PASCAL VERSION 5.3

BY

T.J. BOURNE

MOLIMERX LTD.

1 BUCKHURST ROAD, TOWN HALL SQUARE, BEXHILL-ON-SEA, E. SUSSEX.

Tel: (0424) 220391/223636 TELEX 86736 SOTEX G

Pascal System Manual
Version 5.3

Table of Contents

Pascal System Manual
Version 5.3

1 Introduction

This system provides the TRS-80 Model I or Model III user with efficient Pascal program development and execution facilities. Unlike most other Pascal systems, it compiles your program directly to Z80 machine code, in standard TRS-80 disk load module format. The user may be interested to know that the Editor and Compiler are themselves written in Pascal.

The system has been tested with TRSDOS 2.3, NEWDOS 2.1, VTOS 3.0, LDOS 5.0.2, DOSPLUS 3.3 and NEWDOS/80 2.0 for the Model I, as well as TRSDOS for Model III. As well as DOS facilities, some TRS-80 ROM routines are used; these are common to Model I and Model III. In these days of cheap RAM the system has been structured to use nearly the full 48K bytes of available RAM, and both the Editor and Compiler require this. However, programs produced by the system will run in as little as 16K if they need little storage of their own.

2 Components of the system

The system consists of the following components:

Editor - a screen-based editor (PASEDIT/CMD) is provided. The compiler will also accept source files produced by either Scripsit or Electric Pencil, so long as each source line is terminated by pressing the Enter key.

Compiler - this is the program PASCT/CMD.

Run-time System - this is provided in loadable form in the file PASCAL/CMD. See also PASCAL/OBJ in section 6.

A sample program FILECOMP/PAS is also supplied to demonstrate the use of files and allow the user to verify that all parts of the system are working. Users are advised to avoid confusion by using the file extensions /PAS, /OBJ and /CMD respectively for source, object and command files.

3 The Pascal language supported

3.1 Definition

Those new to the Pascal language are advised to use one of the many excellent introductory texts which are generally available; particularly recommended is 'Pascal for Programmers' by Eisenbach and Sadler, published by Springer-Verlag, most of the examples in which have been run with this system exactly as they stand. The language definitions given here assume some familiarity with the language, and concentrate on the variations from the full language in this implementation.

The base document for this implementation is the Second Draft Proposal for an International Standard (ISO) Pascal (reference in Appendix A), which was based on the Pascal User Manual and Report, by Jensen and Wirth, published in 1975 by Springer-Verlag, referred to from now on as the Report. The lists of restrictions and extensions which follow refer to the syntax definition in Appendix A of the Draft Proposal. A full and formal definition of the language supported by this system is given in Appendix A. It is understood that the British and International Standards for the Pascal language are or will be virtually identical to the Draft Proposal.

3.2 Restrictions

Records are not supported.
All files are implicitly declared by their appearance in the program header as TEXT. No other file types are allowed.
Procedure and function parameters are not allowed.
File access is only by the standard procedures READ, READLN, WRITE and WRITELN. GET and PUT are not available, and file buffers may not be accessed directly.
No type may be referenced before it is declared.
Conformant array parameters are not supported.

3.3 Extensions

A variety of standard procedures and functions are predeclared, several of which yield more efficient object code than would a Pascal equivalent. See Sections 8 and 9 and Appendix C.
PROCEDURE and FUNCTION may be abbreviated to PROC and FUNC respectively.
The files INPUT (keyboard), OUTPUT (screen) and PRINT (line printer) are predeclared and may be omitted from the program header.
Sequential disk files are supported, as described in section 7.
The CASE statement has an optional ELSE clause which is performed if no case is selected.
Hexadecimal constants may be used in source programs (but not in input data). A hexadecimal constant consists of 1-4 hexadecimal digits (0-9,A-F) preceded by &.

3.4 Other points to note

Since square brackets may not be available on the TRS-80, dotted parentheses ((. and .)) may be used in array references, as required by the standard. Square brackets (ASCII values 91 and 93) may also be used, but will appear as arrows on Model I screens.
Since braces (curly brackets) may not be available on the TRS-80, starred parentheses ((* and *)) may be used to delimit comments. Braces (ASCII values 123 and 125) may also be used, but will not appear on unmodified Model I screens. Since the up-arrow on the Model I has the ASCII value 91, rather than the correct 94, two alternatives are allowed for the up-arrow used as a pointer qualifier: @ (at-sign), as allowed by the standard, or ASCII 94, which appears as a right-arrow on Model I screens.
The data types supported are INTEGER (-32768..32767), CHAR (0..255, the usual TRS-80 character set) and BOOLEAN (FALSE and TRUE), plus subranges of these and user-defined scalar types, and also REAL (equivalent to TRS-80 single precision).
Though much type checking is performed by the compiler, some errors may not be detected. The user is particularly cautioned that invalid operations on real data may cause the ROM routines to fail in various ways, possibly including looping.
Note that GOTO may only be used as permitted by the proposed standard. Broadly, this means that a label at other than the outermost level of nesting within a block, i.e. within any FOR or CASE group, may only be referenced by a GOTO within the statement labelled. The compiler will detect most infringements of this rule; those not detected will cause execution errors. Note also that GOTO to a label which is declared but not defined will not be detected by the compiler, and will cause a run time error.

4 Using the Editor

The Pascal Editor PASEDIT is a screen-based editor which produces and accepts as input standard ASCII text files. Each line has a maximum length of 64 characters and is terminated by a carriage return character, decimal value 13. The entire file is terminated by a character with decimal value zero. This format is compatible with the DOS commands LIST and PRINT and with Electric Pencil, also with Scripsit so long as the ASCII option (S,A) is used.

