



**PERQ** Systems  
Corporation

**ACCENT  
LANGUAGES MANUAL  
Volume 1**

**December 7, 1984**

**This manual is for use with Accent Release S6.**

Copyright © 1984 PERQ Systems Corporation  
2600 Liberty Avenue  
P. O. box 2600  
Pittsburgh, PA 15230  
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

This document is not to be reproduced in any form or transmitted in whole or in part without the prior written authorization of PERQ Systems Corporation.

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.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

## **ACCENT LANGUAGES MANUAL**

### **PREFACE**

This manual contains documentation on the languages that are available with Accent, the operating system for the PERQ workstation. Accent was developed jointly by PERQ Systems Corporation and by the Spice Project in the Computer Science Department at Carnegie-Mellon University. "Spice" is an acronym for Scientific Personal Integrated Computing Environment.

Currently the Pascal, C, and Lisp languages can be used with the Accent operating system, and FORTRAN will become available in a future release. This manual contains information on each of the languages for which you have purchased a compiler, except Lisp which is covered in a separate manual.

Before you use this manual you should be familiar with the material in the *Accent User's Manual* and the *Accent Programming Manual*.

Other manuals available for Accent are:

- Accent Microprogramming Manual
- Accent Lisp Manual
- Accent Qnix Manual (forthcoming in a future release)
- Accent System Administration Manual

Throughout the manuals the term "PERQ" refers to all models of the PERQ workstation unless stated otherwise. When a distinction is made between the PERQ workstation and the PERQ2 workstation, the term "PERQ2" refers to both Model LN-3000 and LN-3500.

The following symbols have been used throughout the Accent manuals:

- < > Material that is to be replaced by symbols or text as explained in the accompanying text. Do not type the angle brackets. Example: <filename> indicates that you should type the name of your file.
- [ ] Optional feature. Do not type the square brackets.
- { } 0 to n repetitions of an optional item. Do not type the braces.

#### CAPITALS

Literal, to be reproduced exactly as shown (although it may be reproduced in upper-case or lower-case). Example: <filename.CMD> indicates that the filename must contain the extension .cmd.

| "Or"--choice between the items shown on either side of the symbol.

CTRL Control key

ESC, INS Escape key (labeled as ACC ESC or INS on various models)

DEL Delete key (labeled as REJ DEL on some models)

HELP Help key

LF Linefeed Key

**RETURN** Carriage return

*italics* Input to be typed by the user



## **ACCENT LANGUAGES MANUAL**

### **TABLE OF CONTENTS**

#### **VOLUME 1**

Preface

Pascal:

PERQ / Accent Pascal Extensions

Pascal Library

PasMac: A Pascal Macro-Processor

#### **VOLUME 2**

Pascal (con'd)

Module Index

Kraut: PERQ Pascal Remote

Symbolic Debugger

Matchmaker: The Accent Remote Procedure  
Call Language

C:

C System Interfaces

PERQ C Programming

Index



# **PERQ / ACCENT PASCAL EXTENSIONS**

**December 7, 1984**

Copyright © 1984 PERQ Systems Corporation  
2600 Liberty Avenue  
P. O. Box 2600  
Pittsburgh, PA 15230  
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

This document is not to be reproduced in any form or transmitted in whole or in part without the prior written authorization of PERQ Systems Corporation.

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.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

<u>Table of Contents</u>	<u>Page</u>
<u>1. Introduction</u>	<u>PE-1</u>
<u>2. Lexical Tokens</u>	<u>PE-3</u>
2.1. Identifiers	PE-3
2.2. Numbers	PE-3
2.2.1. Whole Numbers	PE-3
2.2.2. Octal Whole Number Constants	PE-3
2.2.3. Floating Point Numbers	PE-4
2.3. Lexical Alternatives	PE-4
<u>3. Blocks, Scope, and Activations</u>	<u>PE-5</u>
3.1. Block	PE-5
3.2. Scope	PE-6
3.3. Activations	PE-6
<u>4. Constant-Definitions</u>	<u>PE-7</u>
<u>5. Type-Definitions</u>	<u>PE-9</u>
5.1. General	PE-9
5.2. Simple-Types	PE-9

<b>5.2.1. Required Simple-Types</b>	<b>PE-9</b>
<b>5.2.2. Subrange-Types</b>	<b>PE-9</b>
<b>5.3. Structured-Types</b>	<b>PE-9</b>
<b>5.3.1. Array-types</b>	<b>PE-9</b>
<b>5.3.2. String-types</b>	<b>PE-9</b>
<b>5.3.3. Record-types</b>	<b>PE-11</b>
<b>5.3.4. Set-types</b>	<b>PE-11</b>
<b>5.3.5. File-types</b>	<b>PE-11</b>
<b>5.4. Pointer-types</b>	<b>PE-13</b>
<b>5.5. Compatible Types</b>	<b>PE-14</b>
<b>5.6. Assignment-Compatibility</b>	<b>PE-14</b>
 <b>6. Declarations and Denotations of Variables</b>	 <b>PE-17</b>
 <b>7. Procedure, Function, Exception and Handler</b>	
<b>    Declarations</b>	<b>PE-19</b>
<b>7.1. Procedure-Declarations</b>	<b>PE-19</b>
<b>7.2. Function-Declarations</b>	<b>PE-19</b>
<b>7.3. Exception-Declarations</b>	<b>PE-19</b>
<b>7.3.1. ALL EXCEPTION</b>	<b>PE-20</b>
<b>7.4. Handler-Declarations</b>	<b>PE-21</b>
<b>7.5. Parameters</b>	<b>PE-22</b>
<b>7.6. Required Procedures</b>	<b>PE-22</b>
<b>7.6.1. File handling procedures</b>	<b>PE-22</b>
<b>7.6.1.1. REWRITE</b>	<b>PE-22</b>
<b>7.6.1.2. RESET</b>	<b>PE-23</b>
<b>7.6.1.3. CLOSE</b>	<b>PE-23</b>

<b>7.6.2. Dynamic allocation procedures</b>	<b>PE-23</b>
<b>7.6.2.1. NEW</b>	<b>PE-24</b>
<b>7.6.2.2. DISPOSE</b>	<b>PE-25</b>
<b>7.6.3. Transfer procedures</b>	<b>PE-25</b>
<b>7.6.4. PERQ-specific procedures</b>	<b>PE-25</b>
<b>7.6.4.1. StartIO</b>	<b>PE-25</b>
<b>7.6.4.2. RasterOp</b>	<b>PE-26</b>
<b>7.6.4.3. MakePtr</b>	<b>PE-28</b>
<b>7.6.4.4. MakeVRD</b>	<b>PE-28</b>
<b>7.6.4.5. InLineByte</b>	<b>PE-29</b>
<b>7.6.4.6. InLineWord</b>	<b>PE-29</b>
<b>7.6.4.7. InLineAWord</b>	<b>PE-29</b>
<b>7.6.4.8. LoadExpr</b>	<b>PE-30</b>
<b>7.6.4.9. LoadAdr</b>	<b>PE-30</b>
<b>7.6.4.10. StorExpr</b>	<b>PE-30</b>
<b>7.6.4.11. NoPageFault</b>	<b>PE-30</b>
<b>7.7. Required Functions</b>	<b>PE-31</b>
<b>7.7.1. Arithmetic functions</b>	<b>PE-31</b>
<b>7.7.2. Transfer functions</b>	<b>PE-31</b>
<b>7.7.2.1. TRUNC and ROUND</b>	<b>PE-31</b>
<b>7.7.2.2. STRETCH</b>	<b>PE-32</b>
<b>7.7.2.3. SHRINK</b>	<b>PE-32</b>
<b>7.7.2.4. FLOAT</b>	<b>PE-32</b>
<b>7.7.2.5. Type coercion - RECAST</b>	<b>PE-32</b>
<b>7.7.3. Ordinal functions</b>	<b>PE-33</b>
<b>7.7.4. Boolean functions</b>	<b>PE-33</b>
<b>7.7.5. PERQ-specific functions</b>	<b>PE-34</b>
<b>7.7.5.1. WordSize</b>	<b>PE-34</b>
<b>7.7.5.2. LENGTH</b>	<b>PE-34</b>
<b>7.7.5.3. Logical operations</b>	<b>PE-34</b>
<b>7.7.5.3.1. And</b>	<b>PE-35</b>

<b>7.7.5.3.2. Inclusive Or</b>	<b>PE-35</b>
<b>7.7.5.3.3. Not</b>	<b>PE-35</b>
<b>7.7.5.3.4. Exclusive Or</b>	<b>PE-35</b>
<b>7.7.5.3.5. SHIFT</b>	<b>PE-35</b>
<b>7.7.5.3.6. ROTATE</b>	<b>PE-36</b>
 <b>8. Expressions</b>	<b>PE-37</b>
<b>8.1. Mixed-Mode Expressions</b>	<b>PE-37</b>
<b>8.2. Record Comparisons</b>	<b>PE-38</b>
 <b>9. Statements</b>	<b>PE-39</b>
<b>9.1. General</b>	<b>PE-39</b>
<b>9.2. Simple-Statements</b>	<b>PE-39</b>
<b>9.2.1. EXIT</b>	<b>PE-39</b>
<b>9.2.2. RAISE</b>	<b>PE-40</b>
<b>9.3. Structured-Statements</b>	<b>PE-42</b>
<b>9.3.1. Extended case statements</b>	<b>PE-42</b>
 <b>10. Input and Output</b>	<b>PE-45</b>
<b>10.1. READ / READLN</b>	<b>PE-45</b>
<b>10.2. WRITE / WRITELN</b>	<b>PE-46</b>
<b>10.3. The Procedure Page</b>	<b>PE-47</b>
 <b>11. Programs and Modules</b>	<b>PE-49</b>
<b>11.1. IMPORTS Declarations</b>	<b>PE-50</b>
<b>11.2. EXPORTS Declarations</b>	<b>PE-50</b>

<b>12. Command Line and Compiler Switches</b>	<b>PE-53</b>
<b>12.1. Command Line</b>	<b>PE-53</b>
<b>12.2. Compiler Switches</b>	<b>PE-54</b>
<b>12.2.1. File inclusion</b>	<b>PE-54</b>
<b>12.2.2. LIST switch</b>	<b>PE-55</b>
<b>12.2.3. RANGE checking</b>	<b>PE-56</b>
<b>12.2.4. QUIET switch</b>	<b>PE-56</b>
<b>12.2.5. Automatic RESET / REWRITE</b>	<b>PE-57</b>
<b>12.2.6. Global INPUT / OUTPUT switch</b>	<b>PE-57</b>
<b>12.2.7. Mixed-mode permission switch</b>	<b>PE-58</b>
<b>12.2.8. Procedure / function names</b>	<b>PE-58</b>
<b>12.2.9. SCROUNGE (symbol table for                   debugger) switch</b>	<b>PE-59</b>
<b>12.2.10. VERSION Switch</b>	<b>PE-59</b>
<b>12.2.11. COMMENT Switch</b>	<b>PE-60</b>
<b>12.2.12. MESSAGE Switch</b>	<b>PE-60</b>
<b>12.2.13. Conditional compilation</b>	<b>PE-61</b>
<b>12.2.14. ERRORFILE switch</b>	<b>PE-62</b>
<b>12.2.15. HELP switch</b>	<b>PE-63</b>
<b>13. Quirks and Oddities</b>	<b>PE-65</b>
<b>14. References</b>	<b>PE-69</b>



## 1. Introduction

PERQ Pascal is an upward-compatible extension of the programming language Pascal defined in *PASCAL User Manual and Report* [JW74]. The BSI/ISO and IEEE/ANSI standardization organizations have also further formalized and standardized the language as given in *Specification for Computer Programming Language Pascal* [BSI82] and *American National Standard Pascal Computer Programming Language* [IEEE83] respectively. This document describes only the extensions to Pascal applicable to the PERQ workstation. 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. The general outline of this document follows that of both the *BSI/ISO* and the *IEEE/ANSI Pascal Standards* [BSI82] and [IEEE83]. This format should assist the reader in correlating extensions with the standard, underlying, Pascal language.

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* [BSI82], the *UCSD Workshop on Systems Programming Extensions to the Pascal Language* [UCSD79] and, most notably, *Pascal\** [P\*].



## **2. Lexical Tokens**

---

### **2.1. Identifiers**

The underscore character " \_ " may be included as a significant character in identifiers. The syntax for identifiers is:

```
<identifier> ::= <letter> {<letter> | <digit> | _}
```

### **2.2. Numbers**

#### **2.2.1. Whole Numbers**

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.

Arithmetic operations and comparisons are defined for both single and double precision whole numbers.

#### **2.2.2. Octal 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. Thus, when using octal constants, a constant with the 16th bit set is interpreted as a negative INTEGER, not as a positive LONG.

