PEOPLE'S PASCAL II
USER'S MANUAL FOR TRS-80

By KIN-MAN CHUNG and HERBERT YUEN

(The authors wrote, in the September, October and November "Byte" magazine, a three-part article titled: "A 'Tiny' Pascal Compiler", including listing for North Star Basic. Since the "Byte" series, they have re-written their tiny Pascal operating system in tiny Pascal, and compiled it into Z80-native code for TRS-80.)

PEOPLE'S PASCAL OPERATING SYSTEM

Your 'Tiny' Pascal system, hereafter called People's Pascal II, is a complete, self-contained operating system for creating, compiling, running, saving and loading Pascal Programs for the TRS-80. Once you have loaded People's Pascal II, you never need leave the operating system. The People's Pascal II system is composed of three inter-related sections:

Monitor: This is the sub-system which provides run-time support, checks for errors, and provides the necessary utilities to save and load programs to and from cassette tape.

Compiler: This is the program which compiles your Pascal source program into P-code, ready to be executed. The compiler also checks for syntax errors.

Editor: The editor is used to create or modify People's Pascal source programs.

All these sub-systems are loaded simultaneously, and are always present in RAM, unless you choose to overwrite portions to free memory space.

MINIMAL SYSTEM requirements are: Level II, 16K RAM!

The first sections of this users' manual will discuss in detail the three subsystems, what they do, and how to use them. The next section will deal with the specific aspects, limitations and enhancements to People's Pascal II; then follows a chapter on getting started, to help you get through the first time you bring People's Pascal up. Finally, you will find the error codes, syntax diagrams, and the sample programs.

PEOPLE'S PASCAL MONITOR

All operations make at least some use of the monitor, hence we will begin our discussion of the People's Pascal II system with it. The monitor provides run-time support to the entire system, as well as providing you with a means of saving or loading both source programs, and P-code programs from or to cassette tape. From the monitor one also

gives the command to compile a program, or to run that program once it has been compiled. You also invoke the editor from the monitor. Below is a list of the monitor commands and what they do:

E	Edit old source file or create a new one.
C	Compile source program into P-code, ready to be executed. P-code is placed after source in RAM.
C/-P	Compile source, but do NOT generate P-code (useful to check for syntax errors).
C/-S	Compile source, and overwrite the source program (used when you have very large programs).
R	Run the compiled program.
R/-C	Run the compiled program and overwrite the editor and compiler.
LS >*filename*<	Load source program from cassette.
LP >*filename*<	Load P-code program from cassette.
WS >*filename*<	Save source program to cassette.
WP >*filename*<	Save P-code program to cassette.

It should be noted that you are given the ability to overwrite sections of the People's Pascal system if you need the space for large programs. However, you must remember that they are "gone" and you must re-load the entire system again if you are to use them further.

It should also be noted at the time that a filename can be at most six (6) characters long. Errors will result if this is not adhered to.

THE PEOPLE'S PASCAL EDITOR

The text editor provided with your People's Pascal package enables you to create and modify source programs. The text editor is line oriented, but, unlike Basic, does not use line numbers. The maximum number of lines of text that you can have is 600, and the maximum line length is 130 characters.

All editor commands are single characters; some may have numeric arguments following them, or a character string. In our discussion of the editor, "***xx***" refers to integer numbers (1-999), and >***string***< refers to a string. Each command ends with a >***cr***<, carriage return ("ENTER" on your TRS-80 keyboard). Invalid commands are flagged with the message "ILLEGAL". The line pointer always points at the line most recently displayed or modified, or inserted. After a Delete command, the line pointer is moved up one line.

Below is a list of the editor commands. Note: "*" means entirely or "all the way":

>cr<	A carriage return on an empty line will exit from insert mode.
PRINT P	Print the current line.
Pxx	Print xx lines starting from current line.

P*	Print entire file.
Up U	Move up one line.
Uxx	Move up xx lines.
U*	Move up to top of first line of file.
NEXT N	Move line pointer to next line (down).
Nxx	Move line pointer down xx lines.
N*	Move line pointer to last line of file.
Delete D	Delete current line.
Dxx	Delete xx lines starting at current line.
D*	Delete entire file (i.e., "scratch").
Insert I	Enter insert mode (remember, you exit with a >cr<).
	Insert lines after current line pointer. A "?" is displayed to prompt you.
Replace R >st<	Replace the current line by >string<.
Extend X	The current line is displayed and the cursor is at the end of the line, more characters can be appended to the end (similar to Basic),
Status S	Status of current file displayed includes: number of lines, file location, position of line pointer.
QUIT Q	Return to People's Pascal II monitor.

The editor also recognizes two special keys:

<-	the back arrow, for backspace, and
->	the right arrow, for tab, which is three spaces..

These two keys may be used at any time for editing a command or input file.

Expanding on this: if you want to enter a program, you would type "E" from the monitor, then you would type "I" for insert. You then can enter text. To stop entering text, you type a blank carriage return on an empty line.

Hint: When "MEMORY FULL" error occurs while editing or inserting, the source is too big.

You should play with the editor a while to make sure that you completely understand its operation.

PEOPLE'S PASCAL COMPILER

Roughly speaking, a compiler is a program that translates the statements of a high-level language into an equivalent program of machine-readable form. People's Pascal II translates the high-level source program into an intermediate file called P-code. P-code is then interpreted, using the run-time monitor for support. The result is programs which execute at least four times faster, and up to eight times faster than Basic!

People's Pascal II is a subset of standard Pascal. The syntax is essentially identical to its larger brother. Syntax diagrams have been included for those who are just now learning the language. It must be emphasized that this manual is not an instructional text on Pascal programming, but rather an explanation of the limits and special features of People's Pascal. However, we will review some essential points in the next section.

Partial list of books:

Programming in Pascal; Grogono, Addison-Wesley, 1978
Pascal: User Manual and Report; Jensen & Wirth, Springer-Verlag, 1974
A Primer on Pascal; Conway, Gries and Zimmerman; Winthrop Publishers, 1976

COMPILER SPECIFICS

Maximum number of procedure or function parameters is 15; maximum number of procedure nestings is seven levels; the symbol table is restricted to 75 (200 for the big version). ":=" is used for assignment and "=" is used for equality, They are not interchangeable! ";" is used to separate statements, not to end statements. Thus the last ";" in a compound statement:

```
BEGIN STATEMENT
        STATEMENT;
        IF >EXP< THEN >EXP< ELSE
        >EXP<; STATEMENT;
END
```

is not necessary. It is, however, allowed since a Pascal statement can be a null. Note also the absence of ";" before an ELSE or an END in the "CASE" statement.

Expressions may be either arithmetic or logical (Boolean). Thus, the following are perfectly legal:

```
        A := B < C;
        .
        .
        IF A+B THEN ….
```

Note also that the Boolean operator “OR” has the same precedence as the arithmetic operator “+” and “-“; “AND” the same as “*” and “DIV”, etc. It is important to remember that “OR” and “AND” have precedence over “=”, “>”, etc, thus the need for brackets at times as shown below:

```
IF (A > 10) AND (A < 5) THEN …
```

The statement:

```
IF A > 10 AND (A <5) THEN ….
```

would be parsed as:

```
IF A > (10 AND (A < 5)) THEN …
```

thus producing the undesirable result. There are some context-sensitive rules and meanings that cannot be inferred from the syntax diagrams, and may be particular to this implementation:

“(“ and “)” are used in the TRS-80 implementation instead of “{“ and “}”.

Identifier names must start with a letter and may be followed with letters or digits, but only the first four characters are significant. However, reserved words must be typed in full. Identifiers must be declared before used. Identifiers can be declared twice, but only the last one is used. Formal parameters of a procedure need not (and should not) be declared again inside the procedure. Parameters are passed to procedures or functions by value, i.e., a copy of the value of the parameter by the program before the call. The scope rules for identifiers are the same ones used by any block-structure language. The scope of a variable is the procedure that contains its. An inner procedure can reference a variable in an outer procedure.

The only data types People’s Pascal supports are integers are one-dimensional integer arrays. The integers are 16-bit signed, the arrays start at 0. Arrays are NOT checked for “subscript out of range” at run-time.

The meaning of certain operations is:

A DIV B	truncated integer division	:	27 DIV 5 = 5
A MOD B	A – (A DIV B) * B	:	27 MOD 5 = 2
A SHL B	Left shift A by B	:	27 SHL 2 = 54
A SHR B	Right shift A by B	:	27 SHR 2 = 13

The built-in array MEM can be used to read to (if it appears in the left side of an assignment) or from (if it appears in an expression) to or from a specified memory location, such as:

```
A := MEM (24467) + 3;
MEM(T) := 0;
```

A second form of the MEM function is “MEMW”. This enables a two-byte word to be read to or from memory using the same convention as for “MEM”. Note: the low-order byte comes first, in accordance with INTEL convention.

Hex constants are prefixed by % (e.g., %2A00).

Strings are enclosed by single quote (‘), not double. When a string appears in an expression or as a CASE label, it has the value equal to the ASCII value of the first character of the string. When a string appears in the WRITE statement, the entire string would be outputted, such as:

```
X := ‘ABCD’  X would equal ‘A’ = 65
```

The READ and WRITE statements are character-oriented, not line-oriented. More than one character can be placed in the same statement. Decimal numbers or Hex numbers can be read-in from the keyboard by a “#” (decimal) or “%” (hex) after the variable in the READ statement. Similarly, a decimal integer can be printed on the output device by following the expression with the appropriate “#” or “%” for Hex.

```
READ (A,B,C,I#,J%)
```

This would READ three (3) characters, a decimal number, and a hex number.

```
A := 65
WRITE (‘HELLO? ‘,A,’ ‘,A#,’ ‘,A%)
```

would print:

```
HELLO? A 65 0041
```

Since the READ is character-oriented, it is necessary to terminate an integer input by a non-integer character (such as a >cr< or >sp<). To input a hex number, four (4) digits must be typed.

To write on a new line, it is also necessary to output explicitly the ASCII code for >cr< and >lf< to the output device. That is, you must manually insert carriage return line feed. Such as:

```
WRITE('THIS IS A TEST',13,10)
(HERE CR = 13, LF = 10)
```

An expression in the IF, WHILE, and REPEAT statements are said to fulfill the condition if the least-significant bit is 1. This is equivalent to test that the expression is odd. Thus after:

```
IF X THEN A := 1 ELSE A := 100
```

A would have the value of 1 if X is odd, and 100 if X is even.

The relational operators (e.g. "=", "=>" … etc) always produce a value of 0 or 1. Thus after:

```
A := X = 5;
    A= 1 IF X=5, OTHERWISE A=0
```

Comments are delimited by " (* " and " *) ".

What follows is a list of built-in functions to the compiler:

ABS(X)	returns the absolute value of x
SQR(X)	returns square of X
INP(X)	inputs port X, used as A = INP(X)
OUTP(X,A)	outputs A to port X
INKEY	inputs the keyboard, used as in A := INKEY
PLOT(X,Y,A)	plots graphics to screen, using the X-Y coordinates. If A is odd then plot is "set", if A is even then plot is "reset".
POINT(X,Y)	just like Basic: returns a "1" if the point is filled, a "0" if blank
MOVE(B,A,N)	move a block of memory of N bytes from address A to address B.

Screen control characters are the same as TRS-80 Basic. For example, use WRITE (23,31) to clear the screen.

BRINGING UP PEOPLE'S PASCAL

In this section of the People's Pascal users' manual, we will go step by step from loading the tape the first time, to running your first program. Side one of your tape comes with three sample programs, the first is loaded with the system, the second is “HILBER” and the third is “BLOCK”. Side two contains the big version and source to People’s Pascal, “PAS32K” and “COMPS1”, respectively.

STARTUP

1) Turn on your machine. When asked for MEMORY SIZE, respond by hitting the is ENTER key.

2) Type SYSTEM to reach system level. TRS-80 will display the prompt: *?.

3) Make sure that your People's Pascal tape is at the start, and type PASCAL and then ENTER and turn recorder to PLAY.

4) The tape will begin to load, the star will blink every 4 seconds. The entire load will take about 3 min.

5) Once the tape has loaded, type a “/” (slash) and hit ENTER. At this point you should receive the opening message:

    "TINY PASCAL V-1.0"

6) At this point you have successfully loaded the entire People's Pascal operating system, and can proceed to the next section, below.

If you did not get this far try loading the tape again, at various volume settings. Mark down, on the cassette label, the volume setting that was successful. If it will not load, and other commercial tapes will load, return it to CIE for replacement.

CREATING A PROGRAM

1) From the monitor, type "E". This will place you in the editor.

    You will see one of two messages, either:

    EMPTY FILE... ENTER TEXT

    This is when there is no current source program, or you will see

    a set of statistics on the current file.

    On the initial load, the sample program is loaded simultaneously. If this is your very first try, then skip ahead to step 5, otherwise proceed.

2) To "scratch" the sample program which is always loaded with the system, you simply use the editor command: **D*.**

3) At this point you may enter a program.

4) Once your program is entered, you may exit insert mode by hitting an ENTER on the next blank line. This puts you back in the editor command mode.

5) To return to the monitor, in order to compile, etc., you type **Q**.

COMPILING, RUNNING, SAVING/LOADING A PROGRAM

1) Normally, to compile a source program, you type C from the monitor. This creates P-code. If you have any syntax errors, they will show up here.

   If you have syntax errors, the error list on the back of this manual will tell you what they are. You should then go back and re-edit the existing source file, correcting the syntax errors, before re-compiling.

2) Once you have successfully compiled the program, you may run it by typing R from the monitor.

3) To save the program, or the P-code. You may use the appropriate monitor commands. Or you may load a previously saved program.

Remember, you must re-compile a program if you make a change in it!

SPECIAL NOTES

It should be noted that the BREAK key equals a temporary stop of program execution, and that any other key re-starts it. If you hit BREAK twice in a row, you will terminate the run, and return to the People's Pascal monitor (like a control-C on most other systems).

