Pascal-P and PCD System

Referance Manual

Version 3.1.9

Last modified (87/01/26)

Chapter 1

Introduction

This system has been designed for maximum "friendliness", and to avoid unexpected responses and "surprises". The interior design is highly structured, and easily customizable. Close adherance to the ISO standard (with slight extension other than available standard procedures, all compiler detectable) is enforced. The system is portable to other machines.

Pascal-P is a modification of the P4 Pascal compiler developed by Amman, Nori, and Jacobi at the Institut fur Informatik at the at the Eidg. Technische Hochschule in Zuerich. It was adapted for use on the HP3000 by Grant Munsey, Jeff Eastman, and Bob Fraley of Hewlett-Packard Labs, 3500 Deer Creek Rd, Palo Alto, Calif. 94304. It has been further adapted for use with a generalized machine independent P-code interpreter, and for 8080 native code generation, by C.B. Falconer, 680 Hartford Tpk., Hamden, Conn. 06517, USA.

The revised compiler, interpreter, standard procedures, system interfaces, and the present 8080 and P-code code generators (in Pascal) are due to C.B. Falconer, as are any errors therein.

At present the system executes on the HP3000 or on CPM systems with a minimum of 48k memory (Compilation of the compiler requires 63k under CPM). Using the identical compiler it generates code for:

HP3000
P-code interpreter (machine independent)
8080 native code

from the same source files, controlled by various compile time commands.

The P-code codefiles are executable under CPM, or (unchanged) without any disc system when interpreters are linked to appropriate device drivers. Similarly native code files can be linked to the same drivers, when maximum speed is required. Such combinations are suitable for dedicated systems, and are especially attractive when accuracy is critical, because of the extensive compile time and run-time error checking available. In addition such programs can be ported to wildly differing machines and CPUs. Contact C.B. Falconer for further information and licensing.

Such system features as separate compilation, i/o redirection, program segmentation, virtual code-memory, debug and tracing

capabilities are incorporated. No distortion of the Pascal source is required. Program profiles require the addition of about 6 lines in the source text.

Standard Pascal

> This manual is NOT an exposition of the standard Pascal language. The recommended reference manual is "Programming in Pascal", by Peter Grogono. Further useful references are "Pascal User Manual and Report", by Jensen and Wirth, and the ANSI and ISO standards (which are hard reading).
>
> This system implements ISO and ANSI standard Pascal, except for GOTO out of procedures, and passing procedures/functions as parameters.

A fairly extensive set of utilities are available, all written in Pascal, including various non-portable CPM system programs.

- UCSTOCPM which transfers UCSD Pascal text files to the CPM system.
- DISKCOPY which makes complete copies of floppy disks).
- TRANSFER which copies files to and from MS/PCDOS format disks.
- ANSWER, BYE, ENDCALL which implement a remote controlled RCPM system, and which can automatically limit execution to a user defined program.

- programs from PUG (Pascal Users Group) such as..
- COMPARE which compares text files and resynchronizes after differences.
- REFRENCE which shows Pascal program structure and procedure referances.
- ID2ID which replaces identifier names from a list.

- portable programs such as..
- RNF, the text formatter which prepared this manual.

- BINHEX which converts binary files to Intel hex format records.
- PAGER which paginates, labels and dates listings.
- FILEDUMP which lists binary files in hex notation.
- PLOTPROF which plots profiles of program execution.
- XREF which prepares a cross-referance of Pascal programs.
- PCDISASM which dis-assembles "pcd" codefiles.
- WSTOTEXT which converts WordStar document files into standard formats.
- LDIR which list library directories, including datestamps
- LSETDATE which sets datestamps in LU format libraries.
- XREFC which cross-references C programs.
- XREFASM which cross-references assembly programs, and adapts to various machines with an external file (available for 8080, Z80, 8086)

- system programs such as..
- ASSMPCD, the .PCD code generator
- ASSMAP, the native code generator
- TUNE, which dynamically configures code/data space usage in .PCD programs.
- LINKER, a machine independant linker, incidentally used to link .RBM (relocatable binary modules) into .PCD code files.

This manual was written on a wide variety of text editors and finally printed by RNF, a text formatter analogous to the Unix Runoff, and written in Pascal. (The original author of RNF is unknown).

P-code codefiles can be as large as 127 (31 under CPM) segments, each containing a maximum of 127 procedures in a maximum of 32 Kbytes of code. Thus the absolute maximum program file is approximately 4 megabytes (992 kbytes under CPM). This permits large application systems to be created and automatically manipulated by the run-time memory management system. For comparison the compiler occupies less than 42 Kbytes in 28 segments,

and can execute in approximately 6K of codespace. Virtually full compilation speed is attained in approximately 20K of codespace under CPM.

Codefiles are automatically searched over two disks, and one library file under CPM. Since P-code utilities tend to be small, the library system provides significant improvement in disk space utilization by eliminating allocation fragmentation. This can have dramatic effects when the disk allocation unit is large, e.g. hard disks with a 4 kilobyte allocation unit. The use of library code files under CPM has the additional advantage of permitting date-stamping. Upgraded programs may be tested without affecting the original, since the search order prefers a file.

PCD codefiles may optionally be segmented and execute in a virtual memory space. The run-time system automatically performs segment loading and unloading on demand, and chooses segments for discard on a least-recently-used algorithm. Program files can specify the amount of real memory to be allocated for code loading. All code is read directly from diskfiles, and is always re-entrant and pure. Thus no additional disk space need be allocated to the virtual memory system.

Chapter 2

Getting Started

CPM version

This chapter is directed towards CPM 2.2 users. Other systems must make slight modifications, especially MS/PCDOS users.

2.1 Files needed.

To compile and execute Pascal programs you should have available:

1. RUNPCD.COM (and possibly RUNPCDI.COM). These are distributed as INTERP.COM and INTERPI.COM, and may be renamed as is, or customized for access to system timers.

2. PASCALP.PCD, the compiler

3. either ASSMPCD.COM or ASSMPCD.PCD. The .PCD version is much more compact, but significantly slower in execution.

4. (optional) EF (no extension). The error messages file for compilation.

5. (optional) PCDHELP.PCD, which is executed by RUNPCD whenever no code file is specified. This gives on-line information.

6. Your favorite text editor.

For .COM file generation under CPM you will also need:

1. SLRMAC.COM from SLR systems (Z80ASM may also be used).

2. SLRNK.COM from SLR systems

3. PASCLIB.SLR. The run-time library.

4. CPMLINK.SLR. The master interface to CPM

5. (optional) HEAPMARK.SLR (allows reduced object size when no use of the DISPOSE procedure is made).

6. (optional) ERRMSGL.SLR (allows more elaborate run-time error messages. The "L" stands for long.)

Also recommended, for convenience:

1. JOB.COM. The improved replacement for SUBMIT.

2. JOBS.LBR (or extractions of COMPILE.JOB and/or PASCPREP.JOB) to allow single command compilation.

ASSMPCD and ASSMAP.COM

If you use these rather than the .PCD versions simply omit the "runpcd " prefix for them in the following.

2.2 Creating .PCD programs

To compile the source program YOURPROG.PAS to a .PCD file, do:

```
runpcd pascalp (yourprog.pas, con, yourprog.tic)
```

where "con" (the console) will receive the list file, and "yourprog.tic" will receive the temporary intermediate code. At completion enter:

```
runpcd assmpcd (yourprog.tic,yourprog.pcd)
```

If JOB.COM and COMPILE.JOB are available (COMPILE.JOB may be a component of JOBS.LBR) this can all be simplified to:

```
job compile yourprog
```

At completion YOURPROG.TIC may be erased. To then execute yourprog, enter:

```
runpcd yourprog
```

2.3 Creating .COM programs

To compile the source program YOURPROG.PAS to a .COM file, do:

```
runpcd pascalp (yourprog.pas, con, yourprog.tic)
```

just as for a .PCD file. If you have saved the .TIC file from a previous compilation (above) this step can be eliminated.

```
runpcd assmap (yourprog.tic, yourprog.mac)
```

Again, at completion YOURPROG.TIC may be erased, unless required to create a .PCD executable file. If you are going to use Z80ASM in place of SLRMAC add "[64]" to the above command line, replace ".mac" with ".z80", and replace "slrmac" with "z80asm" in the following step. The resultant intermediate source file will be about 10% larger, but the final code will be unchanged.

```
slrmac yourprog.@@z/rf
```

will assemble the .MAC source file. At completion YOURPROG.SLR will have been created, and the .MAC file is no longer needed. (If you have not configured SLRMAC to use the .SLR extension, change the extension accordingly)

```
slrnk /a:100,cpmlink,yourprog,pasclib/s,yourprog/n,/e
```

will do the complete linking, and leave you with YOURPROG.COM.

Again, if JOB.COM and PASCPREP.JOB are available (PASCPREP.JOB may be a component of JOBS.LBR) this can be simplified to:

```
job pascprep yourprog
```

YOURPROG can then be executed just like any other program.

2.4 Using PCDS.LBR

Any .PCD file may be stored in PCDS.LBR, and RUNPCD will find and load it whenever no file.PCD is found. Since .PCD files tend to be small, this can result in significant savings in disk storage.

2.5 Connecting files to programs

The standard input and output may always be redirected by including ">outputfile" and/or "<inputfile" on the command line. This applies to both .COM and .PCD programs. Other external files should have been described in the program header line, and are simply described as parameters to the program by suitable replacements within parenthesis in the command line. Any unspecified file will be attached to the default name, which is the name used in the source program, with no extension. The standard input and output must be text files, but other files may be of any type.

Standard device file names available (on CPM) are:

1. CON the console for output, keyboard for input. Lines are buffered and input lines can be edited.

2. KBD The console for output, keyboard for input. No buffering.

3. RDR The system RDR: device. Input lines are buffered, but may not be edited.

4. PUN The system PUN: device. No buffering.

5. LST The system LST: device. No buffering.

6. AUX The system RDR: device, no input buffering.
   All these file are textfiles.

2.6 Using CCP+ (and possibly DOS+)

If you have installed CCP+ in your system, together with CCPXTEND.SYS, you can omit the "runpcd" prefixes. The system will automatically execute RUNPCD if no .COM file is found.

Chapter 3

Program File Preparation

3.1 Line Numbers and Line Length

Source files for Pascal-P are normally variable line length Ascii files, with or without FRONT numbering (described below). By default any characters past the first 80 of a line are ignored, and treated as comments.

Source lines containing an initial string of 8 digits are treated as numbered, and the line numbers retained and processed by the compiler. These 8 digits are not included in the 80 (or other, see w compiler option) character line length limitation. The compiler assigns sequential line numbers to un-numbered source (but see $include below).

3.2 Indentation coding

Source files may use data compression for indentation, consisting of an initial "DLE" (chr(16)) followed by the printing character chr(ord(' ') + indentationcount). This both compresses source code storage, and speeds up compilation by eliminating scanning of unnecessary blanks. The utility programs "EXPAND" and "COMPRESS" can be used to control this formatting. The above line length limitation applies to the expanded source line.

3.3 Comments

Comments follow the standard Pascal conventions using either (* and *) or { and } as delimiters. Note that "(*" cannot be terminated by "}", nor can "{" be terminated by "*)". This is a deliberate deviation from the ISO standard.

3.4 File inclusion

A line beginning (at the extreme left, apart from line number) with "$include filename" causes that file to be included in the compilation at that point. The remainder of the line is listed but ignored. If the included file is unnumbered further line numbers are assigned at the next multiple of 1000 plus 1.

The "i" compiler option, (See compiler options) avoids the line number control and may be combined with other options on the same line.

At present the master file and two (2) levels of inclusion are available.