### 2.2.3. 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.

## 2.3. Lexical Alternatives

PERQ Pascal supports none of the Lexical Alternatives defined in either [JW74], [BSI82], or [IEEE83]. PERQ Pascal does, however, support the use of the '(\*' and '\*)' tokens as a second pair of comment delimiters. In PERQ Pascal the '(\*') and '})' are NOT synonyms for each other. Thus the '(\*', '\*)' pair may be used to enclose comments delimited by the '{', '}' pair (and vice versa, of course). For example:

```
{ first comment - it contains a second
  /* second comment - embedded within the first */
  end of first comment }
```

is valid PERQ Pascal commentary.

### **3. Blocks, Scope, and Activations**

---

#### **3.1. Block**

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, however, that an identifier be declared before it is used. As in [JW74], 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 11.1, IMPORTS Declarations for a description of  
<import declaration part>.

### **3.2. Scope**

As in *PASCAL User Manual and Report* [JW74].

### **3.3. Activations**

As in *PASCAL User Manual and Report* [JW74].

## **4. Constant-Definitions**

---

PERQ Pascal extends the meaning of a constant to include expressions which may be evaluated at compile-time. Constant expressions are syntactically identical to normal, run-time evaluated expressions. The value of the constant expression is, however, determined during the compilation process. Thus all constant expression operands must be either literal constants, identifiers denoting previously defined constants, or other constant expressions. Valid operators include all of the arithmetic, logical, and relational operators (with the exception of the Set operators). Any of the intrinsic functions which reasonably return a constant result when given a constant argument(s) are also permitted as operands in a constant expression (see 7.7, Required Functions).

The new syntax for constants is:

<constant> ::= <expression>



## 5. Type-Definitions

### **5.1. General**

As in *PASCAL User Manual and Report* [JW74].

### **5.2. Simple-Types**

#### **5.2.1. Required Simple-Types**

As in *PASCAL User Manual and Report* [JW74].

#### **5.2.2. Subrange-Types**

As in *PASCAL User Manual and Report* [JW74].

### **5.3. Structured-Types**

#### **5.3.1. Array-types**

As in *PASCAL User Manual and Report* [JW74].

#### **5.3.2. String-types**

PERQ Pascal includes a character string facility. This facility provides variable length (e.g. the length is dynamic; established at run-time) character strings with a static (e.g. compile-time established) maximum length limit. The default maximum static length of a STRING variable is 80 characters. This may be overridden in the declaration of a STRING variable by appending the desired maximum static 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 static length
                           of 80 characters }

ShortStr : STRING[12];   { maximum static 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 static length of the destination -- the maximum static length of the two strings need not be the same.

The individual characters within a STRING may be selectively read and written as if they were elements of a packed array of characters. 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 static 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 static 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 }
```

### 5.3.3. Record-types

As in *PASCAL User Manual and Report* [JW74].

### 5.3.4. Set-types

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.

### 5.3.5. File-types

PERQ Pascal permits the use of files as the component type of arrays, pointers and record fields. In addition to the standard file type, PERQ Pascal also provides the Generic file type. 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. Passed as an argument to the LOADADR intrinsic.

The following example shows 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
  loadAddr(GenFile);
  storeexpr(UseGenFile.Offset);
  storeexpr(UseGenFile.Segment);
  { Now FileOfInterest can be used }

  .
  .

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

  .
  .

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

## 5.4. Pointer-types

In addition to the standard pointer type, PERQ Pascal also provides the Generic pointer type. 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 domain-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 Procl(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' }
  Procl(I);
  new(C);
  C[1] := 'C';
  C[2] := 'D';
  Procl(C);
end.
```

produces output

AB  
CD

## 5.5. Compatible Types

PERQ Pascal supports strict type compatibility as defined by the BSI/ISO Pascal Standard [BSI82], with one addition: any two strings, regardless of their maximum static lengths, are considered compatible.

## 5.6. Assignment-Compatibility

Single precision whole numbers are assignment compatible with double precision whole numbers and floating point numbers. That is, expressions of type INTEGER (or subrange of INTEGER) may be assigned to variables of type LONG, subrange of LONG, or REAL. Also, INTEGER expressions may be passed by value (only) to LONG, subrange of LONG, or REAL formal parameters. In the same manner, double precision whole number types are assignment compatible with floating point types.

Note that this mixing of whole number and floating point modes may be disabled through the use of the NoMixedModePermitted compiler invocation switch (see 12.2.7, Mixed-Mode Permission Switch).

Double precision whole number types and floating point types are not assignment compatible with single precision whole number types. Likewise, floating point types are not assignment compatible with double precision number types. The type coercion intrinsics SHRINK, ROUND, and TRUNC may be used to convert these modes as needed. See the "Transfer Functions" section of this document for a description of these coercion

intrinsics.



## **6. Declarations and Denotations of Variables**

As in *PASCAL User Manual and Report* [JW74].



## **7. Procedure, Function, Exception and Handler Declarations**

### **7.1. Procedure-Declarations**

As in *PASCAL User Manual and Report* [JW74].

### **7.2. Function-Declarations**

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

### **7.3. Exception-Declarations**

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 (see Handler-Declarations).
3. The exception must be raised (see 9.2.2, Raise Statements).

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

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

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

### 7.3.1. ALL EXCEPTION

If an exception is raised for which no handler is defined or eligible (see Raise Statements), 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. This is accomplished by defining a Handler for the predefined exception ALL:

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

where

- ES is the system segment number of the exception,
- ER the routine number of the exception,
- PStart the stack offset of the first word of the exception's original parameters, and
- 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 succeeding 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.

## 7.4. Handler-Declarations

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 (see Exception-Declarations, above).
2. A handler for the exception must be defined.
3. The exception must be raised (see 9.2.2, RAISE Statements).

A Handler specifies the code to be executed when an exception condition occurs. This is accomplished by defining a handler with the same name and parameter types as the declared exception, for example:

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

(See Section 3.1 for an explanation 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).

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, dictate otherwise. The EXIT and GOTO statements may prove useful here (See 9.2, Simple-Statements).

## 7.5. Parameters

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 repeated, the parameter list must match the previous declaration or a compilation error occurs.

Note that to redeclare procedures / functions that have a function as a parameter, both the parameter list and the function type must be repeated within the redeclared parameter list (if it appears). However, to redeclare procedure / functions that have a procedure as a parameter, their parameter list must NOT appear.

## 7.6. Required Procedures

### 7.6.1. File handling procedures

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

#### 7.6.1.1. REWRITE

The REWRITE procedure has the following alternative 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. The required boolean function, EOF(F), becomes true and a new file may be written. The only difference between PERQ Pascal and *PASCAL User Manual and Report* [JW74] is that REWRITE has the inclusion of the optional filename string.

### 7.6.1.2. RESET

The RESET procedure has the following alternative form:

`RESET(F,Name)`

F is the file variable (of type text) 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.

### 7.6.1.3. 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, any data written to that file is lost.

## 7.6.2. Dynamic allocation procedures

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 workstation's memory architecture.

There are two features of the workstation'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, the workstation supports multiple data heaps from which dynamic allocation may

be performed. This facilitates grouping data together which are to be accessed together, which may improve the PERQ workstation's performance due to improved paging. For further information of the memory architecture and available functions see the "File System" document of the *Accent Programming Manual*.

#### 7.6.2.1. 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 heap number from which the allocation is to be performed. This number is returned to the user when creating a new heap. The value 0 is used to indicate the default data heap.

Alignment specifies the desired alignment. Any power of 2 to 256 ( $2^{**0}$  through  $2^{**8}$ ) is permissible. Do not use zero to specify the desired alignment.

If the extended form of NEW is used, both a segment and an 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.

### 7.6.2.2. 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.

### 7.6.3. Transfer procedures

The transfer procedures Pack and Unpack are not implemented in PERQ Pascal.

### 7.6.4. PERQ-specific procedures

The seven intrinsics, MakeVRD, InLineByte, InLineWord, InLineAWord, LoadExpr, LoadAdr and StorExpr, require that the user have knowledge of how the compiler generates code. These intrinsics are made available only for those who know what they are doing. The programmer who wishes to experiment with these may find that the QCode disassembler, QDIS, is very useful to determine if the desired results were produced.

#### 7.6.4.1. StartIO

StartIO is a special QCode (See the *Pascal/C Machine 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.

### 7.6.4.2. 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, the lower right would be (767,1023) for portrait and (1279,1023) for landscape. 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):

<u>Function</u>	<u>Name</u>	<u>Action</u>
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 XOR NOT Src

The symbolic names are exported by the file "SapphDefs.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 for a portrait screen and 80 for a landscape screen. The specified value must be a multiple of four (4) and within the range 4 through 48 (80 for landscape).

"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 (see 7.6.2.1, New, 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 for a portrait screen or 80 for a landscape screen. The specified value must be a multiple of four (4) and within the range 4 through 48 (80 for landscape).

"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 7.6.2.1, New, for details on buffer alignment.)

#### 7.6.4.3. 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 third parameter is the type of pointer to be created. MakePtr returns a pointer of the type named by the third parameter.

#### 7.6.4.4. 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 document *Pascal/C Machine Reference* in the *Accent Microprogramming Manual*. The variable routine descriptor is left on the expression stack of the PERQ workstation, 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.

#### **7.6.4.5. InLineByte**

InLineByte permits the user to place explicit bytes directly into the code stream generated by the compiler. This intrinsic is particularly 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.

#### **7.6.4.6. InLineWord**

InLineWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particularly 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.

#### **7.6.4.7. InLineAWord**

InLineAWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particularly 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.

#### **7.6.4.8. 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.).

#### **7.6.4.9. 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 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.

#### **7.6.4.10. 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.

#### **7.6.4.11. NoPageFault**

NoPageFault is used in conjunction with InlineByte, InlineWord, or InlineAWord to ensure that the inserted sequence of bytes representing a QCode does not span a virtual memory page boundary. NoPageFault requires one parameter, the number of byte locations needed before the next page boundary occurs. The type of this parameter must be integer or subrange of integer. If the desired number of locations cannot be reserved, the compiler inserts NOOP QCodes to fill the current page, and the desired number of locations is reserved on the next memory page. Note

that when given a location count larger than the size of a virtual page, NoPageFault truncates the count down to the size of a memory page.

## 7.7. Required Functions

### 7.7.1. Arithmetic functions

Recognition of the required arithmetic functions: Sin, Cos, Exp, Ln, Sqrt, and Arctan is not built into the PERQ Pascal compiler. The user may gain access to them by importing the operating system module RealFunctions.

### 7.7.2. Transfer functions

#### 7.7.2.1. TRUNC and ROUND

TRUNC and ROUND, as defined in the standard, convert a floating point number into a whole number, provided that the value of the floating point number is within the legal range of the desired precision. If not, a runtime error occurs. Note that TRUNC (or ROUND) will produce either a single or double precision whole number, dependent upon the context of the expression in which it appears. For example:

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

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

### 7.7.2.2. STRETCH

The STRETCH intrinsic (PERQ specific) converts a single precision whole number (INTEGER) into a double precision whole number (LONG). STRETCH does sign extension.

### 7.7.2.3. SHRINK

The SHRINK intrinsic (PERQ specific) converts a double precision whole number (LONG) into a single precision whole number (INTEGER). The value of the double precision whole number must be within the legal range of single precision whole numbers. If it is out of range a runtime error occurs.

### 7.7.2.4. FLOAT

The FLOAT intrinsic converts a whole number of any precision into a floating point number.

### 7.7.2.5. 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 storage. 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 implicit 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 code generated for the RECAST function will execute correctly.

### **7.7.3. Ordinal functions**

All of the ordinal functions defined in the standard: ORD, CHR, SUCC, and PRED are legal in PERQ Pascal for both single and double precision whole number objects.

### **7.7.4. Boolean functions**

The Boolean function, ODD, as defined in the standard is legal in PERQ Pascal for both single and double precision whole number objects.

## 7.7.5. PERQ-specific functions

### 7.7.5.1. WordSize

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

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

### 7.7.5.2. LENGTH

The predefined integer function LENGTH is provided to return the current 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 to Len.

### 7.7.5.3. Logical operations

The PERQ Pascal compiler supports a variety of logical operations for use in manipulation of the bits of any whole number. The operations supported include: and, inclusive or, not, exclusive or, shift and rotate. With the exception of rotate, all of these operations are applicable to both single and double precision whole number objects. Rotate may only be applied to single precision whole number objects. The syntax for their use resembles that of a function invocation; however the code is generated inline to the calling procedure (hence there is no function run-time call overhead associated with their use). The

syntax for the logical functions is described in the following sections.

