PERQ
Systems
Corporation

PERQ PASCAL EXTENSIONS

March 1984

This manual is for use with POS Release G.5 and subsequent releases until further notice.

PERQ Pascal is an upward-compatible extension of the standard programming language Pascal. This document describes only the extensions to Pascal.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

PERQ Pascal Extensions

December 27, 1983

This manual is for use with POS Release G.3 and subsequent releases until further notice.

PERQ Pascal is an upward-compatible extension of the standard programming language Pascal. This document describes only the extensions to PASCAL.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

TABLE OF CONTENTS

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 for a fundamental definition of Pascal. This document uses the BNF notation used in PASCAL User Manual and Report. 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.

These extensions to the language are derived from the BSI/ISO Pascal Standard [BSI79], the UCSD Workshop on Systems Programming Extensions to the Pascal Language [UCSD79] and, most notably, Pascal* [P*].

2. Declarations

You must declare all data items in a Pascal program. To declare a data item, specify the identifier and then what it represents.

2.1 Identifiers

You can include the underscore character "_" as a significant character in identifiers.

2.2 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 be 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: See "IMPORTS Declaration" in this document.

2.3 Files

PERQ Pascal permits the use of files as the component type of arrays, pointers and record fields.

3. Numbers

3.1 Whole Numbers

Single precision whole numbers are of the predefined type INTEGER. They occupy 16 bits (15 bits and a sign bit) and range in value from -32768 to +32767. Whole numbers of the predefined type LONG are double precision. They occupy 32 bits (31 bits and a sign bit) and range in value from -2147483648 to +2147483647. Arithmetic operations and comparisons are defined for both single and double precision whole numbers. You can only use Longs as assignments in arithmetic expressions. See the section "Extended Constants" for a discussion of single and double precision constants.

3.2 Floating Point Numbers

PERQ Pascal floating point numbers (type REAL) occupy 32 bits and conform to the IEEE floating point format. See [FP80]. Positive values range from approximately 1.1754945e-38 to 3.402823e+38 and negative values range from approximately -1.1754945e-38 to -3.402823e+38 . Arithmetic operations and comparisons are defined for floating point numbers.

3.3 Type Coercion Intrinsics

The STRETCH intrinsic can be used to explicitly convert a single precision whole number into a double precision whole number. The SHRINK intrinsic converts double precision into single precision, provided that the value of the double precision number is within the legal range of single precision. If it is not, a runtime error occurs. The FLOAT intrinsic converts a single precision whole number into a floating point number. TRUNC and ROUND convert a floating point number into a single precision whole number, provided that the value of the floating point number is within the legal range of single precision. If not, a runtime error occurs. (The module CONVERT, described in the PERQ Operating System Manual, provides three functions to convert a floating point number into a double precision whole number or vice versa.)

For example:

```
VAR R : real;
    L : long;
    I : integer;

    .
    .
    .

R := FLOAT(I);
I := TRUNC(R);
I := ROUND(R);
L := STRETCH(I);
I := SHRINK(L);

    .
    .
    .
```

3.4 Assignment Compatibility

Single precision whole numbers are assignment compatible with double precision whole numbers and floating point numbers. That is, expressions of type INTEGER can be assigned to variables of type LONG or REAL and INTEGER expressions can be passed by value (only) to LONG or REAL formal parameters. Double precision whole numbers and floating point numbers are not assignment compatible with single precision whole numbers. The type coercion intrinsics SHRINK, FLOAT and TRUNC can be used for this purpose. The module CONVERT, described in the PERQ Operating System Manual, allows assignments between double precision whole numbers and floating point numbers.

3.5 Mixed Mode Expressions

Mixed mode expressions between single and double precision whole numbers or between single precision whole numbers and floating point numbers are allowed. Mixed mode expressions containing double precision whole numbers and floating point numbers are not allowed. IMPORTANT NOTE: A mixed mode expression is evaluated from left to right (taking normal operator precedence and parentheses into account) in single precision mode until the first LONG (or REAL) is encountered. Starting at that point, all single precision operands are converted to double precision (or floating point) before used in evaluation. Single precision overflow or underflow can occur while in single precision mode. The STRETCH (or FLOAT) intrinsics may be used to avoid this problem.

4. Extended Constants

4.1 Whole Number Constants

Unsigned octal whole number (INTEGER or LONG) constants are supported as are decimal constants. Octal constants 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

NOTE: Unsigned constants do not imply unsigned arithmetic. For example, #177777 has the value -1, not +65535.