One should also note that, once a program has been compiled, only the P-code (that is, the compiled program) need be loaded for execution. In other words, it is not necessary to compile before each execution if you have saved the P-code on tape.

When error 1001 is encountered during compilation, there is not enough memory. You should try using C/-P. Be sure to save the source first!

When MEMORY FULL error occurs on running the program, either cut down array size, or try using R/-C option.

We know that you will enjoy using People's Pascal, and recommend that you “play” with it a while just to get the feel for it, and to become familiar with all of its features.

USING THE 'BIG' PASCAL ON SIDE (B)

On side two of your tape is an expanded People's Pascal compiler. That is, it can handle larger programs. You will need at least 36K RAM to use it.

To use, simply follow the directions "On Bringing Up People's Pascal”, except substitute PAS32K for PASCAL.

The source to the compiler is immediately after PAS32K on side B. It is called: COMPS1. You can then "play" with the source to the compiler. Note: you will need at least 36K to compile the compiler.

IMPORTANT: Source programs are not interchangeable between the two compilers. That is, a program created using the big compiler can NOT be used with the normal compiler, and vice versa.

ERROR CODES:

1 error in simple type
2 identifier expected
3 "program" expected
4) expected
5 : expected
6 illegal symbol
7 error in parameter list
8 OF expected
9 (expected
10 error in type
11 (expected
13 END expected
14 ; expected
15 integer expected
16 = expected
17 BEGIN expected
18 error in declaration part
19 error in field-list
20 , expected
21 * expected

50 error in constant
51 := expected
52 THEN expected
53 UNTIL expected
54 DO expected
55 TO/DOWNTO expected
56 IF expected
57 FILE expected
58 error in factor
59 error in variable

101 identifier declared twice
102 low bound exceeds high bound
103 identifier is not of appropriate class
104 identifier not declared
105 SIGN NOT ALLOWED
106 number expected
107 incompatible subrange types
108 file not allowed here
109 type must not be real
110 tagfield type must be scalar
111 incompatible with tagfield type
112 index type must not be real
113 index type must be scalar
114 base type must not be real
115 base type must be scalar
116 error in type of standard procedure parameter
117 unsatisfied forward reference
118 forward reference type identifier in variable declaration
119 forward declared; repetition not allowed
120 function result type must be scalar
121 file value parameter not allowed
122 forward declared function, repetition not allowed
123 missing result type in function declaration
124 F-format for real only
125 error in type of standard function parameter
126 number of parameters does not agree with declaration
127 illegal parameter substitution
128 result type of parameter function does not agree with declaration
129 type conflict of operands
130 expression is not of set type
131 tests on equality allowed only
132 strict inclusion not allowed
133 file comparison not allowed
134 illegal type of operand
135 type of operand must be Boolean

136 set element type must be scalar
137 set element types not compatible
138 type of variable is not array
139 index type is not compatible with declaration
140 type of variable is not record
141 type of variable must be file or pointer
142 illegal parameter substitution
143 illegal type of loop control variable
144 illegal type of expression
145 type conflict
146 assignment of files not allowed
147 label type incompatible with selecting expression
148 subrange bounds must be scalar
149 index type must not be integer
150 assignment to standard function is not allowed
151 assignment to formal function is not allowed
152 no such field in this record
153 type error in read
154 actual parameter must be a variable
155 control variable must neither be formal nor non-local
156 multidefined case label
157 too many cases in case statement
158 missing corresponding variant declaration
159 real or string tagfields not allowed
160 previous declaration was not forward
161 again forward declared
162 parameter size must be constant
163 missing variant in declaration
164 substitution of standard procedure/function not allowed
165 multidefined label
166 multideclared label
167 undeclared label
168 undefined label
169 error in base set
170 value parameter expected
171 standard file was redeclared
172 undeclared external file
173 (not relevant)
174 Pascal procedure or function expected
175 missing input file
176 missing output file
201 error in RREAL constant: digit expected
202 string constant must not exceed source line
203 integer constant exceeds range
204 (not relevant)
250 too many nested scopes of identifiers

251 too many nested procedures and/or functions
252 too many forward references of procedure entries
253 procedure too long
254 too many long constants in this procedure
255 too many errors on this source line
256 too many external references
257 too many externals
258 too many local files
259 expression too complicated

300 division by zero
301 no case provided for this value
302 index expression out of bounds
303 value to be assigned is out of bounds
304 element expression out of range

398 implementation restriction
399 variable dimension arrays not implemented
1000 . missing
1001 out of memory

USEFUL CALLS, ADDRESSES INSIDE THE MONITOR

Below is a list of useful addresses for those who may wish to use them.

address	**function**
4180 (hex)	starting address of source
4182	ending address of source
4184	start of P-code
4186	end of P-code
4188	address of editor
418A	address of compiler
418C	start address of user source program
418E	address of run-time stack
4190	ending address of run-time stack
4192	end of memory address (7FFF for 16K)
4194	monitor entry point
4196	address of program currently executing
4198	complement of contents of 418E
419A	overflow message flag - default 0

-- I/0 CALLS - - - -

41A0	console in
41A2	console out
41A4	INKEY (input the keyboard - CR [ENTER] not needed)

```
(* SAMPLE TINY PASCAL PROGRAM BY H. YUEN *)
VAR X0, Y0, X, Y, K, F : INTEGER;
BEGIN
  X0 :=13000;
  Y0 := 18000;
  F :=11;
  REPEAT
    X := X0;
    Y := Y0;
    WRITE(15.23,31);
    FOR K := 1 TO 1000 DO
    BEGIN
      X := X + Y DIV 4;
      Y := Y - X DIV 5;
       PLOT(X SHR 8, Y SHR 8,1)
    END;
    X0 := X0 * X0 DIV F;
    Y0 := Y0 + Y0 DIV F;
    F := F + F DIV 6
  UNTIL F > 70;
  WRITE(28,31,'THE SHOW IS OVER’)
END.
```

```
(* PLOT HILBERT CURVES OF ORDERS 1 TO N *)
CONST N = 4, N0 = 32;
VAR I, N, X, Y, X0, Y0, U, V : INTEGER;

PROC MOVE;
VAR I, J : INTEGER;

FUNC MIN(A, B );
BEGIN
  IF A > B
     THEN MIN := B
     ELSE MIN := A
END;

FUNC MAX(A, B);
BEGIN
  IF A < B
     THEN MAX := B
     ELSE MAX := A
END;

BEGIN  (* MOVE *)
  FOR I := MIN(X,U) TO MAX(X,U) DO
  FOR J := MIN(Y,V) TO MAX(Y,V) DO
      PLOT(I,J,1);
  U := X;
  V := Y;
END;

PR0C P(TYP,I);
BEGIN
  IF I > 0 THEN
     CASE TYP OF
     1: BEGIN
          P(4,I-1); X := X – H; MOVE;
          P(1,I-1); Y := Y-H; MOVE;
          P(1,I-1); X := X + H; MOVE;
          P(2,I-1);
        END;

     2: BEGIN
          P(4,I-1); X := X – H; MOVE;
          P(1,I-1); Y := Y-H; MOVE;
          P(1,I-1); X := X + H; MOVE;
          P(2,I-1);
        END;
```

```
     3: BEGIN
          P(4,I-1); X := X – H; MOVE;
          P(1,I-1); Y := Y-H; MOVE;
          P(1,I-1); X := X + H; MOVE;
          P(2,I-1);
        END;

     4: BEGIN
          P(4,I-1); X := X – H; MOVE;
          P(1,I-1); Y := Y-H; MOVE;
          P(1,I-1); X := X + H; MOVE;
          P(2,I-1);
        END;
     END
END;

BEGIN (*MAIN*)
WRITE(15,23,31,13,' HILBERT CURVES');
I := 0;
H := H0;
X0 := H DIV 2;
Y0 := X0;
REPEAT
  I := I + 1;
  H := H DIV 2;
  X0 := X0 + H DIV 2;
  Y0 := Y0 + H DIV 2;
  X := X0 + (I-1) * 32;
  Y := Y0 + H DIV 2;
  X := X0 + (I-1) * 32;
  Y0 := Y0 + 10;
  U := X;
  V := Y;
  P(1,I)
UNTIL I = N
END.
```

```
(* BLOCKADE  BY K. M. CHUNG  4/26/79 *)
VAR I, J, SPEED, ABORT, BLNK : INTEGER;
SCORE, MARK, MOVE, CURSOR :  ARRAY(1) OF INTEGER;

PROC PSCORE;
BEGIN
  WRITE(SCOR(0)#);
  MEMW(%4020) := %3FFE;  (* SET CURSOR *)
  WRITE(SCORE(1)#)
END;

PROC BLINK;
VAR T, K, DELAY : INTEGER;
BEGIN
  T := CUSOR(I)-MOVE(I);
  FOR K := 1 TO 30 DO
  BEGIN
    FOR DELAY := 1 TO 100 DO;
    IF MEMW(T) = BLNK
      THEN MEMW(T) := MARK(I)
      ELSE MEMW(T) := BLNK
  END
END;

BEGIN
  WRITE('SPEED(1-10)');
  READ(SPEED#);
  SPEED := SPEED * 10;
  MARK(0) := '*' + '*' SHL 8;
  MARK(1) := '(' + ')' SHL 8;
  BLNK := ' ' + ' ' SHL 8;
  SCORE(0) := 0;
  SCORE(1) := 0;
  REPEAT
    WRITE(15,28,31);   (* TURN OFF CURSOR, CLEAR SCREEN *)
    FOR I := 9 TO 117 DO
    BEGIN
      PLOT(I,1,I);
      PLOT(I,45,1)
    END;
    FOR I := 1 TO 45 DO
    BEGIN
      PLOT(9,I,1);
      PLOT(10,I,1);
      PLOT(116,I,1);
      PLOT(117,I,1);
```

```
    END;
    CURSOR(0) := %3C00 + 64 * 4 + 12;
    CURSOR(1) := %4000 + 64 * 4 – 16;
    FOR J := 0 TO 1 DO
    MEMW(CURSOR(K)) := MARK(J);
    MOVE(0) := 64;
    MOVE(1) := -64;
    I := 1;
    ABORT := 0;
    PSCORE;
    REPEAT
    UNTIL INKEY <> 0;  (* HIT KEY TO START *)
    REPEAT
      I := 1 – I;
      FOR J := 1 TO SPEED DO
      CASE INKEY OF
        'W' : MOVE(0) := -64;
        'D' : MOVE(0) :=2;
        'O' : MOVE(1) := -64;
        ';' : MOVE(1) := 2;
        'X' : MOVE(0) := 64;
        'A' : MOVE(0) := -2;
        '.' : MOVE(1) := 64;
        'K' : MOVE(1) := -2;
      END;
      CURSOR(I) := CURSOR(I) + MOVE(I);
      IF MEMW(CURSOR(I)) = BLNK
         THEN MEMW(CURSOR(I)) := MARK(I)
         ELSE BEGIN
                SCORE(1-I) := SCORE(1-I) + 1;
                ABORT := 1;
                BLNK
              END
    UNTIL ABORT
  UNTIL SCORE(1-I) >= 10
END.
```

People's Pascal I
TRS-80 People's Pascal System Documentation

Pipe Dream Software, Berwick, Australia

1. Introduction

The TRS-80 People's Pascal system is a program development system for Tiny Pascal, a subset of the Pascal programming introduced by Niklaus Wirth of the Engineering University at Zurich.

Tiny Pascal was defined in Byte magazine- "A 'Tiny' Pascal Compiler", by Kin-Man Chung and Herbert Yuen, in three parts: September, October and November.

The Pascal language is defined in "Pascal: User Manual and Report" by Kathleen Jensen and Niklaus Wirth (Springer-Verlag 1974). A good introductory book on Pascal is "Microcomputcr Problem Solving Using Pascal" by Kenneth L. Bowles (Springer-Verlag 1977).

The following programs are supplied with People's Pascal:

1.1 Text Editor

The editor is line-oriented. Intra-line editing is not provided. Text files may be manipulated in the following ways: create, edit, list, print, merge, copy, read from and write to cassette.

The editor uses a 3,000-character (3K) text buffer and 240-character blocked records on cassette files to achieve its results. Source files of indefinite length may be manipulated, but short files are recommended for convenience and modularity. Editor commands are: insert, delete, replace, list, print, read and merge, write, re-number, compile, and free. Refer to the editor operating instructions and program documentation for details.

1.2 Compiler

Included in the same program as the editor, to save time-consuming swapping between programs during program development, the People's Pascal compiler translates the source program in the text buffer and/or included from one or more source program text files into a P-code object program on cassette.

The compiler accepts the following Pascal subset: AND, ARRAY, BEGIN, CASE, CONST, DIV, DO, DOWNTO, ELSE, END, EOR, FUNC, IF, INTEGER, MOD, NOT, OF, OR, PROC, READ, REPEAT, SHL, SHR, THEN, TO, UNTIL, VAR, WRITE.

Note that character arrays, records, files, reals, programmer-defined types pointers and GOTO are omitted from People's Pascal.

In addition, extensions are provided to read from and write to absolute memory addresses, and to allow the calling of assembly language subroutines at absolute memory addresses, together with limited numeric I/O formatting and the definition of hex constants.

A "$INCL" include-source-file feature is provided to allow modular program development.

The compiler is a one-pass compiler using recursive descent. Refer to language definition, compiler operation and compiler program documentation for details.

1.3 The Interpreter

The interpreter reads a P-code program object file produced by the compiler into memory and interprets the program, performing the actions required of the imaginary P-machine, which has P-code as its instruction set.

The interpreter has the following debugging routines:

SET BREAKPOINT(s)
CLEAR ALL BREAKPOINTS
EXAMINE PROGRAM
GO
EXAMINE STACK CONTENT
EXAMINE NEXT PROGRAM LOCATION
QUIT
RUN
SINGLE STEP

TRACE
EXAMINE PREVIOUS PROGRAM LOCATION
DISPLAY P-MACHINE REGISTERS
DISPLAY BREAKPOINTS

The current version of the interpreter written in Basic is slow, with a double level of interpretation.