When the editor is entered it first offers the option of trying to recover a file which is already in memory. This may be useful in case of finger trouble, hardware malfunction and so on. In most cases this recovery is successful if no other program has been run since the editor was last used. If memory has been corrupted the results are unpredictable, so it is worth checking that the beginning and end of text are as expected before proceeding.

The basic principle of the editor is that text is changed as it is seen on the screen, character by character or line by line, according to the key pressed and the current position of the cursor, which flashes alternately with the character over which it is positioned. In the normal mode, pressing a key causes one of the following actions:

Up-arrow - Move cursor up one line on the screen. If already at the top of the screen, scroll the screen down one line, unless already at the start of the file.

Down-arrow - Move cursor down one line on the screen. If already at the bottom of the screen, scroll the screen up one line. If at the end of the file, add a new blank line to the end.

Left-arrow - Move cursor left one space. If already at the beginning of the line, move to the last non-blank character in the previous line.

Right-arrow - Move cursor right one space. If the line is already full, move to the start of the next line.

NOTE: The preceding four keys may be held down to repeat the cursor movement.

Shift up-arrow - Delete the current line.

Shift down-arrow - Insert a blank line before the current line. *

Shift left-arrow - Delete the current character.

Shift right-arrow - Insert a space before the current character.

Clear - Transfer to command mode (see below). *

Any other key - (in normal mode, with small cursor)
Replace the current character by the new one, and move the cursor right one space.

(in insert mode, with large cursor)
Insert the new character at the cursor position, and move the cursor and the remainder of the line right one position.

* NOTE for VTOS and LDOS users - it will be necessary for you to use shift clear instead of just clear, and shift @ instead of shift down-arrow, because these systems intercept clear and shift down-arrow.

After transferring to command mode by pressing Clear or Shift Clear, any of the following commands may be entered. Unless otherwise stated, the text display is unchanged on completion of the command. In each case the first letter of the command is sufficient.

@ - The decimal value of any character may be input. This character replaces the current character, and the cursor moves on one space. This command may be used to input the arrow characters and others not supported by the standard keyboard routine.

Bottom - Move the cursor to the last character in the file.

Chop - Truncate the current line at the cursor position, by chopping off all characters from the cursor (inclusive) to the end.

Divide - Divide the current line at the cursor position. All characters from the cursor to the end become a new line.

End - Leave the Editor. A warning will be given if the file has been changed and not saved.

Find - Find the next occurrence of a specified string. The Editor will prompt for the search string, which should be terminated by Enter. Pressing Enter without entering a search string causes the previous search to continue where it left off. If the string is found, the cursor is left at the end of the line containing it.

Help - Help the user by listing the commands available.

Insert - Switch to insert mode.

Join - Join the next line on to the end of the current line.

blocK - A number of subcommands are provided for manipulating blocks of text, as indicated below. Note that only one block may be marked at any one time.
B - mark the cursor position as the beginning of a block
C - copy the marked block to the cursor position
D - delete the marked block
E - mark the cursor position as the end of a block
F - forget the current block markers

Load - The Editor will prompt for the input file-spec and load the file into the text buffer, overwriting any previous contents.

Memory - The number of free characters in the text buffer will be displayed.

Number - Find the line with a specified line number. The editor will prompt for the number. This command is useful particularly in conjunction with run time error messages giving line numbers.

Overwrite- Switch from insert mode back to normal mode.

Print - The entire file will be printed on the line printer.

Save - The Editor will prompt for an output file-spec (which may or may not be that used by Load) and write the entire text file to it. The text remains unchanged, and may be saved again to a different file for security.

Top - Move the cursor to the first character in the file.

NOTE - if the message "TEXT SPACE NEARLY FULL" is displayed, the text buffer is undamaged and the current file may safely be saved.

If an error is detected in a file specification during Load or Save, you will be given an opportunity to try again.

5 Using the Compiler

A Pascal source program is put into a disk file using the supplied text editor (PASEDIT), Scripsit or Electric Pencil. Each line of up to 99 characters must be terminated by a return character (13), usually by pressing the ENTER key. New page markers may be used and will be obeyed when listing the program.

The compiler is run by typing its file-spec in the usual way. It will prompt for the input and output file-specs, in standard TRS-80 format, and ask what options are required. These are selected by typing their initial letters, in any order. The following options are available:

A - Append the run time system to the compiler output.
C - Compatibility with version 4. Allows parentheses in array declarations and references, also permits 'falling through' a CASE statement.
D - Debug mode - implies options N, R and S.
M - Allow multiple input files (see below).
N - Generate code to output line numbers for error messages.
P - Print the source listing on the line printer.
R - Generate code to check the range of values being assigned.
S - Generate code to check the values of array subscripts.
T - Turn the trace option on initially.
W - Wait if an error is detected.

The default is to do none of these. For example, replying WP would cause the compiler to print the source listing, and to pause after each error until the Enter key is pressed.

When an error is detected, the line in error will be displayed with an up-arrow pointing at the start of the item where the error was detected. This may be past the actual error. If a printed listing is also being produced, the error will be indicated by a line beginning and ending with ***** and the suspect item indicated by an exclamation mark. In many cases the compiler will attempt to "correct" the error and continue.

NOTE that whenever the compiler is run the file YYPASERR/ZYX must be present throughout the compilation process. If the A option is used the file PASCAL/OBJ must also be available for the concluding stages of the compilation.

