes intended to be read on the LINC-8 can be written on the PDP-12.)

d) All tape transfers are degraded slightly by the presence of the erroneous 129th word. This is because all blocks of the tapes are needlessly longer; extra time is needed to move the tape past all blocks between the present tape block position and the target block during any read or write transfer.

e) Since each block is needlessly longer, the maximum number of blocks that can be utilized is needlessly foreshortened. Several standard tape lengths are generally supported. This will tend to make certain tapes ineligible for extra-long formatting considerations unless 128 words/block is used.

f) As a long-term consideration, there is needless increased wear and tear on tape media and drive tape guides.

2.13 Memory management options

/C Enable internal reconfiguration of assembler resources. This may allow certain assembly configurations on smaller machines to be viable at the expense of overall performance. See below for additional configuration details.

P?S/8 PAL is designed to run on 4K or larger machines; additional memory, if available, is used to allow a larger symbol table which is generally the limiting factor with regard to overall program size that can be successfully assembled.

When P?S/8 PAL is first loaded, the passed command-line option switches are checked to determine the running configuration size with regard to modular assembler components retained once the assembly process begins. Deselected modular sections are removed; retained modular options are then relocated as required to maximize the space available for symbol table and input/output buffers.

Depending on options selected, it is possible that attempts to load the symbol table can overflow the assembler's capabilities; the selected configuration cannot function with standard allocation of input/output buffers.

To achieve a viable configuration, the number of input/output buffers can be reduced; memory space is then made available for additional symbols. The assembly process will take longer due to fewer pages allocated as buffers (but will more likely succeed while handling more symbols).

To gauge the magnitude of the problem on 4K systems, the following is a list of the extra memory allocation requirements (over the minimum needed for an assembly without any modular sections retained):

a) Niceties related to the /N command-line option switch (fully formatted headers on specific aspects of the listing output) requires one additional page.

b) Literal and/or link generation requires three additional pages.

c) Enabling dual mode (LINC) assembly requires three additional pages.

d) Enabling cross-referenced listing output requires two additional pages.

Each page reallocated for symbol table usage adds the ability to support 32 additional symbols; an assembly with no modular options retained can support 288 additional symbols over the included internal symbol table.

If the /C command-line option switch is set, the input and output buffers are reduced to a minimum complement causing the assembly process to take considerably more time; however, sufficient memory space becomes available to support any combination of the modular program options as described above.

P?S/8 PAL is designed to dynamically take advantage of additional memory if available. On all systems with 8K, at least another 3K of memory is available for the symbol table; as such, all modular components can generally be retained without restriction for most assemblies.

Note: Certain P?S/8 configurations reserve the highest 1K (quarter memory field) available for the system device handler extension. For all systems that do not require the handler extension, the full additional 4K of memory can be made available to P?S/8 PAL (and other programs).

When 7K (or more) memory is available, symbols are not stored in field 0. The /C command-line option switch can be used to sacrifice performance while allowing additional symbols residing in field 0.

The maximum number of symbols allowed is 4095 (including all permanent symbols and directives). If 16K of memory is available for symbols (20K total memory or more), there is no point in enabling the /C option other than to perform timing tests to measure the resulting performance degradation; the exact configuration with regard to the modular options becomes moot for larger machines.

Note: PAL8 requires 24K (or more) memory to support the maximum symbol table size of 4095 symbols due to certain design limitations of OS/8 that fractionalize every memory field.

2.20 P?S/8 PAL assembler performance

P?S/8 PAL performance was compared to PAL8 using a deliberately biased configuration (in favor of OS/8) (and a stopwatch for accuracy). While P?S/8 PAL took twice as long to assemble a large file chosen at random from various OS/8 source programs under development at the time, the P?S/8 configuration was based on TC08/TU56 DECtape while the OS/8 configuration was based on the RK8E/RK05 removable cartridge disk (which is a much faster system at the hardware level). Given the overhead of the underlying system devices, it is estimated that P?S/8 PAL can assemble large programs as much as four times as fast as OS/8 PAL8 on identical hardware configurations.

Note: Performance of the P?S/8 BIN Slurp format binary loader was compared to OS/8 *ABSLDR* on the same configuration. The DECtape-based P?S/8 Slurp binary loader successfully loaded binary files into memory **twice as fast** as OS/8 ABSLDR on the RK8E/RK05. This seemingly counterintuitive feat, when demonstrated to various DEC PDP-8 programming group employees, induced a state of cognitive dissonance. They could not comprehend the notion that a DECtape could somehow perform faster than an RK05. (Those familiar with the internal design of both operating systems better understand this predictable result.)

2.30 Additional binary chaining options

Various options of the P?S/8 BIN utility apply during a successful chain operation. Assuming no assembly errors, chaining to P?S/8 BIN is generally used for the purpose of loading the binary output files created during the assembly.

Note: See the description of the /W command-line option switch below for the alternate purpose of chaining to P?S/8 BIN for the purpose of punching binary paper-tapes.

As explained above, chaining to P?S/8 BIN creates a list of binary input files derived from the list of output files created during the assembly; this list may have been obtained from explicitly passed output files, or as a result of some particular combined usage of the /B, /D and /U command-line option switches. As necessary, the list may have been truncated if excess output files were specified.

The following additional command-line options of P?S/8 BIN apply if set in the original P?S/8 PAL command line:

/G Chain to P?S/8 BIN for the purpose of either loading the binary files created during the assembly or punching binary paper-tapes of the binary files (if the /W command-line option switch is also set). The chain is inhibited if there are assembly errors.

/V Enable the virtual Slurp format binary file loader instead of the hardware-specific Slurp format binary file loader (if relevant).

Where possible, P?S/8 BIN loads binary input files using an implementation of the Slurp format binary loader as originally defined by Richard Lary for the R-L Monitor System. The P?S/8 variant supports loading anywhere within the 32K PDP-8 memory space (albeit with restrictions that apply to system-reserved memory areas).

Specifics of the P?S/8 system configuration will indicate whether a hardware-dependent Slurp format binary loader is available; only certain configurations support a device-specific Slurp format binary loader.

On systems that do not include a device-specific Slurp format binary loader, a virtual Slurp format binary loader is used that requires the presence of the system device handler. Other than certain performance issues and minor tradeoffs, the two loaders perform the same overall function.

If the /V command-line option switch is set, the virtual Slurp format binary loader is used instead of the device-specific Slurp format binary loader. This is a user preference option (assuming the device-specific Slurp format binary loader is available); it may be desirable to force usage of the virtual Slurp format binary loader in certain circumstances.

/I If a device-specific Slurp format binary loader is available, the system device handler is reloaded after all binary files are loaded. As a minor restriction, it is not possible to pass the maximum input file count of 17 input files; the maximum input file count is reduced to 16 (which is generally of no consequence to most users and actually impossible when chaining from P?S/8 PAL because at least one input file is required to create binary output).

Note: If the virtual Slurp format binary file loader is in effect, the /I command-line option switch is ignored; the slight restriction on maximum binary input files does not apply.

If a device-specific Slurp format binary file loader is in effect and the /I command-line option switch is not set, the system device handler will not be available; however, a device-specific bootstrap will be placed into locations 07600-07642 to allow program exit to 07600 in a manner consistent with the system device handler being present. (Starting the bootstrap code will reload the P?S/8 keyboard monitor.)

Note: While the starting location of the device-specific bootstrap is always 07600, the ending address is always assumed to be the largest case allowed for any and all P?S/8 system configurations (presently 07642); for certain device-specific bootstraps, this may cause superfluously moving additional words beyond those actually required to function.

Moreover, the largest case allowed is scheduled to be reduced to 07641 in future releases of P?S/8; this is due to considerations where the device-specific bootstrap code is utilized unrelated to binary chaining to P?S/8 BIN from P?S/8 PAL as discussed in this document. As of this writing, the longest existent device-specific bootstrap is slightly shorter than either the present or planned limit address.

For all systems requiring a system device handler extension, it is likely (in all future releases of P?S/8), the bootstrap for all such systems will conform to the following:

```
  CIF  x0                  /CIF TO THE HIGHEST MEMORY FIELD.
  JMP  7600                /CONTINUE THERE.
```

As such, all systems requiring a system device handler extension section will produce the smallest bootstrap starting at 07600. In these instances, moving additional words will be irrelevant. (At least one configuration already conforms to this convention as of P?S/8 Version 8Z.)

/H All user memory is preloaded with 7402 (the PDP-8 HLT instruction) before binary file loading. This is highly recommended when debugging programs under development to prevent unexpected loss of control.

/Z All user memory is preloaded with 0000 (the LINC HLT instruction) before binary file loading. This is highly recommended when the program under development contains numerous LINC instructions or it is expected the binary output may ultimately be saved in core-image format; unloaded locations will show 0000 in any form of memory or storage block dump printout.

P?S/8 can create a core-image of a program currently under development (using the GET /Z command, or as a result of a chaining operation as described above passing the *=7632* value and the /V or /I command-line option switch) for further processing. The resultant core-image blocks can subsequently be transferred to pairs of named TFS files which can later be restored to the % and $ files if deferred execution or debugging is desired.

Note: The P?S/8 GET program is an alias of the P?S/8 BIN program. The only functional difference is the starting address will be forced to 07632 (by clearing the /1 through /7 command-line option switches and forcing the =7632 parameter value) before otherwise starting the P?S/8 BIN utility program (for the purpose of loading binary input files into memory). The user can also invoke these parameters in an explicit keyboard monitor command executing the P?S/8 BIN utility.

Future releases of P?S/8 will include the SHELL overlay (currently under development) which will support versions of the *ENCODE* and *DECODE* programs (currently available as part of the *Kermit-12* program collection for OS/8). Core-image files created in the P?S/8 SHELL environment (through means analogous to the GET command usage as described above) passed through the ENCODE program will produce smaller output files due to the compression of uninitialized storage locations.

If the optional =xxxx numerical value is passed, the value xxxx is used as the program starting address; the default program start field is 0. Assuming the program will be started after loading, specifying any of the following option switches will apply as necessary:

/1 The program will start in field 1 at location xxxx as specified in the =xxxx passed numerical value.

/2 The program will start in field 2 at location xxxx as specified in the =xxxx passed numerical value.

/3 The program will start in field 3 at location xxxx as specified in the =xxxx passed numerical value.

/4 The program will start in field 4 at location xxxx as specified in the =xxxx passed numerical value.

/5 The program will start in field 5 at location xxxx as specified in the =xxxx passed numerical value.

/6 The program will start in field 6 at location xxxx as specified in the =xxxx passed numerical value.

/7 The program will start in field 7 at location xxxx as specified in the =xxxx passed numerical value.

Note: All of the options stated above assume the /W command-line option switch is not set (to allow binary file loading).

If the chain to P?S/8 BIN is for the purpose of an *assemble-and-punch* operation to create binary paper-tapes, the following command-line option switches will apply:

/W The /G command-line option switch is set; the chain to P?S/8 BIN will be used to punch binary paper-tapes if there are no assembly errors (else the chain to BIN will be prevented).

/F The default device for the binary paper-tape output is the low-speed console device paper-tape punch found on various console terminals such as the Teletype Model 33 ASR and Model 35 ASR (device 04). If the /F command-line option switch is set, the binary output is directed to the high-speed punch (device 02).

/R By default, the paper-tape output is punched in DEC BIN format. If the /R command-line option switch is set, the output format used is enhanced DEC *RIM* format (which includes a checksum allowing the binary paper-tape to also be read by the DEC BIN loader).

Note: the /0 command-line option switch is not passed to P?S/8 BIN as it has alternate meaning within P?S/8 PAL. See Section 2.50 for additional details.

2.40 Text management options

/J Prevent generation of a zero-fill word (0000) after text strings with an even count of characters. This applies to all usage of the TEXT and SIXBIT directives.

2.41 Text directive issues

The PAL language defines the TEXT directive which is used with a string argument. The string consists of an indefinite number of printable characters preceded by a delimiter character which cannot be included in the generated text; the string ends with a second usage of the delimiter character.

Note: P?S/8 PAL does not require the trailing delimiter character unless there are multiple statements on the same line.

Early implementations of the TEXT directive restricted the delimiter character to ", ' and in some cases @ (which is not supported by P?S/8 PAL and other assemblers). The use of " or ' as text delimiters is recommended for maximum compatibility with other PAL assembler implementations; P?S/8 PAL and PAL8 support an extension allowing any printing character not contained within the text string to be used as the delimiter character.

All usage of the TEXT directive creates strings of six-bit ASCII text characters packed two per 12-bit word. If the count of characters in the string is an odd number, the last 12-bit word has 00 in the low-order six bits (which generally is used as a string delimiter).

Since the most likely usage of the TEXT directive is to generate strings used with printing subroutines, the default action for string arguments with even character counts is to generate an additional 12-bit word of 0000 to ensure the presence of a string delimiter.

TEXT directive usage is not limited to string creation as described above; some alternate schemes use internal printing characters as delimiters or other control mechanisms. Within the P?S/8 keyboard monitor coding, explicit string length arguments are used in some instances; in other cases the string length is an implicit constant value. The 00 word plays no role in any of these cases; as such, creation of an extra 12-bit word of 0000 can be undesirable.

Note: Early PDP-8 programs such as FOCAL, 1969 were developed on systems incapable of modifying the output of the TEXT directive; as such, the source code includes statements to backup the assembler current location counter to allow the desired contents which differs from the automatically generated 0000 value. Ironically, this form of source-level workaround is incompatible with the /J command-line option switch (which eliminated the problem several years after FOCAL was developed).

As documented above, setting the /J command-line option switch prevents the creation of the extra 12-bit word of 0000 on even count text strings. It is strongly recommended the disposition of the /J command-line option switch be documented at the beginning of all source code; in many cases, this may be the only departure from assembler default settings and could easily be overlooked.

2.42 Other text-related issues

P?S/8 PAL supports the SIXBIT directive which uses the same overall format as the TEXT directive. The only difference is the internal six-bit code used is not the PDP-8 standard six-bit subset of the ASCII character set; instead, an alternate definition of six-bit text (as used on DEC 18-bit and 36-bit computers) applies.

The basic difference between the two character sets is the highest of the six bits used is inverted. Input processing of seven-bit ASCII characters to six-bit code is easier using the PDP-8 standard while output processing of six-bit code to seven-bit ASCII characters is easier using the alternative standard. The PDP-8 standard was chosen largely because there are often multiple input routines while output is generally performed by a single output routine. For most programmers, it is easier to observe TEXT-related six-bit code to relate to the corresponding original seven-bit ASCII characters.

Note: PAL8 and PAL10 do not support the SIXBIT directive.

The DEVICE and FILENA{ME} directives are supported by P?S/8 PAL and PAL8. These are special-purpose text directives to create a four-character and eight-character text string respectively. Early OS/8 programs required the use of the /J command-line option switch (or its equivalent in PAL8 which is the /F command-line option switch); while this allows creation of the intended string arguments, compatibility is not possible with the usual string printing applications, which require even-length string termination with a trailing word of 0000. With the availability of the DEVICE and FILENA{ME} directives, the normal flexibility of text string creation choice is restored.

Note: P?S/8 supports the FILENA{ME} directive for compatibility purposes only. Future P?S/8 systems will support the P?S/8 SHELL with an extended assembler based on P?S/8 PAL to be known as *PALX* (or perhaps *XPAL*). PALX (or XPAL) may support an option to extend the FILENA{ME} directive (or some functional equivalent) to support SHELL directory file name conventions which are as follows:

a) OS/8 file names have six significant six-bit text characters. As such, all of the characters are upper-case alphabetic or numeric characters.