4.2 Single and Double Precision Constants

A constant in the range -32768 to +32767 is considered single precision (an INTEGER). Constants exceeding this range are considered double precision (LONG). The maximums for double precision constants are -2147483648 and +2147483647. When using octal constants, a constant with the 16th bit set is interpreted as a negative integer, not as a positive long integer.

4.3 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, the WORDSIZE intrinsic, and previously defined constants. (See "WordSize" in this manual and RECAST under "Type Compatibility", also in this manual.) All logical operations are performed as full 16-bit operations.

The new syntax for constants is:

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

<constant expression> ::= <csimple constant expression> |
        <constant expression>  <relational operator>
         <csimple constant expression>

<relational operator> ::= = | <> | < | <= | > | >=

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

<adding operator> ::= + | - | OR

<sign> ::= + | -

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

<multiplying operator> ::= * | / | DIV | MOD | AND

<cfactor> ::= <unsigned constant> |
        ( <constant expression> ) | NOT <cfactor> |
        CHR(<constant expression>) |
        ORD(<constant expression>) |
        RECAST(<constant expression>,<type identifier>) |
        WORDSIZE(<name>)
```

5. Type Compatibility

PERQ Pascal now supports strict type compatability as defined by the BSI/ISO Pascal Standard [BSI79], with one addition; Any two strings, regardless of their maximum static lengths, are considered compatable.

5.1 Type Coercion - RECAST

The function RECAST converts the type of an expression from one type to another type when both types require the same amount of space. RECAST, like the standard functions CHR and ORD, is processed at compile-time and thus does not incur run-time overhead. The function takes two parameters: the expression and the type name for the result. Its declaration is:

```
function RECAST(value:expression_type; T:result_type_name):T;
```

The following is an example of its use:

```
program RecastDemo;
type color = (red, blue, yellow, green);
var  C: color; I: integer
begin
I := 0;
C := RECAST(I, color);    { C := red; }
end.
```

Note that RECAST does not work correctly for all combinations of types; use the RECAST function sparingly and always scrutinize the results. Generally, only the following conversions produce expected results:

- longs, reals, and pointers to any other type
- arrays and records to either arrays or records
- sets of 0..n to any other type
- constants to long or real
- one word types (except sets) to one word types (except sets)

Avoid the use of the RECAST function for any other conversion.

WARNING: successful compilation does NOT imply that the RECAST function will execute correctly.

6. Extended Case Statement

Two extensions 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 occurs if any label has multiple definitions.

The extended 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
```

If the selector expression is not in the list of case labels and no OTHERWISE label is used, then the case statement is a no-op. This is different from PASCAL as defined by the PASCAL User Manual and Report, which suggests that this be a fault.

7. Control Structures

7.1 EXIT Statement

The procedure EXIT allows forced termination of procedures or functions. The statement can exit from the current procedure or function or from any of its parents. EXIT takes the procedure or function name to exit from as a parameter. 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 executing 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 exits.

WARNING: The parameter to EXIT can be any procedure or function name. If the specified routine is not on the call stack, the PERQ crashes.

8. Sets

PERQ Pascal supports all of the constructs defined for sets in Chapter 8 of PASCAL User Manual and Report [JW74]. Space is allocated for sets in a bit-wise fashion -- at most 255 words for a maximum set size of 4,080 elements. If the base type of a set is a subrange of integer, that subrange must be within the range 0..4079, inclusively. If the base type of a set is a subrange of an enumerated type, the cardinal number of the largest element of the set must not exceed 4,079.

9. Record Comparisons

PERQ Pascal supports comparison of records with one restriction: No portion of the records can be packed. For example,

```
program P(input,output);
var
    R1,R2 : record
                RealPart : integer;
                Imagine  : integer;
            end;
begin
R1.RealPart := 1;    R1.Imagine := 2;
R2.RealPart := 1;    R2.Imagine := 2;
if R1 = R2 then writeln('Records equal');
end.
```

produces as output

```
'Records equal'
```

10. 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 square brackets 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 regardless of their dynamic and maximum lengths. The resulting comparison is lexicographical according to the ASCII character set. The full 8 bits are compared; hence, 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;
       end;                  { the actual characters go here }
```

10.1 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.
```

assigns the value 35 into Len.