To allow the compilation of programs larger than can be edited as a single file by the Editor or Scripsit, and to permit the inclusion of standard sets of procedures, multiple input files may be compiled. If the M option is selected, the compiler will ask how many: simply enter the appropriate number. There are no restrictions on where the boundary may be between files, except that of course a new file begins a new line. When the end of each file is reached the compiler will prompt for the next File-spec.

The whole of the compilation process may be automated using chaining if supported by your DOS. A sample program PASJCL to achieve this is supplied, and details are given in Appendix H.

6 Running and debugging Pascal programs

To run a generated Z80 program, it is first necessary to ensure that the run-time system is loaded. This may be done by the command PASCAL, or by forming a loadable program which includes the run-time system (see below). Then the program is run simply by typing its file-spec.

To create a loadable program including the run-time system the procedure recommended for previous versions may be followed. Alternatively, the compiler option A may be used to do the job automatically; in this case the output from the compiler will be a complete executable program.

The break key has no effect when a Pascal program is running, unless code has been included in the program to test for it using PEEK. You could test for BREAK using the statement

```
IF ANDBITS(PEEK(14400),4) = 4 (* BREAK *) THEN action
```

within the main execution loop of your program.

A number of facilities have been provided to help you to debug your programs. These include optional range checking of values used in assignments and array references (R and S options respectively) and optional use of line numbers in run time error messages (N option). The D option may be used to imply NRS.

In addition, a trace facility is available. Any number of values (variables, constants or expressions) may be output in each call of the TRACE procedure, which has the same syntax as WRITE. The output, which begins with an asterisk and the line number, may be directed to any text file. Calls to TRACE will have no effect unless tracing has been turned on; this may be done by the compiler option T or by a call to the TRON procedure. Tracing may be turned off by a call to the TROFF procedure. The TRACEON Boolean function may be called within the program to see whether tracing is in effect.

7 Input and output

Input and output operations may be to or from the keyboard, screen, line printer and up to four sequential disk files in each program. The format is the same in each case.

Output is by the WRITE and WRITELN procedures. The first parameter is optionally a file name; if this is omitted OUTPUT (screen) is assumed. Subsequent parameters are the values to be output, and may be any expressions. The allowable value types and their default formats and field widths are as follows:

Character string constants	-	Length of string
CHAR	-	1 character
ARRAY OF CHAR	-	Length of array
BOOLEAN	-	6 characters, TRUE or FALSE
INTEGER	-	As in TRS-80 BASIC
REAL	-	As in TRS-80 BASIC

The default format may be over-ridden in each case by use of a colon followed by a field width value; any additional spaces will be inserted before the value to be output. If stated, the width must not exceed 99. For example,

```
WRITE(PRINT,J:3)
```

will print the current value of J in a field of three digits - if J is 23 then a space followed by the digits 23 will be printed. Contrary to the standard, this system allows you to output less than a full value. For example,

```
WRITE(345:2)
```

will output just 45.

For real values a second width parameter may be supplied, indicating the number of decimal places to appear in the output.

WRITELN forces a new line after all values in the list have been output, while WRITE does not. PAGE sends a form feed character (12) to a disk file or the printer, or does the equivalent of the BASIC CLS if output is to the screen.

Input is by the READ and READLN procedures. The first parameter is optionally a file name; if this is omitted INPUT (keyboard) is assumed. Subsequent parameters are the variables to which values are to be assigned. For character variables the next character or characters of the input data will always be assigned. For real variables the input data will be scanned for a real constant, optionally preceded by a minus sign and/or including an exponent; leading spaces are ignored. For variables of all other types the input data will be scanned for an integer constant, optionally preceded by a minus sign; leading spaces are ignored.

The EOLN and EOF functions may be tested before an input operation to see whether the input pointer is at the end of the line (carriage return, 13) or end of file respectively. READLN forces the remainder of the current line to be discarded after all variables in the list have been given values. READLN(INPUT) or simply READLN may be used to wait for the ENTER key to be pressed.

To enable a program to test for a key being pressed without having to use READ, the INKEY procedure is provided; this works the same as TRS-80 BASIC INKEY$.

The above applies to disk files as well as to INPUT, OUTPUT and PRINT. However, additional procedures are available for disk files:

TITLE must be used before the first reference to each disk file, to assign a disk file to the Pascal file. There are two forms of this procedure; the first, TITLE(filename), prompts the user to input the file-spec from the keyboard, while the second, TITLE(filename,string-literal-or-variable), allows the program to provide the file-spec. Examples of their use can be found in the supplied programs FILECOMP and PASJCL respectively. TITLE may be used again after a file has been in use, and will then automatically close the previous file before assigning a new one.

RESET or REWRITE are used to close a file (if it was open) and then open it for input or output respectively. The appropriate one of these must be used before the first operation on each disk file.

It is advisable to use CLOSE after the last output operation on a file to cause the directory entry to be updated. However, all files will be closed by the system when a program finishes, whether normally or not.

It is easy to make mistakes in typing in file specifications. The ability has therefore been provided to trap such errors and try again. To do this, use the procedure call ERRORTRAP(TRUE) before the relevant RESET or REWRITE, and test the function ERRORCODE afterwards. If ERRORCODE is zero, all is well; otherwise the value returned is the error code returned by DOS, e.g. 24 for file not in directory. To recover, simply repeat whatever process was used to get the file specification. If ERRORTRAP has not been called, or is called again with the parameter FALSE, any DOS-detected error will terminate the program.