#### 7.7.5.3.1. And

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

LAND returns the bitwise AND of Val1 and Val2.

#### 7.7.5.3.2. Inclusive Or

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

LOR returns the bitwise INCLUSIVE OR of Val1 and Val2.

#### 7.7.5.3.3. Not

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

LNOT returns the bitwise complement of Val.

#### 7.7.5.3.4. Exclusive Or

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

LXOR returns the bitwise EXCLUSIVE OR of Val1 and Val2.

#### 7.7.5.3.5. SHIFT

```
Function SHIFT(Value: whole_number_mode;  
Distance: INTEGER ): whole_number_mode;
```

SHIFT returns Value shifted Distance bits. For a single precision Value, Distance must be greater than or equal to -15 and less than or equal to +15. For a double precision Value, Distance must be greater than or equal to -31 and less than or equal to +31. 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.

### 7.7.5.3.6. ROTATE

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

ROTATE returns Value rotated Distance bits to the right.  
ROTATE accepts only a single precision whole number as a  
Value. ROTATE accepts only a positive single precision whole  
number in the range 0 through 15 for Distance. The direction of  
the ROTATE is to the right.

## **8. Expressions**

### **8.1. Mixed-Mode Expressions**

Mixed-mode expressions between single and double precision whole numbers or between whole numbers and floating point numbers are allowed. Each expression is evaluated in the largest precision mode mentioned within that expression. Thus if LONG's and REAL's are mixed together in an expression then that entire expression is evaluated using floating point arithmetic. The compiler may, therefore, produce code which is inefficient for the particular instance of a mixed-mode expression because of this production of the most general precision code. You may control the precision of code generated by the compiler thru the use of explicit type coercion intrinsics. The compiler, when given an explicit coercion (such as STRETCH and / or FLOAT), will treat its operand as an expression separate from the expression in which it is embedded. For example:

```
f := f * (l * i);
```

where f is REAL, l is LONG, and i is INTEGER; will be evaluated using entirely floating point arithmetic. Whereas given:

```
f := f * FLOAT(l * i);
```

The product of l and i will be evaluated using LONG arithmetic and then floating point arithmetic will be used to complete the evaluation.

## 8.2. 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'
```

## 9. Statements

### **9.1. General**

As in *PASCAL User Manual and Report* [JW74].

### **9.2. Simple-Statements**

As in *PASCAL User Manual and Report* [JW74], with the extensions described below.

#### **9.2.1. EXIT**

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 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.

The parameter to EXIT can be any procedure or function name. If the specified routine is not on the call stack, a run-time error occurs.

## 9.2.2. RAISE

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 (see 7.3, Exception-Declarations).
2. A handler for the exception must be defined (see 7.4, Handler-Declarations).
3. The exception must be raised.

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;
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 (see 7.3.1, ALL EXCEPTION).

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 9.2, Simple Statements).

## 9.3. Structured-Statements

As in *PASCAL User Manual and Report* [JW74], with extensions described below.

### 9.3.1. Extended case statements

Three 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.
3. If the selector expression is not in the list of case labels and no OTHERWISE label is used, then the case statement performs no-operation. This differs from [JW74], which suggests that this should be a

fault.

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
```



## 10. Input and Output

### **10.1. 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 whole numbers (and their subranges), floating point numbers and characters, PERQ Pascal also supports the ability to read booleans, packed arrays of characters, and strings.

The literals 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 current 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 current input line is reached or the maximum static length of the string is met. In either case, the dynamic length of the string will be set to the actual number of characters read.

## 10.2. 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 p 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 their subranges), REAL, BOOLEAN, PACKED ARRAY OF CHAR, or STRING. For parameters of type BOOLEAN, one of the literals TRUE, FALSE or UNDEF will be written; UNDEF is written if the internal value of the boolean 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). If e1 is smaller than the number of characters required to print the given value, more space is allocated, unless e is a packed array of char, when 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 their subranges) or REAL. If e is of type LONG or INTEGER (or their subranges) 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 (twos complement); otherwise, the value of e is printed as an 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.

### 10.3. The Procedure Page

The procedure Page is not currently implemented in PERQ Pascal.



## **11. Programs and 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. 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 part. 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 statement part. Every executable group of modules must contain exactly one instance of a program.

Exporting permits a module to provide constants, types, variables, procedures and functions to other modules. Importing permits a module to use the EXPORTS provided by 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 that contain only type and constant declarations cause no run-time overhead, making them ideal for common declarations. Such modules may not be compiled (as errors will be produced); however, they may be successfully imported.

## 11.1. IMPORTS Declarations

The IMPORTS declaration specifies the modules that are to be imported into the module being compiled. 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 12.2, Compiler Switches).

The syntax for the IMPORTS declaration is:

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

## 11.2. EXPORTS Declarations

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 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 3.1, Block; also Chapter 11.2 in the *PASCAL User Manual and Report [JW74]*). The inclusion of "FORWARD;" in the EXPORTS declaration 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 syntax for a unit of compilation in PERQ Pascal 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>



## **12. Command Line and Compiler Switches**

### **12.1. Command Line**

The PERQ Pascal compiler is invoked by typing a compile command line to the Shell. 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 Shell.

`<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>`. If the ".PAS" extension is present, it is replaced with the ".SEG" extension, while 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.

## 12.2. 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 12.1, 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 on the command line preceded by the dash (-) 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.

### 12.2.1. 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" is appended onto the end of FILENAME, and a second attempt is made to find the file.

The file inclusion mechanism may be used anywhere in a program or module, and the results are 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.

Also note that " {" and " }" may be replaced with "(\*" and "\*)" respectively when typing compiler switches in comment form.

### 12.2.2. 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; 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 "}" or "\*" (ignoring leading spaces and tabs); hence no other options may be included in this comment.

### 12.2.3. 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.

### 12.2.4. 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.

### 12.2.5. Automatic RESET/REWRITE

The PERQ Pascal compiler automatically generates a RESET(INPUT) and REWRITE(OUTPUT). This may be disabled 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.

### 12.2.6. Global INPUT/OUTPUT switch

The PERQ Pascal compiler generates code which accesses the built in TEXT file INPUT and OUTPUT as if they were declared locally to the program. This may be disabled if desired with the use of the GLOBALINOUT switch. The format for this switch is:

Default value

Use of global INPUT/OUTPUT is disabled

{\$GLOBALINOUT+} or -GLOBALINOUT  
enables global access of INPUT/OUTPUT

{\$GLOBALINOUT-} or -NOGLOBALINOUT  
disables global access of INPUT/OUTPUT

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

If the comment form of this switch is used, it must precede any  
code of the program.

#### **12.2.7. Mixed-mode permission switch**

The PERQ Pascal compiler allows the mixing of arithmetic modes  
within expressions (see 8.1, Mixed-Mode Expressions). This  
may be disabled if desired with the use of MixedModePermitted  
switch. The format for this switch is:

Default value

Mixed-mode expressions are permitted

-MixedModePermitted

Enables mixed-mode expressions

-NoMixedModePermitted

Disables mixed-mode expressions

The comment form of this switch is illegal and may not be used.

#### **12.2.8. 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.

#### **12.2.9. SCROUNGE (symbol table for debugger) switch**

The SCROUNGE switch causes the compiler to write symbol table information that can be used by the symbolic Pascal debugger. Additional files with extensions .QMAP and .SYM will be generated. The format for this switch is:

Default value

Symbol table files are produced

-SCROUNGE

Enables production of symbol table files

-NOSCROUNGE

Disables production of symbol table files

The comment form of this switch may not be used.

#### **12.2.10. 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 "\*)") or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored.

### 12.2.11. 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.

### 12.2.12. 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.

### 12.2.13. 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 precedes the \$ELSEC; the \$ELSEC signals the end of the \$IFC region. A \$ENDC is