   OS/8 file name extensions are two characters or less in the same character set as the file name.

b) P?S/8 SHELL file names have twelve significant six-bit text characters. The file names support case retention (which is not the same as case-sensitivity). References to SHELL file names can be made in any form of mixed case usage.

   P?S/8 SHELL file extensions are up to four characters expressed in the same character set as the file name.

The PALX (or XPAL) language will include upper/lower case support to allow fully formed P?S/8 SHELL file names.

In the P?S/8 SHELL environment, the file names are packed into six words containing the six-bit upper-case representation of the file names; the case bits are stored separately for the name and the extension.

As such, two additional directives will be required, tentatively proposed as the CASENA{ME} and the CASEEX{TENSION} directives for the file name and file extension portions respectively. The CASENA{ME} directive will return a 12-bit value with the actual case of the corresponding name characters obtained from the last usage of the extended FILENA{ME} directive. The CASEEX{TENSION} directive will return a four-bit value with the actual case of the corresponding extension character obtained from the last usage of the extended FILENA{ME} directive. Since trailing characters in both the name and extension portions of an extended file name may not be present, the case bits will only be set on actual lower-case characters in either portion of the last FILENA{ME} directive usage. (The high-order eight bits of the returned value of CASEEX{TENSION} will always be cleared.)

In a case-retentive system (such as the P?S/8 SHELL), the case of each file name character is retained for optional use in printing utilities; actual search functions are performed in a caseless manner. Other than the length of compare strings, this is consistent with the basic P?S/8 system and OS/8.

Note: To maintain complete compatibility with OS/8, a separate FILNAM{E} directive may be used to support the SHELL file system names; the FILENA{ME} directive will also be available as presently defined.

Certain user-modified assemblers exist that support the *TEXTZ* directive which forces the extra 12-bit word of 0000 even if the /J command-line option switch (or equivalent) is in effect. Future versions of P?S/8 PAL may support the TEXTZ directive and the *SIXBTZ* directive (using the alternate six-bit text code).

Note: As of PAL8 Version B0 (the last version released by DEC within OS/278 Version 2), the TEXTZ directive is not supported.

Most PAL assemblers (including P?S/8 PAL and PAL8) support the " operator in expressions. The result is the seven-bit ASCII code for the next character with the mark parity bit set.

A proposed *ASCII* directive would take a string argument much like the TEXT directive to create a string of seven-bit ASCII characters. Setting the /J command-line option switch would prevent a trailing 12-bit word of 0000.

An additional *ASCIIZ* directive could be implemented to force the creation of the trailing 12-bit word of 0000 regardless of the setting of the /J command-line option switch.

Note: An additional issue is whether to have an option to prevent the mark parity bit within the text string as arithmetic operations are not possible on individual string characters.

Even without any form of ASCII (or related) directive as stated above, it is possible to create seven-bit ASCII text strings (with any of the variations required) using the " operator. When used alone, each character in the string will have the mark parity set; string delimiter characters can be added if desired, such as ending the string with a 12-bit word of 0000 or making the last character of the string negative (as a 12-bit signed integer). For example, by adding an additional term in each expression, it is possible to strip off the mark parity bit as follows:

```
HLOWRLD,"H&177;"E&177;"L&177;"L&177;"O&177;" &177 /HELLO WORLD MESSAGE
        "W&177;"O&177;"R&177;"L&177;"D&177!4000   /NEGATIVE ENDS TEXT.
```

An unnamed additional text directive has been proposed to create packed seven-bit ASCII text strings following the *three for two* packing convention as used in OS/8 (and P?S/8 SHELL) text files. Because of alignment restrictions, additional null eight-bit bytes will be required, regardless of the issue regarding the mark parity bit. If the string is meant to be an image of an actual file section, there must also be a mechanism to append a trailing Control-Z character into the string.

Note: The issue of a three for two directive will likely be taken up in the PALX (or XPAL) assembler which will support upper-case/lower-case text input files. A *PARITY* directive with appropriate argument support can be implemented to handle all mark parity bit (and perhaps the Control-Z character) issues raised above.

2.50 Additional command-line option switches and issues

/0 Automatically generate *200 after the use of the FIELD directive.

Note: Most programs have data or explicit origins after FIELD settings. By using an explicit origin setting, a known serious bug of one of the released DEC BIN loaders (known as the *self-starting* binary loader) can be avoided.

The problem with this version of the DEC BIN loader (which was never fixed) is that when a FIELD setting is encountered, it is applied prematurely; the last data word from the previous address (presumably intended for use with the former field setting in effect) is erroneously loaded into the new field. By ensuring the last data word is an origin setting, this problem can be avoided.

As a practical matter, most well-designed programs use specific origins after FIELD settings for appropriate purposes; as such, this option is generally irrelevant.

/Y The ! operator is changed from inclusive OR (.IOR.) to shift left six bits.

Without the /Y command-line option switch set, 22!44 generates 0066; with the /Y command-line option switch set, 22!44 generates 2244.

The ! operator (with the /Y command-line option switch set) can be used to create short six-bit text expressions:

```
A=      "A&77

B=      "B&77

A!B
```

Each word created using the ! operator in this manner produces the same result as a 12-bit component (two characters) of a text string.

Note: It is not necessary to use the ! operator to create an equivalent result. The following syntax will create the identical value:

```
A^100+B
```

More complex text operations can produce similar results derived directly from the actual seven-bit character values:

```
"A&77^100+"B-300
```

By the use of these alternatives, the standard definition of the ! operator is available (and use of the /Y command-line option switch can be avoided).

Certain programs require the default ! operator when used to embellish combined OPR or IOT instructions:

```
CLA!CLL!CML!RTL             /SET THE ACCUMULATOR TO 0002.

DTSF!DTRB                   /SKIP ON DECTAPE DONE OR ERROR FLAG
                            /SET; READ THE ERROR REGISTER.
```

Programmer preference will determine whether to enable the /Y command-line option switch. As in the case of the /J command-line option switch, if the /Y command-line option switch needs to be set, this should be documented in the beginning of the source code to ensure proper assembly.

2.60 P?S/8 PAL directives

P?S/8 PAL is fully compatible with all standard directives (also known as *pseudo-operations* sometimes abbreviated as *pseudo-ops*) as implemented in PAL8. Certain additional directives are supported that are compatible with other assemblers such as the LAP6-DIAL/DIAL-MS embedded assembler or PAL10. Specific directives are unique to P?S/8 PAL. There are serious issues associated with the *DTORG* directive as discussed below.

Note: A complete list of P?S/8 PAL directives is provided in Appendix B below.

2.61 Issues regarding the DTORG directive

The OS/8 implementation of the DTORG directive is deliberately avoided in P?S/8 PAL. A proposed solution to the underlying problem is presented below which will allow the DTORG directive to be useful to P?S/8 system generation in future releases of P?S/8 PAL.

2.62 PAL8 implementation of DTORG

As currently supported in PAL8, the DTORG directive is completely compatible with PAL10. The DTORG directive is meant to create (128 words/block) absolute block references within the binary output to allow a special-purpose loader (a component of the DEC *Typeset-8* group system programming software) to write Typeset-8 system blocks on DECtape (and perhaps other devices).

The general function is to specify a numbered storage block on a TC01/TC08/TU55/TU56 DECtape followed by the (up to) 128 words to be written to the specified block.

The origin in memory is set prior to the DTORG directive; however, it is generally ignored because no form of standard loader (such as the DEC BIN paper-tape binary loader) is supported. The origin is used to define the upper limit in memory of the (up to) 128 data words that follow; once an entire PDP-8 page has been defined (placing the current location counter on the next PDP-8 page), the intended block contents are finalized.

Typeset-8 programs create a series of small code sections as such. Each is loaded by a special-purpose Typeset-8 generation utility which writes the data to the specified DECtape (or other device) logical block. However, there are important issues:

a) Each section written to a targeted DECtape block is taken from a lengthy punched binary paper-tape created at an OS/8 development site; the tape invariably is the latest release of a major section of the Typeset operating system (or one of its utilities). In-house DEC Typeset-8 machines generally did not include high-speed readers; these systems invariably had Teletype Model 35 ASR terminals to read in the binary data.

b) There is no form of DEC BIN-compatible checksum protection associated with the written data. Since the binary paper-tapes are generally quite lengthy, this is patently foolhardy.

   By the nature of binary paper-tape, it is expected that reading in long paper-tapes will occasionally result in read errors. This is why the DEC BIN loader format includes the standard checksum protection frames.

   Note: The DEC RIM loader, which lacks a checksum mechanism, is only used to load short binary programs such as the BIN loader, which in turn can load in significant binary program data with the confidence provided by passing an overall checksum test. It is unreasonable to expect to use the RIM loader for lengthy binary files; the analogy between RIM binary paper-tape loading and Typeset-8 binary paper-tape loading is troubling.

c) Despite being created by PAL8 in OS/8 standard binary format (as a .BN file), the binary output violates the OS/8 binary file standard; the output of a program containing instances of the DTORG directive cannot be successfully loaded by OS/8 ABSLDR. Attempts to read in the punched binary paper-tape will always end in a checksum error, even if an error-free read was performed. This is because the implementation of the DTORG directive usurps the underlying DEC BIN format that OS/8 converts into *three-for-two* format OS/8 records (384 frames per logical record).

   As such, it is impossible to improve the performance of the Typeset-8 special-purpose loader; reading in the entire binary paper-tape will not yield valid checksum information that could be used to reject attempts to read the lengthy tapes that introduce additional flaws.

Note: It is thought that an alternate checksum scheme can be applied to the present incompatible binary format as described here. Compatible binary paper-tapes do not include FIELD setting data in the checksum while the first frame of the DTORG block argument sets the analogous bits. However, it is generally understood the Typeset-8 group never programmed for extended memory; it would appear that an alternate checksum scheme is impossible if both DTORG and FIELD statements appear in the same binary file. It is not known if the Typeset-8 group programmers even included checksum tests; reading in long binary paper-tapes and expecting the event to be free of read errors represents a serious misunderstanding of the nature of the paper-tape media.

This flawed position is likely based on the irrelevant fact that all products of the Typeset-8 group used premium quality BRPE (*Burpee*) paper-tape punches for six-level paper-tape support using TTS code (the standard of the industry as used by all Typeset-8 customers). It would be bad for their *image* to admit that other DEC products were simply not made to the same standards as was sold to the Typeset-8 customers.

This in part explains the presence of the (over-built) Teletype Model 35 ASR terminals everywhere in the Typeset-8 development labs which occasionally were subject to visits by various customer personnel. Visitors generally had only superficial understanding of the various hardware components but gained confidence in Typeset-8 by first impressions.

Teletype Model 35 ASR terminals look quite rugged (but still may have read errors). The PC04 readers generally attached to many PDP-8 systems do not inspire the same level of visual confidence.

This was also coupled with the false notion of claiming the dominant usage was for relatively small patches related to custom changes for the benefit of end-user systems.

We will likely never know if any development time was wasted by the Typeset-8 group as a consequence of this poor choice of manual data transport, as Typeset-8 group labs were generally located far from either more general PDP-8 development areas or TOPS-10 time-sharing system terminals.

More to the point, DEC management for all 12-bit systems never expected anything to be changed within the Typeset-8 group since, at the time, it was perceived as the only cost center producing a profit while the rest of the software development and related groups were unable to show any equivalent addition to the corporate bottom line.

Note: It is the author's opinion that much of what passed for productivity in the Typeset-8 group would more correctly be described as make-work (or *goldbricking*). (There is ample evidence of this obtained by examining the output of the OS/8 PTP: handler when text files are processed; at the time, the only beneficial usage of this special handling was offline printing on the Teletype Model 35 ASR terminals.)

2.63 Proposed remedies to the DTORG problem

Apparently due to internal DEC politics, no attempt was made to remedy this problem by educating the Typeset-8 programmers regarding their naive design; various well-known mechanisms can easily solve the problem as follows:

a) Instead of implementing the DTORG directive as a violation of the internal binary format of DEC BIN utilities and OS/8, the binary data can be patterned after a variety of utility formats such as certain details of P?S/8 non-system device handlers and all OS/8 device handlers (as required by OS/8 *BUILD*).

   An arbitrary series of origin settings can be used to indicate the intention of DTORG directive statements as follows:

```
DTORG  1234                /DESIGNATE BLOCK 1234 FOR OUTPUT.
```

   This DTORG mechanism would create the equivalent of the following binary output:

```
PREV=  .                    /NEED THE PREVIOUS ORIGIN SETTING.

*7777;*7777;*1234;*PREV     /OUTPUT FLAG ORIGINS FOLLOWED BY
                            /THE DTORG BLOCK NUMBER FOLLOWED
                            /BY THE PREVIOUS ORIGIN.
```

   A special-purpose loader is still required to create Typset-8 system blocks in the proposed format; however, the binary data created is compatible with all loading mechanisms; a lengthy binary paper-tape punched out and then read on a Teletype Model 35 ASR read station could be known to have passed the standard checksum test. Afterwards, all designated blocks can be written out at once (assuming all block images can be simultaneously loaded into memory).

   Note: The Typeset-8 system runs on 4K machines. The binary paper-tape output created for a customized user configuration would tend to be somewhat longer than the binary paper-tape for FOCAL, 1969 (as an arbitrary but likely representative example of comparable size, nearly the entire user memory space of 00000-07577). It is entirely possible far longer tapes could be required depending on how much of the system was being updated at any one time.

b) The example remedy represents an expedient method to solve the DTORG problem. More elaborate schemes could include additional origin settings, perhaps following the last two data-related origins as shown in the example. An overall mini-checksum for the entire string of origin settings could be created if desired. (The final origin must repeat the value in effect prior to the DTORG grouping to allow proper loading.)

   Regardless of specific implementation details, the entire problem of supporting incompatible binary files would be eliminated. All binary files can be used with all available binary format utilities. After punching the lengthy paper-tape, it could be read back using the high-speed paper-tape reader. The resulting file would not only have to pass the standard checksum test, but additionally could be directly compared to the original binary output file to completely confirm the validity of the paper-tape before bringing the tape to the Typeset-8 back-room facility (usually located in a remote building) for the purpose of generating Typeset-8 system components.

c) Without having to support data in an incompatible binary format, the need to store paper-tape binary frames becomes superfluous. OS/8 could have easily been developed using a more efficient internal file format similar to P?S/8 and the *R-L Monitor System* which would essentially be a 256 word equivalent of Slurp binary loader format.

   It is well known to the author (and certain original DEC staff members), that DEC managers perverted certain aspects of the demo project that was eventually to become OS/8. Their demonstrable ignorance of PDP-8 operating systems has left its mark on various forms of internal poor design aspects of that system.

   It is surmised that Richard Lary intended to use 256 word Slurp binary loader format for OS/8 (having just recently demonstrated the *R-L Monitor System* to the same *dim bulb*s who rejected that pioneering system), but the edict to use images of binary paper-tape frames clearly was created in part due to ignorant (and arrogant) demands of the Typeset-8 group.

