PERQ Pascal Extensions

Miles Barel

Three Rivers Computer Corporation

October 27, 1980

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Table of Contents

PERQ Pascal Extensions

1. Introduction

PERQ Pascal is an upward compatible extension of the programming language Pascal defined in PASCAL User Manual and Report [JW74]. This document describes only the extensions to Pascal. Refer to PASCAL User Manual and Report [JW74] for a fundamental definition of Pascal. This document uses the BNF notation used in PASCAL User Manual and Report [JW74]. The existing BNF is not repeated but is used in the syntax definition of the extensions. The semantics are defined informally.

These extensions are designed to support the construction of large systems programs. A major attempt has been made to keep the goals of Pascal intact. In particular, attention is directed at simplicity, efficient run-time implementation, efficient compilation, language security, upward compatibility, and compile-time checking.

Inspiration for these extensions to the language are motivated by the BSI/ISO Pascal Standard [BSI79], the UCSD Workshop on Systems Programming Extensions to the Pascal Language [UCSD79] and, most notably, Pascal* [P*].

2. Declaration Relaxation

The order of declaration for labels, constants, types, variables, procedures and functions has been relaxed. These declaration sections may occur in any order and any number of times. It is required that an identifier is declared before it is used. Two exceptions exist to this rule: 1) pointer types may be forward referenced as long as the declaration occurs within the same type-definition-part and 2) procedures and functions may be predeclared with a forward declaration.

The new syntax for the declaration section is:

```
<block> ::= <declaration part><statement part>

<declaration part> ::= <declaration> |
        <declaration><declaration part>

<declaration> ::= <empty> |
        <import declaration part> |
        <label declaration part> |
        <constant definition part> |
        <type definition part> |
        <variable declaration part> |
        <procedure and function declaration part>
```

(NOTE: Import declaration part is described in section

10.1)

3. Extended Constants

3.1 Unsigned Octal Integers

Unsigned octal integer constants are supported as well as decimal. Octal integers are indicated by a "#" preceding the number.

The syntax for an unsigned integer is:

```
<unsigned integer> ::= <unsigned decimal integer> |
        <unsigned octal integer>

<unsigned decimal integer> ::= <digit>{<digit>}

<unsigned octal integer> ::= #<ogit>{<ogit>}

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ogit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

A change to this extension which would allow integer constants with arbitrary radices is under consideration.

3.2 Constant Expressions

PERQ Pascal extends the definition of a constant to include expressions which may be evaluated at compile-time. Constant expressions support the use of the arithmetic operators +, -, *, DIV, MOD and /, the logical operators AND, OR and NOT, the type coercion functions CHR, ORD and RECAST (RECAST is defined in section 4), and previously defined constants. All logical operations are performed as full 16 bit operations.

The new syntax for constants is:

```
<constant> ::= <string> | <constant expression>

<constant expression> ::= <cterm> | <sign><cterm> |
        <constant expression><adding operator><cterm>

<cterm> ::= <cfactor> |
        <cterm><multiplying operator><cfactor>

<cfactor> ::= <unsigned constant> |
        ( <constant expression> ) | NOT <cfactor> |
        CHR(<constant expression>) |
```

```
ORD(<constant expression>) |
RECAST(<constant expression>,<type identifier>)
```

4. Type Coercion - RECAST

The type coercion function RECAST will convert the type of an expression from one type to another type with the same representation length. RECAST, as well as the standard functions CHR and ORD, is implemented as a compile-time operation. Thus the use of any type coercion function does not incur run-time overhead. The RECAST function takes two parameters: the expression to be coerced and the type name to which the expression is to be coerced. Its declaration is:

```
function RECAST(value:any-type; T:type-name):T
```

5. Extended Case Statement

Two extension have been made to the case statement:

1. Constant subranges as labels.
2. The "otherwise" clause which is executed if the case selector expression fails to match any case label.

Case labels may not overlap. A compile-time error will occur if any label has multiple definitions.

The new syntax for the case statement is:

```
<case statement> ::= CASE <expression> OF
        <case list element> {;<case list element>} END

<case list element> ::= <case label list> : <statement> |
        <empty>

<case label list> ::= <case label> {,<case label>}

<case label> ::= <constant> [..<constant>] | OTHERWISE
```

6. Control Structures

6.1 GOTO Statement

PERQ Pascal has a more limited form of the GOTO statement that is defined in Jensen and Wirth [JW74]. PERQ Pascal prohibits a GOTO statement to a label which is not

within the same block as the GOTO statement.

The following example is illegal in PERQ Pascal:

```
program BADGOTO(input,output);
label 1;
   procedure P;
   begin
   goto 1
   end;
begin
1:
end.
```

A change to eliminate this restriction is under consideration.

6.2 EXIT Statement

The procedure EXIT is provided to allow forced termination of procedures or functions. The EXIT statement may be used to exit from the current procedure or function, or any of its parents. The procedure takes one parameter: the name of the procedure or function to exit from. Note that the use of an EXIT statement to return from a function can result in the function returning undefined values if no assignment to the function identifier is made prior to the execution of the EXIT statement. Below is an example use of the EXIT statement:

```
program EXITEXAMPLE(input,output);
var STR: string;

   procedure P;
   begin
   readln(STR);
   writeln(STR);
   if STR = "This is the first line" then
       exit(EXITEXAMPLE)
   end;

begin
P;
while STR <> "Last Line" do
    begin
    readln(STR);
    writeln(STR)
    end
end.
```

If the above program is supplied with the following input:

```
This is the first line
This is another line
Last Line
```

the following output would result:

```
This is the first line
```

If the procedure or function to be exited has been called recursively, then the most recent invocation of that procedure will be exited.

A change to allow a more general exception handling mechanism is under consideration.

7. Sets

PERQ Pascal supports all of the constructs defined for sets in chapter 8 of Jensen and Wirth [JW74]. Sets (of enumeration values) are limited to non-negative integers only. The maximum set size supported is 32,768 elements. Space is allocated for sets in a bitwise fashion -- at most 2048 words for sets of 32,768 elements.

8. Strings

PERQ Pascal includes a string facility which provides variable length strings with a maximum size limit imposed upon each string. The default maximum length of a STRING variable is 80 characters. This may be overridden in the declaration of a STRING variable by appending the desired maximum length (must be a compile-time constant) within [ ] after the reserved type identifier STRING. There is an absolute maximum of 255 characters for all strings. The following are example declarations of STRING variables:

```
LINE: STRING;    { defaults to a maximum length of 80
                                    characters }

SHORTSTR: STRING[12];    { maximum length of SHORTSTR
                                    is 12 characters }
```

Assignments to string variables may be performed using the assignment statement or by means of a READ statement. Assignment of one STRING variable to another may be performed as long as the dynamic length of the source is within the range of the maximum length of the destination -- the maximum length of the two strings need not be the same.

The individual characters within a STRING may be selectively read and written. The characters are indexed

starting from 1 through the dynamic length of the string. For example:

```
program STREXAMPLE(input,output);
var LINE:string[25];
    CH:char;

begin
LINE:='this is an example.';
LINE[1]:='T';              { LINE now begins with }
                           { upper case T }
CH:=LINE[5];               { CH now contains a space }
end.
```

A STRING variable may not be indexed beyond its dynamic length. The following instructions, if placed in the above program, would produce an "invalid index" run-time error:

```
LINE:='12345';
CH:=LINE[6];
```

STRING variables (and constants) may be compared irrespective of their dynamic and maximum lengths. The resulting comparison is lexicographical according to the ASCII character set. The ASCII parity bit (bit 7) is significant for lexical comparisons.

A STRING variable, with maximum length N, can be conceived as having the following internal form:

```
packed record DYNLENGTH:0..255
                { the dynamic length }
              CHRS: packed array [1..N] of char;
                { the actual characters go here }
       end;
```

8.1 LENGTH Function

The predefined integer function LENGTH is provided to return the dynamic length of a string. For example:

```
program LENEXAMPLE(input,output);
var LINE:string;
    LEN:integer;
begin
LINE:='This is a string with 35 characters';
LEN:=length(LINE)
end.
```

will assign the value 35 into LEN.

9. Procedure/Function Parameter Types

9.1 Procedures and Functions as Parameters

PERQ Pascal does not support passing procedures and functions as parameters to other procedures or functions.

A change to remove this restriction is under consideration.

10. Modules

The module facility provides the ability to encapsulate procedures, functions, data and types, as well as supporting separate compilation. Modules may be separately compiled, and intermodule type checking will be performed as part of the compilation process. Unless an identifier is exported from a module, it is local to that module and cannot be used by other modules. Likewise all identifiers referenced in a module must be either local to the module or imported from another module.

Modules do not contain a main statement body. A program is a special instance of a module and conforms to standard Pascal. Only a program may contain a main body, and every executable group of modules must contain exactly one instance of a program.

Exporting allows a module to make constants, types, variables, procedures and functions available to other modules. Importing allows a module to make use of the EXPORTS of other modules.

Global constants, types, variables, procedures and functions can be declared by a module to be private (available only to code within the module) or exportable (available within the module as well as from any other module which imports them).

10.1 IMPORTS Declaration

The IMPORTS declaration specifies the modules which are to be imported into a module. The declaration includes the name of the module to be imported and the file name of the source for that module. (Note: If the module is composed of several include files(see section 15.2.1), only those files from the file containing the program or module heading through the file which contains the word PRIVATE must be available.)

The syntax for the IMPORTS declaration is:

```
<import declaration part> ::= IMPORTS <module name> FROM
 <file name>
```

10.2 EXPORTS Declaration Section

If a module is to contain any exports, the EXPORTS declaration section must immediately follow the program or module heading. The EXPORTS declaration section is comprised of the word EXPORTS followed by the declarations of those items which are to be exported. These definitions are given as previously specified (see section 2: Declaration Relaxation) with one exception: procedure and function bodies are not given in the exports section. Only forward references (see chapter 11.C in Jensen and Wirth [JW74]) are given. The inclusion of "FORWARD;" in the EXPORTS reference is omitted.

The EXPORTS declaration section is terminated by the occurance of the word PRIVATE. This signifies the beginning of the declarations which are local to the module. The PRIVATE declaration section must contain the bodies for all procedures and functions defined in the EXPORTS declaration section.

If a module is to contain no EXPORTS declaration section, the inclusion of PRIVATE folloing the module or program heading is optional (PRIVATE is assumed). (Note: A module with no EXPORTS would be useless, since its contents could never be referenced -- it only makes sense for a program not to have any EXPORTS.)

The new syntax for a unit of compilation is:

```
<compilation unit> ::= <module> | <program>

<program> ::= <program heading><module body><statement
 part>.

<module> ::= <module heading><module body>.

<program heading> ::= PROGRAM <identifier> ( <file
 identifier>
        {, <file identifier>});

<module heading> ::= MODULE <identifier>;

<module body> ::= EXPORTS <declaration part> PRIVATE
        <declaration part> | PRIVATE <declaration part> |
        <declaration part>
```

11. Dynamic Space Allocation and Deallocation

The PERQ Pascal Compiler supports the dynamic allocation procedures NEW and DISPOSE defined on page 105 of Jensen and Wirth [JW74], along with several upward compatible extensions which permit full utilization of the PERQ memory architecture.

There are two features of PERQ's memory architecture which require extensions to the standard allocation procedures. First, there are situations which require particular alignment of memory buffers, such as IO operations. Second, PERQ supports multiple data segments from which dynamic allocation may be performed. This facilitates grouping data together which is to be accessed together, which may improve PERQ's performance due to improved swapping. Data segments are multiples of 256 words in size and are always aligned on 256 word boundaries. For further information of the memory architecture and available functions see the documentation on the memory manager.

11.1 NEW

If the standard form of the NEW procedure call is used:

```
NEW(Ptr{,Tag1,...TagN})
```

memory for Ptr will be allocated with arbitrary alignment from the default data segment.

The extended for of the NEW procedure call is:

```
NEW(Segment,Alignment,Ptr{,Tag1,...TagN})
```

Segment is the segment number from which the allocation is to be performed. This number is returned to the user when creating a new data segment (see the documentation on the memory manager). The value 0 is used to indicate the default data segment.

Alignment specifies the desired alignment; Any power of two up to 256 inclusive is permissable.

If the extended form of NEW is used, both a segment and alignment must be specified; there is no form which permits selective inclusion of either characteristic.

If the desired allocation from any call to NEW cannot be performed, a NIL pointer is returned.

11.2 DISPOSE

DISPOSE is identical to the definition given in Jensen

and Wirth [JW74]. Note that the segment and alignment are never given to DISPOSE, only the pointer and tag field values.

12. Integer Logical Operations

The PERQ Pascal compiler supports a variety of integer logical operations. The operations supperted include: and, inclusive or, not, exclusive or, shift and rotate. The syntax for their use resembles that of a function call, however the code is generated inline to the procedure (hence there is no procedure call overhead associated with their use). The syntax for the logical functions are described in the following sections.

12.1 And