then used to terminate the \$ELSE 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 };
```

and:

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

#### 12.2.14. 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; 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 compilation errors. This switch allows compilations to be left unattended.

#### **12.2.15. HELP switch**

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



## **13. Quirks and Oddities**

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

1. The last line of any PROGRAM or MODULE must end with a carriage return, or an "Unexpected End of Input" error occurs.
2. Although unique abbreviations are accepted for switches, the following abbreviations cause compilation errors:

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

3. Procedures and functions which are forward declared (including EXPORT declarations) and contain procedure parameters, may not have their parameter lists redeclared at the site of the procedure body.
4. 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, a run-time error occurs.

5. The filename specification given in IMPORTS Declarations must start with an alphabetic character.
6. 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.
7. 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.
8. The RECAST intrinsic does not work with two-word scalars (for example, LONG) and arrays.
9. The code size of a compilation unit cannot exceed

32k.



## **14. References**

---

[BSI82] *Specification for Computer Programming Language Pascal*, British Standards Institute, BS 6192/ISO 7185, 1982

[IEEE83] *American National Standard Pascal Computer Programming Language*, Institute of Electrical and Electronics Engineers, ANSI/IEEE770X397-1983, 1983

[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.



# **PASCAL LIBRARY**

**November 30, 1984**

Copyright © 1984 PERQ Systems Corporation  
2600 Liberty Avenue  
P. O. Box 2600  
Pittsburgh, PA 15230  
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

This document is adapted from the paper by Mary R. Thompson, Sharon Schanzer, and Dean Zarras, *Pascal Library*. Carnegie-Mellon University, Pittsburgh, PA, 1984.

This document is not to be reproduced in any form or transmitted in whole or in part without the prior written authorization of PERQ Systems Corporation.

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.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

<u>Table of Contents</u>	<u>Page</u>
<b><u>1. Introduction</u></b>	<b>PL-1</b>
<b><u>2. AccCall</u></b>	<b>PL-3</b>
<b><u>3. AccentType</u></b>	<b>PL-7</b>
<b><u>4. AccentUser</u></b>	<b>PL-21</b>
<b><u>5. ALoad</u></b>	<b>PL-23</b>
<b>5.1. Procedure ARunLoad</b>	<b>PL-23</b>
<b>5.2. Procedure ShowRun</b>	<b>PL-23</b>
<b>5.3. Function LinkTypeStr</b>	<b>PL-24</b>
<b>5.4. Exception ALoadError</b>	<b>PL-24</b>
<b>5.5. Function DateString</b>	<b>PL-24</b>
<b><u>6. AuthDefs / AuthUser</u></b>	<b>PL-25</b>
<b>6.1. AuthDefs - Exported Types and Constants</b>	<b>PL-25</b>
<b>6.2. AuthUser</b>	<b>PL-27</b>
<b>6.2.1. Function LoginUser</b>	<b>PL-27</b>

<b>6.2.2. Function LogoutUser</b>	<b>PL-27</b>
<b>6.2.3. Function ConfirmUser</b>	<b>PL-28</b>
<b>6.2.4. Function CheckUser</b>	<b>PL-28</b>
<b>6.2.5. Function ChangeUserParams</b>	<b>PL-29</b>
<b>6.3. Function GetUserName</b>	<b>PL-30</b>
<b>6.3.1. Function ListLoggedInUsers</b>	<b>PL-30</b>
<b>6.3.2. Procedure InitAuth</b>	<b>PL-31</b>
<b><u>7. BootInfo</u></b>	<b><u>PL-33</u></b>
<b><u>8. CFileDefs / CLoad</u></b>	<b><u>PL-37</u></b>
<b>8.1. CFileDefs</b>	<b>PL-37</b>
<b>8.2. CLoad</b>	<b>PL-39</b>
<b>8.2.1. General</b>	<b>PL-39</b>
<b>8.2.2. Function CLoadProcess</b>	<b>PL-39</b>
<b><u>9. Clock</u></b>	<b><u>PL-41</u></b>
<b>9.1. General</b>	<b>PL-41</b>
<b>9.2. Function IOGetTime</b>	<b>PL-41</b>
<b><u>10. CommandDefs, CommandParse, and ExtraCmdParse</u></b>	<b><u>PL-43</u></b>
<b>10.1. Introduction</b>	<b>PL-43</b>
<b>10.2. Usage of the CommandParse Module</b>	<b>PL-45</b>
<b>10.2.1. Constants</b>	<b>PL-45</b>
<b>10.2.2. Types</b>	<b>PL-46</b>
<b>10.2.3. Command file routines</b>	<b>PL-46</b>
<b>10.2.4. Parsing routines</b>	<b>PL-47</b>

<b>10.2.5. Word identification routines</b>	<b>PL-48</b>
<b>10.2.6. Miscellaneous pool manipulators</b>	<b>PL-49</b>
<b>10.3. Usage of the ExtraCmdParse Module</b>	<b>PL-50</b>
<b>10.3.1. Routines GetCmd and GetShellCmd</b>	<b>PL-50</b>
<b>10.3.2. Routine GetParsedUserInput</b>	<b>PL-52</b>
<b>10.3.3. Routine GetConfirm</b>	<b>PL-52</b>
<b>10.3.4. Routine GetChPool</b>	<b>PL-53</b>
<b>10.4. CommandDefs</b>	<b>PL-54</b>
<b>10.5. Function Null_CommandBlock</b>	<b>PL-55</b>
<b>10.6. CommandParse</b>	<b>PL-56</b>
<b>10.6.1. Procedure InitCmdFile</b>	<b>PL-56</b>
<b>10.6.2. Function OpenCmdFile</b>	<b>PL-56</b>
<b>10.6.3. Procedure ExitCmdFile</b>	<b>PL-57</b>
<b>10.6.4. Procedure ExitAllCmdFiles</b>	<b>PL-57</b>
<b>10.6.5. Procedure DstryCmdFiles</b>	<b>PL-58</b>
<b>10.6.6. Procedure InitCommandParse</b>	<b>PL-58</b>
<b>10.6.7. Procedure DestroyCommandParse</b>	<b>PL-58</b>
<b>10.6.8. Function AllocCommandNode</b>	<b>PL-59</b>
<b>10.6.9. Procedure DestroyCommandList</b>	<b>PL-59</b>
<b>10.6.10. Procedure AlwaysEof</b>	<b>PL-59</b>
<b>10.6.11. Function ExerciseParseEngine</b>	<b>PL-60</b>
<b>10.6.12. Function ParseChPool</b>	<b>PL-61</b>
<b>10.6.13. Function ParseCommand</b>	<b>PL-61</b>
<b>10.6.14. Function WordifyPool</b>	<b>PL-62</b>
<b>10.6.15. Procedure InitWordSearchTable</b>	<b>PL-63</b>
<b>10.6.16. Procedure AddSearchWord</b>	<b>PL-63</b>
<b>10.6.17. Procedure DeleteSearchWord</b>	<b>PL-64</b>
<b>10.6.18. Procedure DestroySearchTable</b>	<b>PL-64</b>
<b>10.6.19. Function UniqueWordIndex</b>	<b>PL-64</b>
<b>10.6.20. Procedure ConvertStringToPool</b>	<b>PL-65</b>
<b>10.6.21. Function ConvertPoolToString</b>	<b>PL-65</b>

<b>10.6.22. Procedure DestroyChPool</b>	<b>PL-66</b>
<b>10.6.23. Function GetIthWordPtr</b>	<b>PL-66</b>
<b>10.7. ExtraCmdParse</b>	<b>PL-68</b>
<b>10.7.1. Function GetCmd</b>	<b>PL-68</b>
<b>10.7.2. Function GetShellCmd</b>	<b>PL-69</b>
<b>10.7.3. Function GetParsedUserInput</b>	<b>PL-70</b>
<b>10.7.4. Function GetConfirm</b>	<b>PL-71</b>
<b>10.7.5. Procedure GetCharacterPool</b>	<b>PL-72</b>
<b><u>11. Configuration</u></b>	<b>PL-73</b>
<b>11.1. Function CF_IOBoard</b>	<b>PL-73</b>
<b>11.2. Function Cf_Monitor</b>	<b>PL-73</b>
<b>11.3. Function Cf_OldZ80</b>	<b>PL-74</b>
<b>11.4. Function Cf_Network</b>	<b>PL-74</b>
<b><u>12. ControlStore</u></b>	<b>PL-75</b>
<b>12.1. Procedure LoadControlStore</b>	<b>PL-76</b>
<b>12.2. LoadMicroInstruction</b>	<b>PL-76</b>
<b>12.3. Procedure JumpControlStore</b>	<b>PL-77</b>
<b><u>13. Coreload</u></b>	<b>PL-79</b>
<b>13.1. Function CoreRunLoad</b>	<b>PL-79</b>
<b><u>14. DiskUtils</u></b>	<b>PL-81</b>
<b>14.1. Function AddrToBlkNum</b>	<b>PL-85</b>
<b>14.2. Function BlkNumToAddr</b>	<b>PL-86</b>
<b>14.3. Function MapAddr</b>	<b>PL-86</b>

<b>14.4. Function UnMapAddr</b>	<b>PL-86</b>
<b>14.5. Procedure InitDiskUtils</b>	<b>PL-87</b>
<b>14.6. Procedure FinishDiskUtils</b>	<b>PL-87</b>
<b>14.7. Procedure GetDiskInfo</b>	<b>PL-87</b>
<b><u>15. Dynamic</u></b>	<b><u>PL-89</u></b>
<b>15.1. Procedure InitDynamic</b>	<b>PL-89</b>
<b>15.2. Function CreateHeap</b>	<b>PL-89</b>
<b>15.3. Procedure ResetHeap</b>	<b>PL-89</b>
<b>15.4. Procedure DestroyHeap</b>	<b>PL-90</b>
<b>15.5. Procedure DisposeP</b>	<b>PL-90</b>
<b>15.6. Procedure NewP</b>	<b>PL-90</b>
<b>15.7. Procedure CheckHeap</b>	<b>PL-91</b>
<b><u>16. EnvMgrDefs / EnvMgrUser</u></b>	<b><u>PL-93</u></b>
<b><u>17. Except</u></b>	<b><u>PL-95</u></b>
<b>17.1. Procedure InitExceptions</b>	<b>PL-96</b>
<b>17.2. Procedure RaiseP</b>	<b>PL-96</b>
<b><u>18. IFileDefs</u></b>	<b><u>PL-99</u></b>
<b><u>19. IODefs / IOUser</u></b>	<b><u>PL-101</u></b>
<b><u>20. IPCRecordIO</u></b>	<b><u>PL-103</u></b>

<b>20.1. Function SendRecord</b>	<b>PL-103</b>
<b>20.2. Function RecRecord</b>	<b>PL-104</b>
<b><u>21. KeyTran / KeyTranDefs</u></b>	<b><u>PL-105</u></b>
<b><u>22. ModGetEvent</u></b>	<b><u>PL-107</u></b>
<b>22.1. Procedure GetEventPort</b>	<b>PL-107</b>
<b>22.2. Function ExtractEvent</b>	<b>PL-107</b>
<b><u>23. MsgNUser</u></b>	<b><u>PL-109</u></b>
<b><u>24. NameErrors</u></b>	<b><u>PL-111</u></b>
<b><u>25. Net10MBDefs / Net10MBUser</u></b>	<b><u>PL-113</u></b>
<b><u>26. NoEchoInput</u></b>	<b><u>PL-115</u></b>
<b><u>27. NSTypes</u></b>	<b><u>PL-117</u></b>
<b><u>28. OldTimeStamp</u></b>	<b><u>PL-119</u></b>
<b>28.1. Exported Types</b>	<b>PL-119</b>
<b>28.2. Function OldCurrentTime</b>	<b>PL-119</b>
<b>28.3. Function NewToOldTime</b>	<b>PL-119</b>

<b>28.4. Function OldToNewTime</b>	<b>PL-120</b>
<hr/>	
<b>29. PascalInit</b>	<b>PL-121</b>
<b>29.1. Procedure InitPascal</b>	<b>PL-122</b>
<b>29.2. Procedure InitProcess</b>	<b>PL-122</b>
<b>29.3. Function DisablePrivs</b>	<b>PL-122</b>
<b>29.4. Function EnablePrivs</b>	<b>PL-123</b>
<hr/>	
<b>30. PasLong</b>	<b>PL-125</b>
<b>30.1. Procedure ReadD</b>	<b>PL-125</b>
<b>30.2. Procedure WriteD</b>	<b>PL-126</b>
<hr/>	
<b>31. PasReal</b>	<b>PL-127</b>
<b>31.1. Procedure ReadR</b>	<b>PL-127</b>
<b>31.2. Procedure WriteR</b>	<b>PL-128</b>
<hr/>	
<b>32. PathName</b>	<b>PL-129</b>
<hr/>	
<b>33. PMatch</b>	<b>PL-131</b>
<b>33.1. Procedure PattDebug</b>	<b>PL-131</b>
<b>33.2. Function IsPattern</b>	<b>PL-131</b>
<b>33.3. Function PattMatch</b>	<b>PL-132</b>
<b>33.4. Function PattMap</b>	<b>PL-132</b>
<b>33.5. Function NextCh</b>	<b>PL-133</b>
<b>33.6. Function UpCh</b>	<b>PL-133</b>
<b>33.7. Function PattCheck</b>	<b>PL-134</b>

**34. ProcMgrDefs / ProcMgrUser**

**PL-135**

**35. QMapDefs**

**PL-137**

**36. RealFunctions**

**PL-139**

- |                                |               |
|--------------------------------|---------------|
| <b>36.1. Function Sqrt</b>     | <b>PL-139</b> |
| <b>36.2. Function Ln</b>       | <b>PL-140</b> |
| <b>36.3. Function Log10</b>    | <b>PL-140</b> |
| <b>36.4. Function Exp</b>      | <b>PL-140</b> |
| <b>36.5. Function Power</b>    | <b>PL-140</b> |
| <b>36.6. Function Powerl</b>   | <b>PL-141</b> |
| <b>36.7. Function Sin</b>      | <b>PL-141</b> |
| <b>36.8. Function Cos</b>      | <b>PL-141</b> |
| <b>36.9. Function Tan</b>      | <b>PL-142</b> |
| <b>36.10. Function CoTan</b>   | <b>PL-142</b> |
| <b>36.11. Function ArcSin</b>  | <b>PL-142</b> |
| <b>36.12. Function ArcCos</b>  | <b>PL-143</b> |
| <b>36.13. Function Arctan</b>  | <b>PL-143</b> |
| <b>36.14. Function ArcTan2</b> | <b>PL-143</b> |
| <b>36.15. Function SinH</b>    | <b>PL-144</b> |
| <b>36.16. Function CosH</b>    | <b>PL-144</b> |
| <b>36.17. Function TanH</b>    | <b>PL-144</b> |

**37. RunDefs**

**PL-145**

**38. SaltError**

**PL-149**

<b>38.1. Procedure GRWriteStdError</b>	<b>PL-149</b>
<b>38.2. Procedure GRStdError</b>	<b>PL-150</b>
<b>38.3. Procedure GRWriteErrorMsg</b>	<b>PL-150</b>
<b>38.4. Procedure GSErrorMsg</b>	<b>PL-151</b>
<b>38.5. Procedure ErrorMsgPMBroadcast</b>	<b>PL-151</b>
<b>38.6. Function GRStdErr</b>	<b>PL-152</b>
<b><u>39. Sapphire-Related Modules</u></b>	<b><u>PL-153</u></b>
<b><u>40. SeqDefs</u></b>	<b><u>PL-155</u></b>
<b><u>41. Sesame-Related Modules</u></b>	<b><u>PL-157</u></b>
<b><u>42. Spawn</u></b>	<b><u>PL-159</u></b>
<b>42.1. Function Exec</b>	<b>PL-159</b>
<b>42.2. Function Split</b>	<b>PL-160</b>
<b>42.3. Function Spawn</b>	<b>PL-160</b>
<b><u>43. SpawnInitFlags</u></b>	<b><u>PL-165</u></b>
<b><u>44. Spice String</u></b>	<b><u>PL-167</u></b>
<b>44.1. Procedure Adjust</b>	<b>PL-167</b>
<b>44.2. Procedure AppendChar</b>	<b>PL-167</b>
<b>44.3. Procedure AppendString</b>	<b>PL-168</b>
<b>44.4. Function Cat3</b>	<b>PL-168</b>

<b>44.5. Function Cat4</b>	<b>PL-168</b>
<b>44.6. Function Cat5</b>	<b>PL-169</b>
<b>44.7. Function Cat6</b>	<b>PL-169</b>
<b>44.8. Function Concat</b>	<b>PL-170</b>
<b>44.9. Procedure ConvUpper</b>	<b>PL-170</b>
<b>44.10. Function CVD</b>	<b>PL-170</b>
<b>44.11. Function CVH</b>	<b>PL-171</b>
<b>44.12. Function CVHS</b>	<b>PL-171</b>
<b>44.13. Function CVHSS</b>	<b>PL-172</b>
<b>44.14. Function CvInt</b>	<b>PL-172</b>
<b>44.15. Function CvL</b>	<b>PL-173</b>
<b>44.16. Function CVN</b>	<b>PL-173</b>
<b>44.17. Function CVLS</b>	<b>PL-174</b>
<b>44.18. Function CVO</b>	<b>PL-174</b>
<b>44.19. Function CVOS</b>	<b>PL-175</b>
<b>44.20. Function CVOSS</b>	<b>PL-175</b>
<b>44.21. Function CVS</b>	<b>PL-176</b>
<b>44.22. Function CVSS</b>	<b>PL-176</b>
<b>44.23. Function CvUp</b>	<b>PL-177</b>
<b>44.24. Procedure DeleteChars</b>	<b>PL-177</b>
<b>44.25. Function GetBreak</b>	<b>PL-178</b>
<b>44.26. Function Initial</b>	<b>PL-178</b>
<b>44.27. Procedure InsertChars</b>	<b>PL-178</b>
<b>44.28. Function Lop</b>	<b>PL-179</b>
<b>44.29. Function Pad</b>	<b>PL-179</b>
<b>44.30. Function PosC</b>	<b>PL-180</b>
<b>44.31. Function PosString</b>	<b>PL-180</b>
<b>44.32. Procedure ReplaceChars</b>	<b>PL-181</b>
<b>44.33. Function RevPosC</b>	<b>PL-181</b>
<b>44.34. Function RevPosString</b>	<b>PL-182</b>
<b>44.35. Function Scan</b>	<b>PL-182</b>

<b>44.36. Procedure SetBreak</b>	<b>PL-183</b>
<b>44.37. Function ShowBreak</b>	<b>PL-184</b>
<b>44.38. Function Squeeze</b>	<b>PL-184</b>
<b>44.39. Function Str</b>	<b>PL-185</b>
<b>44.40. Function Strip</b>	<b>PL-185</b>
<b>44.41. Function SubStrFor</b>	<b>PL-186</b>
<b>44.42. Function SubStrTo</b>	<b>PL-186</b>
<b>44.43. Function Trim</b>	<b>PL-187</b>
<b>44.44. Function ULInitial</b>	<b>PL-187</b>
<b>44.45. Function ULPosString</b>	<b>PL-187</b>
<b>44.46. Function UpChar</b>	<b>PL-188</b>
<b>44.47. Function UpEQU</b>	<b>PL-188</b>
<b>45. Stream</b>	<b>PL-189</b>
<b>45.1. Procedure StreamInit</b>	<b>PL-189</b>
<b>45.2. Procedure StreamClose</b>	<b>PL-190</b>
<b>45.3. Procedure StreamOpen</b>	<b>PL-190</b>
<b>45.4. Procedure GetB</b>	<b>PL-191</b>
<b>45.5. Procedure GetC</b>	<b>PL-191</b>
<b>45.6. Procedure PutB</b>	<b>PL-192</b>
<b>45.7. Procedure PutC</b>	<b>PL-192</b>
<b>45.8. Procedure PReadIn</b>	<b>PL-192</b>
<b>45.9. Procedure PWriteln</b>	<b>PL-193</b>
<b>45.10. Procedure KBFlushBoardOutput</b>	<b>PL-193</b>
<b>45.11. Procedure StreamKeyBoardReset</b>	<b>PL-193</b>
<b>45.12. Procedure InitStream</b>	<b>PL-193</b>
<b>45.13. Function FullLn</b>	<b>PL-194</b>
<b>45.14. Function StreamName</b>	<b>PL-194</b>
<b>45.15. Procedure WriteNChars</b>	<b>PL-194</b>
<b>45.16. Procedure WriteChars</b>	<b>PL-195</b>

<b>45.17. Function IsStreamDevice</b>	<b>PL-195</b>
<b>45.18. Procedure StreamFlushOutput</b>	<b>PL-195</b>
<b><u>46. SymDefs</u></b>	<b><u>PL-197</u></b>
<b><u>47. TimeDefs / TimeUser</u></b>	<b><u>PL-199</u></b>
<b><u>48. TSDefs / TSUser</u></b>	<b><u>PL-201</u></b>
<b><u>49. ViewKern</u></b>	<b><u>PL-203</u></b>
<b><u>50. ViewPtUser</u></b>	<b><u>PL-205</u></b>
<b><u>51. WindowUtils</u></b>	<b><u>PL-207</u></b>
<b>51.1. Procedure ShowPathAndTitle</b>	<b>PL-207</b>
<b>51.2. Procedure ShowWindowErrorFlag</b>	<b>PL-207</b>
<b>51.3. Procedure RemoveWindowErrorFlag</b>	<b>PL-207</b>
<b>51.4. Procedure ShowWindowRequestFlag</b>	<b>PL-207</b>
<b>51.5. Procedure RemoveWindowRequestFlag</b>	<b>PL-208</b>
<b>51.6. Procedure ShowWindowAttentionFlag</b>	<b>PL-208</b>
<b>51.7. Procedure RemoveWindowAttentionFlag</b>	<b>PL-208</b>
<b>51.8. Procedure StreamProgress</b>	<b>PL-208</b>
<b>51.9. Procedure ComputeProgress</b>	<b>PL-208</b>
<b>51.10. Procedure RandomProgress</b>	<b>PL-209</b>
<b>51.11. Procedure QuitProgress</b>	<b>PL-209</b>

<b>51.12. Procedure MultiLevelProgress</b>	<b>PL-209</b>
<b>51.13. Procedure MultiStreamProgress</b>	<b>PL-209</b>
<b>51.14. Procedure QuitMultiProgress</b>	<b>PL-210</b>



## **1. Introduction**

---

The Pascal library (stored in the LibPascal directory) contains procedures and functions that may be incorporated into other programs.

This document discusses the purpose of each file contained in the library and lists the procedures and functions of interest to application programmers. The files are listed alphabetically, except for a few instances in which related files are discussed in the same chapter. Unless stated otherwise, each file contains one module of the same name as the file.

The routines that provide remote procedure call interfaces to servers are discussed in more detail in the documents for the servers in the *Accent Programming Manual*.

For a cross-listing of modules and files and a listing of all definitions in the Pascal library, see the document "Module Index" in this manual.



## 2. AccCall

The AccCall module contains the exported procedure calls for the Accent primitives that are implemented by system calls rather than by messages. These calls are explained in the document "The Kernel Interface" in the *Accent Programming Manual*.