   Note: PAL10 also implements the creation of incompatible binary paper-tapes; the DTORG support in PAL10 is identical to that of PAL8 and actually was accomplished prior to the existence of OS/8. However, since PAL10 is a cross-assembler, this has no impact on any aspect of the TOPS10 operating system performance; in OS/8 this poor design decision causes a noticeable loss of performance every time binary files are loaded by any OS/8 user.

d) OS/8 *PIP* could easily be modified to convert compatible binary files using DTORG directives (as proposed) into Typeset-8 format binary paper-tapes (assuming lack of cooperation from the Typeset-8 group). Thus, the problems associated with incompatible binary format would only affect the developers of Typeset-8, and not the entire OS/8 community at large.

   While not likely to occur, the binary format of OS/8 utilities could be updated to 256 word Slurp binary loader format which would improve the performance of OS/8 for all users.

e) While the DTORG directive is not useful for OS/8 purposes (every aspect of the present DTORG implementation is responsible for some negative aspect of OS/8 performance), since it addresses 128 logical words/block storage (while OS/8 addresses 256 word logical records), it can be useful for future implementation of P?S/8 system generation utilities. This would follow the overall flow of Typeset-8 system generation methods (without any of the flawed specifics).

   Future releases of P?S/8 PAL (and the SHELL assembler PALX or XPAL) will develop a more useful implementation of the DTORG directive; for the present, P?S/8 PAL merely implements DTORG as one of the nonfunctional directives supported for source-level compatibility reasons, but generates an incompatible subset of the proposed changes.

f) A P?S/8 utility, tentatively to be known as *BINCON*, may be written to read and convert present OS/8 binary files to the proposed format (with respect to the DTORG disposition); the resulting compatible binary format would then be stored as P?S/8 Slurp format binary files. This would implement binary compatibility with the proposed changes to DTORG as will be implemented in future releases of P?S/8 PAL. P?S/8 system generation utilities could then operate with binary paper tapes or other media developed on either operating system.

   Since it is intended that P?S/8 BINCON operate in a bi-directional manner (analogous to the OS8CON text conversion utility), OS/8 users can benefit from the advanced assembly features of P?S/8 PAL when loading (resulting) binary files in OS/8.

   Note: P?S/8 BIN can create binary paper tapes in DEC BIN format on the low-speed or high-speed punch; this can be accomplished by a chained *assemble-and-punch* operation as described in section 2.30 above.

   More information regarding P?S/8 Slurp binary file format is available in the document *P?S/8 Binary File Internal Description* available separately.

2.70 Additional information

P?S/8 PAL supports all mathematical operations commonly available in most PDP-8 assemblers (and several operations that are not available in competing assembler programs) as follows:

a) Unary operations.

```
VALUE= 100              /SET VALUE TO 0100.

VALUE                   /THIS IS 0100.

+VALUE                  /THIS IS 0100.

-VALUE                  /THIS IS 7700.
```

b) expressions.

```
VAL1=   40              /SET FIRST VALUE TO 0040.

VAL2=  100              /SET SECOND VALUE TO 0100.

VAL1+1                  /THIS IS 0041.

VAL2-1                  /THIS IS 0077.

VAL1^4                  /THIS IS 0200.

VAL2%4                  /THIS IS 0020.

VAL1+VAL2               /THIS IS 0140.

VAL1-VAL2               /THIS IS 7740.

VAL2!-VAL1              /THIS IS 7740.

RDE; 7\VAL2             /THIS IS 0702 FOLLOWED BY 7100.
```

Notes:

1) While it is completely correct to use unary operations within compound expressions, cases such as the one shown above are known to fail when using OS/8 PAL8 due to misimplementation (contrary to OS/8 documentation). As such, it may be necessary to perform such operations in stages using intermediate equations when there is a requirement to maintain PAL8 compatibility.

2) Multiply is signified by using the ^ operator.

3) Divide is signified by using the % operator.

4) When using the ! operator in expressions, the results are obtained by performing an inclusive .OR. operation on the elements given. If the /Y command-line option switch is set, the operation performed is changed to left-shift six bits.

5) The last example is used when programming LINCtape operations on a classic LINC or LINC-8 or PDP-12 system that supports the corresponding hardware. *RDE* is a two-word instruction to read a LINCtape block; the second word is a compound word containing the memory segment (quarter of a memory field) for the transfer in the high order three bits and the block number in the low-order nine bits. The RDE symbol is already present in P?S/8 PAL if dual-mode assembly is enabled by use of either the /8 or /9 command-line option switches.

6) The \ operator performs a left-shift nine bits on the supplied value. This is generally used to place the memory segment bits into the second word of a two-word LINCtape read or write instruction.

   P?S/8 PAL only supports the \ operator if dual-mode assembly is enabled by use of either the /8 or /9 command-line option switches.

The reader is directed to various documents available on the Internet covering the fundamentals of PDP-8 programming and the PAL language as implemented on various systems (not all of which are compatible with DEC standards). P?S/8 PAL is generally a superset of all other compatible implementations (to within nitpicks).

Additional information on the P?S/8 PAL assembler is available on the Internet:

http://www.ibiblio.org/pub/academic/computer-science/history/pdp-8/PQS8-related%20Files/1989%20Help%20Files/ASMBLR.HELP

A short summary of all supported command line option switches is contained within this file along with other internal documentation of P?S/8 PAL taken from the source code file (*ASMBLR.PAL* aka *PQSASM.PAL*).

2.80 Issues with regard to using FLIP.EXE to convert line conventions

Proper usage of the online copy of *ASMBLR.HELP* (and many other documents) may require end-of-line conversion due to unix-based conventions used on various servers.

All text files (including the aforementioned ASMBLR.HELP) located within the archive referenced above can be converted to MS-DOS/Windows conventions using Rahul Dhesi's well-known *FLIP.EXE* program.

FLIP can convert files between MS-DOS and unix conventions in both conversion directions. Running the program without arguments from an MS-DOS window will output the program options needed for a successful conversion.

FLIP.EXE is widely available as freeware; a copy of this program is included in the *P?S Enhanced PDP-8 Simulator* (*PEPS*) package for Microsoft Windows in the *\PEPS\Folders\Utilities* directory.

Note: Due to limitations of FLIP.EXE as distributed, it may be necessary to use file names conforming to MS-DOS conventions; many files associated with the PEPS project use naming conventions compatible with the forthcoming P?S/8 SHELL directory structure.

The P?S/8 SHELL uses 12.4 file names with case retention while MS-DOS uses 8.3 file names in upper-case only. OS/8 uses 6.2 file names in upper-case only. Other than potential file date limitations (for files created prior to 01-Jan-1980), SHELL directory files can easily be stored in Windows directories. The files may be renamed to MS-DOS conventions in Windows for further processing with FLIP.EXE, then renamed back to the original file name as required.

Note: FLIP.EXE as distributed is not compatible with 64-bit Windows systems. One remedy is to use Windows *WordPad*[tm] to process the files (which must be accomplished one file at a time).

If the PEPS package is available, ASCII text files may be transferred to either the LPT: or PTP: stream files and then have all potential artifacts stripped out. While this can be a somewhat time-consuming process, this utility will also perform the same operations as FLIP.EXE.

Appendix A - P?S/8 PAL error messages

Code Meaning

BE Current page literal Buffer Exceeded. Only 63 current page literals are allowed per page.

BO Binary output Overflow. Insufficient binary output files passed to complete the assembly.

CO COnditional assembly syntax error or nested < or > count problem.

DT Duplicate Tag error. An attempt was made to modify an existing symbol value using ,.

ER User generated ERror. The program includes internal logical safeguards that were violated. See Note 1.

FI FIxmri error. An attempt was made to improperly define an MRI-type symbol or redefine a permanent symbol.

IC Illegal Character error. Possibly [or (usage without enabling literals.

IE Illegal Expression error. The latest statement is syntactically malformed.

II Illegal Indirect error. The designated pointer is not located on the current page or page zero.

IM Insufficient Memory to load the symbol table. This may be remedied by using the /C command-line option switch.

IP Illegal Pseudo-op (directive) redefinition attempt. All directives are permanent symbols and cannot be redefined.

IR Illegal Reference to an inaddressable memory location by a direct reference instruction (instead of a pointer address).

LG Unsuppressed Link Generated error with link generation enabled. The generated link will also be flagged.

NE Null Expression within a literal reference. The expression must contain at least one term.

NU NUmerical data error. This is usually an attempt to use 8 or 9 when the prevailing radix is octal.

PA PAuse message. A programmed value associated with this message is displayed. See Note 2.

PE Page Exceeded. The code on a current page overlaps the literal table. The literal table (or program) must be made smaller.

PH Conditional assembly PHase error. The assembly terminated with an open failing conditional assembly statement.

PO Pushdown list Overflow. The latest expression is too complex. Simplify the coding using a series of smaller steps.

ST Symbol Table exceeded. More memory is required (or use fewer modular assembly options). This might be remedied using the /C command-line option switch.

SY Miscellaneous SYntax error. The error doesn't fall into any other reported categories.

UF Undefined Field error. The argument to a FIELD directive is undefined.

UO Undefined Origin error. The argument to an origin setting using * is undefined.

US Undefined Symbol error. A symbol was referenced that was not defined during pass one of the assembly.

ZE Page Zero Exceeded. Page zero code overlaps the literal table. The program or literal table must be made smaller.

Appendix A Notes

1) Well-written programs include internal safeguards to prevent scenarios that might violate obscure nuanced details of the program, such as requiring a particular symbol to be maintained within a narrow address range (or at a particular location in memory). Analysis of the error message report may help a maintenance programmer determine what programming blunder was committed.

2) PAuse messages are not actually errors. Well-developed programs occasionally include reports during the assembly for a variety of purposes, including providing additional information about an error condition signaled by the ERROR directive. Extremely long source files may include use of the PAUSE directive to report when an assembly pass starts (or completes).

   Note: PAuse message output unrelated to errors should be avoided during pass two; this is necessary to avoid unexpected disruption of listing output. Techniques to determine which assembler pass is in effect are described elsewhere in this document; these can be utilized to prevent output past pass one.

End of Appendix A

ASMIFM{INUS} *expression*

The ASMIFM{INUS} directive is one of the conditional assembly features of the extended PAL language. See Note 1.

The specified expression is evaluated. If the result is negative, the next line is{E} assembled; if the result is positive, the next line is ignored.

```
ASMIFM  A-B             /COMPARE A TO B.
D=      3               /SET D TO 0003.
```

The next line may be blank or consist of comments only rendering the conditional moot. During the early development of a program, this may deliberately be done to force a specific outcome.

If additional statements are on the same line as the ASMIFM{INUS} directive (using ; to separate the statements) and the conditional test fails, the additional statements are also ignored.

ASMIFN{ONZERO} expression

The ASMIFN{ONZERO} directive is one of the conditional assembly features of the extended PAL language. See Note 1.

The specified expression is evaluated. If the result is non-zero, the next line is assembled; if the result is zero, the next line is ignored.

```
ASMIFN  A-B             /COMPARE A TO B.
D=      3               /SET D TO 0003.
```

The next line may be blank or consist of comments only rendering the conditional moot. During the early development of a program, this may deliberately be done to force a specific outcome.

If additional statements are on the same line as the ASMIFN{ONZERO} directive (using ; to separate the statements) and the conditional test fails, the additional statements are also ignored.

ASMIFZ{ERO} expression

ASMIFZ{ERO} directive is one of the conditional assembly features of the extended PAL language. See Note 1.

The specified expression is evaluated. If the result is zero, the next line is assembled; if the result is non-zero, the next line is ignored.

```
ASMIFZ  A-B             /COMPARE A TO B.
D=      3               /SET D TO 0003.
```

The next line may be blank or consist of comments only rendering the conditional moot. During the early development of a program, this may deliberately be done to force a specific outcome.

Note: If additional statements are on the same line as the ASMIFZ{ERO} directive (using ; to separate the statements) and the conditional test fails, the additional statements are also ignored.

ASMSKP expression

The ASMSKP directive is one of the conditional assembly features of the extended PAL language. See Note 1.

The specified expression is evaluated as an unsigned 12-bit integer value. The next *expression-value* lines of the source code are ignored.

```
ASMSKP  3               /SKIP THE NEXT THREE LINES.
HLT                     /THIS IS IGNORED!
JMP     .+3             /THIS IS ALSO IGNORED!
/                       IGNORED AND ALSO A COMMENT!
```

If the expression evaluates to zero, all lines following the ASMSKP directive will be assembled. If the expression evaluates to a non-zero value and additional statements are on the same line as the ASMSKP directive (using ; to separate the statements), the additional statements are also ignored.

The ASMSKP directive is needed because of the complex nature of conditional assembly using the available set of related directives (certain logical expressions might require the use of multiple conditionals). Unfortunately, since the ASMSKP directive does not work with any form of delimiter mechanism, the programmer must carefully count the lines required to be skipped; slight changes to the program source can cause unexpected results.

CONSOL{E}

The front panel switches are read during the assembly.

The CONSOL{E} directive returns the contents of the front panel switches to allow storage of the value into an assembly-time symbol:

```
SWITS=  CONSOLE          /STORE CONSOLE SWITCHES IN SWITS.
```

Conditional assembly techniques can then be used on the resultant symbolic value for any purpose as determined by the programmer (such as causing PAUSE directive output to appear during the assembly).

DATE

The DATE directive returns the low-order 12-bits of the current system date to allow storage of the value into an assembly-time symbol:

```
CURDAY= DATE             /STORE THE DATE IN CURDAY.
```

Conditional assembly can use the date for a variety of purposes including creating a current version of the program. As of P?S/8 Version 8Z, the format of the returned expression matches that used by current P?S/8 System programs (which requires periodic program maintenance, albeit infrequently).

Future releases of P?S/8 PAL will likely include the *DATEHI* directive to obtain the high-order six bits of the current system date. The two values can be combined to form the complete date (in the range of 01-Jan-1900 through 31-Dec-2411). The proposed format will require alternate calculations but will be free of the need for maintenance.

More details regarding the specifics of calculating the current system date can be found in the P?S/8 Keyboard Monitor Command Guide available as a separate document.

DECIMA{L}

The internal radix for mathematical expressions is set to decimal; the default radix is octal (which can be restored by subsequent use of the OCTAL directive).

DEVICE

The DEVICE directive takes a four-character string argument. It is used to form the device-name string used to reference P?S/8 non-system device handlers and all OS/8 device handlers as follows:

```
DEVICE  DTA1            /REFERENCE TO DTA1:
```

The DEVICE directive creates two 12-bit words containing the packed six-bit ASCII text values of the argument. In the example given, the output would be equivalent to 0424 (DT) followed by 0161 (A1).

The string argument is not processed in the same manner as in the case of TEXT and related directives; as such, the /J command-line option switch has no effect on DEVICE directive usage.

DTORG expression

The DTORG directive is intended to provide target storage device block number information when generating operating system components for Typeset-8 as follows:

```
DTORG   60              /WRITE THIS PAGE TO BLOCK 0060.
```

The expression is evaluated as a 12-bit unsigned integer to designate a block number on a device such as TC01/TC08 DECtape. Up to 128 words (assembled at the current location counter) follow as the intended data to be written to the target block. See Notes 1, 2 and 3.

The implementation of the DTORG directive is extremely problematic. See sections 2.61, 2.62 and 2.63 of this document for more information.

EJECT *optional-string-value*

The EJECT directive causes assembler listing output to advance to the next page.

If used with the optional-string-value, the title on the next page is set to the optional-string-value text (subject to truncation).