3.5 Characters and Identifiers

The compiler recognizes both UPPER and lower case letters. A tab character is treated as a space, but listed as a tab char. A tab is considered a single character in evaluating line length. The compiler considers names which differ only in the case of letters to be identical. The underbar character "_" is considered to be alphabetic in names. Only the first 8 characters of names are significant, although reserved words are checked for exact spelling. Thus "procedur" and "procedure" are distinct, but "procedur" and "procedures" are not distinct. The 8 character names are retained throughout any linking/loading processes which may follow. However non-global procedures use compiler assigned unique names, rather than user names, thus avoiding name collisions in any following assembly and linkage operations.

3.6 Integers and Sets

Integers are currently limited to the range -32768..32767, although 32768 may not be used as a constant in a source program. Sets are limited to the range 0..127. This allows the complete ASCII character set.

3.7 Editors

The CP/m based systems are completely compatible with source files prepared by most editors (but do not use Wordstar under the "D" option, use the "N" option), and with files prepared by the various UCSD editors. The significant line termination character is the

<CR>, and <LF> is ignored. Line indentation coding is identical to that of the UCSD system, but is controlled by application programs, and thus not forced. Note that source files must not have the eighth bit set in characters, and must not contain non-printing characters outside of dle, cr, lf and tab.

CAUTION

For efficiency reasons the compiler (and many other utilities) do not check for EOF except after EOLN. Files with EOF in unterminated lines are illegal according to the various Pascal standards, and will cause run-time errors. Thus source files should always be terminated by a final end-of-line. WordStar users should especially note this, and not inject a final space.

Chapter 4

Compiler Switches & Options

Pascal-P recognizes commands placed in "pseudo-comments" of the form (*$...*) or {$...}. Options are separated by commas with no intervening spaces. Unrecognized options will generate a warning.

Compiler options never affect the sense of the compiled program, but may alter the run-time environment.

The following options are available (* marks defaults):

@nnnn (nnnn is an unsigned integer) Sets initial value for compiler generated labels. Used with separate compilations so that internal procedure names are unique to segments. Default 0. Misuse can cause unusable code. Always assign increasing values, and allow for usage in earlier modules.

B+ * (default). Assign normal file buffer space.

B- Assign no file buffer space beyond that required for file flags and access to f^. Normally used only with 8080 and P-code interpreter systems for device files which are accessed directly.

Bnnn nnn is an unsigned integer. Assign nnn units of file buffer space. Useful for special i/o drivers. Note that this option takes effect during the file type definition.

C+ * (default). Emit object code.

C- No object code. Useful for source syntax checks.

D+ * (default) Emit run-time checks.

D- No run-time checks.

E+ Listing page eject.

E- Ignored.

H+ Accept extensions which allow compilation of source for Per Brinch Hansens Solo system. Also sets the S- option, see below. In this mode the " is recognized as a comment delimiter (start and end), VAR, CONST and TYPE declarations may be intermixed and "local" global variables may be declared (invisible to earlier portions of the source file), "OR" may be used in place of "+" for set union, and characters may be defined within strings by (:nn:) where nn is the ordinal value of the character. All but "(:nn:) within strings and the "local module" VARS are normally executed with warnings in other modes.

H- * (default) Reject the above extensions. If the system was in the H+ or S- modes it is left in the S- mode.

I'filename' A second method for including other source files. This can appear with other options in a single line (which must contain the *) or } characters). This mode avoids the line number setting to a multiple of 1000 mentioned above, and will not be tracked by the XREF cross-referencer program unless the appropriate option bits are set.

L+ * (default). Generate a listing file.

L- suppress listing until an L+ command encountered.

N+ * (default). Emit source line numbers in object code.

N- No tracing line numbers.

P+ * (default). Allow use of nonstandard std procs.

P- Warnings whenever any non-standard "standard Procedure" encountered.

S+ * (default) Warnings and/or errors signalled for any use of nonstandard Pascal features, but not including use of nonstandard "standard procedures".

S- enables use of nonstandard Pascal features including definition of character constants by (:nn:), where nn is the ordinal value of the character, use of the substring construct ARRAY[VAR FOR CONST], use of the second parameter in reset/rewrite and equivalent procedures, use of "OTHERWISE" in case statements.

S'segmentname' Controls segmentation of the generated code. Must appear outside the BEGIN END; of a procedure and before the BEGIN of the main program block.

T+ Print compiler internal tables, showing variables, types, etc.

T- * (default) Suppress table printing.

W+ * (default) Truncate input source at 80 characters.

W- Truncate input source at 72 characters

Wnnn (nnn is an unsigned integer). Truncate input source at nnn characters. nnn <= 108.

X+ save option setting for future restoration. This allows modules to set options as desired, without affecting option settings for the including text.

X- restore option setting saved by previous X+. Note that only one level of storage is available. An X- without a preceding X+ restores the default settings.

Y+ Not for user use. Enables ic listing on prr file, and renders that file unusable for code generation. For compiler debugging only.

Y- * (default). Normal compiler operation.

Z+ Not for user use. Controls phase-in of new features, which may vary or be nonexistent, and may not work.

Z- * (default). Undocumented features etc. are disabled.

^+ As T+, except that pointer variables and types are traced through all nesting levels.

^- * (default) No effect unless ^+ was in effect, when effectively sets T+ option.

Chapter 5

The Translation Process

5.1 Translation Steps

Pascal source programs are translated into runable programs in two or more steps shown schematically below (The PCODE is legible text in a .TIC file):

PCD interpreters (no intrinsic procedures or segmentation):

```
   SOURCE----->PCODE----->PCD=PROGRAM
          !          !
       PASCALP    ASSMPCD
```

PCD interpreters (intrinsic procedures or segmented):

```
   SOURCE----->PCODE----->RBMFILE(s)----->PCD=PROGRAM
          !          !                !
       PASCALP    ASSMPCD          LINKER
```

8080 Native code:

```
                                    LIBRARY----\
   SOURCE----->PCODE----->ASSY------>SLRFILE------>PROGRAM
          !          !          !              !
       PASCALP    ASSMAP      SLRMAC         SLRNK
```

HP3000 (native code):

```
   SOURCE----->PCODE----->SPL----->USL----->PROGRAM
          !          !        !        !
       PASCALP    ASSMSPL    SPL    SEGMENTER
```

On File Sizes

You can expect .TIC files to be roughly the same size as the Pascal source files. For .COM generation the .MAC file will usually be about 3 to 4 times the size of the .TIC file.

5.2 The Compiler Header, showing files used

PASCALP (source, list,prr, ef, input, output) [parm]

PascalP may be operated directly by making the appropriate substitutions in the program header. The pre-defined jobs below combine compiler and assembly execution into one command, and are normally most convenient. A useful specification of source is "CON", which allows entry of options and inclusion of the main program, for example:

```
(*$n-,d-,i'yourprog.pas'*)
```

to suppress line numbers and run-time checks when compiling "yourprog.pas". This type of operation avoids any editing of source files. See the "x" option under Compiler Switches, for a mechanism to set options in specific program areas without affecting the overall option settings. Similar "stub" files may be defined for convenient compilation with various modes set.

5.3 Pre-defined Jobs

Under CPM the submit files COMPILE.JOB, PASCPCD.JOB and PASCPREP.JOB capture the complete (compilation assembly linkage) process into one command. COMPILE.JOB simplifies entry to one filename.

NOTE

"JOB" is similar to the standard CPM "SUBMIT" utility, but allows nested jobs, interactive entry, execution with any default drive assignments, specification of default parameters, uses the comma as a parameter delimiter, and allows "quoted string" parameters. It will also perform the same drive searches as the overall Pascal file system. It is an enhancement of "SUPERSUB".

5.4 Compilation Commands

Using the supplied programs and job streams simplified compilation commands are available.

5.4.1 CPM: Typical commands are:

```
A>JOB PASCPCD source list object scratch

          (where scratch and object may be identical)
or
A>JOB COMPILE source    <<with no extension>>
```

5.5 Options

A number of options have been added to control the compilation process. These options are specified by PARM values in the command. Multiple options may be specified by adding the corresponding values. (Under CPM parm is a number enclosed by "[]" outside the fileparameter section of the command line.)

PARM	Meaning
2	Generate procedure tracing code
4	Continue after compiler errors/warnings
8	Cross-compile (16 bit on 8 or 8 on 16 bit machines)

5.5.1 Under CPM

If no "8" bit is set code for an 8 bit machine (Pcd interpreter or 8080 native code) is generated. The following apply:

16	Force a relocatable PCD file. (Programs with neither intrinsics, externals, nor segmentation will normally generate an executable PCD file, which cannot be linked to external procedures.
64	For .PCD generation, over-ride segmentation commands and force generation of a single segment code file. For .COM generation, cause the intermediate assembly source file to be generated in Zilog rather than Intel mnemnonics. This increases the file size by about 10%.
128	Echo the compiler listing on the TIC (temporary intermediate code) file for subsequent listing during the assembly phase. This option greatly enlarges the TIC file size. If converted to executable PCD via ASSMPCD a complete listing can

be extracted on "codelist" file. See the job file PASCPCD.JOB, which installs all these options. Similarly, if converted to assembly source with ASSMAP, the original compilation listing appears as comments.

5.5.2 Under other systems (e.g. HP3000)

Slightly different conventions apply. Omitted from this manual.

5.5.3 Procedure Tracing

Compiled programs normally do not contain procedure tracing messages. If procedure tracing is desired use the "2" bit option, and run the program with an odd value of PARM specified. Messages of the form "ENTER procedurename" and "EXIT procedurename" will be generated automatically. At present this feature is suppressed, because most systems cannot conveniently supply sense switches for control, however custom interpreters can be supplied which enable the feature. The messages are indented to reflect the dynamic procedure level. Therefore messages from recursive procedures may disappear to the right.

5.5.4 Where options take effect

Of the above "bit" options, only the 4, 8 and 128 bits (no error job aborts, byte object machine, and echo listing on P-Code file) are directly implemented on the compiler. The remaining options are passed to the various assembly programs which generate final code.

Microsoft .REL format

While it is possible to use this format, it is not supported because a: Names are limited to 6 characters, and extremely confusing errors can result from name collisions; b: The LIB80 program is unreliable, and loses portions of large libraries, thus making maintenance of PASCLIB.REL impracticable.

SLR Systems format and librarian avoids all these problems, and is also an order of magnitude faster. Unfortunately the SLR

programs will only execute under Z80 processors (while the PascalP system can execute under 8080, 8085, and v20 processors).

Chapter 6

Compiler Error Messages

6.1 General

The compiler produces error codes, and lists them after the offending line on both output and pasclist formal files. If the L-option is in effect output to pasclist is suppressed. If the translation file (formally "ef", normally EF. under CPM) is available a list of error code translations follows. Note that the position indicated in the source line is that in which the error was detected, and that the actual error may occur earlier.

6.2 Action after errors

If any errors are found the compiler will set JCW (job control word) to the error state, and if warnings are found it will set JCW to the warn state. See the procedure SETJCW under Extensions. This is very useful when executing long compilations under the various batch mechanisms (JOB or SUBMIT under CPM). Under CP/M SETJCW to a negative value causes any submit job to be cancelled on program termination, and the warn state is ignored.

6.3 Some conditions causing errors/warnings

The compiler will always generate a warning if the standard files INPUT or OUTPUT are reset or rewritten (or the equivalent). These files are normally the user console, and cannot be reread nor can the effects of previous writes be erased.

While NIL is implemented as a predefined type, the compiler will forbid any attempts to redefine it, thus giving it the effective status of a reserved word.

In general the compiler will generate warnings where the actual meaning is unambiguous (e.g. use of a feature in the wrong mode,

attempts to reset input, etc.) and errors wherever the possibility of a mis-spelling or faulty punctuation exists.

6.4 Accessing Error Messages

Under CPM the error message file is accessed as file "EF" with a blank extension. If this file is not on the default or system disks it should be specified on the command line with the appropriate disk identifier, otherwise no translation will occur. The error message file may be modified freely (one line per error) to install other languages, clarify errors, etc.