```
{ -----> }    Exports    { -----<
imports AccentType from AccentType;

const
  MAXLOGMESS = 20;
  MAXMSGDATA = 20;
  LOGMSGS = false;

type
  LMsg =
    record
      H:        Msg;
      D:        array [1..MAXMSGDATA] of integer;
    end;
  pLMsg = ^LMsg;

  MsgLog =
    record
      Init:      integer;
      MsgsSent:  long;
      MsgsRec:   long;
      NxtMsg:    integer;
      LMsgs:    array [0..MAXLOGMESS] of
        record
          Sent:     boolean;
          InProg:   boolean;
          GR:       GeneralReturn;
          M:        LMsg;
        end;
    end;
  pMsgLog = ^MsgLog;

Function Send(
  var xxmsg      : Msg;
  MaxWait       : long;
  Option        : SendOption
): GeneralReturn;

Function Receive(
  var xxmsg      : Msg;
```

```
        MaxWait      : long;
        PortOpt     : PortOption;
        Option      : ReceiveOption
    ): GeneralReturn;

Function SetPortsWaiting(
    var ports      : PortBitArray
): GeneralReturn;

Function PortsWithMessages(
    MsgType      : long;
    var ports      : PortBitArray
): GeneralReturn;

Function MoveWords(
    SrcAddr      : VirtualAddress;
    var DstAddr    : VirtualAddress;
    NumWords     : long;
    Delete       : boolean;
    Create       : boolean;
    Mask         : long;
    DontShare    : boolean
): GeneralReturn;

Function SoftEnable(
    NormOrEmerg  : boolean;
    EnOrDis      : boolean
): GeneralReturn;

Function GetIOSleepID(
    var SleepID    : long
): GeneralReturn;

Function EReceive(
    var xxmsg: Msg;
    MaxWait: long;
    PortOpt : PortOption;
    Option: ReceiveOption
): GeneralReturn;

Function RectRasterOp(
    DstRectangle : Port;
    Action       : integer;
    DstX         : integer;
    DstY         : integer;
    Width        : integer;
    Height       : integer;
    SrcRectangle : Port;
    SrcX         : integer;
    SrcY         : integer
): GeneralReturn;

Function RectDrawLine(
    DstRectangle : Port;
    Kind         : integer;

```

```
X1,Y1,X2,Y2 : integer
) : GeneralReturn;

Function RectPutString(
    DstRectangle : Port;
    FontRectangle : Port;
    Action       : integer;
    var FirstX   : integer;
    var FirstY   : integer;
    StrPtr       : Pointer;
    FirstChar    : integer;
    var MaxChar  : integer
) : GeneralReturn;

Function RectColor(
    Rectangle   : Port;
    Action      : integer;
    X          : integer;
    Y          : integer;
    Width      : integer;
    Height     : integer
) : GeneralReturn;

Function RectScroll(
    Rectangle   : Port;
    X          : integer;
    Y          : integer;
    Width      : integer;
    Height     : integer;
    Xamt       : integer;
    Yamt       : integer
) : GeneralReturn;

Function LockPorts(
    LockThem   : boolean;
    Ports       : ptrLPortArray;
    PortsCount  : long
) : GeneralReturn;

Function MessagesWaiting(
    MsgType    : long;
{inout} var Ports   : ptrLPortArray;
{inout} var PortsCount : long
) : GeneralReturn;
```

PERQ Systems Corporation  
Accent Operating System

Pascal Library  
AccCall

### **3. AccentType**

---

Module AccentType in AccentType.Pas contains the types used by the Accent operating system and user interfaces to the system. The primary item of interest in this module is the list of Accent error code numbers on page PL-10.

```
{***** Exports *****}

const
{}
{  Constant:
{    PAGEBYTESIZE, PAGEWORDSIZE, PAGEBITS, DISKBUFSIZE
{
{  Purpose:
{    Constants which define the size of an Accent physical and
{      disk page.
}

PAGEBYTESIZE = 512;
{   Number of bytes in physical page.   }
PAGEWORDSIZE = PAGEBYTESIZE div 2;
{   Number of 16 bit words per page.   }
DISKBUFSIZE  = PAGEBYTESIZE div 2;
{   Number of 16 bit words per disk page.   }
PAGEBITS     = 8; {   Number of bits needed to represent
{      a page.   }

type
{}
{  Type:
{    Bit size types.
{
{  Purpose:
{    Defines all bit types used in the system.
{
}

Bit1        = 0..1;
Bit2        = 0..3;
Bit3        = 0..7;
Bit4        = 0..15;
Bit5        = 0..31;
Bit6        = 0..63;
Bit7        = 0..127;
Bit8        = 0..255;
Bit9        = 0..511;
Bit10       = 0..1023;
```

```
Bit11      = 0..2047;
Bit12      = 0..4095;
Bit13      = 0..8191;
Bit14      = 0..16383;
Bit15      = 0..32767;
Bit16      = integer;

pBit32     = ^Bit32;
Bit32      = packed record
    case integer of
        1: ( DblWord : array [0..1] of integer);
        2: ( Addr   : ptrDiskBuffer);
        3: ( Bit32Ptr: pBit32);
        4: ( AnyPtr : pointer);
        5: ( Byte   : packed array [0..3]
              of Bit8);
        6: ( PageOffset : Bit8;
              LswPage : Bit8;
              MswPage : Bit16
            );
        7: ( Lng     : Long);
        8: ( Blk     : integer;
              Index   : Bit12;
              Imag   : Bit4
            );
        13: ( Field4 : Bit8;
              Field3 : Bit8;
              Field2 : Bit8;
              Field1 : Bit7;
              Field0 : Bit1);
        14: ( Word0  : Bit16;
              Word1  : Bit16);
        15: ( Byte0  : Bit8;
              Byte1  : Bit8;
              Byte2  : Bit8;
              Byte3  : Bit8 );
    end;

Bit64      = record
    lsw     : long;
    msw     : long;
end;

SegID      = long;

SpiceSegKind = (Temporary,Permanent,Bad,SegPhysical,
                 Imaginary,Shadow);
VirtualAddress = long;
PhysicalAddress = long;
MicroSeconds   = long;

DiskAddr     = packed record
    case integer of
        1: (lng     : long);
        2: (byte   : packed array [0..3] of Bit8)
    end;
```

```
DiskInterface = (EIO, CIO, FlopDrives, MultiBus, Enet);

InterfaceInfo = Packed Record Case DiskInterface Of
                  EIO, CIO, FlopDrives : (Unit: Integer);
                  ENet: (EAddr: Packed Array[0..2] of
                           integer);
                  End;

{}

{
  Enumerated type:
  {
    TrapCodes
  }
  Purpose:
  {
    System trap codes.
  }
}
TrapCodes      =
(
  TrapInit,
  TrapReadFault,
  TrapWriteFault,
  TrapSend,
  TrapReceive,
  TrapSetPortsWaiting,
  TrapPortsWithMessages,
  TrapDebugWrite,
  TrapException,
  TrapNothing,
  TrapRectDrawLine,
  TrapRectRasterOp,
  TrapCharRead,
  TrapFull,
  TrapFlush,
  TrapMoveWords,
  TrapRectPutString,
  TrapError,
  TrapClockEnable,
  TrapGPRead,
  TrapGPWrite,
  TrapSoftEnable,
  TrapGetIOSleepID,
  TrapRectColor,
  TrapRectScroll,
  TrapLockPorts,
  TrapMessagesWaiting
);
{}