P-code object programs of up to 8.6K bytes may be interpreted. Refer to interpreter operating instructions and interpreter program documentation for details.

1.4 Translator

The translator reads in a P-code object file produced by the compiler and translates P-code instructions into fast Z-80 code (machine language instructions) using the Z-80 stack pointer for the People's Pascal stack. Translated programs run about five times faster than Level-II Basic. Graphics instructions run about eight time faster.

The translator also has the option to optimize for minimum memory usage, reducing program size to half at the cost of some speed reduction. Refer to translator operating instructions and program documentation for details

1.5 People's Pascal Source Library

The People's Pascal library uses the "$INCL" compiler option to allow an extendable set of standard routines to be incorporated into user programs. The following routines are provided:

SET(ON/OFF,X,Y) (=SET(X,Y))	set/reset graphics
RND(SEED)	pseudo random number generation
AT(cursorposition) (=PRINTE)	cursor control

User-written routines may be added to the People's Pascal library. Other routines also may be provided, such as UCSD "turtle" graphics procedures MOVE(distance), TURN(angle), MOVETO(X,Y), PENCOLOR(white, black/none).

1.6 Run-Time System

This program is written in Z-80 assembly language and provides subroutines called by translated People's Pascal programs for multiply, divide, set, I/O, etc. It occupies about 1 K byte. Both source and object programs are supplied. Refer to run-time-system program documentation for details.

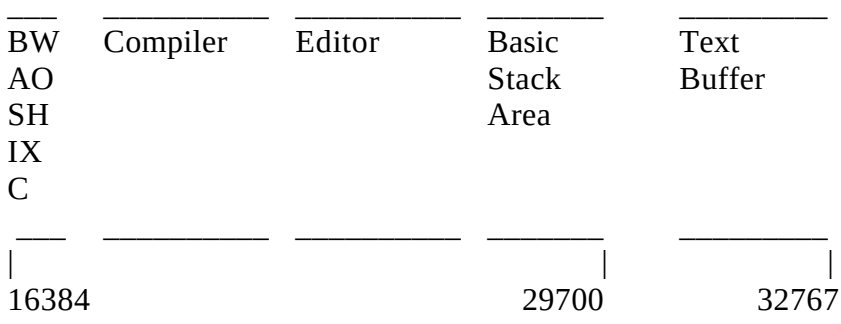
2. People's Pascal Language

It is not the aim of this document to teach the Pascal programming language.

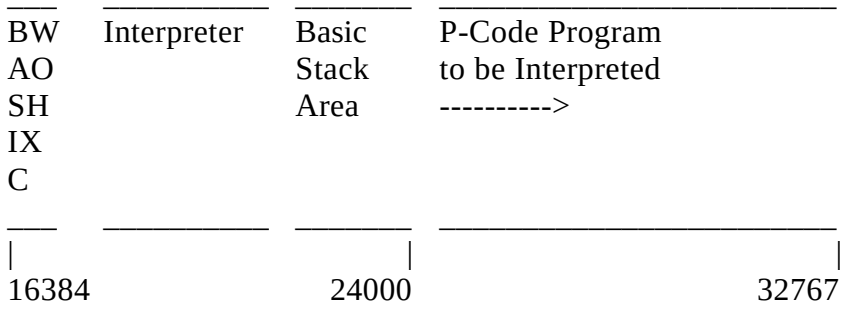
TRS-80 People's Pascal includes the full set of program structuring statements, such as IF, THEN, ELSE, BEGIN, END, WHILE, REPEAT, FOR, CASE, PROC, FUNC, but includes integer and array of integer data types only (16 bit). Refer to the language reference documentation for details.

3. Memory Maps

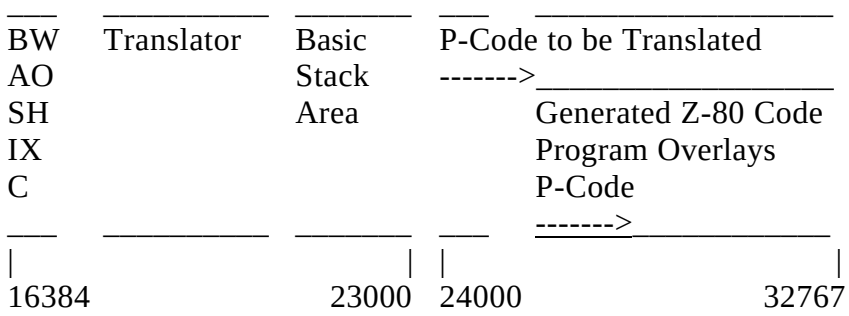
3.1 Editor /Compiler (TPEC)



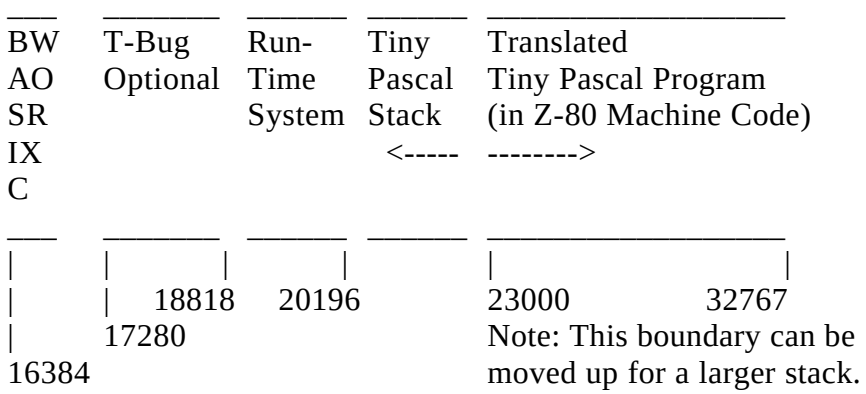
3.2 Interpreter (PPINT)



3.3 Translator (PPTRANS)



3.4 People's Pascal Translated Program at Run Time



4. People's Pascal Stack

The Pascal pseudo machine is stack oriented. For a complete understanding of the system, it is first necessary to understand the P-machine and its stack. The P-machine has two registers, (T) and (B). (T) is the stack pointer, which always points to the top element on the stack. (B) is the base register, which points to (i.e.. holds the address of) a stack location which is the stack base for the "block" (i.e. program, procedure, or function) that is currently executing. The base is used as a reference point for variable addresses.

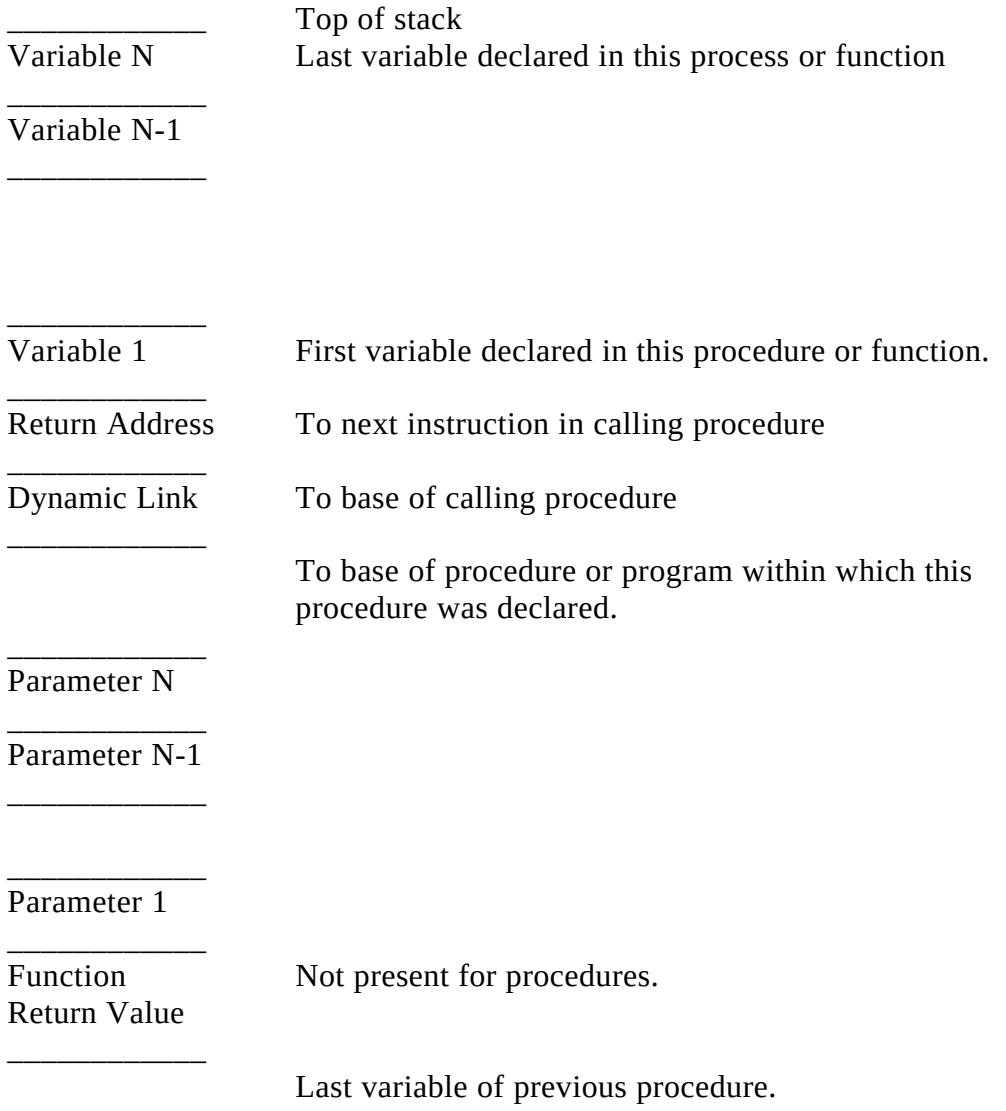
When a block is entered (i.e., when a procedure is called) space for the variables it declares is allocated on the stack in a new "stack frame", which uses the space just above the last-used stack locations.

All variables declared within a block (in a stack frame) are identified within the P-code by an offset from the base of a stack frame, rather than by an absolute address as in some other systems. Variables which were declared in the block which is currently executing can be obtained by adding their offset to the contents of the base register. This forms the absolute address of the variable. On the other hand, variables which were declared in some other block must have their offsets added to the base of the stack frame of that outer block in order to be referenced by their absolute address.

The base of the outer block can be obtained because at the base of each stack frame is a word which contains the absolute address of the previously-entered (outer) stack frame. Thus stack frame bases are linked together in a linked list which descends down the stack to the base of the stack frame of the outermost block (mainline) of the program.

In fact there are two lists: A "static" list, which links stack frames for obtaining variable addresses. This list reflects the lexical structure of the program. i.e., the static nesting of procedure declarations. The second list links stack frames in executation sequence, reflecting the sequence of active procedure calls, at program run time. The second (dynamic) list is used to regain the base of the calling procedure at exit from the called procedure.

5.1 Diagram of People's Pascal Stack Frame



5. The P-codes

P-codes are the machine Language of the imaginary P-machine.

P-codes occupy four bytes each. The first byte is the operation code (op-code). There are nine basic P-code instructions, each with a different op-code. The second byte of the P-code instruction contains either zero or a lexical level offset, or a condition code for the conditional jump instruction. The last two bytes taken as a 16-bit integer form an operand which is a literal value, or a variable offset from a base in the stack, or a P-code instruction location, or an operation number, or a special routine number, depending on the op-code.

5.1 P-code Details

P-code	Hex Op-code	Description
LIT 0,N	00	load literal value onto stack
OPR 0,N	01	arithmetic or logical operation on top of stack
LOD L,N	02	load value of variable at level offset L, base offset N in stack onto top of stack
LODX L, N	12	load indexed (array) variable as above
STO L,N	03	store value on top of stack into variable location at level offset L, base offset N in stack
STOX L,N	13	store indexed variable as above
CRL L,N	04	call PROC or FUNC at P-code location N declared at level offset L
INT 0,N	05	increment stack pointer (T) by N (may be negative)
JMP 0,N	06	jump to P-code location N
JPC C,N	07	jump if C=value on top of stack to P-code location N (C can = 0 or 1)
CSP 0,N	08	call standard procedure number N

Language Description

1. Introduction

People's Pascal is a Pascal subset containing all the program structuring constructs except GOTO, but without many of the data structuring facilities.

2. Pascal Features Not Present in People's Pascal

2.1 Data Types

Integer and array of integer (16-bit) data types are the only data types provided (range -32767 to +32767). Integer arrays may be of single-dimension only. No Boolean, real or CHAR data types. No records, files, or pointer types. No programmer-defined types. No sets.

However, note that integers and integer arrays can be used to store character data, and that single-character literals are accepted, and that a write character string facility is provided, e.g., write ('HELLO THERE'); note also that logical operations are allowed on integers, e.g.

IF A THEN..;
WHILE 1 DO..: (loops forever).

2.2 Program Structure

No GOTO. No statement labels. Structured programming must be used exclusively- this can lead to more easily understood programs. For any function that uses GOTOs, there is another function which performs the same operation without gotos using only sequential, conditional (IF, CASE.) and iterative (WHILE, REPEAT, FOR) structures.

2.3 Parameters Passed by Value

People's Pascal only provides for procedure and function parameters passed by value (i.e. there can be no output parameters from a procedure).

Note that a function can return a value and that procedures can alter global variables (variables declared outside themselves) as alternatives. The second alternative is best avoided where possible to minimize the dependence of the procedure on its environment, and make the program less complex.

2.4 WRITELN

WRITELN is not provided. Use WRITE Instead.

3. Additional Features of People's Pascal

3.1 Access to Memory

A special "built in" array called "MEM" is provided. This array does not need to be declared. The MEM array is mapped onto absolute memory.