Chapter 7

Standard Pascal Features Not Implemented

At present the following features of ANSI standard Pascal have not been implemented. All except "GOTO"s are planned for eventual inclusion.

1. Procedures/functions as parameters.
2. GOTO's leading out of procedure/function bodies.
3. Read/write for non-text files.

7.1 Packing

"PACKED" variables may or may not necessarily be packed. At present no items are packed no more than one item per byte.

The standard procedures PACK and UNPACK may be used, but may actually simply transfer variables.

PACKED and UNPACKED variables are not distinguished at present, although the parameters to PACK and UNPACK must be correctly declared. Thus standard Pascal errors in usage of such variables probably will be undetected.

7.2 Set of Char

A set of char is available, but any attempt to include graphics characters in that set {e.g. chars with ordinal values larger than 127} will result in a run-time error (if checking is enabled) or compilation errors for constants.

7.3 Separate Compilation

The compiler will handle partial source programs for separate compilation. This facility is described in the chapter on Language Extensions under "Procedure Calls". S- mode is required.

The PROGRAM statement is optional under S- mode. If it is omitted, INPUT and OUTPUT files are still available.

7.4 Files in Structured Variables

The user must give special consideration to ARRAYS OF FILE, RECORDS with file components, and pointers to FILE types or other types with FILE components. Before any programmatic reference to these items it is necessary to call the standard procedure FILEINIT(f) for each and every file component. After this the system will function normally, except that no automatic file close on exit from the block in which the file was declared will occur. Again the user must specifically execute the standard procedure CLOSE(f) for each file opened (via reset/rewrite and equivalent procedures). Under CP/m, if the file has only been used for reading (opened via RESET or EXISTS) the final close is not necessary. However inclusion of the close statement will avoid portability problems, and possible problems if program modifications are made later.

7.5 Reads of Real Variables

On input, reads of real variables do not insist on a leading digit, but will accept values starting with ".". This is deliberate, and avoids nuisance run-time errors.

Chapter 8

Runtime File Assignments

8.1 General

A number of run time errors can occur while running Pascal programs. The messages are generally accompanied by a source code line number if the N+ option was in effect on compilation. File errors are also accompanied by the file name.

Files named in the Program line of the Pascal source are external to the Pascal program. They may be defined by the run command, or may be temporary or permanent user files. Files declared within procedures are unnamed temporary files (unless opened with a second parameter in the opening RESET or REWRITE, as described below). The files "input" and "output" normally connect to the user's console.

8.2 On 8080/z80 under CPM:

The program command line is normally of the form:

```
d>RUNPCD program (fileparameters) [parm]; <inputfile >outputfile
```

(with "RUNPCD" omitted for .COM files) but is not restricted to that, i.e. use of the command line is entirely under the control of the application program. Any section after "program" may be omitted, and defaults apply. The file parameters replace the files mentioned in the Pascal PROGRAM heading. If omitted the Pascal internal filename is used. Files input and output cannot be redirected by this mechanism, but use the "Unix" flavored "<" and ">" (comesfrom and goesto) redirection commands. By default input and output connect to the users console. Parm is an integer in the range 0..32767, with a default value of 0. Use is up to the executing program, however odd values are used to cause an initial debug trap, and to enable various run-time debugging aids. Thus it is suggested that application programs rely primarily on even values.

8.2.1 CPM device files

Under CPM and other systems the following device files are normally available (and their names cannot be used for other files). Unless mentioned these are text files. Any devicefile may be used interchangably with a disk file of the same type.

CON	the system console, buffered for input or output.
KBD	the system console keyboard, without buffering or echoing of input. Correct use of this file requires either modification to CPM 2.2. or (interpreters 2.8 up) prevents use of the CNTRL-P CPM function. This avoids loss of input characters during console output. See the STATUS procedure.
RDR	The system "reader" device, line buffered.
PUN	The system "punch" device
LST	The system "list" device
AUX	Identical to RDR device with no line buffering. <lf>s are translated to <nul>. See STATUS procedure.
NUL	a null file (bit bucket). Any type.
CMD	A one line file containing the run command line.

Various other device files can exist at various installations. Examples are:

KBB	Identical to KBD, except that "reset(kbb)" causes all console i/o to be performed through interrupt driven buffers. The close operation on kbb (either specifically performed or by exit from the declaring procedure) restores the normal unbuffered drivers.
AD1..AD8	A set of 8 analog/digital converters. FILE OF real.
RS1, RS2	Direct access to RS232 i/o ports. See STATUS.

R1B, R2B As RS1 and RS2, but buffered via interrupt system. These files will not cause "waits" in the executing program, unless the buffers fill or are empty. This can be pre-checked with the STATUS function.

8.2.2 File Redirection

Under CPM file redirection is available, by substituting the desired files in the program header. Files not specified by this method default to the Pascal filename. A pair of commas can skip file redirection for any one file. Note that INPUT and OUTPUT connect to "CON" by default and cannot be redirected by this mechanism, but that the sequence

```
A>RUNPCD progfile(whatever) <inputfilename >outputfilename
```

will redirect INPUT and OUTPUT.

NOTE

The "lf" character is normally ignored on input. However input of a lf will cause the STATUS function to return a 2 bit (see below) because the hardware is physically loaded. If a get is now performed the system will flush the lf and perform the following get automatically. This can cause unexpected delays when performing direct device I/O. For this reason some device drivers are modified to translate lf into nul and the user must specifically discard it. The AUX file translates all lf's into nulls for this reason.

8.2.3 Under CPM the file search order is:

1. The default disk, if no drive specified,
2. The system disk.

and then, for code files (.PCD) to be executed only:

3. within the library file PCDS.LBR on the default disk.
4. within the library file PCDS.LBR on the system disk ONLY when no library was found on the default disk.

If a drive has been specified the search is limited to the specified drive. If the modified CCP and JOB (for submit) files are installed this search order is implemented at all levels, otherwise it is limited to Pascal program executions.

Files INPUT and OUTPUT default to "CON" (the console) under CPM. See above for run-time re-direction.

Chapter 9

Language Extensions

9.1 Standard Procedures

The following standard procedures have been added to the required Pascal set. Their use usually does not require the S- mode (See use of fname below), and most can normally be replaced (for portability) by user written procedures. {} enclose optional parameters. If P-mode is in effect all procedures not in the standard Pascal set will be flagged by warnings.

NOTE

The standard procedures required by the various Pascal Standards are not discussed here.

In the following fname may be a character string (e.g. 'fname') or a reference to a packed array of char terminated with a blank. Use of fname in calls to reset, rewrite, exists, appendto, update requires S- mode. Note that fname should always be terminated with a blank. Fname should normally begin with an alphabetic character, and contain only alpha-numeric characters. Lower case characters are automatically upshifted before use, but a string variable will not be affected.

9.1.1 File Access

At present APPENDTO is parsed, but not implemented at run-time.

```
RESET(VAR f{, fname});
REWRITE(VAR f{, fname});
```

have been extended to allow the optional 2nd. parameter. This causes attachment to the named external file.

```
EXISTS(VAR f{, fname}) : boolean;
```

is equivalent to "reset", but does not cause a run-time error if the operation fails. It returns true for success, else false.

APPENDTO(VAR f{, fname});
is equivalent to "rewrite", but opens a file for append access. The file must pre-exist, and any further writes append to that file.

UPDATE(VAR f{, fname});
is equivalent to rewrite/reset, but allows direct access to records of a fixed record size file. f may not be a text file. See REPOSITION.

RENAME(VAR f, fname) : boolean;
renames the previously opened file f to fname. Returns true if successful. Also closes the file automatically. Failure may be caused by a previously existing file named "fname", (on the same disk drive under CPM), by illegal fname, by f not being open, and by f being a device-file rather than a disk file.

PURGE(VAR f);
purges the previously opened file f. f must be a disk file.

CLOSE(VAR f);
closes the previously opened file f

FILENAME(f, fname : packed array[1..28] of char);
returns the actual system file name to fname. f must be open. Under CPM, if the file is open on other than the default drive, or a drive was specified when opened, that drive id is returned in fname. Fname is returned upshifted.

STATUS(f) : integer;
returns 0 if the file is not open, otherwise an odd value. Negative values signify some form of error. For positive values various bit positions have special significance, and are especially useful with device files:

1 = file is open;
2 = a get will function without waits;
4 = put will function without waits
8 = writeln will function without waits.

9.1.2 File read/write procedures

OVERPRINT(f, ...);
PROMPT(f, ...);
are syntatically identical to the standard writeln procedure. Overprint causes output without a following line feed, and prompt causes output without a following carriage return or line feed. In all cases the output buffer is flushed to the output device. Prompt should be used whenever a line that has not been completed by "writeln" is intended to appear on the output device (typically the console). Without this the message will probably remain within a system buffer.

READX(f, VAR) : boolean;
is similar to read for integer or real variables, but does not create a run-time error when bad input is found. It returns true when no valid input was supplied, otherwise false. Only one argument may be read, as opposed to read(f, VAR, VAR,..);

READ(f, VAR a : PACKED ARRAY[1..?] OF char);
The read procedure has been extended to allow input of strings. Input will continue until either the string is full, or eoln is encountered. In any case the final character in the string will be a nul (chr(0)), and any remaining portion of the string will be nul filled. Note that this means that the maximum length of the received string is one less than the declared string length. No automatic readln is executed, so that long input lines may be completely received by multiple reads, or flushed by readln. If eoln is true at completion, the complete line was read. The length of the input line can be discovered with the LENGTH function.

LENGTH(VAR a : PACKED ARRAY[1..?] OF char) : integer;
is effectively a special application of the SCANFOR function, with some parameters automatically supplied by the compiler. It is used to discover the length of text lines read into arrays. Note that this describes the actual length of the string, and that proper storage with the terminating marker requires one extra byte.

STRINGCP(VAR s1,s2 : PACKED ARRAY[1..?] OF char) : integer;
compares strings read by the above string read procedure, or other strings terminated by a nul (chr(0)) byte. Returns +1, 0, -1 for s1 greater, equal, or less than s2. Comparison does not include any characters past the length of the shorter string, and a string identical to a shorter string up to the lenth of the shorter string is considered larger.

REPOSITION(f, integer);
repositions file f at record (integer) for further random access. The file must have been opened with the "UPDATE" procedure, and must not be a text file. After reposition a get(f) may be performed to read the desired record, or a put(f) can write into the desired record. A sequence of gets or puts will act as if the file was sequential. To switch from get to put (or from put to get) REPOSITION must first be executed. Under CPM repositioning to a point past the end of file will extend the file and fill the new record with binary zeroes. This allows the use of "sparse" files in databases. The record size is defined by the Pascal declaration of the file.

9.1.3 Unsigned arithmetic.

These procedures operate on values stored as integers, but will not cause integer overflows, and treat all values as unsigned. The use of the type declaration "unsigned = integer;" is suggested. You should imagine the appropriate arithmetic operator inserted between the parameters.

```
UADD(u1, u2 : unsigned) : unsigned;
USUB(u1, u2 : unsigned) : unsigned;
UMULT(u1, u2 : unsigned) : unsigned;
UDIV(u1, u2 : unsigned) : unsigned;

UCMP(u1, u2 : unsigned) : integer;
```

compares two unsigned values, returning +1, 0, or -1 for u1 greater, equal, or less than u2.