All statements containing the EJECT directive are not shown in the assembler listing output.

ENBITS

The ENBITS directive is used in PAL10 to enable the bitmap of the latest memory field when either a new FIELD directive is used or the end of the assembly. This feature is not implemented in other PAL assemblers; the ENBITS directive is provided for source-code compatibility purposes only. See Note 2.

ENDBIN

The ENDBIN directive closes the current binary file. Additional binary output (if any) will be stored in the next binary output file.

The ENDBIN directive is ignored if binary output generation is disabled.

The ENDBIN directive can be used to create additional (optional) program segments ancillary to a master binary file(s) to be initially loaded separately. ENDBIN is supported primarily for compatibility with the PAL III paper-tape system assembler.

ENPUNC{H}

Binary output generation may have been turned off using the NOPUNC{H} directive; the ENPUNC{H} directive will re-enable binary output generation.

The NOPUNC{H} directive is often used when changing the binary current location counter to a value different from the (unrelocated) current location counter used during the assembly. This is used to relocate a code section that initially loads into one memory area to another memory area (by unstated means); the initial loading area is the current location counter defined prior to the use of the NOPUNC{H} directive. The ENPUNC{H} directive is used to restore binary output once the intended target current location counter value is in effect.

The ENPUNC{H} directive is ignored if binary output is already enabled. No error message will occur; this allows ENPUNC{H} directive statements to be used to ensure binary output has been restored.

The ENPUNC{H} directive is ignored if binary output generation is disabled. See Note 4.

ERROR *optional-numeric-expression*

The ERROR directive causes a deliberate error (and error message). If automatic chaining to P?S/8 BIN is in effect (binary output files are being generated and the /G command-line option switch is set), the chaining operation will be prevented.

Well developed PDP-8 programs can contain safeguards to prevent undesirable scenarios such as having a definition fall out of a required narrow range of addresses (or in some cases a unique required value). This may help prevent future program maintainers from committing logical blunders.

The value of the optional-numeric-expression will be displayed as part of the error message; this may help pinpoint the programming problem such as in the example that follows:

```
        *7                      /APPROPRIATE STARTING LOCATION.

LOCTM1, .-.                     /TEMPORARY STORAGE.
LOCTM2, .-.                     /ANOTHER TEMPORARY (WRONG PLACE!)

        AUTOXR= .               /THIS IS WRONG! SHOULD BE 0010.

        MYPTR=  AUTOXR+7        /USED AS AN AUTO-INDEX REGISTER.

        IFNZRO  MYPTR&7770-10   <ERROR MYPTR>
```

The value of *MYPTR* is 0020 instead of staying within the required range of 0010 through 0017; as such, it is not capable of being used as an auto-index register (as the program logic demands). Hopefully, the error message `ER  0020  AT LOCTM2+0001 (0011)` (where 0011 is the current location counter at the point of error in this example), is sufficient to aid maintenance programmers in correcting the mistake. (The *LOCTM2* location is inappropriately placed in the auto-index register area ruining the program logic. By moving LOCTM2 to another area of page-zero memory, the problem is eliminated; program reassembly will not trigger the error-detection safeguard.)

Use of the ERROR directive in this manner can be further embellished if desired. See the description of the PAUSE directive elsewhere in this document for further details.

If the optional-numeric-expression is not used, the error message will be `ER  0000`.

EXPUNG{E}

The EXPUNG{E} directive is used to clear the entire symbol table at the start of an assembly.

All permanent symbols are removed (except directives). The program must define all symbols used (including MRI symbols using the FIXMRI directive).

By deleting the internal symbol table, it is possible that certain programs may be capable of assembly on system configurations with smaller memory size.

As an example of proper usage of the EXPUNG{E} directive, the source code of P?S/8 PAL performs an EXPUNG{E} at the beginning of the assembly; as such, P?S/8 PAL fully defines all symbols used within the program. This guarantees proper program generation by any qualifying assembler program (that properly supports all required features).

As of this writing, only recent implementations of P?S/8 PAL and OS/8 PAL8 support all of the features required for proper assembly of the P?S/8 PAL source code. It is hoped that savvy TOPS-10 programmers can modernize PAL10 to add the few missing features in the current implementation. (To avoid consequential problems, many features of P?S/8 PAL are not used in the P?S/8 PAL source code; where required, the PQS directive is used to produce proper results should the assembly be performed on an alternate assembler program.)

P?S/8 PAL supports many symbols not ordinarily found in other assemblers (such as IOT instructions for several PDP-8 peripherals and the full PDP-8/E *Extended Arithmetic Element* [*EAE*]).

Future releases of P?S/8 will include the optional SHELL overlay; this will include an assembler (partially) based on P?S/8 PAL to be known as PALX (or perhaps XPAL). Either P?S/8-based assembler program will be capable of assembling the source code of either program (although the SHELL-based assembler will support additional features beyond the scope of assembling the assembler).

FIELD optional-numeric-expression

The FIELD directive is used to set a designated memory field for binary loading of the statements that follow. If the optional-numeric-expression is present, it is evaluated to set the desired field; if there is no argument, the next memory field past the present value is selected.

When a FIELD directive is used, all open literal pools are output and reset to default values. If the /0 command-line option switch is set, an automatic *200 will also be generated overriding the previous value of the current location counter.

Memory fields up to a maximum of 256K are supported for compatibility with the KT8A option (which supports addressing memory up to 128K); however, all standard binary loading utilities can only load binary data generated in the 32K address space (FIELD 0 through FIELD 7).

Future releases of P?S/8 will include an extended binary loader that will require the KT8A hardware to operate; due to complexity, an extended binary loader will operate in a manner analogous to the current virtual Slurp loader.

FILENA{ME}

The FILENA{ME} directive takes a string argument in the form of an (up to) six-character file name and a (one or) two-character (optional) file extension. This is used to create a four word text string consisting of an OS/8 six-character file name and a two-character file extension as follows:

```
FILENAME FOOBAR.PA       /STATIC FILE NAME STRING.
```

The FILENA{ME} directive creates four 12-bit words consisting of the packed six-bit ASCII text characters derived from the string argument. In the example given, the output would consist of 0617 (FO), 1702 (OB), 0122 (AR) and 2001 (PA).

P?S/8 PAL supports the FILENA{ME} directive for OS/8 PAL8 compatibility. Future P?S/8 systems will support the P?S/8 SHELL with an extended assembler based on P?S/8 PAL to be designated PALX (or XPAL). PALX will support an option to extend the FILENA{ME} directive to support SHELL directory file names (which are more complex than OS/8 file names).

The string argument is not processed in the same manner as in the case of TEXT and related directives; as such, the /J command-line option switch has no effect on FILENA{ME} directive usage.

FIXMRI

The FIXMRI directive is used to augment a standard equate statement to process the symbol definition as a *Memory Reference Instruction* (*MRI*). MRI symbols modify the expression evaluation so that the PDP-8 addressing rules are applied instead of *inclusive or* (.*IOR*.) processing (which is the default method of expression evaluation) as follows:

```
FIXMRI  INC=   2000      /ISZ PRESUMED TO NEVER SKIP.
```

The FIXMRI directive is also used to define floating-point pseudo-instructions that generally follow the MRI addressing rules as part of several software floating-point packages.

When the EXPUNG{E} directive is used, FIXMRI is needed to redefine all required MRI symbols (such as *AND*, *TAD*, *ISZ*, *DCA*, *JMS* and *JMP*).

FIXTAB

The FIXTAB directive is used after all symbols meant to be considered as an extension of the permanent symbol table are (fully) defined.

All such initialization should occur at the start of the assembly to establish the permanent symbol table (perhaps after the use of the EXPUNG{E} directive).

All permanent symbols will not be included in the cross-reference output (if the /X command-line option switch is in effect) or the symbol table printout at the end of the assembly (if the /S command-line option switch is in effect) unless the /A command-line option switch is (also) set.

I

The I directive is required in all PDP-8 MRI statements using indirect references. This forces bit[3] to be set so the generated instruction value to conforms to the PDP-8 addressing rules. This is done after the evaluated expression is adjusted according to the MRI addressing rules in effect.

Indirect statements using the I directive are subject to errors caused by violation of the PDP-8 addressing rules. See Note 5.

IFDEF *symbol* <*statements*>

The IFDEF directive is one of the conditional assembly features of the PAL language.

The symbol table is searched for the specified symbol. If the symbol is present (and defined), the statements contained within the < and > are assembled. If the symbol is not present (or not defined) at this point in the assembly, the statements contained within the < and > are ignored.

Unlike ordinary symbolic references, attempts to access the specified symbol while undefined (using the IFDEF directive) will not cause Undefined Symbol errors.

The statements can span an indefinite number of lines of source code as necessary. Conditionals are occasionally used within other conditionals in a *nested* manner. The count of < and > must be carefully maintained to achieve an error-free assembly; the contents of comments are scanned for < and > when conditional assembly is in effect.

The IFDEF directive is often used to include other symbols (or data) into an assembly when relevant:

```
        IFDEF   TC01   <          /IF TC01/TC08 DECTAPE.

        NOPUNCH                   /NO BINARY OUTPUT FOR NOW.

        *7754                     /WHERE WC, CA ARE.

WC,     .-.                       /WORD COUNT.
CA,     .-.                       /CURRENT ADDRESS.

        ENPUNCH                   /BINARY OUTPUT ON AGAIN.

                       >          /END OF CONDITIONAL.
```

The controlling symbol *TC01* should be defined in an earlier section of the source file where all global symbols are generally grouped together as follows:

```
/       GLOBAL SYMBOLS HERE.

        TC01=   1                 /USING TC01/TC08 DECTAPE.
```

Since the controlling symbol DECTAP{E} is defined, the conditional section shown above will be assembled.

IFNDEF symbol <statements>

The IFNDEF directive is one of the conditional assembly features of the PAL language.

The symbol table is searched for the specified symbol. If the symbol is not present (or not defined), the statements contained within the < and > are assembled. If the symbol is present (and defined) at this point in the assembly, the statements contained within the < and > are ignored.

Unlike ordinary symbolic references, attempts to access the specified symbol while undefined (using the IFNDEF directive) will not cause Undefined Symbol errors.

The statements can span an indefinite number of lines of source code as necessary. Conditionals are occasionally used within other conditionals in a nested manner. The count of < and > must be carefully maintained to achieve an error-free assembly; the contents of comments are scanned for < and > when conditional assembly is in effect.

The IFNDEF directive can be used to initialize a default symbolic value as follows:

```
IFNDEF  NUDATA <

NUDATA= 1       >        /GET NEW DATA.
```

In the example above, the conditional statement is assembled during pass one of the assembly. During pass two the statement is ignored because the symbol is already defined.

By defining a symbol in this manner, conditional control of other statements located in other parts of the assembly may be created. The example coding shown in the IFDEF directive section above is typical of such statements.

When multiple possibilities exist, a more complex set of statements can chose a desired default as follows:

```
IFNDEF  DF32   <DF32= 0> /ASSUME DF32 NOT PRESENT.
IFNDEF  RF08   <RF08= 0> /ASSUME RF08 NOT PRESENT.
IFNDEF  TC01   <TC01= 0> /ASSUME TC01 NOT PRESENT.

IFZERO  DF32+RF08+TC01 <TC01= 1>   /USE TC01 BY DEFAULT.
```

A global definition near the beginning of the source file can be used to override the (safety) default as used above.

More information on the IFZERO directive is available elsewhere in this document.

IFNZRO expression <statements>

The IFNZRO directive is one of the conditional assembly features of the PAL language.

The specified expression is evaluated. If the result is non-zero, the statements contained within the < and > are assembled. If the result is zero, the statements contained within the < and > are ignored.

The statements can span an indefinite number of lines of source code as necessary. Conditionals are occasionally used within other conditionals in a nested manner. The count of < and > characters must be carefully maintained to achieve an error-free assembly; the contents of comments are scanned for < and > when conditional assembly is in effect.

The IFNZRO directive is often used to check an assembly parameter range as follows:

```
        *7                          /APPROPRIATE STARTING LOCATION.

LOCTM1, .-.                         /TEMPORARY STORAGE.
LOCTM2, .-.                         /ANOTHER TEMPORARY (WRONG PLACE!)

        AUTOXR= .                   /THIS IS WRONG! SHOULD BE 0010.

        MYPTR=  AUTOXR+7            /USED AS AN AUTO-INDEX REGISTER.

        IFNZRO  MYPTR&7770-10       <ERROR MYPTR>
```

The value of MYPTR is 0020 instead of staying within the required range of 0010 through 0017; as such, it is not capable of being used as an auto-index register (as the program logic demands). An error message will be issued (during pass two) when this code section is assembled:

```
ER  0020  AT LOCTM2+0001 (0011)
```

Situations such as the example above often come about when maintenance programmers do not completely understand the design created by the original author(s). Proper safeguards can be designed in to prevent program logic blunders.

IFZERO expression <statements>

The IFZERO directive is one of the conditional assembly features of the PAL language.

The specified expression is evaluated. If the result is zero, the statements contained within the < and > are assembled. If the result is non-zero, the statements contained within the < and > are ignored.

The statements can span an indefinite number of lines of source code as necessary. Conditionals are occasionally used within other conditionals in a nested manner. The count of < and > must be carefully maintained to achieve an error-free assembly; the contents of comments are scanned for < and > when conditional assembly is in effect.

The IFZERO directive is often used to create assembler default values:

```
IFZERO  OURFLD   <

BUFFER= 4000             /USE FIELD 0 VALUE.>

IFZERO  OURFLD-1 <

BUFFER= 6000             /USE FIELD 1 VALUE.>
```

In the above example, a parameter must be set to a value consistent with an overall assembly parameter that can have multiple values.

LMODE

The LMODE directive is provided in the dual mode assembly module to enable LINC mode addressing and certain symbols unique to LINC mode. See Note 1.

NOBITS

The NOBITS directive is used in PAL10 to disable the bitmap of the latest memory field when either a new FIELD directive is used or the end of the assembly. This feature is not implemented in any other PAL assembler; the NOBITS directive is provided for compatibility purposes only. See Note 2.

NOPUNC{H}

The NOPUNC{H} directive turns off binary output generation during the assembly without affecting the assembler's current location counter. Using the ENPUNC{H} directive later in the assembly will restore binary output.

The NOPUNC{H} directive is often used when changing the binary current location counter to a value different from the (unrelocated) current location counter used during the assembly. This is used to relocate a code section that initially loads into one memory area to another memory area (by unstated means); the initial loading area is the current location counter defined prior to the use of the NOPUNC{H} directive. The ENPUNC{H} directive is used to restore binary output once the intended target current location counter value is in effect.

The NOPUNC{H} directive is ignored if binary output generation is not enabled in the overall assembly. See Note 4.

OCTAL

The internal radix for mathematical expressions is set to octal. While the default radix is octal, an earlier section of source code may have changed the radix to decimal using the DECIMA{L} directive. Any source code section where the current radix is uncertain should include a statement using the OCTAL directive to restore the default radix.

PAGE optional-numeric-expression

The PAGE directive is used to set an origin setting at the lowest location on a PDP-8 memory page for binary loading of statements that follow. If the optional-numeric-expression is present, it is evaluated to set the desired page; if there is no argument, the next memory page is selected.

Selecting a new page will cause the output of the literal table of the previous page in effect (if any).