The contents of MEM(X) consists of the byte of memory at absolute address X.

E.g. A:=MEM(X); or MEM(30):=0;

This facility is equivalent to Basic PEEK, POKE.

3.2 Access to Routines in Assembly Language

A "CALL" facility is provided to allow the invocation of assembly language (Z-80 code) routines. E.g., CALL(32650) or CALL(RTN).

All necessary registers arc saved by the People's Pascal run-time system before the user's routine is entered and restored on return. If the routine is called from within a procedure, then the procedure parameters can be accessed on the stack by the called routine.

3.3 Format Control on Read & Write

Without format control, when an integer is written, the result will be that the character whose ASCII value is that of the integer will be output.

I.e.,

WRITE(65) well cause the character "A" to be written to the display.
WRITE(13) will cause a carriage return.
WRITE(23) will cause wide characters.
WRITE (28) will home the cursor.
WRITE(31) will clear the screen from the current cursor position onwards.
WRITE(28,31) will clear the whole screen.

Refer to Level-II Basic Reference Manual pages c/1 and c/2. Note that all special control character values can be declared as constants using "CONST". For purposes of standardization, the following names are recommended:

BS	= 8	backspace and erase
LF	= 10	linefeed/carriage return
FF	= 12	top of form- form feed
CR	= 13	linefeed/carriage return
CON	= 14	cursor on
COFF	= 15	cursor off
WIDE	= 23	convert to 32 chars/line - wide characters
CBACK	= 24	backspace cursor
CFWD	= 25	advance cursor
CDOWN	= 26	move cursor down
CUP	= 27	move cursor up
HOME	= 28	home cursor-move to top LH corner of screen
BLINE	= 29	move cursor to beginning of line
ERASE	= 30	erase to end of line
CLEAR	= 31	clear to end of screen

Similarly, a READ will cause the integer being read to be assigned the ASCII value of the input character.

Read and write formatting are provided to override this facility,

A WRITE (X) will cause a number to appear on the screen equivalent to the value of X. READ (A) will cause the input digits to be converted to a 16-bit integer and stored in A. The “#” is the numeric format indication character. I.e.. WRITE(65#) will cause the characters"6" and "5" to be written to the screen at the current cursor location.

3.4 Hexadecimal Constants

Hexadecimal constants are provided for, and are specified by a leading percent sign. Hex constants must contain four hex digits, e.g., %003A, %FFFF.

3.5 Else on Case Statements

An "else branch" is provided on the CASE statement, which will be taken if the CASE variable does not have a value which matches any of the other specified values. Be especially careful not to use spurious semicolons before this statement, or before the end of the case statement. Look at the syntax diagrams carefully.

E.G.:

```
CASE X OF
    1 : WRITE('X EQUALS ONE',CR);
    2 : Write('X Equals TWO',CR)                (*NO  ';' HERE*)
    ELSE : WRITE (' X OUT OF RANGE')            (*OR HERE*)
END   (*CASE*)
```

If you had an extra case element before the 'ELSE' (e.g., 3:... in the above example), remember to put a semicolon on the end of the previous line (e.g., 2:... in the above example).

4. Tips on Programming in People's Pascal

4. 1 Modules

Break the program up into functional components.

Write these components as procedures or functions. If the procedure or function is of general use, then it can be placed in your own library, or into the People's Pascal library.

Try to connect the procedures to the rest of the program by parameters and function return values.

Declare variables and constants only required within one procedure inside that procedure rather than outside it.

Refer to "Structured Design" (Larry Constantine and Ed Yourdon, Yourdon Inc., 1133 Avenue of the Americas. New York NY 10036) for a thorough discussion of functional module design.

Avoid declaring procedures within other procedures.

People's Pascal optimization in the translator has been designed so that "well structured" (structured in the above manner) programs will execute fastest under "fast" optimization and occupy least memory when optimized with the "small" option. Thus there should be no conflict between good programming practice and efficiency.

With a little ingenuity, procedures and functions can be compiled and tested separately, which can speed up the development process.

To compile a procedure or program mainline which uses other lower-level procedures, "dummy" procedure declarations can be inserted before the main block. Dummy declarations consist of only the procedure name and formal parameter declaration followed by a "begin end;" (e.g. FUNC RND(SEED); BEGIN END;)

This will allow, the main block to compile without having to wait for all its procedures to be compiled first.

Of course. to test any procedure, function or mainline. it will usually be necessary to have compiled in all at the procedures that it uses.

The above process of "dummy" declarations currently only applies to the syntax (grammatical) error detection process. However, if lower-level modules are compiled and tested first, then as development proceeds, these lower level modules are always available for testing the next level.

The "$INCL" feature should be used in conjunction with the separate procedure/function concept. A complete procedure or set of procedures can be put into its own file, and "$INCL"uded into the program.

Note that all procedures must be declared before they are referenced (used). This is a one-pass compiler.

4.2 Initialization of Variables

People's Pascal variables are not cleared when they are allocated on the stack. Thus their initial value is unpredictable (will be whatever happened to be in that stack location). Therefore it is important to explicitly initialize variables (e.g., X:=0;).

4.3 Arithmetic and Stack Overflow

No run-time checking for arithmetic occurs, so an estimate of stack-space requirement should be made and sufficient stack space allocated. Arithmetic overflow may be checked for in the interpreter.

5. The People's Pascal Library

The following routines are available:

SET(ONOFF,X,Y); sets the graphic point at location X,Y, if ONOFF is set on. If ONOFF= 0 then point set off.

RND(seed);	returns a pseudorandom number between 0 and 32767. The seed should be set to the returned value, for the next call.
AT(CURPOS);	sets the cursor to position CURPOS on he screen.

6. People's Pascal's Reserved Words

AND	logical AND operator
ARRAY	array declaration
BEGIN	compound statement opening delimiter
CALL	invoke assembly-language routine
CASE	multiple-statement selection
CONST	constant declaration section keyword
DIV	integer divide arithmetic operator
DO	while statement component
DOWNTO	FOR statement component
ELSE	IF and CASE statement alternative branch
END	compound statement delimiter
FOR	iterative (looping) statement component
FUNC	function declaration keyword
INTEGER	integer data type declaration keyword
MEM	memory array keyword
MOD	arithmetic operator giving division remainder
NOT	logical NOT operator
OF	CASE statement component
OR	logical OR operator
PROC	procedure declaration keyword
READ	READ statement
REPEAT	iterative statement keyword
SHL	logical shift-bits-left operator
SHR	logical shift-bits-right operator
THEN	IF statement component
TO	FOR statement component
UNTIL	REPEAT statement component
VAR	variable declaration section keyword
WHILE	iterative statement keyword
WRITE	WRITE statement

7. People's Pascal Special Symbols

NOTE 1 The square bracket characters used in Pascal for array index delimiters are not available on the TRS-80. The round bracket characters are used instead as in Level-II Basic, rather than the Pascal alternative "(." and ".)".

NOTE 2 The squiggly bracket characters used in Pascal for comment delimiters are not available on the TRS-80. The character combinations "(*" and "*)" are used instead.

#	read/write numeric format indicator (WRITE(A#)
$	compiler directive line indicator ($INCL FRED)
%	hex constant indicator (%A04F)
'	character string delimiter (e.g. 'A')
(	arithmetic or logical expression delimiter
(	array index opening delimiter (e.g. AR(30) := 1;)
(*	comment opening delimiter
)	arithmetic or logical expression delimiter
)	array index closing delimiter
*	multiplication operator
*)	comment closing delimiter
:	variable declaration component
:=	assignment operator
=	equal-to operator
-	unary minus and binary subtraction operator
+	addition operator
;	statement separator
<	less-than operator
<=	less-than-or-equal-to operator
,	separator
>	greater-than operator
>=	greater-than-or-equal-to operator
<>	not-equal-to operator
.	end-of-program indicator

8. People's Pascal Operators

+	addition
-	subtraction and unary minus
*	multiplication
DIV	integer division
MOD	remainder after integer division
SHL	logical shift left (can be used for fast multiplication of positive numbers by a power of two)

SHR	logical shift right (can be used for fast division of positive numbers by a power of two)
NOT	logical-NOT unary operator
OR	logical OR
AND	logical AND
=	equal-to
<	less than
<=	less-than or equal
>	greater than
>=	greater than or equal
<>	not equal to

Operating Instructions
Compiler

1. Introduction

The People's Pascal compiler accepts language statements from an edit-buffer and/or cassette files, and translates these into P-code, which is output to an object file on cassette.

The compiler also produces a screen display both of the source code lines and of the generated object code.

P-codes are machine-language instructions for a simplified stack-oriented virtual (or imaginary) machine. These P-codes can either be interpreted by a program which performs the actions expected of the virtual machine, or they can be translated into code for a different (real) machine. In this case the Z-80 microprocessor of the Tandy TRS80 microcomputer. Both of these options are provided in the Pipe Dream People's Pascal implementation.

In addition. the compiler has the option to compile for syntax errors only (no P-code object file being produced), for increased speed and possibly less operator intervention.

The compiler also has the option to produce a listing on a lineprinter if one is attached to the system.

2. Invoking the Compiler

To compile a Peoples Pascal program, it is first necessary to "CLOAD' and run the "PPEC" editor/compiler program. Refer to People's Pascal editor operating instructions. A source program may be typed into the text buffer for compilation, and/or source code may be compiled from cassette using the "$INCL" include-file option. In addition, a source program may be read into the text buffer for compilation and/or editing.

After any required editing of the source program, the compiler can be started with the "C" command. The compiler starts compilation with the first line of source code in the text buffer. If it is required to compile from an existing source file on cassette, then it will he necessary to enter a line such as:

100$INCL /FILENAME/

into the text buffer before compilation, where /FILENAME/ is the name of the People's Pascal source file which is to be compiled.

Note that currently, the "$INCL" compiler option is not nestable. It can only be used as part of a line of source code in the text buffer (i.e., as part of the program mainline).

Also, the line must appear exactly as specified, without any leading or embedded spaces in the "$INCL" statement, apart from the space before the filename.

Note also that the filename supplied is currently required as an operator aid only. No filename checking is performed in the current version.

3. Object File Option (OBJ FILE?)

After initialization of the "C" command, the compiler will prompt with "OBJ FILE?". If the reply is null (just *ENTER*) then no object file will be produced, and the compile will be for syntax error check only. Any non-blank reply to this prompt will result in a P-code cassette object file being produced.

4. Lineprinter Option (LP?)

The compiler will then prompt with "LP?".

If the reply to this question is "Y", then the input source program lines and the compiled object code will be printed on the lineprinter, as well as being shown on the display.

If the reply to this question is null ,(just *ENTER*) or "N", then no lineprinter output will be generated.

For systems without a lineprinter, it is suggested that this prompt be deleted from the PPEC compiler code, to avoid the annoyance of a prompt to which the answer is always the same.

5. Compiler Operation

After initialization, the compiler will proceed to compile the specified source program. On encountering a syntax or (grammatical) error, the compiler will display a diagnostic error message indicating the type of error. The compiler will then return to the editor, to allow the error in the source code to be corrected, and possibly recompiled.

6. Mounting Object Cassettes (OBJ CAS READY?)

If a P-code object file is being produced, then after it has compiled about 50 P-codes, the compiler will prompt for an output cassette for the object file to be written to.

A previously-erased cassette should be mounted and the recorder placed in record mode. When the output cassette is ready, the *ENTER* key may be pressed, and the compiler will write a block of object code to the cassette and proceed with the compilation.

If no input cassette is being used ($lNCL) then no further operator intervention should be required until the compilation is complete. Assuming a successful compilation, the output cassette will contain a P-code object file version of the program, which may be used as input either to the People's Pascal interpreter to test the program, or to the translator to make a final system-loadable version of the program in Z-80 machine language.

If the compiler is accepting input from a cassette file ($INCL), and an object file is being generated, then it will be necessary to periodically swap cassettes and cassette recorder operating modes when the compiler prompts.

This process is made possible by the blocking of both source and object data on cassette. Data is read in or out a block at a time, and the cassette is stopped on an inter-block gap, at which time it may be safely removed and later remounted without any loss of data.

The process of swapping cassettes is somewhat tedious and human-error prone, so be careful. If an additional cassette recorder is available, then both recorders can be mounted in parallel (with extra jack plugs, etc.) with a ganged switch between them.

One can be left with the object cassette in Record mode, and the other with the source cassette in Read mode, and the switch operated between them on prompt from the compiler.

If an expansion interface is available, as well as a second cassette, then the compiler can be simply modified to write its object files to the second cassette, and no operator intervention will be required for the object cassette after initialization.

7. Mounting Source Cassettes (READ CASE1?)

If the "$INCL" option is used, then the compiler will, on encountering the SINCL line, prompt for the required file as follows:

FILE /FILENAME/ REQD-READCAS?

At this point, the appropriate source cassette should to mounted (remove any object cassette first) and positioned just before the start of the file. The cassette recorder should be placed in Read (replay) mode. When everything is ready, the *ENTER* key may be pressed, the compiler will read the first block of source code from the cassette and proceed with the compilation.

Note that the line numbers in the included file bear no relation to the line numbers in the including file.

If an object file is being generated, then it will be necessary to periodically swap cassettes on prompt from the compiler. Cassettes should not be swapped in anticipation, since it is not always possible to predict which cassette will be required first (refer to previous section).

Note that neither object cassettes nor source cassettes should be rewound or otherwise interfered with during compilation. They should be simply ejected or remounted as required by the compiler.

P-code Interpreter /PPINT/
Operating Instructions

1. Introduction

The People's Pascal P-code Interpreter (PPINT) executes Pascal object code (P-code) output from the People's Pascal compiler. The P-code cassette file output from the compiler is read into memory and then interpreted under operator control. Various debugging commands are provided, such as the setting of breakpoints, to allow monitoring of program execution.