{
  GeneralReturn
}
{
  Purpose:
  {
    Values returned from system calls and system messages.
  }
}
```

```

const AccErr           = 100;

Dummy                = AccErr+0;
Success              = AccErr+1;
TimeOut              = AccErr+2;
PortFull             = AccErr+3;
WillReply             = AccErr+4;
TooManyReplies        = AccErr+5;
MemFault             = AccErr+6;
NotAPort              = AccErr+7;
BadRights             = AccErr+8;
NoMorePorts          = AccErr+9;
IllegalBacklog        = AccErr+10;
NetFail               = AccErr+11;
Intr                  = AccErr+12;
Other                 = AccErr+13;
NotPortReceiver       = AccErr+14;
UnrecognizedMsgType   = AccErr+15;
NotEnoughRoom         = AccErr+16;
NotAnIPCCall          = AccErr+17;
BadMsgType            = AccErr+18;
BadIPCName            = AccErr+19;
MsgTooBig              = AccErr+20;
NotYourChild          = AccErr+21;
BadMsg                = AccErr+22;
OutOfIPCSpace         = AccErr+23;
Failure               = AccErr+24;
MapFull               = AccErr+25;
WriteFault             = AccErr+26;
BadKernelMsg           = AccErr+27;
NotCurrentProcess      = AccErr+28;
CantFork               = AccErr+29;
BadPriority             = AccErr+30;
BadTrap                = AccErr+31;
DiskErr                = AccErr+32;
BadSegType              = AccErr+33;
BadSegment              = AccErr+34;
IsParent               = AccErr+35;
IsChild                = AccErr+36;
NoAvailablePages        = AccErr+37;
FiveDeep               = AccErr+38;
BadVPTable              = AccErr+39;
VPExclusionFailure     = AccErr+40;
MicroFailure            = AccErr+41;
EStackTooDeep           = AccErr+42;
MsgInterrupt            = AccErr+43;
UncaughtException        = AccErr+44;
BreakPointTrap           = AccErr+45;
ASTInconsistency         = AccErr+46;
InactiveSegment          = AccErr+47;
SegmentAlreadyExists     = AccErr+48;
OutOfImagSegments        = AccErr+49;
NotASystemAddress         = AccErr+50;
NotAUserAddress           = AccErr+51;
BadCreateMask             = AccErr+52;
BadRectangle             = AccErr+53;

```

```
        OutOfRectangleBounds      = AccErr+54;
        IllegalScanWidth         = AccErr+55;
        CoveredRectangle          = AccErr+56;
        BusyRectangle              = AccErr+57;
        NotAFont                  = AccErr+58;
        PartitionFull             = AccErr+59;

type GeneralReturn = integer;

{}

{   General error codes to be used
{   by all modules that pass messages.
{}

const
    BADMSGID                 = 1;
    WRONGARGS                 = 2;
    BADREPLY                  = 3;
    NOREPLY                   = 4;
    UNSPECEXCEPTION           = 5; { Message is an exception
                                      on behalf of a server }

{}

{   Constant:
{   MAXPORTS, DEFAULTBACKLOG, MAXBACKLOG
{}

{   Purpose:
{   MAXPORTS                Former maximum number of ports
                                per process.
{   DEFAULTBACKLOG            Left in for type ptrPortArray.
{   MAXBACKLOG                Constant to get kernel's port
                                backlog default.
{   MAXBACKLOG                Maximum allowable port backlog.

{}

const
    MAXPORTS                 = 256;
    DEFAULTBACKLOG            = 0;
    MAXBACKLOG                = 63;

type
    BackLogValue              = 0..MAXBACKLOG;

{}

{   Constants:
{   NORMALMSG, EMERGENCYMSG
{   Purpose:
{   Possible values of MsgType field in a message header.
{   {}

const
    NORMALMSG                 = 0;
    EMERGENCYMSG               = 1;
    NUMMSGTYPES                = 2;
```

```
{}
{   Constants:
{     WAIT,DONTWAIT,REPLY
{
{   Purpose:
{     Possible sending options.
{
{   }
const
  WAIT          = 0;
  DONTWAIT      = 1;
  REPLY         = 2;

type
  SendOption      = WAIT..REPLY;

{   }
{   Constants:
{     PREVIEW,RECEIVEIT,RECEIVEWAIT
{
{   Purpose:
{     Possible sending options.
{
{   }
const
  PREVIEW        = 0;
  RECEIVEIT      = 1;
  RECEIVEWAIT    = 2;

type
  ReceiveOption    = PREVIEW..RECEIVEWAIT;

{   }
{   Constants:
{     DEFAULTPTS,ALLPTS,LOCALPTS
{   Purpose:
{     Possible port options on receive
{
{   }
const
  DEFAULTPTS      = 0;
  ALLPTS          = 1;
  LOCALPT         = 2;

type
  PortOption       = DEFAULTPTS..LOCALPT;

{   }
{   Constants:
{     NULLPORT,KERNELPORT,DATAPORT,FIRSTNONRESERVEDPORT,
{     ALLPORTS
{
{   Purpose:
{     Distinguished local port numbers (or in the case
{     of ALLPORTS a number which implies all ports).
{
{   }
```

```
const
  NULLPORT          = 0;
  KERNELPORT        = 1;
  DATAPORT          = 2;
  FIRSTNONRESERVEDPORT = 3;
  ALLPORTS          = -1;

  {}

  { Constants:
  {   EXPLICITDEALLOC, PROCESSDEATH, NETWORKTROUBLE
  {
  { Purpose:
  {   Reason for a port to be deallocated and/or destroyed.
  }

  const
    EXPLICITDEALLOC      = 0;
    PROCESSDEATH         = 1;
    NETWORKTROUBLE       = 2;

  type
    PortDeath           = EXPLICITDEALLOC..NETWORKTROUBLE;

    { Constants:
    {   READONLY, READWRITE
    {
    { Purpose:
    {   Protection types for virtual memory.
    {
    }

    const
      READONLY            = 0;
      READWRITE           = 1;

  type
    MemProtection        = READONLY..READWRITE;

    {}

    { Constants:
    {   LINEARSTRUCTURE, TYPEINTEGER, TYPEPTOWERSHIP,
    {   TYPEPTRECEIVERIGHTS, TYPEPTALLRIGHTS,
    {   TYPEPT, TYPEUNSTRUCTURED
    {

    { Purpose:
    {   Types of message descriptors.
    {

  }

  const
    TYPEUNSTRUCTURED    = 0;
    TYPEBIT              = 0;
    TYPEBOOLEAN          = 0;
    { TYPESHORT           = 1; }
    { TYPEBIT16            = 1; }
    TYPEINT16             = 1;
    { TYPEINTEGER          = 2; }
```

```
TYPEINT32          = 2;
{ TYPEBIT32        = 2; }
{ TYPEULONG         = 2; }
    TYPEPTOWNSHIP   = 3;
    TYPEPTE RECEIVE = 4;
    TYPEPTALL       = 5;
    TYPEPT           = 6;
    TYPECHAR         = 8;
    TYPEINT8         = 9;
{ TYPEBIT8          = 9; }
    TYPEBYTE         = 9;
    TYPEREAL         = 10;
    TYPEPSTAT        = 11;
    TYPESTRING        = 12;
    TYPESEGID        = 13;
    TYPEPAGE          = 14;

}

{ Constants:
{     PORTDELETED, MSGACCEPTED, OWNERSHIPRIGHTS,
{     RECEIVERIGHTS, INTERPOSEDONE, KERNELMSGERROR
{     GENERALKERNELREPLY
{
{ Purpose:
{     Kernel generated messages ids.
}

const
    M_PORTDELETED      = #100 + 1;
    M_MSGACCEPTED      = #100 + 2;
    M_OWNERSHIPRIGHTS  = #100 + 3;
    M_RECEIVERIGHTS    = #100 + 4;
    M_GENERALKERNELREPLY = #100 + 6;
    M_KERNELMSGERROR  = #100 + 7;
    M_PARENTFORKREPLY  = #100 + #10;
    M_CHILDFORKREPLY   = #100 + #11;
    M_DEBUGMSG         = #100 + #12;

type
{
{ Structure:
{     Msg.ptrMsg
{
{ Purpose:
{     Defines format of message header in user area.
}

{

{ Structure:
{     Port
{
{ Purpose:
{     Port      is the basic data structure describing a port.
}

TypeType = packed record
```

```
case integer of
  1: ( TypeName      : Bit8;
        TypeSizeInBits : Bit8;
        NumObjects    : Bit12;
        InLine        : boolean;
        LongForm       : boolean;
        Deallocate     : boolean
      );
  2: ( LongInteger : long)
end;

Port      = long;
ptrPort   = ^Port;

ptrMsg    = ^Msg;
Msg       = record
  SimpleMsg : boolean;
  MsgSize   : long;
  MsgType   : long;
  LocalPort : Port;
  RemotePort: Port;
  ID        : long;
end;

{}

{
  Types:
  {
    PortArray, PortBitArray, ptrPortArray, ptrPortBitArray
  }
  {
    Purpose:
    {
      Used to handle arrays of ports or port conditions.
    }
  }

ptrPortBitArray = ^PortBitArray;
PortBitArray   = packed array [0..MAXPORTS-1] of boolean;

ptrPortArray   = ^PortArray;
ptrAllPortArray = ^PortArray;
PortArray      = array [0..MAXPORTS-1] of Port;

ptrLPortArray  = ^LPortArray;
LPortArray     = array [stretch(0) .. stretch(#777777)] of Port;

GPBuffer      = packed array [1..8] of Bit8;

DirIOCommands = (DirIOInit,
                  DirIORRead,
                  DirIOWrite,
                  DirIORReadCheck,
                  DirIOWriteCheck,
                  DirIOTrackRead,
                  DirIOTrackWrite,
                  DirIOParamRead,
                  DirIOBootRead,
                  DirIOBootWrite,
                  DirIOCclose);
```

```
ptrDirIOArgs = ^DirectIOArgs;
DirectIOArgs = Record
    IOStatus : Integer;
    UnitNumber : Integer;
    PhysAddress : Long;
    Command : DirIOCommands;
end;

DiskType      = (DUnused,
                 D5Inch,           { 5.25 inch Micropolis }
                 D14Inch,          { 14   inch Shugart on EIO }
                 D8Inch,           { 8    inch Micropolis }
                 DSMD,             { SMD (not implemented) }
                 DFloppy,          { Floppy (done elsewhere) }
);
{ leave room for more - don't pack }

DiskParams    = Record
    DskBootSize: Integer;
                  { Size of boot area };
    DskSectors: Integer;
                  { Num sectors per track };
    DskNumHeads: Integer;
                  { Numheads per cylinder };
    DskNumCylinders: Integer;
                  { Num cylinders on the disk };
    DskSecCyl: Long;
    Case HDiskType: DiskType of
        { the kind of disk }
        D5Inch: (WriteCompCyl: Integer;
                  LandingZone: Integer
        );
        D14Inch: (c24MByte : Boolean);
End;

DiskBuffer     = Packed Array[0..DISKBUFSIZE*2-1] of Bit8;
Header         = packed Array[0..15] of Bit8;

ptrArguments = ^Arguments;
Arguments     = record
    ReturnValue : GeneralReturn;
    case integer of
        1: (
            Msg          : ptrMsg;
            MaxWait     : long;
            Option       : integer;
            PtOption     : integer
        );
        2: (
            Ports        : ptrPortBitArray;
            MsgType     : long
        );
        3: (
            SrcAddr     : VirtualAddress;
            DstAddr     : VirtualAddress;
            NumWords    : long;
        );
    end;
end;
```

```
        Delete      : boolean;
        Create      : boolean;
        Mask        : long;
        DontShare   : boolean
    );
4: (  NumCmds    : integer;
      GPIBBuffer : GPBuffer
);
5: (  NormalOrEmergency : boolean;
      EnableOrDisable : boolean
);
6:
    ( SleepID    : long
);
7: (
    RectPort   : Port;
    Xl          : integer;
    Yl          : integer;
    X2          : integer;
    Y2          : integer;
    Kind        : integer
);
8: (
    SrcRect    : Port;
    DstRect    : Port;
    Action      : integer;
    Height      : integer;
    Width       : integer;
    SrcX        : integer;
    SrcY        : integer;
    DstX        : integer;
    DstY        : integer
);
9: (
    Rect        : Port;
    FontRect   : Port;
    Funct       : integer;
    FirstX     : integer;
    FirstY     : integer;
    MaxX       : integer;
    FirstChar   : integer;
    MaxChar    : integer;
    StrPtr     : Pointer;
    Rslt        : integer
);
10:( 
    LockDoLock : boolean;
    LockPortPtr : ptrLPortArray;
    LockPortCnt : long
);
11:( 
    MsgWType   : long;
    MsgWPortPtr : ptrLPortArray;
    MsgWPortCnt : long
)
```