The default origin of any assembly is 0200; this is equivalent to using the *PAGE 1* statement or a statement of the form:

```
*200                    /START AT 0200.
```

Changing the current page will force a dump of current page literals (assuming literal generation is in effect).

PAUSE optional-numeric-expression

The PAUSE directive has two different usages:

When used without the optional-numeric-expression, the PAUSE directive indicates the rest of the latest input file will be ignored. This usage of the PAUSE directive is provided for compatibility with the paper-tape operating system PAL III assembler.

When used with the optional-numeric-expression, a PAUSE message is displayed during each pass of the assembly. This is a user-defined function and is not an error message per se (although the programmer can define values to specific messages to describe some internal situation related to an error in an internal coding section).

When developing extremely long programs, programmers may use the PAUSE directive to signal the current program section and/or the start (or end) of each assembly pass. For more information see Section 2.03 of this document.

PMODE

The PMODE directive is provided in the dual mode assembly module to enable PDP-8 mode addressing and certain symbols unique to PDP-8 mode. See Note 1.

When P?S/8 PAL is operating in PDP-8 mode, there is generally no difference between dual-mode assembly and the normal single-mode assembly.

For compatibility with programs developed using the embedded assembler utility of the *LAP6-DIAL*/*DIAL-MS* operating system, certain additional directives are included when dual-mode assembly is enabled in P?S/8 PAL.

This includes an alternate set of conditional assembly directives that are not available in any other PAL assembler.

PQS

The PQS directive is a feature unique P?S/8 PAL. It can be used in conditional assembly statements to allow special-purpose programming only supported by P?S/8 PAL; alternate statements can attempt to provide compatibility with other assemblers. Unfortunately, certain powerful features of P?S/8 PAL are not available in other assemblers; crude alternatives have been used to provide a functional (albeit inferior) result where possible.

Features of P?S/8 PAL such as conditional literals have been instrumental in implementing several very large bootable application programs requiring both storage devices and 32K memory; conditional assembly has been used to provide inferior support of statically-determined page zero literals through the use of manual optimization at some point in the development. However, as additional development is performed, the static manual optimization tends to drift while assembly with P?S/8 is always current.

Only additional manual optimization can sync up the relevant (newer) changes. As such, the PQS directive has been extensively used to allow PAL8 to replicate the latest static optimization until further development with P?S/8 PAL can be performed.

The forthcoming P?S/8 SHELL will include a more advanced assembler (partially) based on P?S/8 PAL tentatively to be known as *PALX* (or perhaps *XPAL*). P?S/8 SHELL PALX will also support the PQS directive; however, since XPAL will be a more comprehensive assembler, a companion *PQSX* directive will also be implemented. This will allow programmers to take advantage of the advanced features of both P?S/8 assemblers.

RELOC optional-numeric-expression

The RELOC directive performs much of the functionality of the NOPUNC{H} and ENPUNC{H} directives in a more compact and elegant form. The optional-numeric-expression is evaluated to form a relocation factor that sets up an offset between the assembler's current location counter and the binary output origin.

Note: Since the relocation factor is applied to all binary output, the RELOC directive can be used in conjunction with literals.

Additional usage of the RELOC directive with the optional-numeric-expression causes cumulative relocation, which can be useful in certain obscure situations. The RELOC directive used without the optional-numeric-expression resets the relocation factor to zero.

Note: While PAL8 is the only other assembler that supports the RELOC directive, there are certain differences:

a) PAL8 flags statements detected as *apparently* relocated with *. P?S/8 PAL flags statements in a similar manner, but only if relocation is actually in effect.

If a statement of the form

RELOC .

is used in the source program, relocation is reset to zero; however, PAL8 will only cancel the statement relocation flag when the RELOC directive is used without the *optional-numeric-expression*.

b) PAL8 does not flag statements that are relocated by the use of the NOPUNC{H} and ENPUNC{H} directives. P?S/8 PAL outputs the * on all statements actually relocated regardless of method of relocation.

Note: Certain advanced programming methods include the simultaneous usage of the NOPUNC{H} and ENPUNC{H} directives and the RELOC directive; this is necessary when there are several independent aspects of relocation in effect simultaneously.

SEGMNT optional-numeric-expression

The SEGMNT directive is used to set the field and origin setting to the quarter memory field segment specified by the *optional-numeric-expression* in the range of 00 (field 0, location 0000) through 37 (field 7,location 6000) as required by the LINC architecture of the LINC-8 and PDP-12.

If the *optional-numeric-expression* is present, it is evaluated to set the desired segment; if there is no argument, the next memory segment is selected.

Note: Segment 2 is the default value for LINC mode assembly followed by an origin of 0020 (04020). This is equivalent to SEGMNT 2 followed by *20 (in LINC mode, addresses are specified within a segment). See Note 1.

SIXBIT *text-string*

The SIXBIT directive creates 12-bit words packed with six-bit characters taken from the *text-string* (other than delimiter characters). The encoding is six-bit ASCII text as used with similar directives in assemblers for DEC 18-bit and 36-bit systems (which is not the PDP-8 standard).

Note: If the number of characters in the *text-string* is even, an additional 12-bit word of 0000 is created as a delimiter unless the /J command-line option switch is set. For more information see Section 2.41 of this document.

SKIP optional-numeric-expression

The SKIP directive is used to create deliberately blank lines in the listing output of an assembly. The *optional-numeric-expression* is evaluated as an unsigned 12-bit integer. Blank lines are created as many times as necessary unless the value is zero, in which case no blank lines are created. If the optional-numeric-expression is not present, one blank line will be created.

Note: All lines containing the SKIP directive will not be displayed in the listing.

TEXT *text-string*

The TEXT directive creates 12-bit words packed with six-bit characters taken from the *text-string* (other than delimiter characters). The encoding is six-bit ASCII text as typically used on the PDP-8 (which is not compatible with similar directives in assemblers for DEC 18-bit and 36-bit systems).

Note: If the number of characters in the *text-string* is even, an additional 12-bit word of 0000 is created as a delimiter unless the /J command-line option switch is set. For more information see Section 2.41 of this document.

TITLE *text-string*

The TITLE directive is used to set the title field of the page header used on every listing page if the /L and /N command-line option switches are set. The *text-string* will be used (subject to truncation) starting with the next printed page.

Normally, the title field is taken from the first line of the first input file; the TITLE directive can only affect subsequent output. However, the TITLE directive can be used to override the first page title field by using the XLIST directive beforehand:

```
/    EXAMPLE WHERE THIS IS NOT USED IN THE TITLE FIELD.

XLIST                    /TURN OFF LISTING NOW.

TITLE   MY TITLE         /USE DESIRED TITLE INSTEAD.

XLIST                    /TURN LISTING ON NOW.
```

By use of the TITLE directive, the title field on the listing page headers do not have to start with a "/" character (which all comments must have). All default titles come from the first line of the file; a typical statement would generally be a poor choice for a title field.

XLIST optional-numeric-expression

The XLIST directive is used to hide portions of a listing file (assuming the /L command-line option switch is set). If the optional-numeric-expression evaluates to a non-zero value, the listing is turned off. If it evaluates to zero, the listing output is restored. If no argument is given, the listing output state is reversed from its former state.

Note: All lines containing the XLIST directive are hidden.

Z

The Z directive is provided for compatibility with the PAL II assembler of the paper-tape operating system. All PDP-8 assemblers starting with PAL III are capable of properly setting arguments (or operands) to MRI class instructions according to the rules of the PDP-8 architecture. PAL II only implements a subset where all symbols are erroneously placed on the current page. When necessary, the Z directive is used on all statements that actually address Page Zero to remove the current page bit from the generated value.

Note: PAL III was originally written in PAL II. Later releases of the source code were modified to require a binary copy of PAL III to assemble its own source code.

It is known that PAL8 ignores the Z directive while P?S/8 PAL implements the original intention.

ZBLOCK *numeric-expression*

The ZBLOCK directive is used to create blocks of data set to 0000. The *numeric-expression* is evaluated as an unsigned integer; the value determines how many words are generated.

Note: If the *numeric-expression* evaluates to zero, no words are generated.

1) Certain directives are only available if LINC mode dual assembly is enabled using either the /8 or /9 command-line option switches. They are provided primarily for compatibility with the DIAL-MS dual mode assembler. Due to the nature of LINC mode expressions that use one's complement arithmetic, the programmer must be aware of the attendant quirks (such as positive and negative zero values).

2) Despite not being particularly associated with dual mode assembly, certain directives are provided only if either the /8 or /9 command-line option switches are set. These directives will be moved to the general symbol table load in a future release to avoid this erroneous dependency; for the present, dual mode assembly must be enabled. Additional directives not currently supported may be added to a future release of P?S/8 PAL.

3) The DTORG directive should not be used at this time as its presence in the source code causes the binary output to be corrupted. See Sections 2.61, 2.62 and 2.63 of this document for more information.

4) The NOPUNC{H} and ENPUNC{H} directives are often used to relocate binary output as part of an overlay structure. The assembler current location counter can be updated while the binary output origin remains fixed. The program segment loads into an address space that can be moved or stored elsewhere while properly assembled for its ultimate execution address space:

```
        *6000               /WHERE THE GENERATED CODE LOADS.

        NOPUNCH             /FOOL THE ASSEMBLER.

        *7600               /WHERE THE CODE EXECUTES.

        ENPUNCH             /UN-FOOL THE ASSEMBLER.
```

   Note: The NOPUNC{H} and ENPUNC{H} directives are traditional features of several PDP-8 assemblers including PAL10; literals cannot be used with this form of relocation mechanism.

   The RELOC directive can accomplish a similar form of code relocation including literals if necessary. However, it is only supported by P?S/8 PAL and PAL8.

5) The I directive changes meaning if LINC mode assembly is in effect. Since there are no indirect references in LINC programming, the I directive is redefined to force bit[7] on. In the context of LINC addressing, the I directive means either immediate mode (when used with instructions where the operand immediately follows the instruction) or auto-indexed mode (when used with references to locations 0001-0017).

Many LINC instructions take immediate arguments:

```
     LMODE                /LINC MODE ASSEMBLY.

     LDA I;    3000       /LOAD THE AC WITH 3000.
     ADA  ;    FOO        /ADD THE CONTENTS OF FOO
```

The first instruction above uses the I bit to indicate the argument (3000) is immediately after the instruction. The second instruction does not use the I bit; as such, the address of the operand follows (which ironically means it is an indirect reference to the operand).

The PDP-8 references to locations 0010-0017 are auto-incremented when referenced indirectly. In LINC mode, references to 0001-0017 are ordinary indirect references unless the I directive is used:

```
     LMODE                /LINC MODE ASSEMBLY.

     ADA    17            /ADD WHAT IS POINTED TO.
     ADA I  17            /ADD NEXT VALUE AS WELL.
```

The first instruction uses the present contents of 0017 to point to the operand. The second instruction first increments the contents of 0017 to point to the (next) operand.

Certain LINC instructions have special meaning for references to location 0000; these instructions have specific addressing needs which do not involve the I bit. The auto-increment references to locations 0001-0017 clearly are the inspiration for the PDP-8 auto-increment usage of locations 0010-0017. The range was shortened to allow the PDP-8 interrupt handling vector at the lower addresses. (The LINC was designed with a primitive interrupt structure; the LINC-8 does not support LINC interrupts.)

The PDP-12 defines LINC mode interrupts analogous to the PDP-8 interrupt system (directed to an alternate vector location).

The PDP-8 cannot use indirect references to 0010-0017 without auto-increment; to this limited extent, the LINC instruction set is superior to the PDP-8 instruction set. LINC mode addressing does not allow indirect usage of any other locations (other than pointer locations immediately after the instructions). There is no concept of a page zero in the LINC architecture, and there are only three instructions that address all of memory directly. However, all of memory in this case is limited to 1024 locations, not the PDP-8 norm of 4096 locations.

There is no concept of a subroutine call mechanism in the LINC architecture; the *JMP* instruction attempts to act as a stand-in. However, if additional JMP instructions are used within the subroutine, various fixups (read *kludges*) are required to protect the return address.

All features considered, system architecture experts are welcome to weigh in with their own opinions as to the relative worth of PDP-8 versus LINC architecture; their views may greatly differ from the author's viewpoint.

Few operating systems for the LINC architecture support PDP-8 programming; all known implementations are quite inferior to the various PDP-8 assemblers mentioned elsewhere in this document. To the author's knowledge, none of these systems support mixed dual mode assembly to any extent; rather, PDP-8 programming and binary loading support is considered an independent concept. This is especially disappointing for use on the PDP-12 where mixed dual mode assembly is quite common due to the nature of the architecture.

End of Appendix B

Appendix C - P?S/8 PAL conditional literals and related topics

P?S/8 PAL supports an extension beyond the literal features common to other assemblers as described earlier in this document, which is known as the *conditional* literal.

Note: An alternative term for the conditional literal is the *dependent* literal since the actual code generated is dependent upon specific conditions in effect throughout the assembly.

While the use of conditional literals is an important and powerful feature of practical PDP-8 assembly language programming, conditional literal support is unique to P?S/8 PAL.

Note: Future directions of P?S/8 support include the P?S/8 SHELL overlay which will include a new assembler program partially based on P?S/8 PAL; conditional literal support will be consistent with P?S/8 PAL.

Alternate syntax considerations of conditional literals.

To aid in transitioning from other assemblers (such as Macro-11) to PDP-8 programming, the conditional literal uses syntax similar to that of certain other assemblers:

```
     TAD       #10             /ADD 0010 FROM SOMEWHERE.
```

While addressing rules for the PDP-8 differ from other computer architectures, literals expressed this way are a familiar concept to many programmers accustomed to the software environment of these other systems; the use of conditional literals in P?S/8 PAL will help programmers make the transition to the PDP-8 assembly language environment easier. See Note 1.

Conditional literal syntax and addressing rules.

Conditional literals take the form of the following:

```
     TAD       #3              /ADD 0003 FROM CHOSEN PAGE.
```

In this example, the assembler will make a dynamic decision as to which type of literal to use as follows:

a) If the program already contains a matching Page Zero literal due to other program statements using the identical value, this statement will use that existing Page Zero literal address.

b) Assuming a) fails, If the program already contains a matching current page literal value due to other program statements, the statement will use that current page literal address.

c) Assuming both a) and b) fail, the assembler will create a new current page literal; the new literal address is used.

Note: It is conceivable current page literal statements on the same page could benefit by using the same literal address should the current page literal statement occur later in the assembly.

Unlike Page Zero and current page literals, there is no matching optional trailing character. As such, the seldom-needed feature of nested expression elements is not available when using conditional literals. See Note 1.

In certain cases, it may be necessary to calculate factors of an overall equation to be used with some form of literal statement. Each such factor would be carried out by an equate statement leading to the final expression used in the literal statement. In the case of conditional literals this is unavoidable due to lack of a closing expression character. See Notes 2 and 3.

Source libraries and conditional literals.

Experienced programmers tend to create a series of commonly used routines as starting points for a new programming project. Unlike other architectures (some constrained by ROM-based basic routines), the PDP-8 imposes no particular structure, other than a general convention that the last page of field 0 (07600-07777) is reserved for whatever mechanism is used to load the program.

Note: Depending on the particular operating system and certain specific configuration considerations, there may be additional limitations imposed; however, the emphasis is always on minimal intrusion regarding memory reservation. P?S/8 contains specific information within the kernel memory in 07600-07777 to allow a conforming program to determine what (if any) additional limitations are in effect; other operating systems may implement additional constraints on memory usage.