The interpreter currently is written in Basic. A faster version will soon be available written in People's Pascal. PPINT is not intended for normal running of programs. Once a program has been debugged, it will be translated to Z-80 machine code with the People's Pascal translator for fast execution.

2. Running the Interpreter

Required operator responses are underlined. Here is what you see on the screen:

READY
/SYSTEM
.? /0
MEMORY SIZE? 24000

Mount the PPINT program cassette for read.

READY
/CLOAD

Wait until PPINT has loaded.

/RUN
TRS-80 TINY PASCAL INTERPRETER PIPE DREAM SOFTWARE
P-CODE. START ADDRESS (24000)?

The default of 24000 is shown. If this is OK then just press ENTER, otherwise enter the address you want to use. This is the address that the P-code program will be loaded at, and address of the first P-code instruction for the "R" (RUN) command. In the current version, this address cannot be less than 24000.

READ IN P-CODES (Y)?

The default of Y (yes) is shown. If ENTER is pressed, then the interpreter will read in the P-code program from cassette. If the reply is "N", then it is assumed that the P-code program is already resident in memory and the next question will be bypassed.

MOUNT P-CODE INPUT CASSETTE ON CAS1?

Mount the P-code object file output by the compiler on the cassette deck and press play.

Press ENTER on the keyboard when cassette is ready.

The P-codes will he read into memory. and some "ADD AT" forward-reference fixups should appear on the screen.

INT/

The interpreter is now ready to accept program execution and monitoring commands described below.

3. Interpreter Commands

Interpreter commands consist of single-letter mnemonics terminated by *ENTER*. Some commands will result in further prompting.

3.1 Run Program -R

Initializes the program-counter and runs the program.

3.2 Single Step -S

Executes the next P-code instruction and returns to command mode.

3.3 Go On -G

Continues the program from the current program counter location. The program may have stopped at a break-point, or after a single-step (S) command.

3.4 Display Program Status -X

Displays the program counter (P). The base register (B) and the stack pointer (T) of the Pascal P-machine (which the interpreter is emulating). The top two stack locations also are displayed.

3.5 Display Program Trace -T

Displays the last few P-code instructions executed by the P-code program, in time-of-execution sequence.

3.6 Display Stack Locations -K

Prompts for a stack location (offset from start of stack) and displays six stack locations starting from this point.

3.7 Set Breakpoint -B

Prompts with the breakpoint number for a breakpoint address (P-code program location). When the program counter reaches any value equal to a breakpoint value, the program will be stopped before the execution of the instruction at that location. The status of the program and its variables may be examined. The program may be continued with the "G" (GO) command or with the "S" (SINGLE STEP) command.

3.8 Clear Breakpoints -C

Clear all the breakpoints set by the "B" command.

3.9 Display Breakpoint Location -Y

Displays breakpoint locations previously set with the "B" command.

3.10 Examine P-code Location -E

Prompts for a P-code location, which is loaded into the P-code location display pointer. The P-code instruction at this location is displayed.

3.11 Examine Next P-code Instruction -N

Increments the P-code location display pointer and displays the P-code instruction at that location. Note: if this instruction has been used once, it is only necessary to press *ENTER* to repeat it, stepping on to the next location.

3.12 Examine Last P-code Location - U

Decrements the P-code location display pointer and displays the P-code instruction at that location.

3.13 Quit -Q

Exit from interpreter.

4. P-code Instructions

POP X means remove the top element of the stack and load it into X (the stack is now one smaller).

PUSH X means place the value of X onto the top of the stack (the stack is now one bigger).

LIT	0,NN	Literal: PUSH NN
OPR	0,0	Process and Function return operation
OPR	0,1	Negate: POP A PUSH -A
OPR	0,2	Add: POP A POP B PUSH B + A
OPR	0,3	Subtract: POP A POP B PUSH B – A
OPR	0,4	Multiply: POP A P0P B PUSH B * A

```
OPR     0,5     Divide:
                POP A
                POP B
                PUSH B/A
OPR     0,6     Low Bit:
                POP A
                PUSH (A and 1)
OPR     0,7     Mod:
                POP A
                POP B
                PUSH (B MOD A)
OPR     0,8     Test Equal:
                POP A
                POP B
                PUSH (B=A)
OPR     0,9     Test Not Equal:
                POP A
                POP B
                PUSH (B<>A)
OPR     0,10    Test Less than:
                POP A
                POP B
                PUSH (B < A)
OPR     0,11    Test Greater Than or Equal:
                POP A
                POP B
                PUSH (B>=A)
OPR     0,12    Test Greater than:
                POP A
                POP B
                PUSH (B>A)
OPR     0,13    Test Less than or Equal:
                POP A
                POP B
                PUSH (B<=A)
OPR     0,14    Or:
                POP A
                POP B
                PUSH (B OR A)
                NOTE: these are the logical operators OR, AND and NOT)
OPR     0,15    And:
                POP A
                POP B
                PUSH (B AND A)
```

OPR	0,16	Not: POP A PUSH (NOT A)
OPR	0,17	Shift Left: POP A POP B PUSH (B shifted left by A bits)
OPR	0,18	Shift Right: POP A POP B PUSH (B shifted right by A bits)
OPR	0,19	Increment: POP A PUSH A+1
OPR	0,20	Decrement: POP A PUSH A-1
OPR	0,21	Copy: P0P A PUSH A PUSH A
LOD	L,D	Load: Load A from (base of level offset L)+D PUSH A
LOD	255,0	Load byte from memory address which is on top of stack onto top of stack: POP address Load A with byte from address PUSH A
LODX	L,D	Indexed Load: POP index Load A from (base of level offset L)+D+ Index PUSH A
STO	L,D	Store: POP A Store A at (base of level offset L)+D
STO	255,0	Store in Memory: POP A POP address store low byte of A at address
STOX	L,D	Indexed Store: POP Index POP A STORE A at (Base of level offset L) + D + index
CAL	L,A	Call procedure or function at P-code location A, with base at level offset L

CAL	255,0	Call procedure address in memory: POP address PUSH return address JUMP to address
INT	0,NN	Add NN to stack pointer
JMP	0,A	Jump to P-code location A
JPC	0,A	Jump if true: POP A IF (A and 1) then jump to location A
CSP	0,0	Input 1 character: INPUT A PUSH A
CSP	0,1	Output 1 character: POP A OUTPUT A
CSP	0,2	Input an integer: INPUT A# PUSH A
CSP	0,3	Output an integer: POP A OUTPUT A#

CSP	0,8	Output a character string:
		POP A
		FOR I:=1 TO A DO
		BEGIN
		POP B;
		OUTPUT B;
		END

NOTE: the result of a logical operation such as (A=B) is defined as 1 if the condition was met and 0 otherwise.

P-Code to Z-80 Code Translator
/PPTRANS/

Operating Instruction

1. Introduction

The People's Pascal translator program translates P-code object files output by the People's Pascal compiler (PPEC) into Z-80 microprocessor machine language object programs which can be saved with T-bug onto cassette, and loaded under the "system" command.

The translator has two optimization options. Z-80 code object programs can be optimized for speed, in which case the program occupies about the same space as the P-codes. Alternatively, object programs can be optimized for minimum memory usage, in which case the program occupies about half the memory but runs at about half the speed.

The P-code object program is read into memory. The normal starting address is 24000 (decimal).

After two passes of the P-code to generate a sorted table of P-code jump destinations and their corresponding Z-80 code addresses, Z-80 code is generated and stored in memory, normally starting at address 23000.

If the Z-80 program is large enough then it will overwrite the early portion of the P-code program, which is no longer required. The end of the Z-80 program cannot "catch-up" with the end of the P-code program in a 16 K machine.

For a larger memory, the P-code may be started at a higher address.

Because of the size of the PPTRANS program, Z-80 code cannot be stored at addresses lower than 23000. This leaves about 9.5K bytes for the Z-80 program.

Note that some of the memory below 23000 is used by T-bug and the People's Pascal run-time system (PPRUN). The rest is available for user-generated assembly-language subroutines callable from People's Pascal and/or for stack space - refer to memory maps.

2. Choosing Addresses

Normally, the default addresses shown below are satisfactory for translated People's Pascal programs. However, the translator provides the option to specify other addresses for exceptional cases, such as where a program has a very large stack requirement (i.e., greater than 3 K due to large arrays or use of recursion).

To obtain a larger stack, if the program itself is not too large, then 32500 may be used as the stack address. If the program is large, then the program itself may be created at a higher address than 23000. Note: this may involve reading-in the P-codes at a higher address also, and the now-larger space beneath the program used for stack. Note that this alternative is less desirable since the total amount of memory to be saved onto cassette is correspondingly larger as the runtime system is at a fixed location, and thus the program takes longer to load.

3. Operation

Note: Operator responses are underlined.

1. **READY**
   > SYSTEM

2. ***? /0**

3. **MEMORY SIZE? 23000**

   now mount PPTRNS program cassette

4. **READY**
   >CLOAD

   wait until load is finished

5. **READY**
   >RUN

6. **TRS-80 PEOPLE'S PASCAL TRANSLATOR**
   DEFAULT REPLIES TO PROMPTS ARE SHOWN IN BRACKETS: (parentheses):

   P-CODE START ADDRESS (24000)? < ENTER>
   or your address

7. **Z-80-CODE START ADDRESS (23000)? < ENTER>**
or your address

8. Z-80 STACK ADDRESS (GROWS DOWN) (22999)?
< ENTER> or your address

9. **OPTIMIZATION (F=fast, S=small) (F)? >ENTER< or S**

10. **DISPLAY CODES (Y)? < ENTER > or N**

pptrans runs faster if codes are not displayed

11. **PRINT CODES (N)? <ENTER> or Y**

12. **MOUNT P-CODE INPUT FILE ON CAS1 AND TYPE "RUN"**

Now rewind and remove PPTRANS program cassette and mount P-code object program cassette for input, type "RUN" and "RUN". The translator will read in the P-code cassette, and perform translation. Wait until translation is complete, with the ratio between P-code and Z-80 code, etc., being displayed on the screen.

13. Note the last-used Z-80 address. The translated Z-80 program is now residing in memory starting at address 23000, or your chosen address, at which this code will be executed.

14. Note the last address used by the Z-80 program which is displayed on the screen. Rewind and remove the P-code input cassette. Mount the run-time system object cassette.

15. **READY**

> SYSTEM

The run-tine system will be read into memory. Now memory contains your translated program and the Peoples Pascal run-time system. When loading of the run-time system is complete, rewind and remove the run-time system cassette.

17. At this point it is possible to run the program via the system command. However, it is wise to save two copies of the program first, using T-Bug. To do this, mount the T-Bug cassette for input.

18. ***? TBUG**

    The Tandy T-Bug program will be loaded into memory. At this point, memory contains your program. the run-time system and T-Bug. Using T-Bug, it is possible to make a copy of your program and the run-time system, with or without a copy of T-Bug. This copy will be loadable under the "SYSTEM" command. Rewind and remove the T-Bug cassette. Mount a blank cassette for output.

19. ***? /**

20. *** P 4380 XXXX 4A00 YYYYYY (for program with T-Bug)**

    OR

    *** P 4980 XXXX 4A00 YYYYYY (for program without T-Bug)**

    Where XXXX is the last-used address of the Z-80 program in hex noted after step 13, and YYYYYY is the filename to be assigned to the program on cassette.

21. Wait until T-Bug "I" command is complete, then reposition output cassette and repeat step 20 as many times as required. Rewind and remove the blank cassette and write on it the program name and the cassette tape counter locations of each copy of the program. The date can also be useful.

22. *** J 4A00** *To run the program if required.*

4. Running Translated Pascal Programs

Normally, all that is required to run a People's Pascal program is to load it under the “SYSTEM" command and to run it by typing “/” after it has been loaded into memory.

This causes control to be passed to the address which was specified as the program entry point on the cassette tape file. This normally will be hex 4A00 (decimal 18818), which is the standard entry point of the People's Pascal run-time system.

There is an alternative entry point to the run-time system, hex 4A0E, which allows the user to initialize the People's Pascal stack pointer at other than the fixed value, and/or to run a program which does not start at the standard address (59D8 hex or 23000 decimal).

When entered at this point (4A0E) the run-time system will prompt for these values.

T-Bug may be used for debugging translated programs although it is usually easier to use the P-code interpreter to find bugs.

To use T-bug. type "/17280 after loading the program rather than just “/”. This is the T-Bug entry point. Refer to Tandy T-Bug operating instructions for further help in using T-Bug.

Program Documentation
Text Editor

1. Introduction

The PP editor is a line-oriented text editor using line numbers for text identification. Intra-line editing is not supported in the current version. Lines are stored in a reserve area of high-address memory called the text buffer. The program is written in Level-II Basic with machine language subroutines to move text up and down in the text buffer for speed efficiency.

2. Commands

People's Pascal commands are:

C	COMPILE	Pass control to the compiler.
D	DELETE	Delete line number range.
E	EOF	Write end-of-file mark to cassette file.
F	FREE	Free bytes left in text buffer inquiry.
L	LIST	List lines in text buffer on screen.
N	NUMBER	Renumber lines of text in the buffer.
P	PRINT	Print lines on the line printer.
R	READ	Read block(s) from a cassette file.
W	WRITE	Write line(s) of text to cassette.

3. Record Formats

3.1 Line in Text Buffer

Byte 0	Length of line (0-255) including self and line number bytes.
Byte 1,2	Line number of this line in binary 0-32767 (1-32766 available for user)
Byte 3-N	Text of line

3.2 Text Buffer

Line 1	Dummy line, text = " ", line number = 0
Lines 2 to N-1	actual lines of text
Line N	Dummy line, no text, line number = 32767
Top byte	Buffer has been initialized flag (14= has, any other value = has not)
Top Byte-2-1	Saved copy of FA variable. This is the only variable that needs to be saved over a run command. (FA = address of last byte used in text buffer.