9.1.4 Miscellaneous Procedures

```
SIZEOF(VAR or TYPE id) : integer;
```

a pseudo-function returning the storage requirements of the item in the units of the executing system.

```
TYPEID(expression) : typeid;
```

a pseudo-function, converts expression to a value of type typeid. Expression and typeid must occupy the identical storage space. Primarily used as the inverse of "ORD" to convert an integer to an enumerated type. Other uses are possible, but dangerous. Use of this function requires the S- mode.

```
TERMINATE;
```

terminates program execution whenever executed.

```
SETJCW(integer);
```

sets the system job control word. Causes a running job to terminate at program completion if set to a negative value. No effect in interactive processing.

```
DATER(VAR dl : PACKED ARRAY[1..15] OF char);
```

returns the current date and time formatted as:

```
yy/mm/dd hh:mm    (with a trailing blank)
```

NOTE

This meets ISO standards and collates in ascending time order. If no system timers exist the string "00/00/00 0:00 " is returned.

```
ASL(i, n) : integer;
ASR(i, n) : integer;
LSL(i, n) : integer;
LSR(i, n) : integer;
```

These functions provide various integer shifts. The arithmetic shifts may cause overflows.

```
ALLOCATE(VAR p : ^something);
```

is functionally identical to NEW, except that no heap overflow error (and attendant abort) will occur. If the allocation fails p will be set to NIL. Thus the application program can detect that a failure occurred, and take remedial action.

```
GETMEM(VAR p : ^something, size : integer);
```

is functionally identical to ALLOCATE, except that the user can specify the size of memory to be allocated (in bytes). This is UNSAFE in that no protection now exists against storing items too large for the assigned memory. S- mode is required. The assigned storage may be released with DISPOSE or RELEASE.

```
DEBUG;
```

accesses the system debugger, and is system dependant. On the HP3000 this is the "debug" subsystem. Under CPM the entire system should be executing under DDT or the equivalent, because control is transferred via memory location 038H after the state of the P-machine is displayed.

```
DELAY(seconds : integer);
```

pauses the executing program for seconds. If the system has not been customized to the clock speed (under CPM) the delay period may be in error. Other implementations (e.g. time shared) may pause a process.

```
RANDOM(VAR seed : integer) : real;
```

returns a pseudo-random number in the range 0 to less than 1. The number is depandant on the input value of seed, which should not otherwise be altered.

```
MASK(integer, integer) : integer;
```

performs a bitwise AND over the integers. No overflows can occur.

```
CRC(char, VAR integer);
```

incorporates the byte valued character in a CRC checksum, using the polynomial x^16 + x^12 + x^5 + 1. Useful for communication systems.

```
MARK(p);
```

Where p is of any pointer type. Marks the heap in the current state. The variable p should not be altered until the corresponding "release".

```
RELEASE(p);
```

Releases all items created by NEW since the corresponding MARK(p).

NOTE

Mark and release are found on many Pascal systems, but are not standard.

9.1.5 System Programming Procedures

These procedures are used in various system programs, and are generally useful. They insulate against various run-time environments.

```
MERGEREAL(hi, lo : integer) : real;
MERGEBYTES(hi, lo : integer) : integer;
SPLITREAL(r : real; VAR hi, lo : integer);
SPLITBYTES(i : integer; VAR hi, lo : integer);
```

insulate between differing object machine storage assignment order and patterns. Programs using these are portable, while use of variant records is not. (byte is used as a synonym for char). For example

```
i := mergebytes(ord(hibyte), ord(lobyte));
```

is completely machine independent.

NOTE

A "standard real" for the system is defined as the bit pattern used on the HP3000 (sign, 9 bit exponent offset by 256, 22 bit significand with an implied leading 1 bit), which is not quite identical to the IEEE standard. All real constants in PCD files are of this form, and are automatically converted to resident form at execution time. Native code files use whatever form is resident.

For systems programs format conversions are performed by:

```
STDREAL(r : real) : real; (* resident to standard *)
MYREALSTD(r : real) : real; (* std to resident *)
REAL8080(r : real) : real; (* resident to 8080 *)
MYREAL80(r : real) : real; (* 8080 to resident *)
```

In general the output of these functions is only usable for assignment or manipulation by the split/merge byte/real procedures above.

The sequence

```
r := real8080(myrealstd(standardreal));
```

will convert standard reals to 8080 reals on all machines.

```
POINTERTO(variable);
```

allows generation of a pointer to that variable. S- mode is required. Note that whenever such a pointer is assigned d- mode should be in effect, because the "pointed to" variable is not in the heap, and "invalid pointer" run-time errors will occur. This is non-standard Pascal, and should be avoided wherever possible.

```
MOVETO(VAR char, char; integer);
MOVEUP(VAR char, char; integer);
```

are NON-PORTABLE in general. These avoid all type checking, and allow mass moves of storage content between arrays. The char parameters can be supplied by integer variables, pointers, etc. as desired. No run-time checks are made. Usage is thus inherently unsafe, but provides an escape from rigid Pascal type and bounds checking. MOVETO moves the lower addressed elements first, and can be used to move arrays downwards within themselves. MOVEUP moves the higher addressed elements first, and can be used to move arrays upwards within themselves. "integer" is the number of storage elements to move, in terms of character storage units. The first parameter is the destination (thus "moveto"), and the second the source.

```
SCANFOR (char; VAR char; max : integer) : integer;
SCANWHILE (char; VAR char; max : integer) : integer;
```

are again machine dependant and NON-PORTABLE. NO CHECKING ON the VAR char is performed. Thus this may be supplied by an array referance, or by a referance to an item within an array of char., or any other variable. It is treated as a pointer to element 1 of an array [1..max] of char if max is positive, and as a pointer to element -1 of the array [-max..-1] if max is negative. 0 is returned if the searched-for element is not found (or only found for scanwhile), else the index (measured as above) of the searched element. Note that negative values of max cause backward searches, and return negative, or 0, values. Scanwhile can be considered a search for an element NOT equal to char. No storage is ever altered by the search, which simply returns information.

9.1.6 Special procedures

These allow for precise control of CPM systems, and are non-portable. They should therefore normally be avoided.

```
PEEK(n) : integer;
GETPORT(n) : integer;
```

are functions returning 0..255

```
POKE(n, i);
PUTPORT(n, i);
```

are procedures storing 0..255.

```
INTERRUPTS(onoff : boolean);
```

controls the interrupt system.

```
SYSCALL(fnct : integer; parm : integer) : integer;
IOCALL(entryno : integer; data : integer) : integer;
```

allow for direct connection to CPM services. The parm and data parameters are not checked, so that any type which fits in a single 16 bit word may be used. In particular, to satisfy CPM requirements, these may be pointers generated by the POINTERTO standard procedure (s- mode required). With these procedures interface procedures to the system may be generated, thus isolating system dependant features from the application. Note that no protection against misuse exists.

9.1.7 Super Special Procedures

These are available when suitable connectors have been installed, and are thus not generally portable. However standard connector locations exist in the interpreters and run-time packages. See the file CPMLINK.MAC.

```
CPUTIME(VAR t : ARRAY[0..1] OF 0..maxint);
TIMECLK(VAR t : ARRAY[0..1] OF 0..maxint);
```

return (timeclk) time of day in millisecs after midnight, or (cputime) central processor time used. Under CPM the procedures are identical. ARRAY[0] is the less significant part.

```
TIMESET(hour, min : integer);
DATESET(VAR d : array[0..2] OF integer);
DATEGET(VAR d : array[0..2] OF integer);
```

provide for Pascal program control of system timers. On the HP3000 timeset and dateset are illegal (reserved for system). ARRAY[0] is day, ARRAY[1] is month, ARRAY[2] is year. By convention dateset to an array of zeroes stops the time of day clock.

```
STARTINTERVALTIMER(interval : integer; where : ^boolean);
STOPTIMER;
```

provide for timed input systems. Startintervaltimer causes the boolean "where" to be unconditionally set to TRUE at "interval" periods. This can be used as a flag to acquire a value from A/D converter files, etc. Stoptimer disables this. Not available on HP3000.

```
STARTPROFILER(interval : integer; where : ^storage);
```

initializes the profiler subsystem, and allows measurement of program dynamic execution. The main program must include PROFILER.INC file, declare the auxiliary constants, and call the initializing, stopping, and dumping procedures. The utility program "PLOTPROF" can then create an execution time histogram. This subsystem provides information to guide program optimization, and can avoid wasting effort on pointless optimizations. Note that the

profiler, at present, is incompatible with STARTINTERVALTIMER, since the identical timer hardware is used.

9.2 File System

The file system contains provisions for files as components of structured types, including pointers to files. Use of files in such types requires specific user use of the standard procedure FILEINIT(f) to initialize the file control blocks before any other use of the file is made, and of the standard procedure CLOSE(f) to close the file. The compiler performs the equivalent procedures on directly declared files (e.g. f : text) automatically, but does not detect the presence of the file types within structured variables.

The file system of standard Pascal has been extended to allow direct access and attachment to files. Additional carriage control procedures have been added.

A second parameter may be added to RESET and REWRITE. This parameter is a character string containing the external file name in internal format. The last character of this string must be a blank.

Execution of "write(f,'string':0)" and equivalent statements is a null operation. This is non-standard Pascal, and is never flagged. If string is a single character (ex. write(f,'a':0)) an error occurs. The maximum string length or field size is 255, however multiple writes may create any length of line.

REPOSITION (file, integer) will reposition a file to the indicated record. A subsequent GET will begin at this record, while a subsequent PUT will write to this record. Note that REPOSITION will not modify the EOF flag; it is ignored in determining the validity of the next GET or PUT. Future changes are possible.

Under CPM all disk files are mapped into CPM standard formats and packed into successive 128 byte file records. File items may cross sector and extent boundaries.

Two new formatting procedures have been added. OVERPRINT (file) writes the buffer without advancing the line. The next line will print on top of this one. The procedure PROMPT (file) writes the current line without repositioning the carriage. This allows a subsequent READLN to read from the same line as the output message. Both OVERPRINT and PROMPT deal with the current line (buffer) contents. Like PAGE, the file parameter is optional. Note that lines output by the PROMPT procedure will not have trailing blanks removed.

STATUS(file) returns an Integer value. It is zero if the file has not yet been used, and odd if the file is open. It returns a

negative value for errors (device and system dependant. STATUS can be used to determine whether, for example, a "rewrite" was successful.

NOTE

a rewrite failure always causes an error message, however the system normally allows up to 5 non-fatal run-time errors to occur before aborting.

9.3 Lazy I/O

The "lazyio" system is implemented. This allows normal use of interactive files (e.g. CON) while meeting the Pascal standards for reset, readln, etc. These normally perform logical access to the following character of the file, which is then available for look ahead via eof, eoln, f^ references. The system postpones the physical input of that character/condition until it is actually referenced, thus allowing natural use of prompting messages, etc. While not yet provided, the implementation is such as to allow implementation of a single level UNGET(f) procedure (for textfiles only).

Note that some programming care must be taken to avoid referring to eof, eoln, f^ until such prompts have been made.

Thus any text file may be routed to/from interactive devices (e.g. console), with no special programming considerations.

9.4 Procedure Calls

The procedure call syntax has been extended to allow reference to separately compiled programs and to procedures written in other languages. The word "FORWARD" is replaced by "EXTERNAL" for separately compiled procedures, and by "INTRINSIC" for procedures in other languages.

9.4.1 Separate Compilation

Separate compilation is achieved by removing the main program but leaving the trailing ".". Note that the ";" still terminates the final procedure. S- mode is required. All procedures defined at the outermost level may be referenced in another compilation. Such separately compiled procedures may refer to globals defined within the source code, but it is up to the user to ensure that such globals are identical with those used in other modules. The $include facility (or {$i'name'} pseudo-comment) is useful for this purpose. The resultant code files will be RBM (relocatable binary module) files under CPM. RBM files can be linked with "LINKER" under CPM. See "Program segmentation".

9.4.2 Procedure Parameters

Note that while access to programs written in languages other than Pascal is permitted by using INTRINSIC, it is up to the user to create compatible parameters.

Procedure parameters are stacked in the order declared in the procedure heading. Value parameters are completely evaluated, and may be of any size. They may be treated as initialized variables, but may not be used to control FOR loops. A function value is treated as a parameter preceding all other paramaters, and is not removed on function exit. The user should NOT assume a default functional value, but should always explicitly set it.