P?S/8 adds basic read/write routines to designated storage device(s) that are available by the nature of P?S/8 internal design. By proper adherence to system guidelines, programs can be made compatible with all supported machine models including the *DECmate* series (which is substantially incompatible with all previous 12-bit DEC computers).

As stated above, by conforming to P?S/8 system guidelines, any program can run under P?S/8 on any supported model; however, there is no hard and fast rule that this is required. Any program can be written for any specific supported model(s) constrained only by the designated hardware (and the minimal memory reservation rules of P?S/8, which must be observed to allow P?S/8 to load the executable form of the program into memory).

For practical program execution, foundational programming is needed to support many of the specific structures mentioned above; many elements of this support will be needed nearly all the time while others might be needed less often. In the general case, it is advantageous to develop working modules that can be organized into source libraries for each class of functions desired. See Note 4.

Well-formed source library routines for use in the P?S/8 environment should exclusively use conditional literals wherever possible; existing Page Zero literals defined in the main program will be used to determine the specific binary code generated. As such, the code size of the library routines will tend to be shorter without expending the effort to determine this manually (as would be the case when using other assemblers lacking this feature unique to P?S/8 PAL). See Note 5.

Problematic interaction with incompatible assemblers regarding conditional literals.

Attempts to maintain compatibility with other assemblers can be frustrating when a programming project requires the conditional literal feature to avoid excessive development time. Using P?S/8 PAL (and conditional literals), a nearly finished project can produce a contemporaneous assembly listing which can then be studied to determine instances that were successfully assembled using existing Page Zero literals. This is a good starting point to create compatible source code. However, any further changes to the source code can easily invalidate the inter-symbol relationships that led to the optimization that existed only before making further source code modifications.

Any work performed at this point leads only to a tenuous form of compatibility; changes in the coding may not translate into an efficient program, just merely a program that had been efficient at the time the necessary compatibility statements were written. The result is hardly a good outcome, but is the only way to achieve some measure of compatibility at the expense of efficient design, achieved with minimal effort to this point, then modified with additional effort that only partially supports the tentative compatibility once further changes are made.

Implementation of compatible coding.

Conditional assembly techniques can be used to determine which assembler is processing the source code and choose alternate source code statements where necessary. While this tedious process produces the same binary at the time the effort is spent determining which statements to code both for P?S/8 PAL and also other assemblers (such as PAL8), the alternate statements cannot dynamically adjust to the changing requirements of an ongoing project when future modifications are made.

This is not by any means a completely viable alternative and should only be implemented very close to the end of any project. Those responsible for program maintenance need to be aware of potential consequences of future coding changes.

Note: This is (in part) why certain programmers familiar with P?S/8 PAL abandon PAL8 for some portion of their work.

The following example illustrates how to write code that assembles in a custom manner with P?S/8 PAL:

```
     IFNDEF    PQS  <PQS= 0>  /THIS IS IGNORED BY P?S/8 PAL.

     IFZERO    PQS  <         /ASSEMBLE FOR THE OTHERS.>

     IFNZRO    PQS  <         /ASSEMBLE FOR P?S/8 PAL ONLY.>
```

Various techniques can be implemented for each instance of a known successful conditional literal; the goal is to produce binary code that is identical regardless of which assembler is used (unless and until the overall source code optimization is updated). The following serves as a minimal guideline for constructing compatible dual statement sections:

```
     IFNZRO    PQS  <         /ASSEMBLE FOR P?S/8 PAL ONLY.

     AND       #77            /JUST SIX BITS.

               >

     IFZERO    PQS  <         /ASSEMBLE FOR THE OTHERS.

     AND       [77]           /JUST SIX BITS.

               >
```

All statements similar to the above, where the conditional literal successfully used a Page Zero literal value, must be coded as shown in the example.

Additional techniques, such as surrounding the conditionalized code with judiciously chosen XLIST directive statements can achieve what appears to be unaffected code if desired:

```
     XLIST     OFF            /GENERALLY DON'T WANT TO SEE THIS.

     IFNDEF    OFF  <OFF= 1>  /THIS TURNS LISTING OFF.

     IFNDEF    ON   <ON= 0>   /THIS TURNS LISTING ON.

     XLIST     ON             /WHAT FOLLOWS WILL BE SEEN.
```

These parameters are used to debug the conditional sections. If unexpected output occurs, the definition of *OFF* can be temporarily changed to reveal all conditional sections (at the expense of paging of the code in a listing file) to aid in determining what was miscoded.

Note: The definitions of the controlling variables (OFF and ON*)* are unimportant during pass 1 of the assembly as long as they are eventually defined before the end of the source code.

All dual-coded sections should start with XLIST OFF. Each section starts with an XLIST ON and ends with XLIST OFF. Following the entire conditional section, XLIST ON should be used.

Note: All statement lines with an instance of XLIST are not listed. However, P?S/8 PAL (and PAL10) assembly listings will reveal gaps in the statement numbers. Listings created with PAL8 and CREF have incremental line numbers since there is no ability to detect the consequences of XLIST statements.

The example below implements all of the compatibility features discussed above:

```
     XLIST     OFF             /START OF CONDITIONAL SECTION.

     IFNZRO    PQS  <

     XLIST     ON
     AND       #77             /JUST SIX BITS.
     XLIST     OFF  >

     IFZERO    PQS  <

     XLIST     ON
     AND       [77]            /JUST SIX BITS.
     XLIST     OFF  >

     XLIST     ON              /END OF THIS CONDITIONAL SECTION.
```

Note: Every aspect of this technique must be repeated for each instance of a conditional literal statement successfully using a Page Zero literal value. When assembled by PAL8/CREF the code appears to be innocuous; however, the actual program logic and effort to implement the code is completely hidden.

More advanced techniques.

If the programmer chooses to spend additional effort regarding these issues, statements can be formed to trace conditional literal statements that succeed in producing Page Zero usage.

The following statement is to be used only once early in the source code before any conditional literal considerations:

```
ZCNT=     0              /INITIALIZE SUCCESS COUNT.
```

Within each conditional section being traced, additional statements are added just after the code statement using the conditional literal as follows:

```
IFNZRO    TRC       <    /ASSEMBLED IF THE TRACE IS ENABLED.
ZUSED=    #77            /WILL BE ADDRESS THAT WAS USED.
IFZERO    ZUSED&200 <
ZCNT=     ZCNT+1         /COUNT THIS AS A SUCCESS.
PAUSE     ZCNT           /OUTPUT CURRENT COUNT.
PAUSE     .              /SHOW WHERE WE ARE
                    >
                    >    /END OF TRACE FOR THIS SECTION.
```

This additional coding is intended for all conditional literal statements without regard to the likelihood of Page Zero literal value outcome. The tracing statements are designed to reveal the statements that succeeded; this allows monitoring the effectiveness of conditional literal usage.

The PAUSE directive used with an evaluated argument is a feature exclusive to P?S/8 PAL. Each usage of the PAUSE directive outputs a message to the prevailing listing output device; the printout includes the four digit octal evaluation of the arithmetic expression following the PAUSE directive as well as the current address within the assembly. As used in the example, the count of successful instances appears to *grow* with each new successful case; the address within the assembly will aid in adding the compatibility code changes required as shown above.

Note: PAUSE directive output is not an error message.

Real-world application example.

A large programming project that was materially affected by the use of conditional literals is shown below. Statements were added for backward compatibility (with PAL8) as described above:

A large PDP-8/E system was created for a corporate setting to run several online systems (and an offline backup system that could share data).

This project is one of the most ambitious programs ever written for the 32K PDP-8 environment. It was written as a standalone bootable operating system with external system generation or creation routines; the storage consisted of a pair of RK05 drives on an RK8E with optional third-party serial-line multiplexors. This system is capable of running as many as 40 tasks in a cooperative multitasking environment.

Due to the sheer complexity (which includes overlays of certain maintenance and data recovery tools) this was perhaps the best known example of productivity improvement achieved by the use of conditional literals. Had the feature not been available, development time would have profoundly increased.

However, an unfortunate postscript to this otherwise successful example includes the following:

The small company division that contracted for this system was also occasionally managed by individuals from the parent company. These individuals were not only totally unfamiliar with the local division operation, but also totally ignorant of all things PDP-8. As a direct result of their so-called "executive" actions, the company paid for many additional weeks of needless make-work to implement PAL8 compatibility as described earlier.

They were warned at the time that future modifications would likely result in increased development costs and time. This is exactly what happened during the course of several years as the requirements gradually changed. All of the necessary conditional sections were consistently implemented as described above; the cost predictably increased due to the additional effort to reconcile the code as the project gradually changed direction.

Since the entirety of this large application is completely standalone by nature, such decisions were quite foolhardy.

It is interesting to note the parent company was eventually forced to sell the local division to another large company, and in fact, eventually was forced into bankruptcy. The new owners had notions of their own which did not include the use of any DEC products; they decided to dismantle the equipment and then specified a replacement system that would cost several million dollars. However, since they also seriously devalued the entire local company operation, they were forced to sell it to an even larger company that eventually replaced everything with specific overpriced PC compatible systems that could have easily been produced for far less overall cost.

As of the writing of this document, the corporate world perceives this operation as requiring tens of millions of dollars worth of hardware despite company growth of less than 20% in the over twenty years of operation the author of this document is familiar with; this era ended with the head of the local operation seeking early retirement due to all of the incompetent interference over the years.

Alternative methods to achieve some measure of compatibility.

The use of conditional literals is a useful technique for many PDP-8 programming projects. If there is a need to produce a result that can be used in OS/8, there are appropriate techniques available to allow P?S/8 PAL to be used for program development.

a) P?S/8 generally runs on any configuration that OS/8 can support (including configurations OS/8 cannot run on due to its design limitations regarding system device handler design).

b) P?S/8 *OS8CON* is a bidirectional text file conversion program designed to facilitate moving PAL (and other language) source files between the two systems (in either direction of conversion). While each configuration is different, it is generally possible to create a compatible transfer mechanism on most (if not all) of the devices supported by both systems.

   With minimal effort, it is possible to maintain files in OS/8 and also transfer copies of the files to P?S/8 to allow processing with P?S/8 PAL.

c) Depending on project complexity, there are currently several provisional methods to transfer assembled binary code between P?S/8 and OS/8. This involves taking advantage of fortuitous coincidences between the two systems with regard to field 0 usage coupled with the fact that P?S/8 can boot to OS/8 while preserving the contents of extended memory across the reboot process.

d) For reasons unrelated to OS/8 binary compatibility as discussed within this document, a near-term future component of P?S/8 is being developed known as *BINCON*. BINCON will be capable of converting OS/8 absolute binary files to/from P?S/8 Slurp format binary files. This will allow P?S/8 source development to not only continue to be written conforming to the subset of PAL8 compatibility (where feasible), but will also allow a misimplemented feature of PAL8 (*DTORG* which was originally implemented for the benefit of the Typeset-8 group, which corrupts the binary output for all other OS/8-related purposes) to become useful for P?S/8 development in the near future.

Certain P?S/8 source programs are intended to be assembled and loaded in the OS/8 environment; this produces a stand-alone system generation technique for a portion of current P?S/8 system development. BINCON will allow changes to this process while also maintaining compatibility with the current method. P?S/8 PAL will produce proper P?S/8 binary files which can then be converted to OS/8 format. Programmers can use analogous techniques to take advantage of conditional literals (and perhaps other features such as dual-mode assembly), yet maintain source files (and converted binary files) in OS/8.

e) P?S/8 and OS/8 can be run from the P?S Enhanced PDP-8 Simulator (PEPS) package for Windows (which is available from the author of this document). Due to the way the package is designed, it is possible to punch binary output in DEC BIN format within one system and then read it back into the other system, etc.

   As such, P?S/8 PAL can create an *assemble-and-punch* operation using a single command with the appropriate command-line option switches to produce binary output to the high-speed punch. Exiting the simulator back to the Windows command level, the high-speed punch data file can be reassigned as the high-speed reader data. The simulator can then be run booting OS/8. OS/8 PIP can be used to create an image binary copy of the high-speed reader input using the PTR: handler.

Appendix C Notes

1) The syntax of the conditional literal was proposed as introduced by the \ character. As of P?S/8 PAL Version 8P, the # character is used. A cosmetic advantage of the original proposal was to allow pseudo-symmetrical statements to be formed:

```
TAD     \3/         /ADD 0003 FROM DYNAMICALLY CHOSEN PAGE.
```

   No actual symmetry exists; in the example above, the line ends with a comment starting just after the end of the conditional literal with the first / character; to the casual eye, the comment starts with the second / character (in the usual place further to the right).

   This syntax was abandoned when dual-mode assembly was added to P?S/8 PAL. LINC mode code requires the \ character be used for left-shift 9 operations associated with LINCtape programming.

2) PAL8 has several bugs in its arithmetic expression handling that render it incompatible with P?S/8 PAL and PAL10, which largely implement most expression features compatibly; however, the weaknesses of PAL8 are avoidable for most (but not all) PDP-8 programming projects.

   Note: Areas of weakness and incompatibility are the focus of this section; the goal is to guide all users by providing useful workarounds that can be deployed in all relevant operating systems.

   In general, P?S/8 program development features maintain compatibility with the PAL8 implementation as released with *OS/278 Version 2*. Certain bugs exist in earlier versions of PAL8 which were never fixed; however, these are generally esoteric in nature and for the most part easily avoided. In a few specific instances, the exact syntax in statements within P?S/8 source code files was slightly compromised to maintain compatibility with PAL8; however, this is admittedly a relatively minor issue. As described above, both PAL10 and PAL8 lack conditional literal support, which is a key conceptual feature that is crucial to realistic large-scale program development.

   A large PDP-8 project was written attempting to overcome this serious issue using conditional assembly techniques to manage the incompatibility (with no ability to fully overcome the problem). During development, many assembly listings were analyzed to edit conditional assembly statements to best serve the project. Since the project managers demanded that PAL8 be used, this was a less than satisfying outcome wasting much time and effort.

   Note: In hindsight, the programmers should have fought management more aggressively to prove their decisions were detrimental to the company's bottom line.

3) The use of parenthetical expressions in the context of literals is an intended design goal. Unfortunately, PAL8 parenthetical expression support is poorly implemented and has many unexpected limitations and odd quirks. It is generally recommended that such expressions be avoided unless the only assemblers used are PAL10 or P?S/8 PAL; both of these assemblers properly implement Page Zero and current page literals regardless of complexity.

   By design and where applicable, conditional literals must be calculated using separate equate statements; however, the use of parenthetical expressions is an obscure feature of the PAL language which is seldom employed. Unfortunately, this led to failure to test poor implementations such as PAL8. With regard to arithmetic expressions in general, PAL8 has several blunders besides literal expressions, such as a flawed implementation of unary expressions.

   Note: When few programmers use obscure features, bugs tend to go unnoticed and are less likely to be fixed.

4) Due to the versatility of the PDP-8 instruction set (especially the *JMS* instruction) there are often multiple ways to create callable subroutines that, while similar in function, are not identical in form or usage. At the whim of the programmer, preferred variations can be placed into a source library, each with different tradeoffs; as a project is developed, choices are made as to which routines are best to use.