3.3 Line Record in Cassette File Block

Byte 0	Length of text of line in bytes. If length would be equivalent to certain ASCII characters such as quote ("), then one space is added to line and length is incremented by 1 to avoid trouble with Level-II Basic I/O.
Bytes 1 to M	Line number in ASCII numeric characters (0 <M < 6)
Bytes M to N	Text of line

3.4 Cassette File Block Format

Maximum Size - 240 characters

Byte 0	Quote symbol (') to hold block together through Level-II Basic 1/0
Bytes 1 to N	Lines of text

4. Program Variables

L$	Current line
LN	Current line number
LG	Length of a string
V	Varptr of a variable
W	Varptr of a variable and temporary variable
X	16-bit number (work variable)
P	Pointer to (holds address of) current line record in text buffer
TA	Top address (32767)
FA	Address of last byte used in text buffer
SA	Address of start of text buffer
ML	Maximum line number allowed (32767)
YY$	Used in sneaky transfer of line from text buffer to L$
LM	Last cassette I/O mode (read/write source/object cassette)

C M	Current Cassette I/O mode
BR	Bottom of line number range
TR	Top of line number range
A$	Temporary string variable
QL	Line number of current line in text buffer
BL$	String to hold cassette source file I/O block
B$	Temporary string variable
PO	Pointer to last (old) current line in buffer
Q1	Text buffer move parameter- source address
Q2	Text butter move parameter - destination address
Q3	Text butter move parameter - byte count (HL=Q1, DE=Q2, BC=Q3, FOR LDIR, LDIR Z-80 instructions)
z8$	String variable used to hold Z-80 code subroutines executed via USR (0)

5. Program Routines

Note: The program has been renumbered from 1 with an increment of 1 to reduce its memory size.

Because of the long lines allowed (256 bytes) in Level-II Basic, additions and changes are still possible.

if the program is renumbered, then lines 1-30 should still start at 1 In increments of 1 to avoid undue expansion.

It is suggested that other lines be renumbered so that the new numbers equal the old numbers times 10, so patches etc. can still be applied, yet the new version still resemble the old.

194-201	Once only program initialization
203	Input command prompt (mainline loop)
204	Command interpreter
205	Line insertion/deletion
209	Extract line number (LN) from line
211	Position P pointer at line with line number LN in text buffer
213	Decode line number range in L$ into BR and TR
218	Display line on screen/printer
219	Interpret line number range and find first line in text buffer
220	List (L command) routine
222	Delete (D) routine
224	Write (W) routine
230	Write end-of-file mark (E) routine
231	Re-number (N) routine
234	Write block to cassette (BL$)

236	Write line to cassette (append line to BL$)
242	Read (R) routine
243	Put lines from current block (BL$) into text buffer
244	Read a block (BL$) from cassette
247	Spilt next line from BL$
248	Point to next line in text buffer
250	Copy current line in text buffer into L$, LN
251	Insert (replace, delete) line in text buffer
254	Delete line from buffer
256	Restore FA from reserved high memory after a run command (which wipes all variables)
257	Save value of FA In reserved high memory
258	Put low, high byte of X Into Z8$
259	Set up Z-80 machine language move routine for text buffer and execute this routine by obtaining address via VARPTR for USR(0)
260	Execute Z-80 machine language routine in Z8$ via USR(0)

Tiny Pascal Compiler
Program Documentation

1. Introduction

For a full discussion of the principles of operation. of his compiler, refer to "Byte magazine, October, 1978, "A Tiny Pascal Compiler - Part 2: the P-Compiler", by Kin-Man Chung and Herbert Yuen. This program is largely based on the program listed in that article, but recoded in Level-II Basic and optimized for minimum memory usage.

The compiler is a one-pass compiler using a technique called recursive descent. Tandy Microsoft Level-II Basic is used recursively.

The compiler has its own stacks. one for strings and the other for numeric variables. For maximum speed and memory efficiency, all numeric variables are declared to be of integer type.

In effect, to compile a program, the compiler simply follows the syntax diagrams (railroad diagrams) of the language, deciding which route to take by looking at the source program text, and emitting object code like smoke as it goes.

One disadvantage of the compiler is that it does not have the ability to recover and continue after an error in the source program. To provide this facility would increase the complexity of the compiler, and thus its memory requirement, cutting in to the size of the text buffer, or the ability to correct source program errors without having to load in a different program for editing.

2. Program Variables

T$()	Symbol Table - Identifier name string array
S()	Stack-Compiler's number stack
S$()	Stack- Compiler's string stack
T1()	Symbol Table - Absolute program lexical level at which identifier was declared
T2()	Symbol Table-Value if constant, or displacement from base if variable table, or P-code location if process or function
T3()	Symbol Table-Array size for array, else number of parameters for process or function identification
S9	Numeric Stack Pointer
P8	String Stack Pointer
M$	P-code Operator Mnemonics string values

W0$	People's Pascal Reserved Words string values
T0	Maximum Number of Symbols (size of symbol table) checked for
FL	Nested File Level for "$INCL" (max 1 in current version)
T1	Pointer into Symbol Table Arrays T1(). T2(), T3()
K1	Number of Parameters in previous process, function
OF$	Object File Flag -Non-null => object file to be produced
OB$	Object File Cassette output block area
BZ	Pointer into OB$ object file block area
N0	Number of Reserved Words in W0$
N1	Maximum Value of an integer
N2	Length of Identifier
I$	Constant String of value "IDENT"
Y9	
LN	Current Program Line Number
L$	Current Line of program text
CI	Character Pointer into L$
X$	Current Character of Program Text (also used to hold "expected" in error section)
R	Strung Value of Next Token expected by the compiler
E	Error Code Number
U,V,W	P-Code Generation- Parameters to code-generation routine: U-opcode, V-relative level, W –value
O	String Variable containing next program token (also used to hold "missing" in error section)
ML	Maximum Program Line Number
BL$	Cassette Input file block area
CM	Current Cassette I/O Mode (refer to LM)
I,J,K	Temporary Loop and work variable
A$	Next Program Text Token, returned by scanner
T	ASCII Value of X$
B$	Temporary String Variable
Z$	Temporary String Variable
N3	Value of Token for "NUM" type tokens
C$	Value of a String Literal
K$	Symbol Table Entry Type - C-constant, A-array, P-process, Y=function. V-Variable
Y$	Temporary String Variable to hold parameter to be pushed onto, or having been popped from the string stack S$ ()
TT$	Temporary String Variable to hold symbol table entry type (refer K$)
X	Temporary Variable to hold value to be pushed onto or to be popped from number stack S()
K2	Procedure or Function Call - Number of actual parameters
K3	Procedure or Function Call-Index of entry in symbol table
C1	P-Code Location Pointer

I1	Case Statement-Number of case labels
I2	Case Statement-Number of nested case statements
F9	Flag 1=TO, 0=DOWNTO; also 1=parameters. 0=no parameters
D0	Pascal Stack Location Holder
L1	Absolute Static (lexical) level of procedure or function declaration
CC	Next Byte of P-Code to be output
N4	VARPTR of W
LG	Address of OB$ (output block area)
LM	Last Cassette I/O mode (refer CM) 1=Write source file (editor) 2=Write object file (compiler) 3=Read source fife (editor) 4=Read source fife (compiler)
LP	Line - Printer Output Flag – l=print, 0=don't print

3. Program Routines

2	Check that current token is as required (R) and issue error message number (E) if not
3	Get next token, cheek that it is as expected and issue error message if not
4	Push X onto numeric stack
5	Pop X from numeric stack
6	Get next character of program text into X$ and ASCII value into T
7	Issue error message number (E)
8	Analyze expression
9	Code generation-output 4-byte P-code specified by V, V, W
10	Get next token
11	Analyze a statement
12	Enter Symbol in A$ into symbol table at position T
13	Generate P-code with V (level offset)=0
14	Push string in Y$ onto stack
15	Pop siring from stack into Y$
16	Analyze array index expression
17	Code Generation-generate variable-level reference portion of P-code
18	Generate OPR P-code (U=1, V=0)
19	Generate LIT P-code (U=0, V=0)
20	Generate P-Code with W=0 and V=0
21	Scan for start of array index expression
22	Scan for left parenthesis
23	Scan for right parenthesis
24	Initialize various compiler variables
25	Start of compiler execution-INIT

26 Compiler mainline-compile block + "." at end, re-run program to clear all variables
28 Check that current token is as required, and emit error message if not
34 Input a new line of source code
35 Initialize $INCL(ude) cassette file input
36 Read line from $INCL file
38 Get next token from source program into string variable O (no dollar ($) for brevity since this is so common)
69 Search symbol table for identifier
70 Analyze constant (CONST) declaration
71 Obtain value of constant
76 Analyze single VAR(iable) declaration
77 Analyze simple expression
83 Analyze term
88 Analyze factor
103 Analyze expression
111 Analyze statement
113 Analyze variable assignment (A:=B)
119 Analyze write statement
124 Analyze read statement
138 Analyze IF statement
140 Analyze compound statement
141 Analyze compound statement (BEGIN…END)
143 Analyze repeat statement
145 Analyze WHILE statement
146 Analyze CASE statement
155 Analyze FOR statement
159 Analyze block
162 Analyze CONST declaration
164 Analyze CONST declaration
167 Analyze ARRAY declaration
170 Analyze PROC declaration
171 Analyze FUNC declaration
177 Analyze BEGIN
181 Code Generation - Output 4-byte P-code to object file
187 Output "Fix up forward reference" pseudo P-code to object file and display
188 Output 1 byte of P-code in CC to cassette output block
189 Output block of object code in OB$ to cassette and reinitialize OB$

NOTES:

Every attempt has been made to reduce to minimum the size of the compiler.

This is the reason for the "jump table" at the front of the program. These short line numbers are used frequently and take less space.

Whenever a subroutine ends with GOSUB XXXX: RETURN, this code has been replaced with GOTO XXXX, which is functionally equivalent, takes less space, but tends to make the program messy to read However, these occurrences are recognizable, with a bit of practice.

The construct RETURNELSERETURN has been used at the end of IF statement lines to avoid the memory overload of using another program line.

Some IF Statements involving the comparison of quoted logical operators, etc., have caused Level-II Basic a few headaches, and will not work without embedded spaces.

The current version of the compiler is combined with the editor program, but these two programs are relatively separate, only sharing certain initialization code, and the routines for finding the next line in the text buffer and copying that line into LN,L$. The cassette source read routine is also shared together with the line unpack routine.

Conversion to Disk

The following tasks would be required/desirable:

1. Separate editor and compiler into two separate programs.

2. Add disk file access capability for editor, compiler, translator and interpreter, for both source and P-code object files.

3. Add capability to write translated Z-80 object code files to disk either as a translator facility, or as an extra program or as an option of the run-tine system.

4. Allow greater depth of nesting of "$INCL" (ude)s.

5. Alter emphasis in compiler from minimum memory requirement to higher speed.

Conversion for Additional Memory

The initialization of TA (top address) would need to be altered from 32767.

Care would be required with address calculation in integer mode when handling addresses over 32767. It might be necessary to use floating-point data types for such variables.

If the text buffer were to be significantly enlarged it would be desirable to use a machine-language routine to replace the Basic routine used to position pointer Pa at the address of the line in the text buffer with a given line number.

This simple function could be easily implemented and would eliminate any apparent delay to most commands.

P-Code Interpreter /TPRINT/
Program Documentation

1. Introduction

The People's Pascal P-code interpreter (PPINT) executes Pascal object code (P-code) output from the People's Pascal Compiler. The P-code cassette file output from the compiler is read into memory and then interpreted under operator control. The program currently is written in Level-II Basic.

2. Program Variables

SZ	Size of stack array for program to be interpreted.
S1	Size of stack at which overflow message is emitted. A little less than SZ.
S()	Stack array.
M$	Holds P-code instruction mnemonics.
PS	P-code start address.
PP	P-code pointer-points to current P-code during read-in from cassette.
Z$	Temporary string variable.
P1	First byte of 4-byte P-code.
P2	Second byte of P-code.
P3	Third byte of P-code.
P4	Fourth byte of P-code.
U	Size of trace array.
BL	Maximum number of breakpoints allowed.
TR()	Trace array stores last few P-codes executed.
BR()	Breakpoint array stores breakpoint locations.
BA	Copy of base for Level L.
B	Base register of current stack frame holds address of base of stack frame of current block.
L	Level offset.
A	P-code second (16-bit) operand.
Z	
T	Stack pointer.
P	Program counter (Holds P-code locations).
ST	Stop execution flag, 0=OK, 1=stop.
P0	P-code location display pointer.
TP	Trace array pointer - circulates around TR() trace array as instructions are executed and stored In TR().
K	
X	P-code address in memory.
N1	16-bit operand.

F	P-code op code.
IX	LODX, STOX indexing flag. 0=not, 1=indexing.
SA	Top of stack 16-bit word.
SB	Top-1 of stack 16-bit word.
M1	Temporary variable.
H	Parameter for hex input/output.
PC	Parameter for hex.
PC	P-code location parameter.
N	Pointer parameter into M$.
I	Temporary variable.
J	Temporary variable.
BP	Number of breakpoints currently set.
CM	Command mnemonic string.
IB$	Input data block from cassette file.
IP	Pointer to next byte in IB$ input block area.
Z9$	String to hold Z-80 machine code routine to read in a block of data from P-code input file.
LN	Length of input block IB$ in characters.
ZZ	Temporary variable.

3. Program Routines

100	Initialization.
1000	Initialization - parameter input.
1013	Cassette file read-in to memory. Forward reference fix-ups output to the P-code object file are fixed up in memory as they are encountered. Pseudo-P-codes 253 and 254 are used to label these items. Pseudo P-code 255 is used as an end-of-program indicator.
9900	Initialization.
20040	Routine to bind the base address corresponding to a given level offset.
20060	P-code program initialization.
20090	“Execute P-code instruction" routine. Ends at line 20680
20120	P-code or op-code branch out depending on value of op-code.
20140	LIT- Execute literal instruction.
20150	OPR-Execute OPR instruction.
20520	LOD Execute load instruction.
20530	STO-Execute store instruction.
20540	CAL- Execute call instruction.