11. Generic Types

Generic types are general forms of more specific types. PERQ Pascal provides two predefined generic types:

1) Pointers

2) Files

11.1 Generic Pointers

Generic pointers provide a tool for generalized pointer handling. Variables of type POINTER can be used in the same manner as any other pointer variable with the following exceptions:

1) Since there is no notion of type associated with the reference variable of a generic pointer, generic pointers cannot be dereferenced.

2) Generic pointers cannot be used as an argument to NEW or DISPOSE.

3) Any pointer type can be passed to a generic pointer parameter. To make use of a generic pointer, RECAST should be used to convert the pointer to some usable pointer type.

The following is a sample program utilizing generic pointers:

```
program P(input,output);
type
    PtrInt = ^integer;
    PAOfChar = packed array[1..2] of char;
    PtrPAOfChar = ^PAOfChar;
var
    I : PtrInt;
    C : PtrPAOfChar;

    procedure Proc1(GenPtr : pointer);
    var W : PtrPAOfChar;
    begin
        W := recast(GenPtr,PtrPAOfChar);
        writeln(W^[1],W^[2])
    end;

begin
new(I);
I^ := 16961;    { First byte = 'A', second = 'B' }
Proc1(I);
new(C);
C^[1] := 'C';
C^[2] := 'D';
Proc1(C);
end.
```

produces output

```
AB
CD
```

11.2 Generic Files

Generic files have very restricted usage. Their purpose is to provide a facility for passing various types of files to a single procedure or function. Generic files may only appear as routine VAR parameters. Their type is FILE. They can be used in only two ways:

1) Passed as a parameter to another generic file parameter,or

2) As an argument to the LOADADR intrinsic.

The following examples show two ways of using generic files:

```
program P;
type
    FOfInt = file of integer;
var
    F : text;
    F2 : file of boolean;

    procedure Proc1(var GenFile : file);
    var
        UseGenFile : record
                       case boolean of
                          true  : (FileOfInterest : ^FOfInt);
                          false : (Offset : integer;
                                   Segment : integer);
                     end;
    begin
    loadAdr(GenFile);
    storexpr(UseGenFile.Offset);
    storexpr(UseGenFile.Segment);
    { Now FileOfInterest can be used }
                  .
                  .
                  .
    end;

    procedure Proc2(var GenFile : file);
    const
        STDW = 183;
    var
        FileOfInterest : ^FOfInt;
    begin
    loadAdr(FileOfInterest);
    loadAdr(GenFile);
    InLineByte(STDW);
    { Now FileOfInterest can be used }
                  .
                  .
                  .
    end;


begin
Proc1(F);
Proc1(F2);
Proc2(F);
Proc2(F2);
end.
```

12. Procedure/Function Parameter Types

12.1 Parameter Lists

PERQ Pascal permits, but does not require, the parameter list of procedures and functions which have been forward declared to be repeated at the site of the actual declaration. If given, the parameter list must match the previous declaration or a compilation error occurs. For redeclaration of function parameters, both the parameter list and the function type must be repeated.

12.2 Function Result Type

PERQ Pascal functions may return any type, with the exception of type FILE.

12.3 Procedures and Functions as Parameters

PERQ Pascal supports passing procedures and functions as parameters to other procedures or functions, as described by the BSI/ISO Pascal Standard [BSI79].

You must put the full description of the procedure parameter in the routine header (just as if it was defined by itself). For example:

```
Procedure EnumerateAll (directory: String; Function for each one
                        (filename: String): boolean; all:boolean);
```

Note that if the procedure is forward declared or in an export section, the routine header for this procedure cannot be repeated.

13. 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 the definition of a program given by the PASCAL User Manual and Report [JW74]. 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).

Modules which contain only type and constant declarations cause no run-time overhead; making them ideal for common declarations. It should also be noted that such modules may not be compiled (as errors will be produced), however they may be successfully imported.

13.1 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 file for that module. When compiling an import declaration, the source file containing the module to be imported must be available to the compiler.

Note: If the module is composed of several INCLUDE files, only those files from the file containing the program or module heading through the file which contains the word PRIVATE, must be available. (See "Compiler Switches" in this manual.)

The syntax for the IMPORTS declaration is:

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

13.2 EXPORTS Declaration Section

If a program or 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 with one exception: procedure and function bodies are not given in the exports section. Only forward references are given. (See "Declaration Relaxation" in this manual.) (See Chapter 11.2 in the "PASCAL User Manual and Report" [JW74].) The inclusion of "FORWARD;" in the EXPORTS reference is omitted.