9.5 Case Default

The syntax of the CASE statement has been extended to allow a default action to be taken if the expression value is not represented in the list of labelled statements using an "OTHERWISE" clause at the termination of the CASE statement (as defined in the draft ISO standard) This extension requires S- compilation mode. In the absence of this statement any execution of a case statement with an undefined case variable will cause a run-time error (with or without runtime checks enabled).

9.6 Complex Comparisons

Structured variables may be compared for (in)equality, as in "IF array1 = array2", but not for other relations. This is an extension to standard Pascal, and at present is not detected by the S+ compiler option. For future compatibility such use should be bracketed by the S- .... S+ options in source code to prevent generation of warnings. Note that "array1 := array2" is legal Pascal, enforcing equality.

9.7 Strings

An ISO standard compatible string facility has been included. See the discussion of READ, LENGTH, and STRINGCP procedures above. The design is such that programs using these extension may be ported to any standard Pascal system by writing appropriate procedures. However, such replacement procedures will have the normal strong typeing, rather than be universal.

9.8 Substrings

A limited substring facility has been provided. A modified subscript specifies the initial index and length within a string. In keeping with normal Pascal philosophy, the substring length must be a compile-time constant. A[I FOR 10] specifies 10 elements of array A starting at element I. The substring variable may be used on either side of an assignment, or as a procedure parameter. The same notation may be used for arrays of any type, but no further subscripts or field selections may be applied. This notation is portable, but more restrictive that the MOVETO and MOVEUP procedures noted above. S- mode is required.

Chapter 10

Program Segmentation

10.1 Overview

Program segmentation is completely independant of program structure, and is controlled solely by the placement of (*$s'segname'*) compiler commands. Any procedure, whether nested or global, can be placed in any segment. All segmentation may be performed after a program is operational. The source text never need be distorted to allow segmentation.

Under CPM and the interpreter system no final decision on segmented operation need be taken until after compilation. The output of the compiler is a TIC (temporary intermediate code) file which contains ".SEG" pseudo codes, created from the s'segname' commands.

If this is converted into executable code with ASSMPCD no segmentation will be present in the final file. This is often useful for debugging. The output from ASSMPCD, however, will be in relocatable form, and must be passed through LINKER to create an executable file (This may be prevented by use of [parm=64] when running ASSMPCD). Whenever a ".SEG" pseudo-op is encountered ASSMPCD assumes that a segmented file is being generated, and creates such a linkable object file.

Programs created from separately compiled modules will normally be segmented and use the techniques described in this chapter. The modules may themselves be segmented. See the chapter on Separate Compilation.

10.2 PCD segmentation (for interpreter execution)

At present, to implement segmentation the "TIC" file is split into multiple single segment files with the utility program SPLITTIC. Splittic will create a file for each named segment in the input code, with the extension .TIC, and using the segment name truncated to 8 characters (or to 7 characters with a user specified prefix character). Each of these segment files must now be converted to a

RBM (relocatable binary module) with ASSMPCD, just as a single segment program. The results are then combined into a single program file by LINKER. The first file linked must be either "SEGS15.RBM" or "SEGS31.RBM", which contains the outline of a segment table for a 15 or 31 segment codefile.

10.2.1 Splitting the TIC file

Execute:

```
B>runpcd splittic (ticfilename)
```

or

```
B>splittic (ticfilename)
```

and select a suitable prefix character when requested (a CTL-Z reply, or redirecting input from nul with "<nul" will omit the prefix). The split files will occupy at least as much space as the original. At completion the original TIC file may be discarded. The following assumes the selection of a prefix letter "X" (must be upper case) and that segments are named "segfile". SPLITTIC reports on the actual filenames created during execution.

10.2.2 Assembly

For each output file generated execute (omit "runpcd " if using the .COM versions) :

```
B>runpcd assmpcd (xsegfile.tic,xsegfile.rbm)
```

10.2.3 Linking

Execute linker as follows:

```
B>runpcd linker (objfile.pcd, loadmap)
```

replacing objfile and loadmap with appropriate file names. Loadmap may be designated con, lst, or a diskfile. The information on it will be used in the next step, and therefore it should normally not be routed to the console.

Reply to the first "filename" prompt with "SEGS15.RBM", and to the following prompts with the names of the just assembled segments. Terminate the entries with an empty line. The object file prompt is a last chance to change your mind on the output file name. The reply to the "loadpoint" prompt should be "0", and carriage returns

can be entered for all other prompts. LINKER will scan the files, announce that .CODSIZE, .DATASIZ, .SEGS., .WHERE01 through .WHERE15 are undefined, and ask whether more modules are available. Reply "n". LINKER will now generate the output file and the loadmap. All intersegment procedure linkages have now been resolved.

NOTE

For programs using more than 15 segments replace SEGS15.RBM with SEGS31.RBM.

10.2.4 Installing the segment map

The user must patch into the final file the addresses of the segments in the code file, which are the loadpoint addresses of the segments in the LINKER output loadmap divided by 128. These are CPM record numbers in the code file, and are the various ".WHEREnn" undefined in the preceding step. The addresses are installed at locations (segmentnum*8) and (segmentnum*8+1), high byte first. Program "DISKEDIT" can be used for this. The two byte value at locations 0 and 1 (most significant byte first) must be set to the negative count of segments present. This is the undefined ".SEGS" in the previous step. At present no more than 31 segments can be used, and this section generally assumes a maximum of 15. If errors are made in calculating these values the program will not load. Any further operations may be performed by the "TUNE" utility program.

10.2.5 Initial memory allocation

The value at locations 4 and 5 of the codefile must be patched to specify the total codespace allocated. This is the undefined ".CODSIZE" from the linking step. (.DATASIZ is unused). The run time system will then automatically swap code segments in and out of main memory on demand, and select segments for discard based on a least recently used algorithm. No more than the specified code space will be used for the program code. A system that buffers disk tracks will greatly enhance performance.

The critical quantity to be selected is the codespace above. If this is too small the system will thrash, while if too large sufficient execution time data space will not be available. It must be at least as large as the largest single segment in the codefile, or the program can never be loaded. Normally this space is selected sufficiently large to hold the main execution portions of the program without any initialization or rarely used error handling segments. This will automatically swap such segments out of main memory, and have negligible performance effects.

The user can also control the time span used by the memory management system by altering byte 2 of the code file (range 2 to 255). This measures time in terms of inter-segment transfers performed.

NOTE

> Since PCD files are quite compact, only the rare program will require segmentation. Very large program systems can be written as a collection of procedures, pass data through a global area, and be supervised and co-ordinated by a small outer block using this segmentation scheme. There will be no necessity for duplicated code within sections of such a system if suitably segmented.

10.3 Prestored alternative settings

By convention, alternative settings for the memory management period and codespace values are stored in locations (8*i+5) and (8*i+6) to (8*i+7) for i := 1 to numberofsegments. If these values are present the utility program "TUNE" can be quickly used to alter the memory-requirements/performance balance of the program. TUNE may also be used to initially install these values.

The appearance of such a segment map may be examined by:

```
A>diskedit pascalp.pcd
```

followed by the command

```
r0
```

terminate the program by "q". Use a copy of Pascalp in case of error.

NOTE

> Other areas within the segment map are used by the memory manager. Therefore at run-time the map contents will be different.
>
> Numerical input to diskedit is normally decimal, but may be prefixed by "#" for hexadecimal input. In some cases characters enclosed by single quotes may also be used. Diskedit provides no input line editing, and was built as a crude tool.

After segment addresses and segment count have been entered via diskedit, further alterations may be performed by:

```
A>RUNPCD tune (codefilename.ext)
```

using the unprompted "I" and "D" commands (? gets a prompt). This is much safer than diskedit.

10.4 The structure of a code file/segment

All executable PCD files begin with a single descriptive integer, which is stored high byte first, in 2's complement notation.

If this integer is positive the PCD file consists of a single segment, and the integer is the length of that segment. The segment proper is enclosed by the integer, an unused word, and (at the end) a single byte describing the number of transfer vectors. These five (total) bytes are not included in the length descriptor.

10.4.1 The segment map.

If this integer is negative it must be in the range -1 to -127 (only -31 for now), and describes the number of segments in the PCD file. The following is based on a 15 segment limitation. The initial 128 bytes of the file are a segment map, consisting of 16 8-byte entries which can be described as:

```
   ARRAY[0..15] OF RECORD
     CASE n OF
0: (   segcount     : -1..-15); (* hi byte first integer *)
       tunevalue    : byte;
       entryseg     : byte; (* id of outer block *)
       codespace    : hifirstinteger;
       dataspace    : hifirstinteger; (* not used yet *) );
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15:
   (   fileaddress  : hifirstinteger; (* sector in PCD file *)
       initflags    : byte;
       workarea     : ARRAY[1..5] OF byte;)
```

By convention bytes 3, 4, 5 of workarea are used to store optional values for tunevalue and codespace. Eventually initflags will be able to describe segments not to be swapped out and machine language segments. In operation this byte also holds segment usage history.

The remainder of the PCD file holds segments organized as the single segment PCD file above, but always filling complete sectors. Thus the fileaddress value can address 128 * 65536 bytes of code. This is an absolute limitation on the total size of a code file. Similarly no segment can reference more than 127 procedures (sum of internal and external), and no segment can exceed 32K.

Data segments are entirely separate, and are addressed via base registers. The heap and stack areas must be contiguous, but can range up to 64K total size.

10.4.2 The transfer vector.

The last byte of each segment (located by the segment length descriptor) holds a count of transfer points. 0 means that only the main program entry is present. This final byte is preceded by n (where n is the contents of the count byte + 1) HIBYTEFIRST 16 bit transfer vectors. They may be of two types (and a code listing will show the type - See appendix C):

10.4.2.1 PCD transfers.

These vectors are used by the CUP (call user procedure) (and its variants, CLP and CGP call local and call global) P-codes. If the value is negative, it is a self-relative pointer to the code within the segment. If the value is positive, it contains two one byte fields, holding the segment number and entry number within the external segment.

10.4.2.2 Intrinsic transfers.

These vectors are used solely by the CIP (call intrinsic procedure) P-code. It contains the absolute machine address of an external procedure. Note that the external procedure is called with NO stack marker, but with the appropriate parameters (and possible function return value space) allocated on the stack. The intrinsic is responsible for clean-up.

The zeroth transfer is only used for entry to the main program. In other segments it will normally point to a CSP STP (halt) instruction.

Chapter 11

FILTERS and CONVENTIONS

11.1 General

Various "filter" programs are provided. These have the common characteristic that they use only "input" and "output" files, and perform some sort of translation of the input file. "Input" and "output" are always textfiles. Filters have no extraneous files over which to prompt the user, and no added verbiage is generally desired on the output, thus they are unable to prompt the user.

The programs are generally executed by using the i/o redirection facilities ">" and "<" (goes_to and comes_from respectively). A minor exception is "TYPETEXT", as detailed below. COPYCOLS also provides for additional parameters in the command line.

11.2 LINE TERMINATORS

CPM textfiles universally use the convention that lines are terminated by a <cr><lf> pair. Some systems generate files that omit the <lf>. Any such files are legible to Pascal programs, and after passing through these filters will have the conventional line terminators inserted.

WARNING:

> The ISO and ANSI Pascal standards specify, and for good reason, that an EOF must immediately follow an EOLN in a non-empty textfile, i.e. all lines must be terminated by EOLN (normally a <ret> under CPM). Files created by other systems may not respect this. WordStar is a notable offender, where the user must specifically terminate the last line in a file and not add any invisible blanks at the end. When these programs read such files they will

probably abort with a "READ PAST EOF" error. The cure is to be sure to enter the final carriage return. Programs can be written to correct this, but may require an enormous overhead to check EOF and decide what to do after every input character.