8 Miscellaneous facilities

A number of procedures and functions have been provided to make it easy to do things which would otherwise be difficult or impossible to achieve in Pascal. Please note that use of these facilities will make your programs more difficult to convert to run on other Pascal systems.

CALL This allows you to call a routine written in Z80 machine code and stored at a known location in memory. It needs one parameter, the address of the routine.

CMD This allows you to exit from your program, passing a character string constant which will be executed as a DOS command. This may be the name of the next program in a series.

FAIL This terminates your program and returns to DOS, also ending a DO or CHAIN sequence if one was in effect.

INKEY This returns a character value corresponding to the key pressed if any, otherwise CHR(0).

INP This returns the value read from the specified port, as in BASIC.

LPEN This needs one parameter, a screen position in the range 0 to 1023. It is designed for use with the light pen marketed by Molimerx. It returns TRUE if the light pen is detected at the specified screen position. Users are advised to call this function twice, with a brief delay between the two calls, to avoid erroneous results caused by timing problems.

MEMORY This returns the minimum amount of free memory since the start of the program. If there is more than 32767 bytes free, it returns 32767.

OUT This sends the specified value to the specified port, as in BASIC.

PLOT This is equivalent to the S-80 BASIC commands SET and RESET. The first parameter is the X position (0 to 127), the second the Y position (0 to 47) and the third a flag with the value 1 for SET and 0 for RESET.

RANDOM Reset the random number seed.

RND — The single parameter is an integer as in S-80 BASIC. If 0, RND returns a real value between 0 and 1, otherwise it returns an integer value between 1 and the parameter value (inclusive).

SOUND — This takes two parameters, a frequency and a duration. It makes a sound in the usual S-80 way if a sound box is connected to the cassette lead.

STOP — This may be used to end a program from within a procedure or function.

9 Writing efficient programs

Since a compiled Pascal program is automatically much faster than the equivalent interpreted BASIC program, the efficiency of your Pascal programs may not often be important. However, when it is, you may find it useful to note some ways in which you can help the compiler to generate the most efficient code possible (and to use the minimum of memory).

Using the right variable type for the job is important. All arrays are automatically packed if possible; this means that character and boolean arrays are always packed into a single byte per element, as are integer subrange arrays with values in the range 0-255 and user-defined types with 256 values or less. Use of these types where possible not only saves space but also causes more efficient code to be generated in many cases. Real variables should be used only when strictly necessary, since every operation on them involves at least one ROM call, and often several.

The location of the variable declaration is also important, particularly for scalars. The most efficient access is always to variables declared in the main program, since space for them is allocated statically within the program code. Next best are parameters and local variables (those declared in the same procedure or function as the reference). When declaring arrays, note that a lower bound of zero is most efficient, with a value of one next best. Any other lower bound will require a subtraction at every element reference.

Some of the Pascal control structures are more efficient than others. The simplest code is generated for CASE and FOR statements, while complex logical expressions in IF, WHILE or REPEAT may involve several calls to the run-time system. CASE statements where the selector expression gives a single-byte value (see paragraph above) produce particularly compact code.

A number of functions and procedures have been provided specifically to provide efficent ways of manipulating bits and bytes directly. As well as Pascal equivalents of the BASIC PEEK and POKE, these include LOW, HIGH and WORD, three functions for separating a word (integer value) into its high-order and low-order bytes and for building a word from two bytes. Bit handling functions ANDBITS, ORBITS, NOTBITS, SHLBITS and SHRBITS are also provided; see Appendix C. The procedures MOVEUP and MOVEDN, each with the three parameters From address, To address and Length, give access to the Z-80 instructions LDIR and LDDR respectively, and allow the efficient movement of large blocks of data; they may be used in conjunction with VARPTR, which always returns the address of the first byte of the data concerned. Most of these procedures and functions generate in-line code.

Appendix A - Formal Language Definition

The Pascal language supported by this compiler is based on that defined in the Second Draft Proposal for a Pascal standard (ISO DP7185.1, BSI document reference 81/60021). This is very similar to, and is based on, Appendix D, Syntax, of the Pascal User Manual by Jensen and Wirth (full reference in section 3), which may be easier to obtain. It is believed to be virtually identical to the recently published Pascal standard, BS 6192.

The following conventions are used in the definitions which follow:

1 Lower case words, which may include hyphens, refer to rules occurring elsewhere in the syntax definition.
2 Items occurring within parentheses in a definition may occur zero, one or more times.
3 Where a number of items, which may be grouped between less-than and greater-than signs (< and >), are separated by OR, one and only one of the items must appear.
4 Anything occurring within double quotes (") must appear exactly as shown.
5 Anything occurring within dotted parentheses (. and .) may occur once or not at all.