   For example, a subroutine may be defined for the purpose of passing a text string meant to be displayed on the system console. Every aspect of the mechanism of the subroutine is subject to change. A subroutine entry point may be located near the end of Page Zero intended to be addressed directly by each caller using *JMS 0177* or similar. Alternatively, a pointer may be placed on Page Zero to allow callers scattered throughout memory equal access to the subroutine; multiple callers on several memory pages would justify the use of a Page Zero pointer. By using conditional literals, a proper dynamic decision can be made with regard to appropriate generation of subroutine linkage pointers.

   An implementation detail might specify the use of information passed in the accumulator, such as an upper limit on message print length (in case the message could exceed some restricted amount). Alternately, the accumulator could contain flag bits indicating what action to take afterwards (such as displaying <CR> and <LF>). Other possibilities could include the disposition of specially crafted text message strings capable of mixed upper-case and lower-case display.

The calling sequence could dictate the string exists inline past the subroutine call; the subroutine should return to just past the word containing the string terminator character.

Alternatively, the address of the string could be passed inline and the subroutine returns to the location following the inline pointer. If the scope of the subroutine includes extended memory support, additional bits may be passed in a variety of ways to form a full 15-bit address of where the string is located within the potential 32K memory space.

Not every variation is needed in every situation and all decisions are subject to preference. As such, source libraries are not only highly specific to a programmer's needs, there are many ways to carry out common functions and a sophisticated programmer may expend effort developing elegant mechanisms to exploit in future projects.

Note: Some features may inherently be somewhat inefficient of memory utilization in exchange for functionality; experienced programmers know what is the most suitable for any given situation. With experience, additional projects may mandate additional methods to expand programming techniques.

5) Note: Assuming several uses of a common value within library sections, as long as one usage anywhere in the program is "hardwired" as a Page Zero literal, all related instances will also effectively become Page Zero literals. In some cases, it may be desirable to include comments about likely combinations within the relevant library routines.

For example, two independent library routines may both use a common literal value; if only one routine is referenced in the main program, the use of actual Page Zero literal references would depend on the usage of a hardwired Page Zero literal in the main code. If both sections are included in the program, changing any usage to a hardwired Page Zero literal would ensure all other instances would become Page Zero references without regard for the main program code. Conditional assembly techniques can be applied to elegantly manage situations of this nature.

Assume subroutine #1 contains the first (potentially Page Zero) conditional literal while subroutine #2 contains a second occurrence of the same (potentially Page Zero) conditional literal.

The programmer could create a mechanism to account for subroutines used in the library such as the following:

```
   SUB1=   1        /SUBROUTINE #1 IS USED IN THIS PROGRAM.

   SUB2=   1        /SUBROUTINE #2 IS USED IN THIS PROGRAM.
```

The body of code in the library for subroutine #1 would be bracketed within the following statements.

```
   IFNZRO  SUB1    <

   /       THIS IS WHERE THE ACTUAL SUBROUTINE #1 CODE GOES.

                   >
```

Similarly for subroutine #2:

```
   IFNZRO  SUB2    <

   /       THIS IS WHERE THE ACTUAL SUBROUTINE #2 CODE GOES.

                   >
```

For the actual conditional literal section in question, management statements similar to the following can implement the example described above:

(This is the section where the potential Page Zero conditional literal is located within subroutine #1.)

```
   IFZERO  SUB2    <

   AND     #77              /REMOVE UPPER SIX BITS.

                   >

   IFNZRO  SUB2    <

   AND     [77]            /REMOVE UPPER SIX BITS.

                   >
```

The second subroutine uses the conditional literal form of a similar statement; the actual binary code generated within subroutine #2 is a Page Zero literal if a) the main program uses the same Page Zero literal, or b) subroutine #1 uses the explicit Page Zero reference because it was aware of the identical usage within subroutine 2.

Future releases of P?S/8 PAL may make mechanisms such as this slightly easier to implement. There exists an obscure assembler for PDP-8 code including literal support and support for the instructions of the FPP-12 (and FPP-8/A, E). It was designed and implemented by Jack Burness for use with *DIAL-MS* (the operating system he wrote exclusively for the 8K PDP-12 with one of a small class of specific disk storage devices).

This assembler is known as *FPPASM* and differs from the embedded DIAL assembler in the following particulars:

a) The DIAL assembler supports LINC mode assembly, FPPASM does not. (P?S/8 PAL supports LINC mode assembly as a modular option which can be enabled by the use of command-line option switches.)

b) The DIAL assembler does not support literals of any form while FPPASM supports Page Zero and current page literals in a manner generally compatible with PAL10, PAL8 and P?S/8 PAL.

c) The DIAL-MS assembler and the FPPASM assembler support certain directives and conditional assembly mechanisms incompatible with PAL10, PAL8 and P?S/8 PAL (when dual assembly mode is disabled). All of the directives unrelated to conditional assembly represent alternative syntax for the functionality supported by the three assemblers as stated above; in some cases, these features are less powerful and represent older language elements generally abandoned in favor of the newer directives. Any program can become fully compatible among the three assemblers by incorporating appropriate substitutions.

P?S/8 PAL supports most of the FPPASM conditional directives if the proper command-line switches are used.

Additional directives available in FPPASM may be implemented in future releases of P?S/8 PAL which are helpful when supporting source code libraries.

The exact syntax of these directives will be made more compliant with the other standard PDP-8 assembler directives; other directives used within FPPASM have already been added to P?S/8 PAL for full compatibility with the DIAL assembler.

The IFREF directive assembles code contained within angle brackets in a manner consistent with the IFNDEF, IFDEF, IFNZERO and IFZERO directives. The determinant is whether or not a stated symbol exists in the symbol table.

The IFNREF directive assembles code contained within angle brackets if the stated symbol does not exist in the symbol table.

While the equivalent management logic can be implemented using the IFNDEF and IFNZRO directives (for most source library statements), the IFREF and IFNREF directives can simplify the implementation of source code library support, especially if the various code sections make use of conditional literals.

Called subroutines are included in the overall program because they are referenced in the main source code section. Properly utilized, source library maintenance can be an automatic process. Library routines can make use of the XLIST directive to prevent inclusion of unreferenced sections in the assembly listing.

Note: FPPASM was primarily written to support source libraries for the FPP-12 hardware as part of the original FPP-12 software package support for DIAL-MS. Some of the actual library files are available on certain online software archive sites. See below for the site maintained by the author of this document. Source code of both the FPPASM assembler and various FPP-12 support libraries can be found at the following web site:

http://www.ibiblio.org/pub/academic/computer-science/history/pdp8/FPPASM%20Files/

End of Appendix C

Appendix D - P?S/8 PAL command-line option switch ordered summary

/A If a symbol table or cross-reference is in effect, all symbols will be used.

/B If explicit output files are not specified, the first binary output file is % on the system device unit. See the description of the /D and /U command-line option switches.

/C Enable internal reconfiguration of P?S/8 PAL resources to favor symbol table capability over performance.

/D If explicit output files are not specified, the second binary output file is $ on the system device unit. See the description of the /B and /U command-line option switches.

/E Do not retain literal extents when leaving the current page.

Note: This command-line option switch is ignored unless some combination of the /Q and /O command-line option switches are in effect.

/F If P?S/8 PAL chains to P?S/8 BIN for the purpose of punching binary paper-tape output, the output device is the high-speed punch. See the description of the /R command-line option switch.

/G Chain to P?S/8 BIN for the purpose of either loading the binary files created during the assembly or punching binary paper-tapes from the binary files (if the /W command-line option switch is also set).

Note: Chaining to P?S/8 BIN is inhibited if there were errors detected during the assembly.

/H If chaining to P?S/8 BIN and the /W command-line option switch is not set, all user memory is preloaded with 7402 (the PDP-8 HLT instruction) prior to binary file loading. See the description of the /G command-line option switch.

/I If chaining to P?S/8 BIN and the /W command-line option switch is not set and using a hardware-dependent Slurp format binary loader, the system device handler will be reloaded after all binary files are loaded.

Note: This command-line option switch is ignored on systems that only support the virtual Slurp format binary loader.

/J Prevent generation of a 12-bit word containing 0000 after text strings with an even count of characters as used with the TEXT and SIXBIT directives.

/K Enable the change of title on assembly listing from every input file. By default, only the first input file supplies the title field for the entire listing.

Note: This assumes listing output is enabled. If there is neither binary file generation nor listing enabled, force a two-pass assembly.

/L Assembly listing output is produced during pass two of the assembly process. See the description of the /N, /P and /X command-line option switches.

/M Chain to P?S/8 *MAP* to create a bitmap of all binary output generated during the assembly.

Note: When chaining to P?S/8 *MAP* then P?S/8 BIN, the chain to P?S/8 BIN will be inhibited if there were errors detected during the assembly; however, the chain to P?S/8 MAP will always be performed.

/N Enable *neatness* (*niceties*) options of an assembly listing. Page headers are added including the current date, day of the week and page sequence numbers. See the description of the /T option switch.

/O Enable link generation. Links will be flagged as assembly errors unless the /Q command-line option switch is also set.

/P Enable wide-carriage listing output; where applicable, the output will be oriented towards 11" x 17" ledger/tabloid format. See the description of the /L and /S command-line option switches.

/Q Enable generation of Page Zero, current page and conditional literals. Link generation is disabled. See the description of the /O command-line option switch.

/R If P?S/8 PAL chains to P?S/8 BIN for the purpose of punching binary paper-tape output, the tape is punched in enhanced RIM format; this is readable by both the RIM loader and the BIN loader. See the description of the /F command-line option switch.

/S Symbol table output will be created at the end of the assembly. See the description of the /A, /K, /N and /P switches.

/T Listing output is forced to the system console. If the /N option switch is set, tear-off lines will be printed on every page.

/U If explicit output files are not specified and the /B or /D command-line option switches are in effect, the logical unit used is the system device logical unit number .XOR. 1.

/V If chaining to P?S/8 BIN and the /W command-line option switch is not set, the virtual Slurp format binary loader will be used to load Slurp format binary files.

Note: Using the /V command-line option switch causes the /I command-line option switch to be moot. The virtual Slurp format binary loader is used by default on certain system configurations.

/W Chain to P?S/8 BIN to punch binary paper-tapes from the binary output files. See the description of the /F and /R switches.

Note: Chaining to BIN is inhibited if there were errors detected during the assembly.

/X Enable cross-reference listing output of user symbols. Listing output width is increased to accommodate statement numbers. See the description of the /A, /N and /P command-line option switches.

/Y The ! operator is changed from *inclusive OR* (.OR) to shift left six bits.

/Z If chaining to P?S/8 BIN and the /W command-line option switch is not set, all user memory is preloaded with 0000 (the LINC HLT instruction) prior to binary file loading. See the description of the /G command-line option switch.

/0 Automatically generate *200 after use of the FIELD directive. By default, only the field change is generated.

/1 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 1. See the description of the =xxxx parameter below.

/2 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 2. See the description of the =xxxx parameter below.

/3 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 3. See the description of the =xxxx parameter below.

/4 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 4. See the description of the =xxxx parameter below.

/5 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 5. See the description of the =xxxx parameter below.

/6 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 6. See the description of the =xxxx parameter below.

/7 If P?S/8 PAL chains to P?S/8 BIN for the purpose of loading the binary output created by the assembly, the program will start in field 7. See the description of the =xxxx parameter below.

/8 Enable LINC and PDP-8 dual assembly; the default operating mode is PDP-8 mode (PMODE). The default assembly address is 0200.

/9 Enable LINC and PDP-8 dual assembly; the default operating mode is LINC mode (LMODE). The default assembly address is 4020.

= If the =xxxx parameter is explicitly given on the command-line, the program will be started at location xxxx in field 0 (unless one of the extended memory command-line option switches /1 through /7 described above are set, in which case the starting field is set to the corresponding field).

Note: The program will not be loaded (or started) if there were errors detected during the assembly.

End of Appendix D

1. BIN format

   The standard binary loading format in the DEC *Paper-tape operating system* is known as DEC *BIN* format which is optimized for loading data from paper-tape frames. This format includes both leader and trailer frame codes to delineate sets of data. The highest tape channel is used as a visual aid to help position paper-tapes in the reader; all leader and trailer frames have only the high-order bit punched. Data frames may or may not have this bit punched, but clearly have some combination of the other bits also punched. All leader frames are ignored. Loading ends when the first trailer frame is encountered. DEC BIN format incorporates all necessary elements needed to load 12-bit words into memory including a 12-bit checksum at the end of the tape; memory field change frames are included as required.

   Note: field change frames have the two highest bits punched. Additionally, field change frames are not included in the overall tape checksum. It is surmised this came about because the format was upgraded when machines larger than 4K memory were first produced, this could have had compatibility issues with DEC BIN format tapes produced somewhat earlier.

   DEC BIN format is inherently compromised in terms of loading efficiency because of the need to support leader and checksum codes that are superfluous in an operating system environment (such as P?S/8). Storage devices inherently have device parity checking for all data; leader and trailer frame codes serve no purpose when data is stored in blocks on a mass storage device.

   While certain systems use an adaptation of DEC BIN format for binary files, P?S/8 uses the more versatile and efficient Slurp binary loading format described elsewhere; Slurp format is optimized for efficient loading. The P?S/8 binary utility program (also known as *BIN*) includes utilities to convert between paper-tape data and binary file data (in both conversion directions).

   Note: Throughout this document, BIN refers to the P?S/8 BIN utility as mentioned above. More information about P?S/8 BIN can be obtained from other documents such as the *P?S/8 System Programs Guide* available separately.

2. Edit buffer

   P?S/8 includes a set of embedded commands in the keyboard monitor which allow editing of a text file directly (without loading a separate editing program). This design concept allows for rapid editing of portions of a larger project that generally extends across several TFS files. A file can be created from scratch or an existing file can be loaded into the edit buffer within the keyboard monitor. Files are tentative until the edit buffer contents are saved to a named file in the TFS directory of the system device (or one of the other supported logical device units).

   While there are some tradeoffs when compared to more traditional file editing methods, generally there is only incidental need for additional storage space; most files are updated in the edit buffer and then written back to the same file slot on the system device. Certain users also create backup copies of important files to ensure the preservation of the file under edit, but this practice is not mandatory.

   Note: All files of a project may be saved to an alternate logical unit with the same (or variant) file names. Backing up project files (periodically) in this manner generally makes the creation of individual backup files moot. Many users do not bother with any form of file backup.

3. Extended-length text files

   Extended-length files partially resemble TFS text files except no line number or line pointer data exists within the file. Additionally, there are no length restrictions on extended-length files (other than the applicable limits of the storage device).

4. Logical Console Overlay

   The Logical Console Overlay is an optional component of P?S/8. When enabled, all conforming programs will direct all system console (and lineprinter) input/output to subroutine calls to designated memory locations instead of the usual physical devices. While the use of the Logical Console Overlay requires the availability of extended memory, certain advanced features become available as described below:

   a) If the P?S/8 system device handler detects an error, the usual action is to halt the CPU (often displaying device-dependent status bits in the accumulator); pressing CONTINUE on the system front panel will safely retry the latest read/write operation. Users need to be familiar with the particular system device handler in use to gain insight as to what is causing the specific problem by examining the contents of the accumulator (if feasible).

Note: A system option is available to disable the system halt, resulting in an infinite retry after any error; depending on hardware particulars, this setting is generally not recommended. (Ironically, using this setting causes error handling to resemble the mediocre error recovery techniques usually implemented in many OS/8 device handlers.)