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

If a program is to contain no EXPORTS Declaration section, the inclusion of PRIVATE following the 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>
```

14.0 Exceptions

PERQ Pascal provides an exception handling facility. Exceptions are typically used for error conditions. There are three steps to using an exception:

1) The exception must be declared.

2) A handler for the exception must be defined.

3) The exception must be raised.

An exception is declared by the word EXCEPTION followed by its name and optional list of parameters. For example:

```
EXCEPTION DivisionByZero(Numerator : integer);
```

Exceptions can be declared anywhere in the declaration portion of a program or module.

The second step is to specify the code to be executed when the exception condition occurs. This is done by defining a handler with the same name and parameter types as the declared exception, for example:

```
HANDLER DivisionByZero(Top : integer);
<block>
```

(See "Declaration Relaxation" in this document for a definition of <block>.) Essentially, a handler looks like a procedure with the word HANDLER substituted for the word PROCEDURE. Handlers may appear in the same places procedures are allowed, with one difference. Handlers cannot be global to a module (they may, however, be global to the main program). The number and type of parameters of a handler must match those of the corresponding exception declaration but the names of the parameters may be different. Multiple handlers can exist for the same exception as long as there is only one per name scope. The exception must be declared before its handler(s).

Raising an exception is analogous to a procedure call. The word RAISE appears before the name and parameters of the exception, for example:

```
RAISE DivisionByZero(N);
```

causes the appropriate handler to execute. The appropriate handler is determined by the current subprogram calling sequence. The run-time stack is searched until a subprogram containing a handler (of the same name) is found. The search starts from the subprogram which issues the RAISE.

For example:

```
program ExampleException;

exception Ex;

handler Ex;
  begin
  writeln('Global Handler for exception Ex');
  end;

procedure Proc1;
  begin
  raise Ex;
  end;

procedure Proc2;
  handler Ex;
    begin
    writeln('Local Handler for exception Ex')
    end;
  begin
  raise Ex;
  Proc1;
  end;

begin
raise Ex;
Proc2;
Proc1;
end.
```

produces the following output:

```
Global Handler for exception Ex
Local Handler for exception Ex
Local Handler for exception Ex
Global Handler for exception Ex
```

Handlers which are already active are not eligible for reactivation. In this case the search continues down the run-time stack until a non-active handler is found. A handler cannot, therefore, invoke itself by raising the same exception it was meant to handle. If a recursive procedure contains a handler, each activation of the procedure has its own eligible handler.

If an exception is raised for which no handler is defined or eligible, the system catches the exception and invokes the debugger. A facility is provided to allow the user to catch such exceptions before the system does. Handlers can be defined for the predefined exception ALL:

```
EXCEPTION ALL(ES,ER,PStart,PEnd : integer);
```

where

1) ES is the system segment number of the exception,

2) ER the routine number of the exception,

3) PStart the stack offset of the first word of the exception's original parameters and,

4) PEnd the stack offset of the word following the original parameters.

Any raised exception that does not have an eligible handler in the same or suceeding level (of the calling sequence), in which an ALL handler is defined, is caught by that ALL handler. The four integer parameter values are calculated by the system and supplied to the ALL handler. Extreme caution should be used when defining an ALL handler, as the handler also catches system exceptions. All cannot be raised explicitly. The ability to define ALL handlers is intended for "systems hackers" only.

The operating system provides several default handlers. Information on these handlers can be found in the "Program System" and "Module Except" descriptions in the Operating System Manual.

Since exceptions are generally used for serious errors, careful consideration should be given as to whether or not execution should be resumed after an exception is raised. When a handler terminates, execution resumes at the place following the RAISE statement. The handler can, of course, explicitly dictate otherwise. The EXIT and GOTO statements may prove useful here (See "Control Structures" in this manual.)

15.0 Dynamic Space Allocation and Deallocation

The PERQ Pascal Compiler supports the dynamic allocation procedures NEW and DISPOSE defined on page 105 of PASCAL User Manual and Report [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 are 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.

15.1 NEW

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

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

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

The extended form 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. The value 0 is used to indicate the default data segment.

Alignment specifies the desired alignment; Any power of 2 to 256 (2**0 through 2**8) is permissable. Do not use zero to specify the desired alignment.

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 usually returned. However, if memory is exhausted, the FULLMEMORY exception may be raised. If the call to NEW fails and raises FULLMEMORY, the user program will abort unless it

includes a handler for FULLMEMORY.

15.2 DISPOSE

DISPOSE is identical to the definition given in PASCAL User Manual and Report [JW74]. Note that the segment and alignment are never given to DISPOSE, only the pointer and tag field values.

16. Single Precision Logical Operations

The PERQ Pascal compiler supports a variety of single precision (INTEGER) logical operations. The operations supported 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.

16.1 And

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

LAND returns the bitwise AND of Val1 and Val2.

16.2 Inclusive Or

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

LOR returns the bitwise INCLUSIVE OR of Val1 and Val2.

16.3 Not

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

LNOT returns the bitwise complement of Val.

16.4 Exclusive Or

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

LXOR returns the bitwise EXCLUSIVE OR of Val1 and Val2.

16.5 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 0, and likewise when performing a right shift, the Most Significant Bit is filled with a 0.

16.6 Rotate

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

ROTATE returns Value rotated Distance bits to the right. ROTATE accepts only a positive integer in the range 0 through 15 for distance. Note that the direction of the ROTATE is the opposite of SHIFT.

17. Input/Output Intrinsics

PERQ's Input/Output intrinsics vary slightly from PASCAL User Manual and Report [JW74].

17.1 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 PASCAL User Manual and Report[JW74] REWRITE is the inclusion of the filename string.

17.2 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. RESET assigns the value of the first element of the file to F^. EOF(F) becomes false if F is not empty; otherwise, EOF(F) becomes true and F^ is undefined.

17.3 READ/READLN

PERQ Pascal supports extended versions of the READ and READLN procedures defined by PASCAL User Manual and Report [JW74]. Along with the ability to read longs, 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. Mixed upper and lower case are permissible.

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 fewer 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 maximum 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.

17.4 WRITE/WRITELN

PERQ Pascal provides many extensions to the WRITE and WRITELN procedures defined by PASCAL User Manual and Report [JW74]. Due to the scope of these extensions, the WRITE and WRITELN procedures are completely 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, long, 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 0 nor 1.

7. e1, the minimum field width, is optional. In general, the value e is written with e1 characters (with preceding 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 are printed.

8. e2, which is optional, is applicable only when e is of type long, integer (or subrange of integer) or real. If e is of type long or 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 twos 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.

17.5 CLOSE

The CLOSE intrinsic closes an output file. The intrinsic has the following form:

```
CLOSE(F)
```

F is the file variable to be associated with the file to be closed. Note that if you do not close a file for which a rewrite was performed and the program exits, or aborts, the data is lost.

18. Miscellaneous Intrinsics

18.1 StartIO

STARTIO is a special QCode (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.

18.2 RasterOp

RasterOp is a special QCode which is used to manipulate blocks of memory of arbitrary sizes. 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.

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. 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 operating system module Screen exports useful parameters.

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

```
RASTEROP(Function,
         Width,
         Height,
         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 precede 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 RasterOp 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. The specified value must be a multiple of four (4) and within the range 4 through 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. (See "New" in this document for details on buffer alignment.)

"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. The specified value must be a multiple of four (4) and within the range 4 through 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. (See "New" in this document for details on buffer alignment.)

18.3 WordSize

The WordSize intrinsic returns the number of words of storage required for any item which has a size associated with it. This includes constants, types, variables and functions. The intrinsic takes a single parameter, the item whose size is desired, and returns an integer.

Note: WordSize generates compile time constants, and hence may be used in constant expressions.

18.4 MakePtr

The MakePtr intrinsic permits the user to create a pointer to a data type given a system segment number and offset. Its use is intended for those who are familiar with the system and are sure of what they are doing. The function takes three parameters. The first two are the system segment number and offset within that segment to be used in creating the pointer, respectively, given as integers. The last parameter is the type of the pointer to be created. MakePtr returns a pointer of the type named by the third parameter.

Note: The next seven intrinsics, MakeVRD, InLineByte, InLineWord, InLineAWord, LoadExpr, LoadAdr and StorExpr, require that the user have knowledge of how the compiler generates code (which will not be discussed here). These intrinsics are intended for "system hacking", and are made available for those who know what they are doing. The programmer who wishes to experiment with these may find the QCode disassembler, QDIS, is very useful to determine if the desired results were produced.

18.5 MakeVRD

MakeVRD is used to load a variable routine descriptor for a procedure or function. (See "Routine Calls and Returns" for a description of LVRD and CALLV in the QCode Reference Manual.) The variable routine descriptor is left on the expression stack of the PERQ, and any further operations must be performed by the user. This procedure takes one parameter, the name of the function or procedure for which the variable routine descriptor is to be loaded. The use of this intrinsic assumes that the programmer is familiar with QCode (primarily a "hacker's" intrinsic).