```
Function LAND(Val1,Val2: integer): integer;
```

LAND returns the bitwise AND of Val1 and Val2.

12.2 Inclusive Or

```
Function LOR(Val1,Val2: integer): integer;
```

LOR returns the bitwise INCLUSIVE OR of Val1 and Val2.

12.3 Not

```
Function LNOT(Val: integer): integer;
```

LNOT returns the bitwise complement of Val.

12.4 Exclusive Or

```
Function LXOR(Val1,Val2: integer): integer;
```

LXOR returns the bitwise EXCLUSIVE OR of Val1 and Val2.

12.5 Shift

```
Function SHIFT(Value, Distance: integer): integer;
```

SHIFT returns Value shifted Distance bits. If Distance is positive a left shift occurs, otherwise a right shift occurs. When performing a left shift, the least significant bit is filled with a Ø, and likewise when performing a right shift, the most significant bit is filled with a Ø.

12.6 Rotate

```
Function ROTATE(Value, Distance: integer): integer;
```

ROTATE returns Value rotated Distance bits. If Distance is positive a right rotate occurs, otherwise a left rotate occurs. Note that the direction is opposite of SHIFT.

13. Input/Output Intrinsics

PERQ's Input/Output intrinsics vary slightly from Jensen and Wirth [JW74]. Only the differences are discussed below.

13. REWRITE

The REWRITE procedure has the following form:

REWRITE(F,Name)

F is the file variable to be associated with the file to be written and Name is a string containing the name of the file to be created. EOF(F) becomes true and a new file may be written. The only difference between the PERQ and Jensen and Wirth [JW74] REWRITE is the inclusion of the filename string.

13.2 RESET

The RESET procedure has the following form:

RESET(F,Name)

F is the file variable to be associated with the existing file to be read and Name is a string containing the name of the file to be read. The current file position is set to the beginning of file, i.e. assigns the value of the first element of the file to F^. EOF(F) becomes false if F is not empty; othewise, EOF(F) becomes true and F^ is undefined.

13.3 READ/READLN

PERQ Pascal supports extended versions of the READ and READLN procedures defined by Jensen and Wirth [JW74]. Along with the ability to read integers (and subranges of integers), reals and characters, PERQ Pascal also supports reading booleans, packed arrays of characters and strings.

The strings TRUE and FALSE (or any unique abbreviations) are valid input for parameters of type boolean (both upper and lower case are permissable).

If the parameter to be read is a PACKED ARRAY[m..n] of CHAR, then the next n-m+1 characters from the input line will be used to fill the array. If there are less than

n-m+1 characters on the line, the array will be filled with the available characters, starting at the m'th position, and the remainder of the array will be filled with blanks.

If the parameter to be read is of type STRING, then the string variable will be filled with as many characters as possible until either the end of the input line is reached or the static length of the string is met. If there are not enough characters on the line to fill the entire string, the dynamic length of the string will be set to the number of characters read.

13.4 WRITE/WRITELN

PERQ Pascal provides many extensions to the WRITE and WRITELN procedures defined by Jensen and Wirth [JW74]. Due to the extensiveness of these extensions, the entire WRITE and WRITELN procedures are redefined below:

1. write(p1,...,pn) stands for write(output,p1,...,pn)

2. write(f,p1,...,pn) stands for BEGIN write(f,p1); ... write(f,pn) END

3. writeln(p1,...,pn) stands for writeln(output,p1,...pn)

4. writeln(f,p1,...,pn) stands for BEGIN write(f,p1); ... write(f,pn); writeln(f) END

5. Every parameter pi must be of one of the forms:

```
e
e : e1
e : e1 : e2
```

   where e, e1 and e2 are expressions.

6. e is the VALUE to be written and may be of type char, integer (or subrange of integer), real, boolean, packed array of char or string. For parameters of type boolean, one of the strings TRUE, FALSE or UNDEF will be written; UNDEF is written if the internal form of the expression is neither Ø nor 1.

7. e1, the minimum field width, is optional. In general, the value e is written with e1 characters (with preceeding blanks). With one exception, if e1 is smaller than the number of characters required to print the given value, more space is allocated; if e is a packed array of char, then only the first e1 characters of the array will be

printed.

8. e2, which is optional, is applicable only when e is of type integer (or subrange of integer) or real. If e is of type integer (or subrange of integer) then e2 indicates the base in which the value of e is to be printed. The valid range for e2 is 2..36 and -36..-2. If e2 is positive, then the value of e is printed as a signed quantity (16 bit 2's complement); otherwise the value of e is printed as a full 16 bit unsigned quantity. If e2 is omitted the signed value of e is printed in base 10. If e is of type real then e2 specifies the number of digits to follow the decimal point. The number is then printed in fixed point notation. If e2 is omitted, then real numbers are printed in floating point notation.

14. Miscellaneous Intrinsics

14.1 StartIO

There is a special QCode (STARTIO - See the PERQ QCode Reference Manual) which is used to initiate input/output operations to raw devices. PERQ Pascal supports a procedure, STARTIO, to facilitate generation of the correct QCode sequence for I/O programming. The procedure call has the following form:

STARTIO(Unit)

where Unit is the hardware unit number of the device to be activated (for further information see the documentation on IO Programming).

14.2 Raster-Op

Raster-Op is a special QCode which is used to manipulate arbitrary size blocks of memory. It is especially useful for creating and modifying displays on the screen. RasterOp modifies a rectangular area (called the "destination") of arbitrary size (to the bit). The picture drawn into this rectangle is computed as a function of the previous contents of the destination and the contents of another rectangle of the same size called the "source". The functions performed to combine the two pictures are described below.

In order to allow RasterOp to work on memory other than that used for the screen bitmap, RasterOp has parameters that specify the areas of memory to be used for the source and destination: a pointer to the start of the memory block

and the width of the block in words. The height of the block is not needed. Within these regions, the positions of the source and destination rectangles are given as offsets from the pointer. Thus position (0,0) would be at the upper left corner of the region, and, for the screen, (767, 1023) would be the lower right.

The compiler supports a RASTEROP intrinsic which may be used to invoke the Raster-Op QCode. The form of this call is:

```
RASTEROP(Function,
         Width,
         Heigth,
         Destination-X-Position,
         Destination-Y-Position,
         Destination-Area-Line-Length,
         Destination-Memory-Pointer,
         Source-X-Position,
         Source-Y-Position,
         Source-Area-Line-Length,
         Source-Memory-Pointer)
```

(NOTE: the values for the destination PRECEED those for the source.)

The arguments to RasterOp are defined below:

"Function" defines how the source and the destination are to be combined to create the final picture stored at the destination. The Raster-Op functions are as follows: (Src represents the source and Dst the destination):

Function	Name	Action
0	RRpl	Dst gets Src
1	RNot	Dst gets NOT Src
2	RAnd	Dst gets Dst AND Src
3	RAndNot	Dst gets Dst AND NOT Src
4	ROr	Dst gets Dst OR Src
5	ROrNot	Dst gets Dst OR NOT Src
6	RXor	Dst gets Dst XOR Src
7	RXNor	Dst gets Dst XNOR Src

The symbolic names are exported by the file "Raster.Pas".

"Width" specifies the size in the horizontal ("x") direction of the source and destination rectangles (given in bits).

"Height" specifies the size in the vertical ("y")

direction of the source and destination rectangles (given in scan lines).

"Destination-X-Position" is the bit offset of the left side of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Y-Position" is the scan-line offset of the top of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Area-Line-Length" is the number of words which comprise a line in the destination region (hence defining the region's width). The appropriate value to use when operating on the screen is 48.

"Destination-Memory-Pointer" is the 32 bit virtual address of the top left corner of the destination region (it may be a pointer variable of any type). This pointer must be quad-word aligned, however.

"Source-X-Position" is the bit offset of the left side of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Y-Position" is the scan-line offset of the top of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Area-Line-Length" is the number of words which comprise a line in the source region (hence defining the region's width). The appropriate value to use when operating on the screen is 48.

"Source-Memory-Pointer" is the 32 bit virtual address of the top left corner of the source region (it may be a pointer variable of any type). This pointer must be quad-word aligned, however.

15. Command Line and Compiler Switches

15.1 Command Line

The syntax for the compiler command line is:

[<OutFile>=]<InFile>{/SW}

<InFile> is the source file to be compiled. If the name <InFile> does not end with ".PAS", ".PAS" will

automatically be concatonated onto <InFile> before attempting to open <InFile>.

<OutFile> is the name to be given to the output of the compiler. ".SEG" will be concatonated onto the end of <OutFile> if it is not already present. If no <OutFile> is given, <InFile> will be used, replacing ".PAS" with ".SEG".

Any number of compiler switches may follow the input file specification. The format and functionality of available switches are defined in the following sections.

15.2 Compiler Switches

PERQ Pascal compiler switches may be set either according to the convention described on pages 100-102 of Jensen and Wirth [JW74] or on the command line described above (see section 15.1). Compiler switches may be written as comments and are designated as such by a dollar sign character ($) as the first character of the comment, or after the input file specification in the command line preceeded by the slash character (/). The actual switches provided by the PERQ Pascal compiler, although similar in syntax, bear only little resemblance to the switches described in Jensen and Wirth [JW74].

The following sections describe the various switches currently supported by the PERQ Pascal Compiler.

15.2.1 File Inclusion

The PERQ Pascal compiler may be directed to include the contents of secondary source files in the compilation. The effect of using the file inclusion mechanism is identical to having the text of the secondary file(s) present in the primary source file (the primary source file is that file which the compiler was told to compile).

To include a secondary file, the following syntax is used:

```
{$I FILENAME}
```

The characters between the "$I" and the "}" are taken as the name of the file to be included (leading spaces and tabs are ignored). The comment must terminate at the end of the filename, hence no other options can follow the filename.

The string ".PAS" is automatically concatonated onto the end of the filename if it is not already there.

The file inclusion mechanism may be used anywhere in a

program or module, and the results will be as if the entire contents of the include file were contained in the primary source file (the file containing the include directive).

The PERQ Pascal compiler can support only a limited number of nested file inclusions (the current maximum is 8). Note: this limit also includes the nesting of IMPORT files.

Note: There is no form of this switch for the command line, it may only be used in comment form within a program.

15.2.2 List Switch

The List switch controls whether or not the compiler will generate a program listing of the source text. The default is no list file generation. The format for the List switch is:

{$L FILENAME}

or

/LIST:FILENAME

where FILENAME is the name of the file to be written. The string ".LST" will be concatonated to FILENAME if it is not present. Like the file inclusion mechanism, the filename is taken as all characters between the "$L" and the "}" (ignoring leading spaces and tabs); hence no other options may be included in this comment.

With each source line, the compiler prints the line number, segment number, procedure number, and the number of bytes or words (bytes for code, words for data) required by that procedure's declarations or code to that point. The compiler also indicates whether the line lies within the actual code to be executed or is a part of the declarations for that procedure by printing a "D" for declarations and an integer to designate the lexical level of the statement nesting within the code part.

The list file is required by the debugger to associate code locations with source lines.

15.2.3 Range Checking

This switch is used to enable or disable the compiler from generating additional code to perform checking on array subscripts and assignments to subrange types.

Default value: Range checking enabled

$R+ or /RANGE enables range checking

$R- or /NORANGE disables range checking

If "$R" is not followed by a "+" or "-" then "+" is assumed.

Note that programs compiled with range checking disabled will run slightly faster, but invalid indices will go undetected. Until a program is fully debugged, it is advisable to keep range checking enabled.

15.2.4 Quiet Switch

This switch is used to enable or disable the compiler from printing the names of each procedure and function as it is compiled.

Default value: Printing of procedure and function names enabled

$Q+ or /VERBOSE enables printing of procedure and funtion names

$Q- or /QUIET disables printing of procedure and function names

if "$Q" is not followed by a "+" or "-" then "+" is assumed.

15.2.5 Symbols Switch

This switch is used to set the number of symbol table swap blocks usedby the compiler. As the number of symbol table swap blocks increases, compiler execution time becomes shorter, however physical memory requirements increase. Any number of symbol tables blocks from 1 to 24 may be used. The default is 15. The format for this switch is:

/SYMBOLS:<# of Symbol Table Blocks (1-24)>

Note: There is no comment form of this switch, it may only be used on a command line.

REFERENCES

[BSI79] "BSI/ISO Pascal Standard," Computer, April 1979.

[JW74] K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer Verlag, New York, 1974.

[P*] J. Hennessy and F. Baskett, "Pascal*: A Pascal Based Systems Programming Language," Stanford University Computer Science Department, TRN 174, August 1979.

[UCSD79] K. Bowles, Proceedings of UCSD Workshop on System Programming Extensions to the Pascal Language, Institute for Information Systems, University of California, San Diego, California, 1979.

INDEX

(entries entirely in upper case are reserved words or predeclared identifiers)