The following is a complete list of the syntax rules applicable to this implementation, with an indication of those additional rules included in the standard language definition:

```
actual-parameter ::= expression OR variable-access
actual-parameter-list ::= "(" actual-parameter
          ( "," actual-parameter ) ")"
adding-operator ::= "+" OR "-" OR "OR"
apostrophe-image ::= "''"
array-type ::= "ARRAY" "(." index-type ( "," index-type )
          ".)" "OF" component-type
array-variable ::= variable-access
assignment-statement ::= < variable-access OR
          function-identifier > ":=" expression
base-type ::= ordinal-type
block ::= label-declaration-part constant-definition-part
          type-definition-part variable-declaration-part
          procedure-and-function-declaration-part
          statement-part
boolean-expression ::= expression
bound-identifier -- not implemented
buffer-variable -- not implemented
case-constant ::= constant
case-constant-list ::= case-constant ( "," case-constant )
case-index ::= expression
case-list-element ::= case-constant-list ":" statement
```

```
case-statement ::= "CASE" case-index "OF" case-list-element
          ( ";" case-list-element ) ( ";" )
          (. "ELSE" statement .) "END"
character-string ::= "'" string-element ( string-element ) "'"
component-type ::= type-denoter
component-variable ::= indexed-variable
compound-statement ::= "BEGIN" statement-sequence "END"
conditional-statement ::= if-statement OR case-statement
conformant-array-parameter-specification -- not implemented
conformant-array-schema -- not implemented
constant ::= (. sign .) < unsigned-number OR constant-identifier >
          OR character-string
constant-definition ::= identifier "=" constant
constant-definition-part ::= (. "CONST" constant-definition ";"
          ( constant-definition ";" ) .)
constant-identifier ::= identifier
control-variable ::= entire-variable
digit ::= "0" OR "1" OR "2" OR "3" OR "4" OR "5" OR "6" OR "7"
          OR "8" OR "9"
digit-sequence ::= digit (digit )
directive ::= "FORWARD"
domain-type -- type-identifier
else-part ::= "ELSE" statement
empty-statement ::=
entire-variable ::= variable-identifier
enumerated-type ::= "(" identifier-list ")"
expression ::= simple-expression
          (. relational-operator simple-expression .)
factor ::= variable-access OR unsigned-constant OR
          function-designator OR set-constructor OR
          "(" expression ")" OR "NOT" factor
field-designator -- not implemented
field-identifier -- not implemented
field-list -- not implemented
field-specifier -- not implemented
file-type ::= "FILE" "OF" component-type
file-variable ::= variable-access
final-value ::= expression
fixed-part -- not implemented
for-statement ::= "FOR" control-variable ":=" initial-value
          < "TO" OR "DOWNTO" > final-value "DO" statement
formal-parameter-list ::= "(" formal-parameter-section
          ( ";" formal-parameter-section ) ")"
formal-parameter-section ::= value-parameter-specification OR
          variable-parameter-specification
function-block ::= block
function-declaration ::= function-heading ";" directive OR
          function-identification ";" function-block OR
          function-heading ";" function-block
function-designator ::= function-identifier
          (. actual-parameter-list .)
```

```
function-heading ::= < "FUNCTION" OR "FUNC" > identifier
          (. formal-parameter-list .) ":" result-type
function-identification ::= < "FUNCTION" OR "FUNC" >
          function-identifier
function-identifier ::= identifier
functional-parameter-specification -- not implemented
goto-statement ::= "GOTO" label
hexadecimal-constant ::= "&" hexadecimal-digit
          ( hexadecimal-digit )
    NB    A maximum of four hexadecimal digits may appear.
hexadecimal-digit ::= digit OR "A" OR "B" OR "C" OR "D" OR "E"
          OR "F"
identified-variable -- pointer-variable "@"
identifier ::= letter ( letter OR digit )
identifier-list ::= identifier ( "," identifier )
if-statement ::= "IF" boolean-expression "THEN" statement
          (. else-part .)
index-expression ::= expression
index-type ::= ordinal-type
index-type-specification -- not implemented
indexed-variable ::= array-variable "(." index-expression
          ( "," index-expression ) ".)"
initial-value ::= expression
label ::= digit-sequence
label-declaration-part ::= (. "LABEL" label ( "," label ) ";" .)
letter ::= any upper-case or lower-case letter
member-designator ::= expression (. ".." expression .)
multiplying-operator ::= "*" OR "/" OR "DIV" OR "MOD" OR "AND"
new-ordinal-type ::= enumerated-type OR subrange-type
new-pointer-type -- "@" domain-type
new-structured-type ::= (. "PACKED" .) unpacked-structured-type
new-type ::= new-ordinal-type OR new-structured-type OR
          new-pointer-type
ordinal-type ::= new-ordinal-type OR integer-type OR
          boolean-type OR char-type OR ordinal-type-identifier
ordinal-type-identifier ::= identifier
packed-conformant-array-schema -- not implemented
pointer-type -- new-pointer-type OR
          pointer-type-identifier
pointer-type-identifier -- type-identifier