```
end;

{*****
  Process management constants and types
*****}

const

{}

{ NB: The following constants are carefully arranged so that
|     the PCBHandle has all that the microcode needs to access.
|     If you make changes here, you
|     must make corresponding changes in the PCBHandle definition
|     and in the microcode kernel process manager.
| }

MAXPROCS      = 63;           { should be power of two
                             minus one }

NUMPRIORITIES = 16;
NUMSLEEPQS    = 32;           { must be power of two }
NUMQUEUES     = NUMSLEEPQS + NUMPRIORITIES + 5;

type

  ProcState   = ( Supervisor, { 00 - supervisor with
                               privileges |
                               Privileged, { 01 - user with
                               privileges |
                               BadSupervisor, { 10 - supervisor without
                               privileges |
                               User);       { 11 - user without
                               privileges }

  ProcID       = integer;
  PriorID     = 0..NUMPRIORITIES-1;
  QID          = 0..NUMQUEUES;

  ptrInteger   = ^integer;
  ptrBoolean   = ^boolean;

  PStatus      = record
    State        : ProcState;
    Priority     : PriorID;
    MsgPending   : boolean;
    EMsgPending  : boolean;
    MsgEnable    : boolean;
    EMsgEnable   : boolean;
    LimitSet     : boolean;
    SVStkInCore  : boolean;
    QueueID      : QID;
    SleepID      : ptrInteger;
    RunTime      : long;
    LimitTime    : long
  end;
```

```
{*****  
{     Device management constants and types  
{*****  
  
const  
    MAXPARTCHARS      = 8;      { maximum length for  
                                a partition name }  
    MAXDPCHARS        = 25;     { maximum length for  
                                dev:part name }  
    MAXPARTITIONS     = 30;     { maximum partitions  
                                mountable }  
    MAXDEVICES        = 5;      { maximum number of devices }  
    MAXDISKS          = 4;  
  
type  
    PartString         = string[MAXPARTCHARS];  
    DevPartString      = string[MAXDPCHARS];  
    PartitionType      = (Root,UnUsed,Segment,PLX {...} );  
  
    PartInfo   = record           { entry in the PartTable}  
        PartHeadFree : DiskAddr; {pointer to  
                                Head of Free List}  
        PartTailFree : DiskAddr; {pointer to tail  
                                of Free List}  
        PartInfoBlk  : DiskAddr; {pointer to  
                                PartInfoBlock}  
        PartRootDir  : SegID;    {SegID of Root  
                                Directory}  
        PartNumOps   : integer;  {how many  
                                operations  
                                done since last  
                                update of  
                                PartInfoBlock}  
        PartNumFree  : long;     {HINT of how many  
                                free pages}  
        PartInUse    : boolean;  {this entry in  
                                PartTable is  
                                valid}  
        PartMounted  : boolean;  {this partition is  
                                mounted}  
        PartDevice   : integer;  {which disk this  
                                partition is in}  
        PartStart    : DiskAddr; {Disk Address of  
                                1st page}  
        PartEnd      : DiskAddr; {Disk Address of  
                                last page}  
        PartKind     : PartitionType; {Root or Leaf}  
        PartName     : PartString; {name of this  
                                partition}  
        PartExUse   : boolean;  { Opened  
                                exclusively }  
        Unused       : long;     { Port is not  
                                returned }  
        PartDiskRel  : boolean;  
    end;
```

```
PartList = array[1..MAXPARTITIONS] of PartInfo;  
ptrPartList = ^PartList;
```

## **4. AccentUser**

---

Module AccInt contains routines for interfacing with the kernel.  
They are explained in the document "The Kernel Interface" in  
the *Accent Programming Manual*.



## **5. Aload**

Module ALoad provides facilities that are used to load and execute Pascal programs. This module is used by the Shell and is not of general use to most users.

### **5.1. Procedure ARunLoad**

```
Procedure ARunLoad(RunFileName: Path_Name; p: pointer;
                   filesize: long; hiskport: port;
                   LoadDebug: boolean);
```

#### Abstract:

Load a process with a run file image.

#### Parameters:

##### RunFileName

Name of the Pascal run file to be loaded. This may be null if you wish to load the process with a file that is mapped into memory.

P A pointer to a run file structure that is in memory. To load a Pascal program from a file this parameter must be nil.

FileSize Size of the file, in bytes, to be loaded

HisKPort Kernel port of the process that is to be loaded

##### LoadDebug

If this parameter is true, print information about the loading process.

### **5.2. Procedure ShowRun**

```
Procedure ShowRun(p: pointer; MapFileName: Path_Name);
```

#### Abstract:

This procedure is used to display information about a Run file.

#### Parameters:

P A pointer to an in-memory image of a run file

**MapFileName**

Name of the file that will be written with the information about the run file

### 5.3. Function LinkTypeStr

```
Function LinkTypeStr(typ: LinkFileType): string;
```

**Abstract:**

Translate a LinkFileType into a string.

**Parameters:**

Typ        A type field indicating the type of link block

**Returns:**

The string name of the type Typ

### 5.4. Exception ALoadError

```
exception ALoadError(s: string_255);
```

**Abstract:**

This exception is raised when there is an error during the loading process.

**Parameters:**

S        A string that will be set to contain an indication of why the load failed

### 5.5. Function DateString

```
Function DateString(date : Internal_Time) : String;
```

**Abstract:**

Converts a date in internal format into a string.

**Parameters:**

Date        A date in system internal format

**Returns:**

String data that is represented by Date

## **6. AuthDefs / AuthUser**

---

Modules AuthDefs in AuthDefs.Pas and Auth in AuthUser.Pas provide the client interface to the Authentication Server. The Authentication server port, SysAuthPort, is obtained through the name server function Lookup.

### **6.1. AuthDefs - Exported Types and Constants**

```
CONST
  Auth_Var_Size = 30;

  No_User    = 0;           { files owned by "nobody" }
  First_User = 1;          { first valid user }
  Max_Users  = 1023;
  Unknown_User = 1023;

Type
  Auth_Var = String[Auth_Var_Size];

  User_ID   = No_User..Max_Users; { must be "bit10" }

  PassType = Long;           { a two word value }
                           { 4 chars for a password }

  UserRecord = record
    Name:      Auth_Var; { Name of user }
    UserID:    User_ID; { User ID of user }
    EncryptPass: PassType; { Encrypted password}
    Profile:   APath_Name; { Path name of the
                           { profile file }
    NameOfShell: APath_Name; { Name of the
                           { Shell.RUN file }
  End;

  Machine_Name = String[255];

  Check_Type = (Check_Login,     { user is logging in }
                Check_User); { user is changing }
                           { Parameters }

  Logged_User = record         { one logged-in user }
    UserID      :User_ID;
    UserName    :Auth_Var;
    MachineName :Auth_Var;
  End;
```

```
Logged_User_Array = array[0..0] of Logged_User;
Logged_User_List  = ^Logged_User_Array;
```

```
CONST
  Auth_Error_Base    = 5000;
  UserNameNotFound  = Auth_Error_Base + 1;
  PassWordIncorrect = Auth_Error_Base + 2;
  AuthPortIncorrect = Auth_Error_Base + 3;
```

## 6.2. AuthUser

### 6.2.1. Function LoginUser

```
Function LoginUser(
    ServPort      : Port;      { SysAuthPort }
    UserName     : Auth_Var;
    Password     : Auth_Var;
    MachineName  : Auth_Var;
    Var UserAuthPort
    Var UserRec   : UserRecord
): GeneralReturn;
```

#### Abstract:

Logs a user in to the authentication server.

#### Parameters:

Servport Authentication Server port

Username Name of the user that we want to check

Password Password for the user

UserAuthPort

Port returned to the user if his or her name and password match

UserRec Is filled with the user information if the user name and password match

#### Returns:

Success Valid user - logged in

UserNameNotFound

Invalid user

### 6.2.2. Function LogoutUser

```
Function LogoutUser(
    ServPort      : Port;      { User port }
): GeneralReturn;
```

#### Abstract:

Logs a user out.

#### Parameters:

ServPort User's connection to the Authentication Server

**Returns:**

Success

AuthPortIncorrect

### **6.2.3. Function ConfirmUser**

```
Function ConfirmUser(
    ServPort : Port;                      { SysAuthPort }
    UserAuthPort : Port;
    var UserID : User_ID;
    Var UserMachineName : Auth_Var
): GeneralReturn;
```

**Abstract:**

Checks the UserAuthPort to ensure that the user is logged in. If it is, returns useful information about the user.

**Parameters:**

ServPort    Port for the authentication server

UserAuthPort

            Signature port

UserID     Returns the ID number for the user

UserMachineName

            Returns the name for the machine the user is logged on

**Returns:**

Success    If the user is valid

### **6.2.4. Function CheckUser**

```
Function CheckUser(
    ServPort : Port;                      { User Port }
    UserName : Auth_Var;
    PassWord : Auth_Var;
    Var UserRec : UserRecord
): GeneralReturn;
```

**Abstract:**

Verifies a user-name / password pair and returns information about that user.

**Parameters:**

ServPort    Authentication Server port

Username Name of the user that we want to check  
Password Password for the user  
UserRec Filled with the user information if the user name and password match

**Returns:**

Success Valid user

UserNameNotFound

Invalid user

### 6.2.5. Function ChangeUserParams

```
Function ChangeUserParams(
    ServPort      : Port;           { User Port }
    UserName      : Auth_Var;
    CurrentPassword : Auth_Var;
    ChangePassword : boolean;
    NewPassword   : Auth_Var;
    NewProfile    : APath_Name;
    NewShell      : APath_Name
) : GeneralReturn;
```

**Abstract:**

This function changes the parameters for a logged-in user.

**Parameters:**

ServPort User's authentication port

Username Name of the user whose information we want to change

CurrentPassword

    Current password in the user's record

ChangePassword

    True -> change password;

    False-> don't change password

NewPassword

    A new password to replace the current one

NewProfile

    Path name of the profile file for this user

NewShell Name of the shell to be stored in the user record

**Returns:**

Success If the user was added or changed

AuthPortIncorrect

If the user did not have the proper access rights to add or change a user

**Side Effects:**

This function will change the file PassFile.

### 6.3. Function GetUserName

```
Function GetUserName(
    ServPort : Port;           { SysAuthPort }
    UserID   : User_ID;
    var UserName : Auth_Var
): GeneralReturn;
```

**Abstract:**

Gets the user name corresponding to a User ID.

**Parameters:**

ServPort Authentication server port

UserID User ID

UserName Returns name for user

**Returns:**

Success

UserNameNotFound

#### 6.3.1. Function ListLoggedInUsers

```
Function ListLoggedInUsers(
    ServPort      : Port;      { SysAuthPort }
    Var UserList  : Logged_User_List;
    Var UserList_Cnt : Long
): GeneralReturn;
```

**Abstract:**

Returns all of the users currently logged in to this Authentication Server.

**Parameters:**

**ServPort**    AuthServer service port

**UserList**    Returns a list of user <-> ID <-> Name <-> Machine

**UserList\_Cnt**

              Returns the number of users logged in

**Returns:**

Success

### **6.3.2. Procedure InitAuth**

**procedure InitAuth(RPort : port);**

**Abstract:**

**InitAuth** provides the remote procedure call initialization for client access to the authentication server. It should be called once by the client program before any other routines in **AuthUser** are used.



## 7. BootInfo

---

Module BootInfo in BootInfo.Pas provides the definition of the Boot Information block. This block is set up, in part, by MakeVMBoot when a boot file is created. The machine specific fields are filled in by the system microcode when the machine is booted. You must have physical memory privileges to access these structures.