11.3 INDENTATION CODING

A convention widely used in the system is the "indentation_code", patterned after the coding used in the UCSD system. This expresses an indentation, for a text line, as the character 010h, or control-p, or 16 decimal, or dle (all synonyms) followed by the printing character (space + number_of_spaces_to_indent). This is only a convention, rather than forced as in the UCSD system. All system programs respect this (PASCALP, PAGER, XREF, REFRENCE, COMPARE, COPYCOLS, TYPETEXT, EXPAND) and perform the output expansion. This avoids much useless and time-consuming scanning of blanks in text files. Program COMPRESS creates the indentation code from plain text. Of the above programs, COPYCOLS, TYPETEXT, EXPAND, COMPRESS and REFRENCE are all filters for textfiles. Note that COPYTEXT simply passes on any indentation code.

11.4 NUMBERED LINES

A further convention used in this system is that textfile lines which have an initial string of 8 digits (all must be digits) are numbered, and that the first 5 digits are the line number. The remaining digits express a fractional line number. Programs that process source text, such as PASCALP, XREF, REFRENCE, COMPARE, PAGER, XREFASSM respect this convention.

11.5 THE PIP [b] (buffered) OPTION:

To co-operate with PIP, several programs provide for the PIP [b] buffered option. If the programs are executed with a non-zero value of parm they will, at intervals, emit a dc3 character and then pause for approximately parm seconds. At file completion they will emit a cntrl-z character. These cause PIP to flush buffers and terminate, and enable simple-minded information transfers in printing characters. Programs that include this feature are: COPYTEXT,

STRIPNUM (both filters), BINHEX and RBMTOHEX (documented elsewhere). The time delays assume execution via RUNPCD on a 2 Mhz machine (unless the CPMLINK portion has been customized), or on the HP3000 (or other machines with process blocking). An even value of parm should normally be specified to avoid an initial debug trap.

WARNING:

some earlier versions of these programs exist without the delay feature. They should be replaced when found.

11.6 MINI-MANUALS

11.6.1 COPYCOLS (left_column, right_column) <infile >outfile

copies left_column through right_column from infile to outfile. Left_column defaults to 1, and right_column to 132 (the maximum). Indentation codes are expanded before measuring column position. Lines longer that 132 characters cannot be processed.

11.6.2 COPYTEXT <infile >outfile [optional_parm]

simply copies all input to the output. No indentation codes are expanded. There is no limit to line length. The PIP [b] option is supported (see above).

11.6.3 TYPETEXT <infile [>outfile may be specified, but is useless]

copies infile to "output", normally the user console, and halts every 20 lines until a <ret> is entered. Indentation codes are expanded, and lines longer that 80 characters are wrapped into multiple lines. TYPETEXT actually uses another file (KBD) for non-echoing interaction with the operator. This interaction reduces its portability. Some versions may wrap at 79 characters to allow for terminals that wrap at 80 characters without detecting that no more characters follow. An EndofFile (control-z under CPM) when <ret> is expected ends the pauses permanently. A control-c at this point exits the program. [parm] may optionally be specified, and

alters the default pagesize of 20. Use an even number to avoid an initial debug trap.

11.6.4 EXPAND <infile >outfile

is logically equivalent to COPYTEXT, but expands indentation codes. There are no provisions for delays etc.

11.6.5 COMPRESS <infile >outfile

is the inverse of EXPAND. Files occupy less disk space, and require less transmission time after this, but are still perfectly legible to compilers etc. They remain comprehensible, but not pretty, to human viewers.

11.6.6 STRIPNUM <infile >outfile

removes any line numbers from infile. See above for the definition of a line number. The PIP [b] option is supported, see above.

11.6.7 ADDNUMS <infile >outfile [optional_parm]

adds sequential line numbers to a textfile. If a [parm] is specified it sets the initial line number -1, i.e. the first line will be numbered one higher. This is consistent with the default zero value of parm.

11.6.8 WSTOTEXT <infile >outfile [optional_parm]

is a preliminary program. It converts the special characters in a WordStar document file to the normal Ascii set, removing any un-used "soft" hyphens, suppressing trailing blanks on lines, etc. In addition any "dot commands" (i.e. lines beginning with a "."), and pagination are deleted. EOF is correctly set in the output file (see the WARNING below). [parm], if specified, serves the same function as the WordStar ".po" dot command, and inserts blank spaces at the left of each line. This document was prepared on WordStar and passed through WSTOTEXT.

11.6.9 REFRENCE <infile >outfile

is a highly specialized filter for showing the structure of Pascal source programs, and respects indentation codes. It also follows all $include filename statements, and, if parm=100 was specified, follows all (*$i'filename'*) include commands. See the Pascal-P documentation for the specific syntax. REFRENCE is a customization of a program by Arthur Sale.

All these filters, with the exception of REFRENCE, are small and simple programs. P-code versions of these filters are all executed by the prefix "runpcd ". Native code versions are executed as shown.

Chapter 12

RBM files and HEX files

12.1 General

The fundamental relocatable binary modules are known as RBM files. They have been designed to be compatible with the INTEL hex standard, and to be machine independant, in that the only assumption made is that the host machine addresses in units of bytes. Modules cannot exceed 32768 bytes in length, but no restriction on final code size is made, except that external values must be expressable in two bytes for arithmetic to be performed. Limitations on name length have been expressly avoided, although present software truncates all names to 8 characters.

The structure of RBM files is best explained by first explaining HEX files.

A HEX file is made up entirely of printing characters, and can be tranmitted and manipulated as a text file. A record, in a HEX file, begins with the ":" character, has several fields containing only the hexadecimal characters '0' through '9' and 'A' through 'F'. The fields are as follows:

colon	The character ":", beginning a record.
length	2 hex characters, describing a value in the range 0 to 255 only. This value specifies the length of the remainder of the record.
address	4 hex characters, describing an address in the range 0 to 65535 (under some circumstances this is considered a signed integer in the range -32768 to 32767).
type	2 hex characters, describing a value in the range 0 to 255 only. The interpretation of this value is central to the use of RBM files.

data (2 times length) hex characters. Interpretation varies with use.

checksum 2 hex characters, such that when each character pair since the ":" is considered as a number in the range (0..255), the sum MODULO 256 will be zero.

anything except a ":" may follow the checksum, and may be used for formatting, comments, etc. It is always ignored.

Note that every data item, after the initial colon, can be expressed as a one byte value, thus reducing storage requirements, transmission time, etc. by at least a factor of 2.

The original INTEL standard defined record types 0 and 1. Type 0 is an absolute load code record, in which "address" describes the machine address to be loaded with the first byte in the data field. Type 1 is an end-of-file record, with length always zero, and address describing an address to which execution control is to be transferred. Type 0, when length is zero, is treated in the same manner. By convention, a transfer address of zero in these end-of-file records signifies that no control transfer is to be made.

12.2 RBM files

RBM records are exact images of HEX records, except that the leading colon and trailing checksum have been discarded, and that the information is presented as 8 bit bytes in place of pairs of hexadecimal characters. No trailing "anything" field is permitted, and a new record begins immediately after the previous record ends. Verification of storage is left up to the storage system on which the files reside (typically CRC checksums over the storage blocks). Thus RBM records can be discussed in exactly the same terms as HEX records, and conversion between the systems is easy.

12.2.1 RBM record types

To preserve compatibility RBM files retain the original INTEL definitions of record types 0 and 1, and add further types starting at type 128 (080 hex). In all cases the length byte describes the number of bytes in the data field, which may be zero. Additional types are:

128 Relocatable data record. Exactly analogous to the absolute data record (0), but the address field describes the location with respect to the base of the current module.

129 End module record. Address and length are zero.

130 Relocatable Code Module Header. The relocatable records which follow are to be placed in a code segment. The address field describes the total length of the following code. The data field, (which may be empty) may hold a name for the segment of up to 60 characters.

131 Entrypoint descriptor. Address is a value relative to the base of the current segment. Data holds the name of the entrypoint, in Ascii characters.

132 Absolute entry. Address is an absolute value, which is to be used to resolve any referances to the name in the data field (again in Ascii characters).

133 External referance. Address is an index value used by the linkage records which follow. The data field holds a name, in Ascii, whose actual value is described in some other module. Note that the index in the address field will never be less than 2, because indices 0 and 1 are reserved to describe code and data module relative relocation.

134 Data module header. As type 130, except that the following data records are to be placed in the data segment.

135 Alignment Record. Must only occur immediately after a module header record. Causes the module to adjust its location so that the absolute location, modulo the address field, is zero. The data field is unused.

136 Pcd module entry point. At linkage time this value must have the high order 8 bits set to the module number in which it occured, in the range 1 to 127 (only 31 at present). The data field contains the entry name. These records are used to describe PCD linkages in terms of segment/entrynumber pairs. The occurance of such an entry point causes LINKER to assume that the output is to be a PCD program.

137 Equate names. Not presently implemented. Address field is unused. The data field contains two Ascii names separated by the "=" character, as in "name1=name2". Causes all referances to name1 to be resolved as referances to name2. The data field should not exceed 60 characters, thus effectively limiting names to less than 30 characters.

138

thru Reserved, not presently assigned.

143

Types 144 through 159 are reserved as linkage records, in which the type modulo 16 is used as an operator. The address field (except for types 152 through 154 below) contains an index, which refers to an external referance record active within this module (or 0 or 1 to specify the current code or data segment bases), and the data field contains a list of 16 bit addresses, in high byte first format, specifying a location relative to the current module beginning which is to be adjusted. All operators discard any carrys and borrows, and thus never cause arithmetic overflows. The operators currently assigned are:

144 Add lobytefirst words. The value of external referance is added to the two bytes of the module.

145 Subtract lobytefirst. The value of external referance is subtracted from the two bytes of the module.

146 Add byte. The value of external referance, low order 8 bits only, is added to the byte of the module.

147 Subtract byte. The value of external referance, low order 8 bits only, is subtracted from the byte of the module.

148 Add high byte. The value of external referance, high order 8 bits only, is added to the byte of the module.

149 Subtract high byte. The value of external referance, high order 8 bits, is subtracted from the byte of the module.

150 Add hibytefirst words. Similar to type 144, but the module contents is treated as a high byte first integer. Note that this does not affect the value of the external.

151 Subtract hibytefirst words. As 150, but external referance is subtracted from the module content.

Types 152 through 154 are anonymous linkage records. Address contains the value of the "external", rather than an index to it's entry record. These records permit construction of one pass assemblers and codegenerators, by postponing "fixup" operations to linkage time. Since modules are in their most compact form at linkage time, and since general code generation always requires a two pass algorithm at some point, the linkage step is the most efficient point at which to implement the second pass.

152 Add lowbytefirst fixup linkage.

153 Add hibytefirst fixup linkage.

154 Add byte (8 bits only) fixup linkage.

155

thru Unassigned operators.

159

160

thru Reserved for future use.

191

192 up Available for system dependant special operations which cannot be handled by the existing types. No reliance on portablity should be made when these types are used.

12.3 Utility Programs

The utility programs HEXTORBM and RBMTOHEX perform conversions between RBM and HEX files, and are principally used to transfer RBM files over transmission links (e.g. RS232 lines). RBM files are generated by assemblers, compilers, etc, and linked into executable

programs by LINKER. LINKER operation is documented separately. RBMLOAD converts RBM files which contain only absolute loader records, into program files. HEXLOAD is the equivalent for HEX files, and similar to the CPM standard program LOAD. However HEXLOAD and RBMLOAD use the file redirection systems, and are thus much more flexible. RBMTOHEX also has provisions for co-operation with the PIP [b] option. See the manual on FILTERS.