18.6 InLineByte

InLineByte permits the user to place explicit bytes directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of actual QCodes into a program. InLineByte requires one parameter, the byte to be inserted. The type of this parameter must be either integer or subrange of integer.

18.7 InLineWord

InLineWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of direct QCodes into a program. InLineWord requires one parameter, the word to be inserted. This word

will be inserted immediately as the next two bytes of the code stream (no word alignment is performed). The type of this parameter must be either integer or subrange of integer.

18.8 InLineAWord

InLineAWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of direct QCodes into a program. InLineAWord requires one parameter, the word to be inserted. This word is placed on the next word boundary of the code stream. The type of this parameter must be either integer or subrange of integer.

18.9 LoadExpr

The LoadExpr intrinsic takes an arbitrary expression as its parameter and "loads" the value of the expression. The result of the "load" is wherever the particular expression type would normally be loaded (expression stack for scalars, memory stack for sets, etc.).

18.10 LoadAdr

The LoadAdr intrinsic loads the address of an arbitrary data item onto the expression stack. The parameter to LoadAdr, the item whose address is desired, may include array indexing, pointer dereferencing and field selections. The address which is left on the expression stack will be a virtual address if the parameter includes either the use of a VAR parameter or a pointer dereference; otherwise a 20-bit stack offset will be loaded.

18.11 StorExpr

StorExpr stores the single word on top of the expression stack in the variable given as a parameter. The destination for the store operation must not require any address computation; the destination must be a local, intermediate or global variable; it must not be a VAR parameter; if it is a record, a field specification may be given.

18.12 Float

The Float intrinsic converts an integer into a floating point number.

18.13 Stretch

The Stretch intrinsic converts a single precision integer to a double precision integer.

18.15 Shrink

The Shrink intrinsic converts a double precision integer to a single precision integer. If the double precision integer is outside the range of -32768 to +32767, a runtime error occurs.

19. Command Line and Compiler Switches

19.1 Command Line

The PERQ Pascal compiler is invoked by typing a compile command line to the PERQ Operating System. The syntax for the compile command line is:

COMPILE [<InputFile>] [~ <OutputFile>] {</Switch>}.

<InputFile> is the name of the source file to be compiled. The compiler searches for <InputFile>. If it does not find <InputFile>, it appends the extension ".PAS" and searches again. If <InputFile> is still not found, the user will be prompted for an entire command line. If <InputFile> is not specified, the compiler uses for <InputFile> the last file name remembered by the system.

<OutputFile> is the name of the file to contain the output of the compiler. The extension ".SEG" will be appended to <OutputFile> if it is not already present. Note that if <OutputFile> is not specified, the compiler uses the file name from <InputFile>. Then, if the ".PAS" extension is present, it is replaced with the ".SEG" extension, else if the ".PAS" extension is not present, the ".SEG" extension is appended. If <OutputFile> already exists, it will be rewritten.

</Switch> is the name of a compiler switch. All compiler switches specified on the command line must begin with the "/" character. Any number of switches may be specified, and if a switch is specified multiple times, the last occurrence is used. Also, if the /HELP switch is specified, the other information on the command line is ignored. The available switches are defined in the following sections.

The interface to the compiler is slightly different when invoked from a command file. If there is an error in the command line, it does not prompt for input, but rather logs an entry in a file called

CmdFilePascal.ERR

in the local directory. If this file does not exist in the local directory, the compiler creates the file and inserts the entry. Otherwise, the compiler simply appends the entry to the existing file. The compiler never deletes this file.

19.2 Compiler Switches

PERQ Pascal compiler switches may be set either in a mode similar to the convention described on pages 100-102 of PASCAL User Manual and Report [JW74] or on the command line described above (see above Section, "Command Line"). The first form of compiler switches may be written as comments and are designated as such by a dollar sign character ($) as the first character of the comment followed by the switch (unique abbreviations are acceptable) and possibly switch parameters. The second form is given after the input file specification in the command line preceded by the slash (/) character. The actual switches provided by the PERQ Pascal compiler, although similar in syntax, bear little resemblance to the switches described in PASCAL User Manual and Report [JW74].

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

19.2.1 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:

{$INCLUDE FILENAME}

The characters between the "$INCLUDE" 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.

If the file FILENAME does not exist, ".PAS" will be concatenated onto the end of FILENAME, and a second attempt will be made to find the file.

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).