pointer-variable -- variable-access
procedural-parameter-specification -- not implemented
procedure-and-function-declaration-part ::=
          ( < procedure-declaration OR function-declaration >
          ";" )
procedure-block ::= block
procedure-declaration ::= procedure-heading ";" directive OR
          procedure-identification ";" procedure-block OR
          procedure-heading ";" procedure-block
procedure-heading ::= < "PROCEDURE" OR "PROC" > identifier
          (. formal-parameter-list .)
procedure-identification ::= < "PROCEDURE" OR "PROC" >
          procedure-identifier
```

```
procedure-identifier ::= identifier
procedure-statement ::= procedure-identifier
          (. actual-parameter-list .)
program ::= program-heading ";" program-block "."
program-block ::= block
program-heading ::= "PROGRAM" identifier
          (. "(" program-parameters ")" .)
program-parameters ::= identifier-list
read-parameter-list ::= "(" (. file-variable "," .)
          variable-access ( "," variable-access ) ")"
readln-parameter-list ::= (. "(" < file-variable OR
          variable-access > ( "," variable-access ) ")" .)
record-section -- not implemented
record-type -- not implemented
record-variable -- not implemented
record-variable-list -- not implemented
relational-operator ::= "=" OR "<>" OR "<" OR ">" OR "<=" OR ">="
          OR "IN"
repeat-statement ::= "REPEAT" statement-sequence
          "UNTIL" boolean-expression
repetitive-statement ::= repeat-statement OR while-statement OR
          for-statement
result-type ::= simple-type-identifier OR
          pointer-type-identifier
scale-factor ::= signed-integer
set-constructor ::= "(." (. member-designator
          ( "," member-designator ) .) ".)"
set-type ::= "SET" "OF" base-type
sign ::= "+" OR "-"
signed-integer ::= (. sign .) unsigned-integer
signed-number ::= signed-integer OR signed-real
signed-real ::= (. sign .) unsigned-real
simple-expression ::= (. sign .) term ( adding-operator term )
simple-statement ::= empty-statement OR assignment-statement OR
          procedure-statement OR goto-statement
simple-type ::= ordinal-type OR real-type
simple-type-identifier ::= type-identifier
special-symbol ::= "+" OR "-" OR "*" OR "/" OR "=" OR "<" OR ">"
          OR "(." OR ".)" OR "." OR "," OR ":" OR ";" OR "("
          OR ")" OR "<>" OR "<=" OR ">=" OR ":=" OR ".."
          OR word-symbol
statement ::= (. label ":" .) <simple-statement OR
          structured-statement >
statement-part ::= compound-statement
statement-sequence ::= statement ( ";" statement )
string-character ::= any character from the TRS-80 character set
string-element ::= apostrophe-image OR string-character
structured-statement ::= compound-statement OR
          conditional-statement OR repetitive-statement
structured-type ::= new-structured-type OR
          structured-type-identifier
structured-type-identifier ::= type-identifier
```

```
subrange-type ::= constant ".." constant
tag-field -- not implemented
tag-type -- not implemented
term ::= factor ( multiplying-operator factor )
type-definition ::= identifier "=" type-denoter
type-definition-part ::= (. "TYPE" type-definition ";"
          ( type-definition ";" )
type-denoter ::= type-identifier OR new-type
type-identifier ::= identifier
unpacked-conformant-array-schema -- not implemented
unpacked-structured-type ::= array-type OR set-type OR file-type
unsigned-constant ::= unsigned-number OR character-string OR
          constant-identifier OR "NIL"
unsigned-integer ::= digit-sequence
unsigned-number ::= unsigned-integer OR hexadecimal-constant
          OR unsigned-real
unsigned-real ::= unsigned-integer "." digit-sequence
          (. "E" scale-factor .) OR
          unsigned-integer "E" scale-factor
value-parameter-specification ::= identifier-list ":"
          type-identifier
variable-access ::= entire-variable OR component-variable OR
          identified-variable
variable-declaration ::= identifier-list ":" type-denoter
variable-declaration-part ::= (. "VAR" variable-declaration ";"
          ( variable-declaration ";" ) .)
variable-identifier ::= identifier
variable-parameter-specification ::= "VAR" identifier-list
          ":" type-identifier
variant -- not implemented
variant-part -- not implemented
variant-selector -- not implemented
while-statement ::= "WHILE" boolean-expression "DO" statement
with-statement -- not implemented
word-symbol ::= "AND" OR "ARRAY" OR "BEGIN" OR "CASE" OR "CONST"
          OR "DIV" OR "DO" OR "DOWNTO" OR "ELSE" OR "END"
          OR "FILE" OR "FOR" OR "FUNC" OR "FUNCTION" OR "GOTO"
          OR "IF" OR "IN" OR "LABEL" OR "MOD" OR "NIL" OR "NOT"
          OR "OF" OR "OR" OR "PACKED" OR "PROC" OR "PROCEDURE"
          OR "PROGRAM" OR "RECORD" OR "REPEAT" OR "SET" OR "THEN"
          OR "TO" OR "TYPE" OR "UNTIL" OR "VAR" OR "WHILE"
          OR "WITH"
write-parameter ::= expression (. ":" expression
          (. ":" expression .) .)
write-parameter-list ::= "(" (. file-variable "," .)
          write-parameter ( "," write-parameter ) ")"
writeln-parameter-list ::= (. "(" <file-variable OR
          write-parameter > ( "," write-parameter ) ")" .)
```