When the Logical Console Overlay is enabled, the default action taken by the system device handler is transformed into issuing a status report on the system console device (which is potentially redefined by the Logical Console Overlay configuration to be directed to alternative hardware). This generally includes details of the latest read/write attempt. The user can choose which action to take (at the point of error) by entering one of the options appearing on the system console keyboard. This allows the user to correct the problem (if possible) and retry the latest read/write operation, or to abort the latest operation (if necessary). Even without having specific knowledge regarding the vagaries of the current system device, any user can easily manage all system device handler recovery operations (perhaps after consulting some device-specific documentation).

b) Not every system device handler is capable of optimal performance; this is especially true of system device handlers designed to run on 4K systems (without requiring loading of a portion of the handler into extended memory). Due to the limited memory space on 4K systems, it is often necessary to compromise performance to obtain basic functionality.

When the Logical Console Overlay is enabled, additional functionality (designed into the system device handler for this purpose) is also enabled. This generally implements additional performance improvements that should overcome the original design compromises needed to obtain basic functionality.

Note: On hardware configurations where the system device handler requires an extended-memory segment, the Logical Console Overlay is designed to load into lower memory locations within the same (highest) memory field. As such, for conforming programs (such as P?S/8 PAL) attempting to locate extended memory fields, the presence of the Logical Console Overlay does not change the count of fully available memory fields. In certain circumstances, programs may take advantage of the 3/4 memory field available if the Logical Console Overlay is disabled. For example, P?S/8 PAL may support up to 768 additional symbols if this memory space is available.

c) Certain configurations of the Logical Console Overlay may use alternate console hardware (such as a console terminal with interface device codes differing from 03/04). In certain configurations (based on terminal emulators), an error panel screen may temporarily replace the normal display of console output until the error is resolved.

5. RIM format

While BIN format (as described elsewhere in this document) is the standard format for loading most binary paper-tape programs, *RIM* format is generally used to first load the BIN loader. While RIM format is more inefficient, this is generally not an issue when loading relatively short programs.

Some users want a minimal overall solution to loading certain particular programs; as such, the binary data might be punched in RIM format, eliminating the need for the BIN loader as an intermediate loading program.

RIM format paper-tapes use the same overall frame layout as that used with the BIN loader; the differences are as follows:

a) Auto-update of the memory loading pointer is not supported. Each loaded word must include an origin setting preceding the data word.

b) There is no support for an overall checksum in a RIM binary paper tape.

   Note: There are various techniques available to include an optional checksum at the end of a RIM format binary paper tape; this adds the ability for the BIN loader to load the same program as an alternative to the RIM loader.

c) No support for Field settings. The user must set the loading field from the front panel Data Field switches or equivalent means; this setting applies throughout the entire RIM loading session.

d) Two equally short versions of the DEC standard RIM loader exist. One supports the high-speed reader; the other supports the low-speed reader of the model ASR 33 (or ASR 35) Teletype often used as the system console device. All versions of the BIN loader are far longer programs; either of the RIM loader variants can be used to quickly load the BIN loader; as such, the RIM loader is often hand-toggled into memory using the computer's front panel switches.

6. Slurp binary file format and Slurp binary file loader

Slurp is the name given to the P?S/8 binary file format by its inventor, Richard Lary. (The name is suggestive of the mechanism by which the loader functions.) Mr. Lary and his colleague Lenny Elekman are the principal authors of the *R-L Monitor System*. P?S/8 is (partially) based on this earlier system. Both operating systems support Slurp format binary files as created by P?S/8 PAL and other utilities; due to implementation restrictions within the R-L Monitor System, there is no support for extended memory loading.

Note: The assembler used in the R-L Monitor System is the DEC Paper-Tape Operating System assembler known as *PAL III*. Certain modifications were made to write binary output to the system device in Slurp binary format. Although binary output is limited to designated system blocks, additional commands exist to transfer the binary output to user-designated Slurp format binary files in the TFS directory (when and where feasible, which is not always the case due to fixed file length considerations). P?S/8 PAL supports Slurp format binary output files directly, which eliminates this problem; the default binary output design was remedied as follows:

a) The block layout of the R-L Monitor System is such that Slurp format binary output is always written starting at absolute block 0022. There is a fixed file that can be generally referenced (as if it were in the TFS directory) located at blocks 0040-0057; in all command scenarios, this is accessed using the *$* symbol. If sufficiently long Slurp binary output is created, it can easily overlap the storage space of the $ file; as long as the length is less than 0020 blocks (meaning that no more than blocks 0040-0041 have been overwritten within the $ file), the Slurp format binary data can be transferred to a named file in the TFS directory. However, some programs create larger Slurp format binary data (which had been allowed to be as long as blocks 0022 through 0057 without damage to the operating system). Such programs had to be reassembled and then loaded from the default Slurp binary area should there had been any activity that might change these specific circumstances (such as writing over the $ file).

   Due to an implementation restriction of the R-L Monitor BIN utility, it is impossible to load the contents of the actual $ file; the syntax used to access it instead refers to the Slurp format binary output area starting in block 0022. (The BIN program reinterprets the starting block of the first input file as 0022 if the actual passed value is 0040.)

Note: R-L Monitor system program names are limited to two characters, thus R-L Monitor *RU{n}* commands typically reference *PA{L}* and *BI{N}*, etc. Additional characters are allowed on the command line, but will be ignored.

b) P?S/8 changes certain aspects of the operating system block layout such that blocks 0020-0037 can be referenced with the symbol % which is analogous to the $ file (which references blocks 0040-0057) in all relevant command contexts.

The storage blocks occupied by the two files (% and $) are also accessed as the virtual image of field 0 once created by the *GET* command (or equivalent). In this context, direct usage of the two file names should be avoided to prevent destruction of the core image; however, the image contents can be saved (and restored at a later time if desired) by using a series of keyboard monitor FE{tch} and WR{ite} commands as required. If there is no requirement to preserve the core image, the two files can be used independently in lieu of TFS files (which are generally faster to access due to close proximity to the beginning of the storage device).

Note: while the P?S/8 SHELL overlay will (generally) not reference the TFS file structure, the field 0 virtual area is identically defined as the same blocks used in the basic P?S/8 system. This can be useful when changing between the basic P?S/8 and the P?S/8 SHELL for further program development and related logistic issues.

P?S/8 PAL includes the /B, /D and /U command-line option switches to access these files without having to provide explicit output file references (where applicable).

Thus, any reference to the % file (whether explicitly as a stated output file, or by use of the /B command-line option switch passed to PAL) is the closest equivalent to the original functionality in the R-L Monitor System (without the potential file destruction problem).

Note: The basic P?S/8 execution-class commands access system programs with up to six character names. For programs with exactly six characters in the name, the commands can be embellished using additional characters of the user's choosing (all of which are ignored). As such, this is not an issue for relevant programs as discussed here including PAL, BIN, GET, START, ODT (since all of them have shorter program names).

The fastest loading of binary data into randomly accessed memory locations is achieved using the Slurp binary format method. Actual loading efficiency approaches 6/7 of block-oriented (core-image) binary format while allowing the loading of arbitrary data ranging from individual memory locations through large blocks of contiguous memory without requiring any form of block structure. Memory can be freely loaded anywhere in the entire PDP-8 32K memory space (other than system reserved memory areas).

Note: All implementations of the Slurp loader prior to P?S/8 were limited to loading into field 0 only; P?S/8 uses an extension to the format that doesn't impact on 4K usage. An appropriate *CDF XX* instruction to the new memory loading field is required when the loading field changes.

Since the P?S/8 Slurp binary file loader is a superset of the original 4K-only versions as used is several other PDP-8 systems, files transferred from the R-L Monitor System can be loaded without internal conversion.

Since P?S/8 PAL supports chaining to P?S/8 BIN, all loading options of the Slurp binary file loader are available during a chain operation; additional details are described elsewhere in this document.

A list of other operating systems that support a Slurp format binary loader follows:

a) The R-L Monitor System (also known as *MS/8* as submitted to DECUS). This is the earliest system to support a Slurp format binary loader, albeit limited to 4K loading. This system did not provide a virtual Slurp format binary loader option and is only implemented on TC01/TC08 DECtape for a single drive unit. Most notably, MS/8 was not a commercial product, as it was entirely written by students of the *Polytechnic Institute of Brooklyn* (now a division of *New York University*). Most of these students eventually became employees of *Digital Equipment Corporation* (*DEC*); however, all work on the R-L Monitor System was performed by students in the New York City area (including the author of this document who added support for FOCAL, 1969 just before the submission as DECUS 8-466).

b) While DEC was totally unaware of this situation, *DIBOL-8* (the precursor of *COS-300/310*) was actually a hacked-up variant of the R-L Monitor System; binary output of the DIBOL compiler (and the DIBOL run-time system) are loaded by the original R-L Monitor System Slurp format binary loader. The keyboard monitor commands were modified to be consistent with the proposed COS systems released somewhat later.

Note: Early releases of P?S/8 are faithful to the original keyboard monitor commands of the R-L Monitor System; however, all recent releases are consistent with the DIBOL/COS systems. Where relevant, documentation of the COS keyboard commands also applies to P?S/8.

c) *POLY BASIC* is a stand-alone BASIC-only operating system for a 4K PDP-8 including any of several storage devices such as the TC01/TC08 DECtape, PDP-12 LINCtape, DF32 and RF08. The POLY BASIC source code was originally created on The R-L Monitor System by the same students who created the original R-L Monitor System while students (just before their employment began at DEC in Maynard Mass).

Note: Since POLY BASIC is a standalone operating system, binary output from PAL III is loaded into memory; when the system halts, a scratch DECtape is mounted on drive unit 0 to allow writing out the components of the POLY BASIC system.

The internal run-time system and binary output of the POLY BASIC compiler are loaded into memory using a Slurp format binary loader limited to 4K memory.

Note: For devices where a virtual loader would be required in P?S/8 (due to hardware limitations), POLY BASIC takes advantage of an internal buffer known to be available while file loading is in progress. (The buffer area is later used for dynamic program storage.)

A minimal distribution cleanup and eventual submission to DECUS was accomplished later when the (former) students were DEC employees (as opposed to the bulk of the work done earlier as students).

DEC management unfairly exploited this situation and claimed the project as DEC intellectual property without any compensation to either the students or the school. This situation never sat well with certain individuals associated with this incident; it was felt at the time that DEC was too powerful to lodge any complaints against (and the inherent intimidation caused by fear of being fired).

d) *EDUSYSTEM 30* is DEC's brazenly commercial release of POLY BASIC with little to no modifications to the original student project.

e) *EDUSYSTEM 15-30* is a specific variant of EDUSYSTEM 30 that runs only on the TD8E DECtape. Either 8K of memory or the MR8E-C support ROM is required.

Note: When the MR8E-C ROM is used, it is located in field 7 of memory in locations 77400-77777. As such, the host PDP-8/E (or similar) system must support extended memory for either configuration. It is believed that, out of expediency, the technique of using an internal buffer during Slurp loading is deployed in this system despite the ability to implement an appropriate hardware-specific version; P?S/8 implements a TD8E-specific Slurp format binary loader for the identical hardware configurations analogous to other devices such as the RX01 which also transfer data solely by program transfer techniques.

7. Three-for-two

This is an informal reference to the method used in certain PDP-8 file systems to pack three bytes of seven (or eight) bits each into two adjacent 12-bit words within a larger structure (such as a storage device block or record). This packing scheme is used inside of OS/8 binary files and also text files in both OS/8 and the P?S/8 SHELL environments.

The first character occupies the low-order eight bits of the first twelve-bit word. The second character occupies the low-order eight bits of the second twelve-bit word. The third character is split into the upper-most four bits and the lower-most four bits. The upper-most four bits are placed into the four high-order bits of the first 12-bit word; the lower-most four bits are placed into the four high-order bits of the second 12-bit word.

While several other (less than satisfactory) methods of implementing three- for-two packing and unpacking have been devised, clearly the most elegant way is to use co-routines such as how the author of this document implemented file support in the OS/8 program *Kermit-12*. The coding space occupied by both routines together is slightly less than half of that used by commonly used routines nominally written for the same purpose in other OS/8 programs, yet the co-routines are also faster and have higher overall data throughput. This is one of the few counter-examples where speed and small size are not in tradeoff opposition.

The P?S/8 SHELL file system will store text format files in a manner similar to OS/8 with certain considerations as follows:

a) OS/8 (nearly) always sets the high-order bit of every frame to (over) simplify the process of conversion of the data to paper-tape frames; however, since this is a trivial consideration (and is also often ignored), it is always necessary to mask off the high-order bit (which contains no useful information) when processing the latest byte.

b) All OS/8 files are meant to be terminated with trailing Control-Z characters (with the high-order clear); however, this convention is often misimplemented with the high-order bit inadvertently set; in some cases, files end at the last byte of the last block of the file, entirely lacking the trailing Control-Z character. The co-routines used are able to correct for these occasional file formatting blunders.

c) P?S/8 SHELL text files will always clear the high-order bit of every byte; in special cases, setting the high-order bit will be allowed for special formatting purposes using proprietary extended character codes beyond the normal seven-bit ASCII character set.

d) P?S/8 SHELL text files will always include trailing Control-Z characters with the high-order bit clear. Co-routine implementation makes this easy to achieve in minimal code space.

e) P?S/8 SHELL binary files will use Slurp binary loading format as is used in the basic P?S/8 TFS file system. Some slurp format binary loading or other utilities are best implemented using co-routines; three- for-two packing methods play no role in Slurp format binary files (which are multiples of 12-bit words).

8. Tiny File System (TFS) and the TFS directory

The *Tiny File System* (*TFS*) is the primary file format used in the basic P?S/8 system. Each TFS file is of fixed size allowing the TFS directory structure to consist of allocatable slots that are adjacent and cannot overlap.

If a TFS directory filename is modified or deleted, the underlying storage is not changed (unless an explicit write operation is performed). This allows files to be easily renamed as needed.

There are unique advantages to the TFS structure which, for certain users, requires some measure of orientation and experience. Once mastered, rapid editing of large projects can proceed with little additional effort.

Files are loaded into the keyboard monitor edit buffer to modify the contents. The process is not finalized until appropriate commands are issued to write the edit buffer contents to a slot in the TFS directory. This generally means that the same file slot is reused; as such, there will be no need to access the TFS directory.

Note: For those requiring the notion of a file backup, the updated file can be written to a different TFS directory slot. Appropriate filename changes can be made later to maintain ongoing file names. Advanced users are known to pre-allocate additional directory slots for this purpose.

The TFS directory structure is used only for user-created files, including the Slurp format binary files produced by P?S/8 PAL assembly of TFS text files (and/or extended-length text files). System programs are stored in an independent file structure elsewhere on the system device and/or other logical device units.

For example, it is expected that PAL assembly projects of nearly any size will require several TFS files. The contents of multiple TFS files are concatenated together and passed to P?S/8 PAL (as required) to carry out the current assembly project.

Note: While certain P?S/8 system programs address the associated line numbers within TFS text files, P?S/8 PAL (and most other P?S/8 system programs) ignores line number data. When necessary, extended-length files can be passed to P?S/8 PAL to allow assembly of vary large source programs. As necessary, the source file input stream passed to P?S/8 PAL can be a mixture of TFS text files and extended-length text files.

End of Glossary of Terms

[End-of-file]