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

19.2.2 List Switch

The List Switch controls whether or not the compiler generates a program listing of the source text. The default is to not generate a list file. The format for the List switch is:

{$LIST <filename>}

or

/LIST [= <filename>]

where <filename> is the name of the file to be written. The extension ".LST" will be appended to <filename> if it is not already present. If <filename> is not specified, the compiler uses the source file name. If the ".PAS" extension is present, it is replaced with the ".LST" extension, else if the ".PAS" extension is not present, the ".LST" extension is appended. Like the file inclusion mechanism, in the comment form of the switch, the filename is taken as all characters between the "$LIST" 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, and procedure number.

19.2.3 Range Checking

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

Default value: Range checking enabled

$RANGE+	or /RANGE	enables range checking
$RANGE-	or /NORANGE	disables range checking
$RANGE=		resumes the state of range checking which was in force before the previous $RANGE-or $RANGE+ switch.

If "$RANGE" is not followed by a "+", "-", or "=", then "+" is assumed.

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

19.2.4 Quiet Switch

This switch is used to enable or disable the Compiler from displaying the name of each procedure and function as it is compiled.

Default value: Display of procedure and function names enabled

$QUIET+ or /VERBOSE	enables display of procedure and function names
$QUIET- or /QUIET	disables display of procedure and function names
$QUIET=	resumes the state of the quiet switch which was in force before the previous $QUIET- or $QUIET+ switch.

If "$QUIET" is not followed by a "+", "-", or "=", then "+" is assumed.

19.2.5 Symbols Switch

This switch is used to set the number of symbol table swap blocks used by the Compiler. As the number of symbol table swap blocks increases, compiler execution time becomes shorter; however physical memory requirements increase (and the Compiler may abort due to insufficient memory). The format for this switch is:

~/SYMBOLS = <# of Symbol Table Blocks>

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

The default number of symbol table blocks and the maximum number of symbol table blocks are both dependent on the size of memory. For systems with 256k bytes of main memory, the default number of symbol table blocks is 24 and the maximum number of symbol table blocks is 32. Note that you can specify more than 32 symbol table blocks with a 256k byte system, but performance usually degrades considerably. For systems with 512k or 1024k bytes of main memory, the default number of symbol table blocks is 200 and the maximum number of symbol table blocks is also 200.

19.2.6 Automatic RESET/REWRITE

The PERQ Pascal compiler automatically generates a RESET(INPUT) and REWRITE(OUTPUT). This may be disabled if desired with the use of the AUTO switch. The format for this switch is:

Default value: Automatic initialization enabled

$AUTO+ or /AUTO enables automatic initialization

$AUTO- or /NOAUTO disables automatic initialization

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

If the comment form of this switch is used, it must precede the BEGIN of the main body of the program.

19.2.7 Procedure/Function Names

The PERQ Pascal compiler generates a table of the procedure and function names at the end of the ".SEG" file, if so directed. This table may be useful for debugging programs. The format for this switch is:

Default value: Name Table is generated

$NAMES+ or /NAMES enables generation of the Name Table

$NAMES- or /NONAMES disables generation of the Name Table

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

Note: currently two programs, the debugger and the disassembler, use the information stored in the Name Table.

19.2.8 Version Switch

The Version Switch permits the inclusion of a version string in the first block of the ".SEG" file. This string has a maximum length of 80 characters. Currently this string is not used by any other PERQ software, however, it may be accessed by user programs to identify ".SEG" files. The format for this switch is:

$VERSION <string> or /VERSION = <string> to set the Version string.

When using the $VERSION form of the switch, the version string is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored. If the /VERSION form is used, the version string is terminated by either the end of the command line or the occurrence of a '/' character (hence a '/' may not appear in the version string).

19.2.9 Comment Switch

The Comment Switch permits the inclusion of arbitrary text in the first block of the ".SEG" file. This string has a maximum length of 80 characters. It is particularly useful for including copyright notices in ".SEG" files. The format for this switch is:

$COMMENT <string> or /COMMENT = <string> to set the comment string.

When utilizing the $COMMENT form of the switch, the comment text is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored.

19.2.10 Message Switch

The Message Switch causes the text of the switch to be printed on the user's screen when the switch is parsed by the compiler. It has no effect on the compilation process. The format for this switch is:

$MESSAGE <string> to print <string> on the console during compilation

The message is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored.

Note: There is no command line form for this switch, it may only be used in its comment form.