Note: if it is an absolute call, and the address is that of the graphics "SET' routine, then a SET/RESET will be performed instead of. call.

20550	INT- Execute increment-stack pointer instruction.
20560	JMP-Execute JUMP instruction.
20570	JPC - Execute conditional jump instruction.

20580	CSP- Perform CSP function.
20690	Get 2nd P-code instruction operand (16-bit).
20710	Display P-code instruction at location PC.
20760	Check if a breakpoint has been encountered.
20820	MAINLINE-Accept and execute operator commands.
20830	Input command and execute it.
20840-20970	Command interpreter.
30010	Get next P-code from cassette file.
30070	Z-80 machine language routine to read-in a block of data from cassette input file. Level-II basic I/O is bypassed to avoid records being truncated if certain values (e.g. (")) occur in data.
30080	Routine to read Z-80 routine into Z9$.
30100	Fix up forward reference item encountered on cassette input file.
30210	Routine to execute machine language subroutine in Z9$ which reads a block of data into the Level-II basic 256-character I/O buffer at address 16870, and to copy this data into block area IB$.
30300	Routine to call an assembly language subroutine whose address is on top of the Pascal stack, unless the address is that of the graphics "SET” routine, in which case, a level-II Basic SET/RESET instruction is performed instead.

P-to-Z8O Code Translator /PTRANS/
Program Documentation

1. Introduction

The P-code to Z-80 code translator program (PTRANS) translates a P-code program into a Z-80-microprocessor machine language program. The P-code program is input from a P-code object cassette file generated as output by the People's Pascal compiler.

A People's Pascal program, when translated to Z-80 machine language, will typically run about five times faster than an equivalent Level-II Basic program.

The following People's Pascal statements executes in about 5 seconds:

```
FOR I:=0 TO 127 DO BEGIN
FOR J:=0 TO 47 Do BEGIN
      SET(ON,I,J);
END; (*FOR*)
END; (*FOR*)
```

Whereas the equivalent Level-II Basic statements:

```
FOR 1=0 TO 127:
FOR J=0 TO 47:
SET (I,J):
NEXT J:
NEXT I
```

take about 42.5 seconds,

2. Description

The following actions are performed:

1. Initialization - Translation parameters are prompted for and saved, then initialization code is deleted and the program run again. Note: a "CSAVE" after running the program will not produce a viable copy.

2. P-codes are read-in from cassette and stored in memory normally starting at address 24000. Forward reference fix-ups, generated by the one-pass Pascal compiler are fixed up as they are encountered in the cassette file. These forward reference fix-ups ("add X at Y") are stored as pseudo P-codes in the P-code cassette file using op-codes 253 and 254.

3. Pass-1: establish table of P-code jump or call destination locations by looking for JMP, JPC and CAL op-codes; remove duplicates and sort table into ascending P-code location sequence (= ascending Z-80 address sequence).

4. Pass-2: generate Z-80 addresses corresponding to P-code locations in table by translating P-code to Z-80 code and obtaining the length of each Z-80 code.

5. Pass-3: generate Z-80 codes, including correct addresses from table; store in memory normally starting at address 23000, and list-out P-codes with equivalent Z-80 codes in hex and addresses in decimal and hex.

3. Program Variables

DI	Display flag: 0=don't display object code, 1=do.
LP	Print flag: 0=don't print object code, 1=do.
OP	optimization flag: 0=optimization for speed, 1=minimum memory use.
PA	P-code address table-array.
ZA	Z-80 code address table-array.
JT	Run-time system jump-table address.
CO$	P-code op-code mnemonics stored in string.
PS	P-code storage start address.
ZS	Z-80 code storage start address.
PP	Current P-code pointer.
ZP	Current Z-80 code pointer.
ZZ$	Temporary string variable.
P1	Value of first byte of current P-code.
P2	Value of second byte of current P-code.
P3	Value of third byte of current P-code.
P4	Value of fourth byte of current P-code.
P5	Value of third and fourth bytes of current P-code taken as a 16-bit integer.
Z8$	Storage area for bytes of current Z-80 code (the Z-80 instruction's equivalent to the current P-code).
A$	Temporary string variable.

PC	Current P-code pointer.
I	Temporary loop variable.
J	Temporary loop variable.
AN	Actual number of addresses in address table.
K	Temporary variable.
N1	Temporary variable.
CL	Current Z-80 code length in bytes.
AP	Index into address tables PA and ZA.
X	P-code indexed LOD/STO operation (LODX/STOX) flag. Also work variable in initialization.
LT$	P-code literal string accumulation area for CSP 8.
XX	Pointer to address within jump table.
RT	Run-time system routine number.
XL	Low byte of XX (also temporary variable).
XH	High byte of XX (also temporary variable).
P6	-2*P5.
P7	Low byte of P6.
P8	High byte of P6.
HX$	Holds two-character hex string equivalent to one binary byte.
IX	Index into P-code address table.
BY	Byte to be converted to hex.
BH	Four-bit "nibble" of BY.
HB	Hex base. ="0" or “A”.
HX	Four-bit nibble to be converted to hex character.
Z9$	String to hold Z-80 read-cassette machine language routine required to bypass Level-I1 Basic input routine.
LN	Length of block read from cassette (P-code input file).
IB$	Area to hold P-code block read from cassette.

4. Program Routines

1-25	Initialization - input parameters.
27	Prompt for number showing default value. Accept reply and save it.
29	Save a value in high memory.
31-35	Read-in P-code cassette fife.
37	Append "Push HL" Z80-code onto Z8$.
39	Restore a value from high memory after "run”.
41	Further initialization, mainline.
45	Pass 1, mainline.
47	Pass 2, mainline.
49	Pass 3, mainline.
53	Termination, mainline.
59	Scan P-codes in memory for P-code jump destinations and store these in P-code location table.

69 Bubble sort P-code location table.
73 Obtain current P-code into P1, P2, P3, P4, P5.
77 Calculate P5 from P3, P4.
79 Display current P-code on screen.
85 Pass-2: Calculate Z80 addresses corresponding to P-code locations and store in ZA.
97 Pass-3: Generate, display and store Z80 codes.
103 Display current Z80 code on screen.
105 Generate Z80 code corresponding to current P-code.
113 Translate LIT P-code to Z80 code.
115 Translate OPR P-code to Z80 code.
125 Translate LOD P-code.
135 Translate S'I'O P-code.
143 Translate CAL P-code.
149 Translate JMP P-code.
159 Translate JPC P-code.
165 Translate CSP P-code.
173 Convert XX to XL and VH low and high bytes.
175 Append A "CALL XX" Z80 code to Z8$.
177 Put a "LA HL, [P6]" Z80 code into Z8$.
181 Put a "LD HL, [P5]" Z80 code into Z8$.
183 Put a "LD L, (IX + [P7])
LD H, (IX + [P7 + 1])
PUSH HL" Z80 code into Z8$
185 Put a "POP HP
LD (IX+[P7]), L
LD (IX+[P7+1]),H" code unto ZS$.
187 Calculate P7 from P5.
188 Append a
"LD A, [P2]" Z80 code into Z8$.
191 Append a
"JP XX" Z80 code to Z8$
193 Find Z80 address in table ZA corresponding to P-code location held in P5 by looking up this P-code location in table PA (linear search).
195 Look up P-code location held in P5 in table PA.
199 Display Z80 code in Z8$ in hex plus current Z80 address in decimal and hex and store Z80 code in memory at current Z80 address.
209 Convert binary byte in BY to two hex characters in HX$.
211 Convert 4-bit nibble in HX to hex character and append to HX$.
215 Store a 4-byte P-code at the current P-code location.
217 Display the contents of the PA and ZA tables.
219 Get next P-code from cassette in P1, P2, P3, P4.

231	Z80 machine language routine to read a block of P-codes from the cassette input file into the Level-II Basic I/O buffer area.
233	Routine to read Z80 machine language routine into Z9$.
237	Routine to apply forward reference fixup "pseudo-P-codes" (op-codes 253 & 254) to P-code in memory as these are encountered on the P-code cassette input file.
239	Routine to execute the Z80 machine language cassette-read routine held in Z9$ with the USR(0) function, and transfer the data read into the string area IB$.

5. Z-80 Codes Generated for Each P-Code

Note: IX register is used for Pascal current stack (B) base register. SP is used for stack pointer (T). HL is used for argument register, A is used to hold level offset.

5.1 Optimizing for Speed

MNEMONIC	OPERATION	Z80-Code
LIT 0,NN	load literal onto stack	LD HL,NN PUSH HL
OPR 2	add operation	POP DE POP HL ADD H,DE PUSH HL
OPR 19	Increment operation	POP HL INC HL PUSH HL
OPR 20	decrement operation	POP HL DEC HL PUSH HL
OPR 21	copy top of stack	POP HL PUSH HL PUSH HL
OPR N	arithmetic or logical operation	CALL OPRN

LOD 0,N	load variable onto stack
	(-64 < N < 64)
		LD L,(IX+[-N*2])
		LD H, (IX+[-N*2+1])
		PUSH HL

LOD 0,NN	load variable onto stack	LD HL,NN CALL LOD

LOD L,M	load Level L variable onto stack	LOD HL,NN LD A, L CALL LOD1
LODX 0,M	load current level indexed (array) variable onto stack	LD H,M CALL LODX
LODX L,M	load Level L indexed variable onto stack	LD A, L CALL LODX1

STO 0,N	store current level variable from top of stack (-64 < N < 64)	POP HL
		LD (IX+[-N*2]),L
		LD (IX+[-N*2+1]),H

STO 0,NN	store current level variable from top of stack	LD HL,NN CALL STO

STO L,M	store level L variable from top of stack	LD HL,NN
		LD A,L
		CALL STO1

STOX 0,M	store current level indexed (array) variable from top of stack	LD HL,M CALL STOX

STOX L,M	store level L indexed variable from top of stack	LD HL,M
		LD A,L
		CALL STOX1

CALL 0,M	call procedure or function at	CALL CAL JP [Z80 ADDR] P-code loc. M
CALL L,M	call procedure or function declared at level L	LD A,L CALL CAL1 JP [Z80 ADDR]
CALL 255,0	call machine language subroutine	CALL CALA
JMP 0,M	Jump to P-code location M	JP [Z80 ADDR]
JPC 0,M	Jump if condition false to P-code location M	POP AF JNC [Z80 ADDR]
JPC 1,M	Jump if condition true to p-code location M	POP AF JC [Z80 ADDR]
CSP 0,N	call standard procedure number N	CALL CSPN

INT -1	adjust stack	POP BC
INT –2	pointer	POP BC
		POP BC
1NT –3		3 X POP BC
INT 1		DEC SP
		DEC SP
INT 2		4 X DEC SP
INT M		LD HL, [-M*2]
		ADD HL,SP
		LD SP,HL

5.2 Optimizing for Minimum Memory Use

The same code as above is produced except as follows:

mnemonic	operation	Z-80 Code
LIT 0,N	load small positive	RST 4; RESTART 4
	literal onto stack	RESTART 4
	(0 <= N < 256)	DEFB N
LOD 0,N	load variable	RST5;
	with small offset	RESTART 5
	at this level	DEFB –2*N
	(-64w<64)	
STO 0,N	store variable	RST 6;
	with small offset	RESTART 6
	at this level	DEFB -2*N
	(-64<N<64)	
LOD 1,N	store variable	RST 7;
	with small positive	RESTART 7
	offset at	DEFB -2*N
	one level higher	
	(0<=N<128)	
STO 1,N	store variable	RST 1;
	with small positive	RESTART 1
	offset at one	DEFB -2*N
	level higher	

NOTE: In order to make the best use of the minimum-memory option, the programmer may use the following techniques:

1) Do not declare procedures within procedure. All procedures should be declared at the outermost block Level. (This rule will also make programs run slightly faster, and is quite sensible from a human point of view, as well as being compatible with the single level of the $INCL

compiler option. Usually there is no need to declare procedures and functions at any other than the outermost block level.

2) Declare all single variables before declaring any array variables. This will generally ensure that all variables have an offset of less than 64 stack locations from the base and therefore allow the translator to make use of the "small" option. The size of the offset of array variables does not matter.

Program Documentation
Run-Time System

1. Introduction

The People's Pascal run-time system provides subroutines which are called by translated People's Pascal Programs. Subroutine are provided for such functions as multiply and divide, keyboard input, etc.

Code for these functions could be inserted “in-line” into the program by the translator, but then People's Pascal programs would be very large. In general, the factor which decides whether a given function should be performed in-line or as a subroutine, is the size of the code required to perform the function. The larger the code is, the more economical it is to have only one copy of it as a subroutine, and the less the proportional overhead in execution time of the actual subroutine call and return instructions against the code executed to perform the function.

The run-time system is entirely self contained apart from two Level-II Basic routines which are used to input a character from the keyboard and to output a character to the screen. To convert to computer such as the Sorcerer, it should only be necessary to provide the equivalent of these two routines.

As well as providing subroutines, the run-time system is entered initially when a People’s Pascal program is run. Certain initialization functions are performed before control is passed to the program.

2. The Jump Table

Most subroutines within the run-time system are accessed via a jump table included in the run-time system. This allows modification of subroutine locations within the system without modifying the addresses of the subroutine entry points.

This also allows modifications to the run-time system without modifying the translator program or previously-translated programs, providing of course that the jump table itself is not moved. Also, subroutine entry routines within the jump table are at a constant offset from the starting address of the jump table. Thus if ever the jump table is moved, (re-assembled with a different origin), then the only parameter to be changed to the translator is the address of the start of the jump table (JMPTAB).