The typical execution command is

```
B>runpcd hextorbm(hexfile, rbmfile)
```

or

```
B>runpcd rbmtohex(rbmfile, hexfile); [parm]
```

For example, to transfer a rbm file named MYFILE.RBM over an RS232 link, such as a modem, on the PUN device, with 10 second delays for buffer flushing by the receiver, enter:

```
B>runpcd rbmtohex(myfile.rbm, pun); [10]
```

If no files can be found for the input files (hexfile or rbmfile) the programs will prompt for their names, and request confirmation before purging any previous versions of the destination file. RBMLOAD and HEXLOAD act in the identical manner.

BINHEX (binfile, hexfile) can be used to convert binary files to absolute load hex files. By default the origin is set at 0100h. The output may be converted back to a binary file by HEXLOAD. [parm] may be used to co-operate with PIP operating with the [b] option. See "FILTERS".

See the separate documentation on LINKER and its companion SCANRBMS.

Chapter 13

LINKER v1.1.4

13.1 GENERAL

LINKER (and the companion program SCANRBMS, documented separately) scans relocatable binary module files and forms an absolute load module. The command structure is flexible, and designed to permit either interactive operation or complete control in the command line, using the standard file redirection facilities, together with the "indirect" and library files detailed below. The "RBM" (See RBMFILES documentation) format code files are machine independant, but do expect a byte addressing system with addresses up to 65535. Byte and word externals/relocation, with either high byte or low byte first word order, separation of data areas from code areas, module alignment, segmented .PCD files, and "fixups" from one pass code generators are catered for. Name length is controlled by a compile time constant, and is presently set to 8 characters.

File names are accepted up to an empty line. Any filename preceded by "@" specifies indirect access, i.e. that file contains a list of files, which may in turn use indirect access.

In interactive (normal) mode, following the empty filename prompts are made for the output file name, code and data load points. After the input has then been scanned, if any external names remain undefined, the operator is prompted for further module names. A "no" reply here causes any such undefined names to be evaluated as zero.

A file name beginning with "-" specifies a library file. The default extension is ".LBR". The name "-" alone specifies the default library file, which is "RBMS.LBR". Note that the "-" is not part of the filename, but must directly precede it without any intervening spaces. The default library file is automatically selected unless a "-nul" (or other empty or non-existant file) file name is specified. Note that nul is the system defined bit-bucket, and always exists.

13.2 SEARCH ORDER:

Files and modules are searched in the following order (unless a specific drive has been specified, when the search is limited to a file on that drive):

1: On the default drive.
2: On the system drive.
3: If a module (not a library) is not yet found, it is searched in the current library. When the library was selected it used the above search pattern.

If a file is found on other than the default drive, it's name is revised to show the drive. If a module is found as a file it's name is upshifted. If found in a library the name is unchanged.

Thus files, modules, indirect files, may exist on up to two drives, and modules and indirect files may be found as files or as modules. This allows an existing library module to be re- placed by creating a file with the same name, without altering the library in any way. If all is well the library may then be permanently altered with the public domain LU (library utility) program and the replacement module file removed.

13.3 BATCH OPERATION:

A numerical input (hex with a leading digit) to the "filename" prompt is a specification of the load point. A second numerical input specifies the data load point. Any such numerical input causes the system to operate in the batch mode, and no further prompts are output. In particular no opportunity then exists to resolve undefined labels by adding further modules. Similarly any end of file on "INPUT" (or the file from which it has been redirected) causes operation in the batch mode.

13.4 CONNECTION TO PREVIOUSLY LINKED MODULES:

Any file name whose group/extension begins with ".CON" is considered a connector file. Any such file must contain only absolute entry points, (NOT CHECKED), and is used only to resolve undefined external referances at that point in the loading process. A connector file can be created from the loaded module by specifying

file "cnct" in the run command with PARM=2. Such a "cnct" file is in RBM format, specifying absolute values.

13.5 RUN-TIME OPTIONS:

Several run-time options may be selected by numerical "parm" values, specified within "[]" on the command line. For multiple options use the sum of the values.

Value	Effect
2	Generate a "cnct" file with a listing of all entry points. See above.
4	Generate an output listing (on "loadmap") of the complete symbol table.
10	Generate output code in "RBM" absolute loader format, rather than as a "COM" format file. This format is forced if initialized data segments exist in the input modules, since a continuous output binary image cannot be generated.

13.6 MACHINE DEPANDANCIES:

On word addressing machines where an integer occupies only one unit of storage (e.g. HP3000) the sense of the parm=4 and parm=10 bits is reversed. Such output is normally used to down-load other machines, and a binary file is useless. Time and disk space to generate complete symbol table listings are less critical, and the table is normally examined later by listing it. In addition, on such machines no libraries are presently used, nor do drive specifiers or searches apply.

13.7 EXAMPLE:

To link and relocate modules "a.y" and "b.y" to origin 0100h on codefile "ab.com"

```
B>linker (ab.com, con)
LINKER (objfile,  loadmap,  cnct, input, output); Ver. 1.1.4
```

```
filename >a.y
filename >b.y
filename >
Loadfile (code)   (default "objfile") = ?
Code address (default 100H)       hex = ?
Data address (default after code) hex = ?

<<output showing modules loaded and starting addresses>>

<<output showing the final loadmap, with data relocated>>
<<this output is routed to formal file "loadmap", and  >>
<<may contain a symbol table.  See parm=4 above        >>
```

Appendix A

Validation Suite Results (Preliminary 3.0.79)

Pascal Processor Identification
- Machines : 8080 under CPM 2.2, Interpreter 2.2.6
  HP3000 under MPE HP32002C.G0.C3
- Compiler : Pascal-P V3.0.79 (Revised to 3.1.1)
- Level : 0
- Date : 8 Mar. 1983
- Tests by : C.B.Falconer
- Test Version : 3.1

NOTE

system maxset is 127. Tests 6.7.1-9 & 6.7.2.4-6 modified accordingly.

Conformance Tests:	HP3000	8080 CPM
Number of tests passed =	166	159
Number of tests failed =	13	20

Details of failed tests:

Six primary reasons (apply to both systems) :

1. functional and procedural parameters are not implemented. Affects tests:
   6.6.3.1-4 6.6.3.4-1 6.6.3.4-2
   6.6.3.5-1

2. Variables are identified by their initial 8 chars. only. Affects:
   6.1.3-2 6.4.3.5-11

3. (* *) and {} comments are separate, and one type may be nested within the other, although comments of any single type may not be nested. Affects:
   6.1.9-1

4. GOTO's out of a procedure/function block are not implemented. Affects:
   6.8.2.4-1

5. File types can be used in structured variables, but only if the system special standard procedure FILEINIT(f) is executed on each such file, i.e. array of file. Test is successful with this change. Affects:
   6.4.3.5-4

6. CASE table size restricted to 1000 entries. Affects:

6.8.3.5-2

Implementation errors detected (both systems):

7. Functional values may only be assigned within the main function block. Affects:
   6.2.2-6

8. Real output formatting is non-standard.
   6.9.3.5.1-1 6.9.3.5.1-2

8080 CPM implementation errors detected:

9. In the CPM file environment, for non TEXT files, no accurate EOF marker is available. Thus eof is not necessarily set after read-back from a non-text file. 6.4.3.5-11 already failed for identifier length. Affects:
   6.4.3.5-5 6.5.3.5-6 6.4.3.5-7
   6.4.3.5-8 6.5.3.5-9 6.4.3.5-10
   6.4.3.5-11 6.4.3.5-12

NOTE

Except as noted above for system maxset, no textual changes whatsoever were made to the validation suite programs.

Appendix B

Compilation Example

```
  This is the sole operator input for this example
  ------------------------------------------------
B>job pascpcd typetext.pas con typetext.pcd typetext.pcd
JOB V1.2

B>; PcdCompile source listing pcd tic codelist options
B>; For unsegmented programs with no intrinsic/external calls.
B>; Use "NUL" for any unwanted files
B>; Tic may later be used to create assembly source with ASSMAP.
B>; Tic and pcd may be identical to save disk space
B>;   since assmpcd does not use pcd until tic has been read.
B>; Set options 128 for listing on assmpcd listing
B>;   (8 bit compiles for 16 bit machine(HP3000) with increasing stack)
B>;   (16 bit forces RBM format output on pcd)
B>RUNPCD pascalp (TYPETEXT.PAS,CON,TYPETEXT.PCD);[]


PascalP system Ver. 2.3.0 Copyright (C) 1982
CP/M installation rev. 2.3
PASCALP (pasctext, pasclist, prr, ef, output); V 3.1.0
83/09/07  9:57               PASCAL-P Universal Compiler Ver. 3.1.0

    1000     0:d   PROGRAM typetext(kbd, input, output);
    2000     0:d   (* Modification of "EXPAND" to paginate to crts    *)
    3000     0:d   (* and wrap over-long lines into multiple lines.   *)
    4000     0:d   (* Converts textfiles, replacing indention codes   *)
    5000     0:d   (* by spaces.       dle, ' '+i represents i spaces *)
    6000     0:d   (* Revised 14 July 83 to handle the sequences      *)
    7000     0:d   (*   dle eoln     and     dle code eoln            *)
    8000     0:d   (* both are mapped into simply eoln.  This avoids  *)
    9000     0:d   (* anomolies generated by a UCSD format editor.    *)
   10000     0:d   (* Assumes no non-printing characters in input     *)
   11000     0:d
   12000     0:d     LABEL 1;
   13000     0:d
   14000     0:d     CONST
   15000     0:d       dle      = 16;
   16000     0:d       pagesize = 24;   (* lines *)
   17000     0:d       linesize = 80;   (* columns *)
   18000     0:d
   19000     0:d     VAR
   20000     0:d       c        : char;
   21000     1:d       linenum,
   22000     1:d       column   : integer;
   23000     5:d       kbd      : text; (* for continue/terminate control
   24000   191:d
   25000   191:d   (*$n-,d- No runtime checks or linenos for speed *)
   26000   191:d
   27000   191:d     (* 1--------------1 *)
```

```
   28000   191:d
   29000   191:d     PROCEDURE pause;
   30000     0:d
   31000     0:d       BEGIN (* pause *)
   32000     0: 2      IF eof(kbd) THEN terminate
   33000    11: 4      ELSE readln(kbd);
   34000    21: 2      linenum := 1;
   35000    25: 2      END; (* pause *)
   36000    27: 2
   37000    27: 2    (* 1--------------1 *)
   38000    27: 2
   39000    27: 2    PROCEDURE linewrap;
   40000    27: 2
   41000    27: 2      BEGIN (* linewrap *)
   42000     0: 2      IF column > linesize THEN BEGIN (* linewrap *)
   43000    11: 4        column := 1; linenum := succ(linenum);
   44000    22: 4        IF linenum > pagesize THEN pause;
   45000    32: 4        writeln; END;
   46000    37: 2      END; (* linewrap *)
   47000    39: 2
   48000    39: 2    (* 1--------------1 *)
   49000    39: 2
   50000    39: 2    BEGIN (* typetext *)
   51000     0: 1    reset(kbd); linenum := 1; column := 1;
   52000    41: 1    WHILE NOT eof DO BEGIN
   53000    50: 3      WHILE NOT eoln DO BEGIN
   54000    59: 5        read(c);
   55000    67: 5        WHILE c = chr(dle) DO BEGIN
   56000    75: 7          IF NOT eoln THEN read(c);
   57000    92: 7          IF eoln THEN GOTO 1 (* dle eoln & dle code eoln
   58000   103: 9          ELSE BEGIN
   59000   106: 9            IF c > ' ' THEN BEGIN
   60000   114:11              write(' ' : ord(c)-ord(' '));
   61000   125:11              column := column + ord(c) - ord(' '); END;
   62000   137: 9            read(c); END;
   63000   145: 7          END;
   64000   147: 5        write(c); column := succ(column);
   65000   163: 5        IF (column > linesize) AND NOT eoln THEN linewrap
   66000   182: 3  1:  readln; column := 1; linenum := succ(linenum);
   67000   198: 3      IF linenum > pagesize THEN pause;
   68000   208: 3      writeln; END;
   69000   215: 1    END. (* typetext *)

NO. ERRORS=0 WARNINGS=0 Program size(pcode bytes)=288

NO. ERRORS=0 WARNINGS=0 Program size(pcode bytes)=288

Exit Pascal system, Max heap use @9F0C
B>RUNPCD assmpcd (TYPETEXT.PCD,TYPETEXT.PCD,);[]


PascalP system Ver. 2.3.0 Copyright (C) 1982
CP/M installation rev. 2.3
EXECUTABLE Code size (bytes) is 301 = 012D (hex)
```

```
Exit Pascal system, Max heap use @6E7E
B>era temp0001.$$$
B>
```