19.2.11 Conditional Compilation

The PERQ Pascal conditional compilation facility is implemented through the standard switch facility. There are three switches which are used for conditional compilations. The first is the $IFC switch, which has the following form:

{$IFC <boolean expression> THEN}

This switch indicates the beginning of a region of conditional compilation. If the boolean expression, evaluated at compile time, is true, the text to follow is included in the compilation. If the boolean expression evaluates to false, then the text which follows is not included.

The region of conditional compilation is terminated by the $ENDC switch:

{$ENDC}

Upon encountering the $ENDC switch, the state of compilation returns to whatever state was present prior to the most recent $IFC.

The remaining switch is the $ELSEC switch, and it functions much in the same way as the else clause in an IF statement. If the boolean expression of the $IFC switch is true, then the $ELSEC text is ignored, otherwise it is included.

If a $ELSEC switch is used, no $ENDC preceeds the $ELSEC; the $ELSEC signals the end of the $IFC region. A $ENDC is then used to terminate the $ELSEC clause.

Conditional compilations may be nested.

The following are two examples of the conditional compilation mechanism:

```
Const CondSw = TRUE;
PROCEDURE Test;
begin
{$IFC CondSw THEN}
    Writeln('CondSw is true');
{$ENDC}
end { Test };

TYPE Base = record i,j,k:integer end;
{$IFC WORDSIZE(Base) = 3 THEN}
    Cover = array[0..2] of integer
{$ELSEC}
    Cover = array[0..10] of integer
{$ENDC};
```

19.2.12 Errorfile Switch

When the compiler detects an error in a program, it displays error information (file, error number, and the last two lines where the error occurred) on the screen and then requests whether or not to continue. The /ERRORFILE switch overrides this action. When you

specify the switch and the compiler detects an error, the error information is written to a file and there is no query of the user. However, the compiler does display the total number of errors encountered on the screen.

The format for this switch is: /ERRORFILE [= <filename>]

where <filename> is the name of the file to be written. The extension ".ERR" will be appended to <filename> if it is not already present. If <filename> is not specified, the compiler uses the source file name. If the ".PAS" extension is present, it is replaced with the ".ERR" extension, else if the ".PAS" extension is not present, the ".ERR" extension is appended.

The error file exists after a compilation if and only if you specify the /ERRORFILE switch and an error is encountered. If the file <filename>.ERR already exists from a previous compilation, it will be rewritten, or deleted in the case of no compiliation errors. This switch allows compilations to be left unattended.

19.2.13 Help Switch

The Help switch provides general information and overrides all other switches. The format is /HELP.

20.0 Quirks and Other Oddities

The following are descriptions of known quirks and problems with the PERQ Pascal compiler. Future releases may correct these problems.

1. FOR loops with an upper bound of greater than 32,766 never terminate.

2. The last line of any PROGRAM or MODULE must end with a carriage return, or an "Unexpected End of Input" error occurs.

3. Although unique abbreviations are accepted for switches, the following abbreviations cause compilation errors:

Switch	Bad Abbreviation
$ELSEC	$ELSE
$ENDC	$END
$IFC	$IF
$INCLUDE	$IN

4. Procedures and functions which are forward declared (this includes EXPORT declarations) and contain procedure parameters, may not have their parameter lists redeclared at the site of the procedure body.

5. The compiler currently permits the use of an EXIT statement where the routine to be exited from is at the same lexical level as the routine containing the EXIT statement. For example:

```
program Quirk;

    procedure ProcOne;
    begin
    end;

    procedure ProcTwo;
    begin
    exit(ProcOne)
    end;

begin
ProcTwo
end.
```

If there is no invocation of the routine to be

exited on the run-time stack, the PERQ hangs and must be re-booted.

6. The filename specification given in IMPORTS Declarations must start with an alphabetic character.

7. Record comparisons involving packed records (illegal comparisons) will not be caught unless the word PACKED appears explicitly in the record definition. For example, records with fields of user-defined type Foo, where Foo contains packed information, are considered comparable by the compiler when in actuality they are not.

8. Reals and longs cannot be used together in an expression.

9. The compiler will not detect an error in the definition or use of a set that exceeds set size limitations. If such a set is used, incorrect code will be generated.

10. Many functions that exist for integers (for example, LAND) are not implemented for longs. Also, you can only use Longs as assignments in arithmetic expressions.

11. The RECAST intrinsic does not work with two-word scalers (for example, LONG) and arrays.

20. References

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

[FP80] "An Implementation Guide to a Proposed Standard for Floating Point", Computer, January 1980

[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)