3. Restart (RST) Instructions

An exception to the use of the jump table is the use of RST instructions in People's Pascal programs that have been translated with the minimum-memory usage optimization option. For certain common functions, the restart (RST) instructions (1-byte subroutine calls to fixed low-memory addresses) are used, as follows:

LIT 0,N	(0<=N<256)	RST 4, DEFB N
LOD 0,N	(-64<N<64)	RST 5, DEFB –2*N
STO 0,N	(-64<N<64)	RST 6, DEFB –2*N
LOD 1,N	(0<=N<128)	RST 7, DEFB –2*N
STO 1,N	(0<=N<128)	RST 1, DEFB –2*N

Use of the RST instructions is made possible by the flexible approach taken by Microsoft in designing Level-II Basic.

RST instructions jump to low memory (ROM) addresses, but at these locations, Microsoft has put jump instructions out into RAM locations 4000 hex onwards for RST 1 to RST 7 (RST 0 is not used in this way). These locations at 4000 are set to jump back into ROM, or perform other functions when the memory size question is answered.

On initialization, the People's Pascal run-time system overwrites these locations at 4000 hex with the addresses of the relevant subroutines itself. These addresses are restored when the Level-II keyboard/screen I/O routines are called. This feature is not used by programs optimized for speed.

4. Program Variables and Constants

STK	Slack location used by PPRUN during initialization.
CR	Carriage return code.
KBUFL	Number of characters in keyboard buffer.
KBUFP	Pointer to next character in keyboard buffer.
KBUF	keyboard buffer area (max 64 characters).
RST	Area containing restart table overwrite data. This is copied to 4000 hex on initialization and after keyboard I/O.
NORST	Area containing copy of Level-11 Basic version of restart table. This is copied to 4000 hex on keyboard I/O.
K10	Table of powers-of-ten for binary to decimal conversion for CSP3 (write #).

5. Register Usage

SP	Used for People's Pascal stack pointer (T). It is also used for subroutine return linkage.
HL	Generally, used as an argument register. It is used in code called by RST instructions to hold addresses of trailing arguments.
DE	General purpose.
BC	General purpose.
A	Used to hold relative level offset when not 0. Also general purpose.
IX	Used for People's Pascal base register (B).
IY	Frequently used to save subroutine return address popped from stack at start of subroutine and jumped to at end.

Alternate register - Used In CSP1only.

6. Program Routines

START	Normal initialization entry point,
INAD	Alternate entry point - allows override of stack address and entry of non-standard program start address.
DORST	Overwrites Level-II Basic restart table at 4000 hex.
UNDO	Overwrites 4000 hex with original Level-II contents. Note NORST must be in HL.
LITB	Small literal - only for minimum-memory translation option.
CALA	Absolute memory address call (CALL(MEM)). Note IX register is saved. Any other register can be overwritten, so programmer does not need to worry about destroying register values in his subroutine.
CAL1	Call procedure at non-zero level offset.
CAL	Call procedure at 0 level offset (CAL 0,N).
OPR0	Subroutine return.
OPR1	Negate top of stack (TOS).
OPR2	Add-not currently used-in line instead.
OPR3	Subtract
OPR4	16-bit signed multiply.
OPR5	16-Bit signed divide.
OPR6	Test TOS for odd value.
OPR7	MOD (uses divide, multiply and subtract).
OPR8	Compare equal.
OPR9	Compare not equal.
OPR10	Compare less-than.
OPR11	Compare greater-than or equal.
OPR12	Compare greater than.

OPR13	Compare less-than or equal.
OPR14	OR operation.
OPR15	AND operation.
OPR16	NOT operation.
OPR17	Shift left operation.
OPR18	Shift right operation.
OPR21	Not used – inline instead.
KBIN	Routine to input a line of characters from the keyboard, echoing then to the screen and allowing the delete key to operate if required.
CSP0	Input a character. Calls KBIN to get next character out of input line.
CSP1	Output a character. Also resets KBIN input line pointer and length to zero so next call to CSP1 will cause a new read.
CSP2	Read a number. Calls KBIN to get characters of number. Number is terminated by first non-digit character.
CSP3	Write a number.
CSP8	Output a string of characters. These arc supplied in form of a trailing argument terminated by a null (0) byte. Also clears KBIN input line length and pointer, so next read will cause true input to be done.
M8	Unsigned 16-bit-by-8-bit multiple.
NEGHL	Negate the HI register (internal subroutine only).
LODA	Load from absolute memory address (:=MEM(X)).
LOD1B	Load from small offset at previous level (small option only).
LOD1	Load, level < > 0.
LODB	Load from small offset at current level (small option only).
LOD	Load, Level=0.
LODXI	Indexed (array) load, level <>0.
LODX	Indexed load, level=0.
BASE	Find base register value corresponding to level offset supplied in A register and return base value in BC register.
STO1B	Store to small offset, Level=1 (small option only).
STO1	Store, level < > 0.
STOB	Store to small offset, level = 0.
S TO	Store, level =0.
STOX1	Store indexed (array), level <> 0.
STOX	Store indexed, level = 0.
JMPTAB	Jump table.

7. Special Subroutines

The following two subroutines are used in the Level-II Basic ROM:

0033 hex	Output character in A-register to screen.
002B hex	Try for character from keyboard. A-register will have character if there was one, otherwise A-register will be zero. This is called in a loop until A < > 0.

These are the only external facilities used by the run-time system, and equivalent routines would need to be supplied in their stead for a different micro system. Also, some modification would probably need to be made to the restart system for minimum-memory optimization if this feature was to be retained under a different system.

Editor/Compiler
Operating Instructions

1. Introduction

The People's Pascal editor is a line-oriented editor. Edit commands operate on tines of text in a text buffer, which has room for just over 3,000 characters or 50-200 lines of text, depending on line length. Intra-line editing is not provided.

Lines of text in the text buffer may be:

- inserted from keyboard cassette files
- replaced from keyed-in or tape files
- deleted
- renumbered
- written to a cassette file
- listed on the screen
- printed on the lineprinter
- compiled

Files of any length may be created or edited. PPEC files are not loadable via the "CLOAD" command or the "SYSTEM" command, nor are they compatible with the Tandy editor/assembler. However, they may easily he read by a Level-II Basic program. PPEC cassette flies are blocked for efficiency.

2. Line Numbers

In the line-oriented PP editor, lines of text are identified by line numbers. Lines always occur in line-number sequence both in the text buffer and in cassette files. Line numbers may range from 1 to 32,766.

Many of the editor commands operate on text lines having line numbers falling within a line number range. A line number range is expressed as a starting line number followed by a single dash (i.e.. "-") character, followed by a final line number (e.g. 500-1000).

The following variations of this form are allowed:

A. No final line number, (e.g. 500). In this case, the final line number will default to the value of the starting line number, and the command will operate on that line only.

B. No starting or final line number. In this case, the starting line number will default to 1, the final line number will default to the highest line number allowed (32,766), and the command will operate on all lines in the text buffer.

C. Startling line number replaced by a full stop (full point or " ") character, (e.g. -500). In this case, the starting line number will take the value of the current line number, and the command will operate from the current line to the final line number.

D. Missing final line number, but a dash character present (e.g. 100- or .). In this case the final line number will default to the largest line number allowed, and the command will operate on lines from the starting line number to the end of the buffer.

3. Commands

People's Pascal editor commands consist of a single letter, possibly followed by a line number range or other numeric argument.

The following commands are accepted:

C	COMPILE	Compile People's Pascal program in the text buffer.
D	DELETE	Delete line(s) from the text buffer.
E	EOF	Write an end-of-file mark to the output-cassette file.
F	FREE	Enquire how many bytes "free" (available) in the text buffer.
L	LIST	List line(s) in the text buffer on the screen.
N	NUMBER	Re-number lines in the text buffer.
P	PRINT	Print line(s) in the text buffer on the line printer.
R	READ	Read block(s) from the input cassette file, and insert or replace lines in the text buffer.
W	WRITE	Write line(s) from the text buffer to the output cassette file.

Commands must be typed precisely. Leading spaces are not allowed. Embedded spaces are not allowed between command mnemonics ("C", "D'. "E", etc.) and the numeric arguments. An unrecognized command will cause a message from the editor.

4. Inserting and Replacing Lines

NOTE: The current version of the editor requires that any limes containing the characters “,” or ":” be preceded by the quote sign ("), otherwise the Level-II Basic I/O will truncate the line at the comma or colon, and emit this message: "EXTRA IGNORED”. It is good practice to precede every line containing source code (text) by a quote sign. The quote is "thrown away" by the Level-II Basic input routine. The editor lists lines in alignment with lines typed in this fashion, for the consistent appearance of People's Pascal block-structure indentation of the file on the screen. It is not necessary to precede command lines by a quote, since these never contain a comma or a colon.

A line to be inserted into the text buffer is typed preceded by its line number. If there was already a line in the buffer with this number, then the new line will replace the old line. Otherwise, the new line will be inserted into thc buffer in position according to its line number.

5. Deleting Lines - D

To delete a single line, simply type its line number This line will be deleted from the buffer. To delete several lines, the D command can be used. D alone will delete line 100 (same effect as just typing 100). D100-500 will delete lines 100 to 500 inclusive. Some delay may be noticed when many lines are deleted at once.

6. Listing Lines -L

The L command is used to list lines in the text buffer on the screen. Just L will list all the lines in the buffer. L100 will list line 100. L100-500 will list lines 100 to 500 inclusive. "L." (without quotes) will list the current line. L.- will list from the current line to the end of the text buffer. Note that the ENTER key will cause the operation of the list command to be terminated, with the current line being the last line listed (i.e. L.- will continue the listing again from the point where it was terminated by hitting ENTER.

7. Renumbering Lines - N

The N command is used to assign new line numbers to lines in the text buffer. This can be useful if it is desired to insert text from a cassette file into a given location. Lines can also be moved by saving them on cassette, deleting them, renumbering the remaining lines, and then reading back in the original lines from cassette.

Lines are renumbered starting from a base number until the end of the buffer. Just N will cause all lines in the buffer to be renumbered. N500 will cause line 500 onwards to be renumbered. When the N command has been entered, the editor will ask for the new base and increment. Lines will be renumbered starting from this base with this increment. The new base must be greater than the line number of the line below the lines being renumbered.

8. Reading Text from Cassette -R

Lines on PPEC cassette files are stored as variable-length records in variable-length blocks of up to 240 characters. The R command will cause the next block(s) on the cassette to be read into the text buffer in position according to their line numbers. If there is already a line in the text buffer with the same number as a line being read from cassette, then the old line will be replaced. Just R will read in the next block. R100 will read in the next 100 blocks or stop at the next end-of-file mark, or until the text buffer becomes full, whichever occurs first.

9. Writing Lines to a Cassette File -W

The W command will cause lines to be written to cassette. Just W will cause all the lines in the buffer to be written to the cassette in blocks. W100-500 will cause lines 100 to 500 to be written to cassette. Lines are written as variable-length records in variable-length blocks of up to 240 characters. A "short" block may be written as the last block of a line number range. In addition, a line starting with a "$" sign will always be the last line in its block (Refer to " $INCL" in compiler documentation).

On completion of the write command, the editor will ask whether the lines written out to cassette should be deleted from the text buffer ("DELETE?"). A reply of "Y" will cause these lines to be deleted from the text buffer. Any other reply will leave these lines unchanged in the text buffer. This option is provided for the editing of files that are too big to all fit in the edit buffer at once.

If the write command was a write to the end of the text buffer, then TPEX will prompt with "EOF?"" on completion of the write. If the reply to this prompt is "Y" then an end-of-file mark will be written to the file at this point (same effect as E command).

10. Compile - C

The C command will pass control from the People's Pascal editor to the compiler, which will attempt to compile lines from the text buffer. If it is required to compile a file from cassette, then it will be necessary to insert a line such as "100$INCL FRED" into the text buffer before invoking the compiler. Before compilation commences, the compiler prompts with an "LP?". If the reply is "Y", then the listing output will be printed on the lineprinter rather than being displayed on the screen.

The compiler then prompts with an "OBJ FILE?". If the ENTER key is pressed then no object file will be generated, and the compile will be for syntax errors detection only. Any other reply will cause an object file to be generated. In one cassette systems, where source input is being accepted from cassette, it will be necessary to change cassettes and cassette drive operating modes (play/record) when the compiler prompts. between the source file cassette(s) and the object file cassette.

11. Print on Lineprinter Text Buffer Line(s) - P

This command is used in exactly the same way as the L command, except that the output appears on the lineprinter rather than the screen.

12. Creating, Maintaining Large Files

It is possible to create and maintain cassette files of indefinite length with the People's Pascal editor, even with only one cassette drive, although large files of People's Pascal source code are not recommended (refer to compiler documentation on modular programming).

To create a large file, type lines into the text buffer until it becomes nearly full, mount an output cassette and write the contents of the buffer out with the W command, deleting the text that has been written out. Type more lines into the text buffer, with higher line numbers, and repeat the process until the complete file has been written out. Write an end-of-file mark to the output cassette.

To edit a large file, a new copy of the file is made on a fresh cassette, as follows. Mount the input cassette containing the current version of the file. Read several blocks of the file into the text buffer and edit them. Remove the input cassette and mount the chosen output cassette. Write out the edited lines from the text buffer to the output cassette. Remove the output cassette and remount the input cassette. Read in some more text from the input cassette and repeat the process. Repeat until all input text has been processed (end-of-

file-or "#EOF") encountered, and all edited text has been written to the output cassette. It is advisable to keep one old version of a file, in case the most up-to-date version is lost, accidentally erased, or becomes unreadable. Alternatively, a copy of the current version may be made by using the above process without any editing.

Important Note:

PPEC text files are blocked on cassette. This means that the file consists of blocks of data with "gaps" in between. It is this feature which makes many of the features of the program possible. It is especially important that cassettes that are to be used as output files from the editor should be erased before they are used. Bulk erasure is not necessary. Simply rewind the cassette, set the recorder on manual (disconnect computer remote control) and record over the whole cassette. This can be done any time the recorder is not being used by the computer.