Appendix C

Compilation with codelisting

This is the sole operator input for this example

```
B>job pascpcd typetext.pas nul typetext.pcd typetext.pcd con 128
JOB V1.2

B>; PcdCompile source listing pcd tic codelist options
B>; For unsegmented programs with no intrinsic/external calls.
B>; Use "NUL" for any unwanted files
B>; Tic may later be used to create assembly source with ASSMAP.
B>; Tic and pcd may be identical to save disk space
B>;   since assmpcd does not use pcd until tic has been read.
B>; Set options 128 for listing on assmpcd listing
B>;   (8 bit compiles for 16 bit machine(HP3000) with increasing stack)
B>;   (16 bit forces RBM format output on pcd)
B>RUNPCD pascalp (TYPETEXT.PAS,NUL,TYPETEXT.PCD);[128]


PascalP system Ver. 2.3.0 Copyright (C) 1982
CP/M installation rev. 2.3
PASCALP (pasctext, pasclist, prr, ef, output); V 3.1.0

NO. ERRORS=0 WARNINGS=0 Program size(pcode bytes)=288

Exit Pascal system, Max heap use @9F0C
B>RUNPCD assmpcd (TYPETEXT.PCD,TYPETEXT.PCD,CON);[128]


PascalP system Ver. 2.3.0 Copyright (C) 1982
CP/M installation rev. 2.3
ASSMPCD (assmtext, rbmfile, listfile, output) Ver. 1.1.8
;    1000     0:d   PROGRAM typetext(kbd, input, output);
;    2000     0:d   (* Modification of "EXPAND" to paginate to crts    *)
;    3000     0:d   (* and wrap over-long lines into multiple lines.   *)
;    4000     0:d   (* Converts textfiles, replacing indention codes   *)
;    5000     0:d   (* by spaces.       dle, ' '+i represents i spaces *)
;    6000     0:d   (* Revised 14 July 83 to handle the sequences      *)
;    7000     0:d   (*   dle eoln     and     dle code eoln            *)
;    8000     0:d   (* both are mapped into simply eoln.  This avoids  *)
;    9000     0:d   (* anomolies generated by a UCSD format editor.    *)
;   10000     0:d   (* Assumes no non-printing characters in input     *)
;   11000     0:d
;   12000     0:d     LABEL 1;
0006              PGM  TYPETEXT
;   13000     0:d
;   14000     0:d     CONST
;   15000     0:d       dle      = 16;
;   16000     0:d       pagesize = 24;   (* lines *)
;   17000     0:d       linesize = 80;   (* columns *)
;   18000     0:d
```

```
;   19000     0:d     VAR
;   20000     0:d       c        : char;
;   21000     1:d       linenum,
;   22000     1:d       column   : integer;
;   23000     5:d       kbd      : text; (* for continue/terminate control
;   24000   191:d
;   25000   191:d   (*$n-,d- No runtime checks or linenos for speed *)
;   26000   191:d
;   27000   191:d     (* 1--------------1 *)
;   28000   191:d
;   29000   191:d     PROCEDURE pause;
;   30000     0:d
;   31000     0:d       BEGIN (* pause *)
0006                FWD  PAUSE
;   32000     0: 2      IF eof(kbd) THEN terminate
0006                PRO  PAUSE
0006                NTR  PAUSE
                   PAUSE:
0006 F70000         ENT  1,@5
0009 66FF41         LAO  191
000C F00C           EOF
000E F8FFF0         FJP  @6
;   33000    11: 4      ELSE readln(kbd);
0011 F025           CSP  TRM
0013                PAR  0
0013 FAFFEB         UJP  @7
                   @6:
0016 66FF41         LAO  191
0019 F01B           CSP  RLN
001B                PAR  1
                   @7:
;   34000    21: 2      linenum := 1;
001B 01             LDCI 1
001C 73FFFB         STOI 5
;   35000    25: 2      END; (* pause *)
001F 00F6           RET  0
                  @5=0

;   36000    27: 2
;   37000    27: 2    (* 1--------------1 *)
;   38000    27: 2
;   39000    27: 2    PROCEDURE linewrap;
;   40000    27: 2
;   41000    27: 2      BEGIN (* linewrap *)
0021                FWD  LINEWRAP
;   42000     0: 2      IF column > linesize THEN BEGIN (* linewrap *)
0021                PRO  LINEWRAP
0021                NTR  LINEWRAP
                   LINEWRAP:
0021 F70000         ENT  1,@9
0024 7BFFFD         LDOI 3
0027 50             LDCI 80
0028 CB             GRTI
0029 F8FFD5         FJP  @10
```

```
;   43000    11: 4        column := 1; linenum := succ(linenum);
002C 01           LDCI 1
002D 73FFFD       STOI 3
0030 7BFFFB       LDOI 5
0033 E0           INCI 1
0034 73FFFB       STOI 5
;   44000    22: 4        IF linenum > pagesize THEN pause;
0037 7BFFFB       LDOI 5
003A 18           LDCI 24
003B CB           GRTI
003C F8FFC2       FJP  @11
003F F301         CGP  1,PAUSE
                 @11:
;   45000    32: 4        writeln; END;
0041 7A000C       LDOA -12
0044 F01D         CSP  WLN
0046              PAR  1
                 @10:
;   46000    37: 2      END; (* linewrap *)
0046 00F6         RET  0
                @9=0

;   47000    39: 2
;   48000    39: 2    (* 1--------------1 *)
;   49000    39: 2
;   50000    39: 2    BEGIN (* typetext *)
;   51000     0: 1    reset(kbd); linenum := 1; column := 1;
0048 F027         MAI  TYPETEXT
                 TYPETEXT:
004A F70000       ENT  1,@13
004D 01           LDCI 1
004E 01           LDCI 1
004F 5D86         LDCI 134
0051 0C           LDCI 12
0052 5C084B424420
0058 20202020     LCA  'KBD     '
005C 08AA         MVS  8
005E 66FF41       LAO  191
0061 F026         CSP  FIN
0063              PAR  9
0063 66FF41       LAO  191
0066 00           LDCA NIL
0067 F019         CSP  RES
0069              PAR  2
0069 01           LDCI 1
006A 73FFFB       STOI 5
006D 01           LDCI 1
006E 73FFFD       STOI 3
;   52000    41: 1    WHILE NOT eof DO BEGIN
                 @14:
0071 7A000A       LDOA -10
0074 F00C         EOF
0076 EC           NOT
0077 F8FF87       FJP  @15
```

```
;   53000    50: 3      WHILE NOT eoln DO BEGIN
                 @16:
007A 7A000A       LDOA -10
007D F00B         CSP  ELN
007F              PAR  0
007F EC           NOT
0080 F8FF7E       FJP  @17
;   54000    59: 5        read(c);
0083 66FFFF       LAO  1
0086 7A000A       LDOA -10
0089 F016         CSP  RDC
008B              PAR  2
;   55000    67: 5        WHILE c = chr(dle) DO BEGIN
                 @18:
008B 79FFFF       LDOC 1
008E 10           LDCI 16
008F B1           EQUC
0090 F8FF6E       FJP  @19
;   56000    75: 7          IF NOT eoln THEN read(c);
0093 7A000A       LDOA -10
0096 F00B         CSP  ELN
0098              PAR  0
0098 EC           NOT
0099 F8FF65       FJP  @20
009C 66FFFF       LAO  1
009F 7A000A       LDOA -10
00A2 F016         CSP  RDC
00A4              PAR  2
                 @20:
;   57000    92: 7          IF eoln THEN GOTO 1 (* dle eoln & dle code eol
00A4 7A000A       LDOA -10
00A7 F00B         CSP  ELN
00A9              PAR  0
00A9 F8FF55       FJP  @21
00AC FAFF52       UGO  @3
;   58000   103: 9          ELSE BEGIN
00AF FAFF4F       UJP  @22
                 @21:
;   59000   106: 9            IF c > ' ' THEN BEGIN
00B2 79FFFF       LDOC 1
00B5 20           LDCC ' '
00B6 C9           GRTC
00B7 F8FF47       FJP  @23
;   60000   114:11               write(' ' : ord(c)-ord(' '));
00BA 20           LDCC ' '
00BB 79FFFF       LDOC 1
00BE 20           LDCC ' '
00BF A0           SBI
00C0 7A000C       LDOA -12
00C3 F01F         CSP  WRC
00C5              PAR  3
;   61000   125:11               column := column + ord(c) - ord(' '); END;
00C5 7BFFFD       LDOI 3
00C8 79FFFF       LDOC 1
```

```
00CB 9E          ADI
00CC 20          LDCC ' '
00CD A0          SBI
00CE 73FFFD      STOI 3
                 @23:
;   62000   137: 9            read(c); END;
00D1 66FFFF      LAO  1
00D4 7A000A      LDOA -10
00D7 F016        CSP  RDC
00D9             PAR  2
                 @22:
;   63000   145: 7          END;
00D9 FEB1        UJS  @18
                 @19:
;   64000   147: 5        write(c); column := succ(column);
00DB 79FFFF      LDOC 1
00DE 01          LDCI 1
00DF 7A000C      LDOA -12
00E2 F01F        CSP  WRC
00E4             PAR  3
00E4 7BFFFD      LDOI 3
00E7 E0          INCI 1
00E8 73FFFD      STOI 3
;   65000   163: 5        IF (column > linesize) AND NOT eoln THEN linewra
00EB 7BFFFD      LDOI 3
00EE 50          LDCI 80
00EF CB          GRTI
00F0 7A000A      LDOA -10
00F3 F00B        CSP  ELN
00F5             PAR  0
00F5 EC          NOT
00F6 EF          AND
00F7 F8FF07      FJP  @24
00FA F302        CGP  0,LINEWRAP
                 @24:
00FC FE7D        UJS  @16
                 @17:
;   66000   182: 3  1:  readln; column := 1; linenum := succ(linenum);
                 @3:
00FE 7A000A      LDOA -10
0101 F01B        CSP  RLN
0103             PAR  1
0103 01          LDCI 1
0104 73FFFD      STOI 3
0107 7BFFFB      LDOI 5
010A E0          INCI 1
010B 73FFFB      STOI 5
;   67000   198: 3      IF linenum > pagesize THEN pause;
010E 7BFFFB      LDOI 5
0111 18          LDCI 24
0112 CB          GRTI
0113 F8FEEB      FJP  @25
0116 F301        CGP  0,PAUSE
                 @25:
```

```
;   68000   208: 3      writeln; END;
0118 7A000C       LDOA -12
011B F01D         CSP  WLN
011D              PAR  1
011D FE53         UJS  @14
                 @15:
;   69000   215: 1    END. (* typetext *)
011F 66FF41       LAO  191
0122 F00A         CSP  CLO
0124              PAR  1
0124 F024         STP
                @13=192

0126              END
0126 FEFAFEDDFF1D
012C 02

  0 TYPETEXT       0048 global
  1 PAUSE          0006 global
  2 LINEWRAP       0021 global
EXECUTABLE Code size (bytes) is 301 = 012D (hex)

Exit Pascal system, Max heap use @6E7E
B>era temp0001.$$$
B>
```

→→e