Appendix B - Reserved words

The following words have specific meanings in Pascal and may not be used as identifiers, even though they may not all be recognised by this version of this particular compiler:

AND	FILE	NIL	SET
ARRAY	FOR	NOT	THEN
BEGIN	FORWARD	OF	TO
CASE	FUNC	OR	TYPE
CONST	FUNCTION	PACKED	UNTIL
DIV	GOTO	PROC	VAR
DO	IF	PROCEDURE	WHILE
DOWNTO	IN	PROGRAM	WITH
ELSE	LABEL	RECORD	
END	MOD	REPEAT	

Appendix C - Standard and other supplied procedures and functions

The compiler operates as if the following were all declared in an outer block encompassing the entire program. Though these identifiers may be used for other purposes, the user is strongly advised not to do so.

ABS(expr)	Integer or Real Function - Absolute value
ANDBITS(e1,e2)	Integer Function - Logical AND of two integer values
ARCTAN(expr)	Real Function - Arc Tan (radians) (As BASIC ATN)
AT(expr)	Procedure - Position cursor on screen (as TRS-80 BASIC PRINT @)
CALL(expr)	Procedure - Call routine at fixed address (use PEEK/POKE to pass parameters)
CHR(expr)	Char Function - Convert to character (as TRS-80 BASIC CHR$)
CLOSE(file)	Procedure - Close a disk file
CMD(string)	Procedure - Exit, passing a command to DOS
COS(expr)	Real Function - Cosine (angle in radians)
DISPOSE(ptr)	Procedure - Free addressed variable
EOF(file)	Boolean Function - True if at end of file
EOLN(file)	Boolean Function - True if at end of line
ERRORCODE	Integer Function - DOS error code after ERRORTRAP
ERRORTRAP(bool)	Procedure - Turn I/O error trap on or off
EXP(expr)	Real Function - Exponential (as BASIC EXP)
FAIL	Procedure - Exit via DOS error exit 4030 Hex (will abort chaining)
HIGH(expr)	Char Function - Get high-order byte of value
INKEY	Char Function - Return key value if one pressed (as TRS-80 BASIC INKEY$)
INP(port)	Integer function - As BASIC INP
LN(expr)	Real Function - Natural Log (as BASIC LOG)
LOW(expr)	Char Function - Get low-order byte of value
LPEN(pos)	Boolean Function - Senses whether the light pen is at the specified screen position
MARK(ptr)	Procedure - Remember current heap position
MEMORY	Integer Function - Minimum amount of free memory (will never return a value greater than 32767).
MOVEDN(f,t,l)	Procedure - Block move downwards (Z80 LDDR) From and To addresses should be top of block
MOVEUP(f,t,l)	Procedure - Block move upwards (Z80 LDIR) From and To addresses should be bottom of block
NEW(ptr)	Procedure - Allocate storage for addressed variable
NIL	Constant - Pointer to nothing
NOTBITS(expr)	Integer Function - Logical NOT of an integer value
ODD(expr)	Boolean Function - True if expression is odd
ORBITS(e1,e2)	Integer Function - Logical OR of two integer values

```
ORD(expr)          Integer Function - Convert to integer
                     (similar to TRS-80 BASIC ASC, but
                      may be used for any type except Real)
OUT(port,val)      Procedure - As BASIC OUT
PAGE(file)         Procedure - Start new page or screen
PEEK(expr)         Integer Function - As TRS-80 BASIC PEEK
PLOT(x,y,flag)     Procedure - Plot TRS-80 graphics
                     (for SET use PLOT(x,y,1),
                      for RESET use PLOT(x,y,0))
POKE(addr,val)     Procedure - As TRS-80 BASIC POKE
PRED(expr)         Function (Any type except Real) - One less than
                     value of expression
RANDOM             Procedure - Reset random number seed (as BASIC)
READ(...)          Procedure - Standard input procedure
READLN(...)        Procedure - Standard input procedure
RELEASE(ptr)       Procedure - Reset heap to pointer value (from MARK)
RESET(file)        Procedure - Open disk file for input
REWRITE(file)      Procedure - Open disk file for output
RND(expr)          Real Function - As TRS-80 BASIC RND(n) where n
                     is an integer.
ROUND(expr)        Integer Function - Round a real value
SHLBITS(e1,e2)     Integer Function - Shift the first value left the
                     number of bits given by the second value
SHRBITS(e1,e2)     Integer Function - Shift the first value right the
                     number of bits given by the second value
SIN(expr)          Real Function - Sine of angle (in radians)
SOUND(f,d)         Procedure - Make a sound of the specified frequency
                     and duration
SQR(expr)          Real or Integer Function - Square of a number
SQRT(expr)         Real Function - Square root
STOP               Procedure - Exit from program without getting
                               to the end of the main block
SUCC(expr)         Function (Any type except Real) - One more than
                     value of expression
TITLE(file)        Procedure - Get file-spec from keyboard
TITLE(file,string or char array)
                   Procedure - Use supplied string as file-spec
TRACE(...)         Procedure - If trace flag is set, output the
                     specified values to the specified file
TRACEON            Boolean Function - Indicate whether trace flag is
                     set
TROFF              Procedure - Set trace flag off
TRON               Procedure - Set trace flag on
TRUNC(expr)        Integer function - Truncate real value to integer
WORD(high,low)     Integer Function    Build word from two bytes
WRITE(...)         Procedure   Standard output procedure
WRITELN(...)       Procedure   Standard output procedure
```

Appendix D - Other predeclared identifiers

Files - INPUT - The TRS-80 keyboard
OUTPUT - The TRS-80 display
PRINT - Line printer (if available)

Types - BOOLEAN - (FALSE,TRUE)
CHAR - Any ASCII character, byte value 0..255
INTEGER - Any value from -32768 to 32767
LOWINT - Integer values in the range 0..255
REAL - As TRS-80 BASIC single precision
TEXT - Equivalent to FILE OF CHAR

Constants - MAXINT - 32767

Appendix E - Use of memory

The system requires 48K bytes of RAM. Generated Pascal programs may run in as little as 16k depending on their data requirements. The run-time system occupies approximately 4K bytes starting at 5500 hex, and generated Z80 code starts at 6500 hex. Input/output buffers are located from 5200 to 54FF on Model I, and at the end of the generated code on Model III. The run-time stack, which includes most variable storage as well as return addresses and so on, starts at the top of available memory and grows downwards. The highest address to be used is taken from the DOS location HIGH$ at 4049/A Hex on Model I or 4411/2 Hex on Model III. Lower case drivers and other routines in high memory will be safe so long as they set HIGH$ correctly, as in VTOS and Newdos 80. The Editor and Compiler allow at least 200 bytes free for device drivers etc. (nearer 1K bytes on Model I).

Appendix F - Size Limits

The following limits apply to the size of programs which may be handled by the system:

Editor		
Length of source line	64	
Total number of characters in text	29000	
Compiler		
Length of a source line	99	
Length of a string constant	97	
Length in an output format	99	
Length of keyboard input up to ENTER	128	
Maximum length of an identifier	99	
Number of user-defined identifiers	370	(see note below)
Length of user-defined identifiers	1500	
Number of active cases + 3 * number of nested blocks	200	
Lowest value in a set	0	
Highest value in a set	255	

Note that the space for user-defined identifiers is also used to hold details of array dimensions and any other types which are declared implicitly, e.g.

```
VAR A: ARRAY(0..7,0..9) OF (YELLOW,RED,BLUE);
```

takes one space for A, two for the two levels of array, one each for the implicitly declared types 0..7 and 0..9, one for the implicitly declared type (YELLOW,RED,BLUE) and one each for YELLOW, RED and BLUE, a total of nine entries.

Appendix G - Error codes and messages

G.1 Compilation errors

2 Identifier expected
3 PROGRAM expected
4) expected
5 : expected
8 OF expected
9 (expected
11 (. expected (i.e. left square bracket)
12 .) expected (i.e. right square bracket)
13 END expected
14 ; expected
15 Integer constant expected
16 = expected
17 BEGIN expected
18 Type expected
20 . expected
25 , expected
26 .. expected
29 Undeclared type
33 Aggregate not allowed here
38 A string constant must contain at least one character
46 Scalar constant required
50 Error in constant
51 := expected
52 THEN expected
53 UNTIL expected
54 DO expected
55 TO or DOWNTO expected - TO assumed
58 Illegal factor in expression
62 Filename expected
91 String constant too long
92 Unexpected end of program - check for unmatched string or comment delimiters
99 Compiler error
102 Invalid range - lower bound exceeds upper bound
103 Invalid use of identifier
104 Undeclared identifier
106 Numeric value required here
109 Real value or variable not allowed in this context
114 Base type must not be real
116 Parameter value is not of required type
119 When FORWARD is used, no declaration of parameters or returned value is allowed with the procedure body
126 Wrong number of parameters
130 Set expression expected
131 Illegal comparison
135 Boolean value required
137 Error in type of set element
138 Too many subscripts
139 Subscript does not match declared index type
141 Pointer value expected

143 Illegal loop control variable
145 Type conflict
146 File must be of char - in this version files may only be declared as TEXT or FILE OF CHAR
165 Label is already defined
166 Label/GOTO nesting conflict - see GOTO restrictions in Section 3
167 Label not declared
247 Total length of all identifiers is too high
248 Value out of declared range
249 Input line too long
250 Too many identifiers
251 Too many nested blocks or cases
252 Too many disk files
253 Feature not yet implemented

G.2 Run-time errors

The following messages may be produced by the run-time system. Since this is used by all the programs in the system, they may also be produced by the Editor or Compiler.

Out of memory	- The stack has extended down as far as the end of the program. The cure is usually to reduce the size of arrays.
Error on file n	- Where n is a number from 1 to 4 indicating the relative position in the PROGRAM header (not counting INPUT, OUTPUT and PRINT) of the offending file. Usually a DOS message appears first which fully explains the error. If not, it is a logical error such as writing to a file which was RESET.
Divide by zero	- Integer or real division by zero.
Maths error	- Error in use of standard procedure or real arithmetic, such as SQRT of negative value.
Source error n	- A serious error of type n was detected at the line stated during compilation.
Value n out of range	- Compiler option R or S was selected, and either a value to be assigned is outside the range declared for the target, or an array subscript is outside the range of the index type.
No case for value n	- A CASE statement did not include an ELSE clause, the C option was not specified and the value of the case-index did not match any of the specified case-constants.
No digits read	- A READ or READLN for a non-character value found an invalid character before the first digit.
No value for label n	- The given label is the subject of a GOTO, and was declared but not defined.

In each case the message will be followed by

at line n

if the line number option was selected or the error was detected at compile time.

Appendix H - Use of chaining (CHAIN or DO)

In this system steps have been taken to facilitate the use of chaining with operating systems such as NEWDOS/80 (which is the one used by the author). In particular, the FAIL procedure has been provided; when called, this terminates the program by signalling an error to DOS, in such a way that chaining will be terminated. This is used by the compiler if it has found more than trivial errors during compilation, so that standard JCL or DO files can be used for compiling a program and then running a test or generating a new version.

A program PASJCL is supplied in source and command forms to illustrate this technique. When run using the PASJCL command, it asks for the program name, and then offers the options of including a test run, or alternatively either generating a new version (program/CMD) or simply doing a test compilation. It then displays the compiler options available and invites a selection. A file PASRUN/JCL is created by the program and then invoked by CMD('DO PASRUN'), and a file TEMP/OBJ is required (or created) by the first and third options. You may of course need to tailor this approach to fit your operating system, and are free to do so. As it stands the program works with NEWDOS/80 Version 2. Note the use of TITLE(JCL,'PASRUN/JCL') to save the need for you to type in a file-spec..

This approach could be extended to build a JCL file for compiling a program consisting of more than one source file. Simply change the options line to MW, add a line saying how many input files are present, and include any additional source file-specs after it, one per line.

Appendix I - Possible incompatibilities with previous versions

Upper and lower case letters are now treated as equivalent everywhere except in character strings and comments, though they will be listed as entered.
Dotted parentheses ((. and .)) should now be used where the language requires square brackets. Ordinary parentheses will still be accepted if the compiler's C (compatibility) option is selected.
Bit-handling functions ANDBITS, ORBITS, NOTBITS, SHLBITS and SHRBITS have been added. The operators AND, OR and NOT may now only be used with boolean operands; the non-standard operators SHL and SHR are no longer available.