```
const
  BootBlockLocation = #200000000000;

type ASTRecord = integer;

type
  MachineInfoRec = packed record
    case boolean of
      false: (int: integer); { configuration comes in a word }
      true: (
        WCSSize: 0..15; { 0 -> 4K WCS }
                    { 1 -> 16K WCS }
        Reserved: 0..3;
        IsPortrait: Boolean; { True -> Portrait screen }
                      { False ->
                        Landscape screen }
        BoardRev: 0..31; { IO Board revision /
                           disk type:
                           { 0 -> CIO board with
                             Shugart disk
                           }
                           { 1 -> CIO board with
                             Micropolis disk
                           }
                           { 16 -> EIO board
                           }
                           { }
                         }
        OldZ80: boolean; { True -> Old
                           Z80 protocol }
                      { False -> New
                        Z80 protocol }
        CMUNet: boolean; { True for CMU network
                           environment}
                      { False for standalone
                        10MBit net}
        Reserved2: 0..3
      )
    end;
  { Boot Information Record }
```

```

BIRecord = packed record
  case integer of
    1: ( IntBlk: array [0..255] of integer );
    2: (
      OvlTable : array [0..11] of VirtualAddress;
                  { Overlays}
      VP :       VirtualAddress;
                  { Address of VP table }
      PV :       VirtualAddress; { " of PV table }
      PVList :   VirtualAddress; { " of PV list }
      Sector :   VirtualAddress; { " of sector
                                  headers}
      PCB :      VirtualAddress; { " of PCB
                                  Handles }
      AST :      VirtualAddress; { " of AST }
      AccentQueue: VirtualAddress; { " of Queue
                                     headers }
      AccentFont : VirtualAddress; { " of Font }
      AccentCursor :VirtualAddress; { " of Cursor }
      AccentScreen :VirtualAddress; { " of Screen
                                     segment}
      ScreenSize : integer;
      FreeVP :    integer;        { Initial FreeVP }
      FreeAST :   integer;
      SchedProc : integer; { High level scheduling
                           process}
      InitProc :  integer; { Initial process }
      BootChar :  integer; { Character used in boot }
      NumProc :   integer; { Number of processes
                            set up }
      StackSize : integer; { Size of sup. stack in pages}
      GlobalSize :integer; { Size of sup. global area
                            in pages}

      NumSVReg : integer; { Number of regs for
                           SVCall }
                           {** NumSVRegs is obsolete -
                           GGR 11/16/81 **}

      TrapCode:  integer;
      TrapArgs:  VirtualAddress;
      MemBoard:  integer; { number of K of memory }
      AccentStdCursor: VirtualAddress;
      AccentRoTemp:  VirtualAddress;
      DefaultPartitionName: String[19];
      IgnoreRunFile: Boolean;
      MachineInfo:  MachineInfoRec;
      Filler:     array [0..49-WordSize(AstRecord)] of integer;

      FirstAst:  ASTRecord;
      EtherIOArea: VirtualAddress;
      UserPtr:   VirtualAddress; { Start of user
                                 process }

      SVContext: record

```

```
      SV_CS:           integer;
      SV_GP:           integer;
      SV_LP:           integer;
      SV_LocalSize:   integer;
      SV_TrapCount:   integer;
      SV_FirstRN:     integer;
      SV_PC_Vector:   array [0..121]
                       of integer
    end
  )
end;

ptrBIRecord = ^BIRecord;
```



## **8. CFileDefs / CLoad**

Modules CFileDefs in CFileDefs.Pas and CLoad in CLoad.Pas provide the facilities that are used to load a process with a C program. These procedures are used by the Shell and are not of general use to other programs.

### **8.1. CFileDefs**

#### **Types and Constants**

```
type

  LString      = String[ 255];

  pFirstBlock  = ^FirstBlock;
  FirstBlock =
    record
      FileVersion     : integer;           { major version # }
      FileSize        : long;             { size of this file
                                            in bytes}
                                            {C}
      FileTimeStamp   : Internal_Time; {C}
      SymbolAreaSize  : long;            { in pages }
      TextAreaSize    : long;            { in pages }
      DataAreaSize    : long;            { in pages } {C}
      BSSAreaSize     : long;            { in pages } {C}
      DestAddr        : long;            { word address of
                                            file in process }
      StartAddr       : long;            { of program
                                            (i.e. 'crt0') }

      MainAddr        : long;            { of program
                                            (i.e. 'main') }

      InitialLocalSize : long;          { words needed for
                                            first stack frame
                                            locals }

      StackBaseAddress : long;          { word address }
      StackSizeInPages : long;

    end;

{ $IFC (WORDSIZE(FirstBlock) > 256) THEN}
  ?Error: First block is too big
{$ENDC}

const
```

```
CFileVersion      -4:  
{  
{ CFileVersion is the major version number for 'C'. It should  
{ not conflict with the version number in 'Pascal' run files.  
}  
  
{  
{ Symbol entry codes  
{  
PrimaryDef          = 10;  
    {}  
    { Symbol Kinds (kind of (primary) entry codes)  
    {}  
    PascalProcedure     = 1;  { UnUsed }  
    LocalProcedure      = 2;  
    GlobalProcedure     = 3;  
    InitializedSymbol   = 5;  
    LocalLabel          = 6;  
    UndefinedSymbol     = 7;  
        { synonyms }  
    UndefinedGlobal      = UndefinedSymbol;  
    DefinedGlobal         = InitializedSymbol;  
    DefinedLocal          = LocalLabel;  
OffsetDef           = 11;  
ChainHead           = 12;  
AbsoluteDef         = 13;  
AbsoluteLocalDef    = 14;  
LibraryDef          = 90;  
  
{  
{ Area Designators  
{  
TextState            = 0;  
DataState            = 1;  
Data2State           = 2;  { Internal to asm }  
    { synonyms }  
PCInText             = TextState;  
PCInData              = DataState;  
PCInData2             = Data2State;  
  
{  
{ Misc. stuff  
{  
OFFSETBASE           = 16000; { Maximum computed  
                           offset value. }
```

## 8.2. CLoad

### 8.2.1. General

Module CLoad in CLoad.Pas provides the interfaces that are used to load a process with a C program.

### 8.2.2. Function CLoadProcess

```
Function CLoadProcess( FileName : APath_Name;
                      var FileInMem: pointer;
                      var FileSize : long;
                      Proc      : Port;
                      LoadDebug: Boolean): GeneralReturn;
```

#### Abstract:

This procedure is used to load a process with a C program.

#### Parameters:

FileName Name of the C run file to be loaded. This may be null if you wish to load the process with a file that is mapped into memory.

FileInMem Pointer to the memory image of the file to load or NIL. It may return a pointer to the memory image if the file is not a C file.

FileSize Size of the file, in bytes, to be loaded. It may return the size of the memory image if the file is not a C file.

Proc Kernel port of the process that is to be loaded

LoadDebug

If this parameter is true, print information about the loading process

#### Returns:

Success File was loaded - process is set up and ready to run. The file image has been freed.

CLoadNotCFile

File was not a C file. FileInMem points to a copy of the file, and FileSize contains the size of the copy in bytes. (The memory image can then be passed to the Pascal loader.)

other The process was not loaded. All memory in the parent process has been freed.

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**Clock**

## **9. Clock**

---

### **9.1. General**

Module Clock in Clock.Pas provides a quick 60-hz clock routine.

### **9.2. Function IOGetTime**

Function IOGetTime: long;

Returns:

Clock value Number of screen refresh cycles since system was started



## **10. CommandDefs, CommandParse, and ExtraCmdParse**

### **10.1. Introduction**

Modules CommandDefs, CommandParse and ExtraCmdParse (in CommandDefs.Pas, CommandParse.Pas, and ExtraCmdParse.Pas, respectively) provide several routines intended to ease the task of developing utilities that conform to the "standard" command syntax conventions.

Command parsing occurs in four distinct phases:

1. The Shell transforms the user's input command into a list of words (Tokenization / partial Syntax Analysis).
2. The ParseCommand routine of CommandParse further processes this word list into lists of inputs, outputs, and switches (final Syntax Analysis).
3. Your utility should scan the three lists to ensure correctness of the parsed information (Semantics Analysis).
4. Your utility executes the command (Execution).

In the case that your utility must prompt for user's input, the first two of these phases (Tokenization and Syntax Analysis) are performed by the other parsing routines provided by CommandParse and ExtraCommandParse.

On the surface, it might seem to you that the last two phases of parsing could be merged for better efficiency. You are strongly urged not to merge them! The user is permitted, within the

"standard" syntax, to enter a boolean switch and then later on override that switch with another. Thus, the last switch value is the value to be used. If phases 3 and 4 were to be merged you might have performed an operation under the first switch value when the last was intended. An example is the command:

**DELETE -NOCONFIRM \*.\* -CONFIRM**

Clearly the user meant to perform this deletion with confirmation. Thus the Delete utility can not actually execute any operation until scanning of the command is complete.

Scan the entire parsed list of switches first. The results of this scan should be retained for use in the actual execution of the command. Every effort should be made to define switches in a context insensitive manner. That is, switches should apply to the command as a whole rather than applying to individual parameters within the command. The last occurrence of boolean-valued switches should be used, as in the above example of Delete. Note that exceptions to this context insensitivity rule exist in the Linker and with the HELP switch.

Execute the HELP switch first by displaying a helpful message, discarding everything else on the command line, and then do either: (1) return to the shell; (2) if your utility takes a long time (more than 10 seconds) to initialize itself (like the compiler) prompt the user for an input and them process it; or, (3) like FTP and Floppy, prompt the user for another command as you normally would.

The routines in CommandParse do not directly interact with the user. Rather, they process a "pool" of characters which has been read-in by some other means (either by the Shell or by the routines in ExtraCmdParse). The routines in CommandParse are divided into four functional groups:

1. those that deal with nested command files.
2. those that perform the actual parsing.
3. those that deal with identifying words.
4. other miscellaneous routines that deal with character pools.

The routines in ExtraCmdParse do directly interact with the user via the keyboard / display and then return the parsed data structures representing that input.

## **10.2. Usage of the CommandParse Module**

### **10.2.1. Constants**

CommandParse exports definitions of the separator characters of the "standard" syntax so that applications may provide their own recognizers utilizing these characters if they so desire. Note that the identifiers beginning 'shell\_' represent characters that are only recognized by the Shell. They are defined in this module for completeness and so that all are aware of characters that may have special meanings in the syntax.

CommandParse also exports two characters to be used in User prompts. The 'CmdChar' should be used as the last character of user prompts caused from outside a nested command file. The 'CmdFileChar' should be used inside any nested command file. The routines 'GetCmd' and 'GetParsedUserInput' in ExtraCmdParse deal with these characters automatically, while 'GetCharacterPool' does not.

### 10.2.2. Types

CommandParse exports two string types and pointers to them for use in manipulating the text of command words. The caller should take great care *not* to attempt to store into a ‘Word\_String,’ nor should any attempt be made to deallocate data pointed to by a ‘pWord\_String.’ These strings may refer to a portion of a larger “pool” of characters (e.g., the word list passed to your utility by the Shell).

Always copy the command word string into a string variable before performing any string manipulations that might change the text of the command word string.

### 10.2.3. Command file routines

The command file routines provide support for a LIFO stack of nested command files. This stack is maintained by these routines as a singly-linked list of file descriptors, ‘Command\_File\_List.’ This list always ends in a node associated with the current default input file to act as a backstop.

InitCmdFile initializes the command stack to contain the backstop node associated with the default input file. This routine must be called before any usage of the command file stack.

OpenCmdFile locates, opens, and pushes a new file onto the command file stack.

ExitCmdFile closes and pops the top file off of the command file stack. An attempt to exit the backstop results in simply the re-opening of the default input file and the backstop stack entry is retained.

ExitAllCmdFiles exits all of the files currently on the command file stack and re-opens the default input file as the backstop.

DstryCmdFiles exits all stacked command files INCLUDING the backstop. InitCmdFile must be called prior to any further command file stack usage.

#### 10.2.4. Parsing routines

These routines transform an unbounded "pool" of characters into three linked lists containing the command inputs, outputs, and switches. The rules of the "standard" command syntax are enforced. The routine ExerciseParseEngine is the heart of the parsing activity. It scans the given "pool" until an end-of-file condition is detected. If all of the characters in the given "pool" are consumed, then ExerciseParseEngine invokes a user-supplied routine to obtain the next "pool" to be processed.

The linked lists resultant from parsing contain a pointer to the text of the command word and, for switches, links describing the context in which the switch occurred.

InitCommandParse initializes the data table that forms the state machine utilized by ExerciseParseEngine. This routine should be called before any usage of the parsing routines.

DestroyCommandParse deallocates the store used by the parsing state machine. InitCommandParse must be called before any more parsing activity can occur.

AllocCommandNode allocates the storage for a Command\_Word\_List entry. None of the items within the allocated node are initialized by this routine.

DestroyCommandList deallocate all of the nodes on a given Command\_Word\_List.

AlwaysEof a utility routine which may be used in connection with ExerciseParseEngine. This routine signals an end-of-file

condition to the parser.

ExerciseParseEngine is the core routine of the parsing activity. All other routines that perform parsing utilize this one to actually perform the parse. It scans an unbounded "pool" of characters, asking the user for more if it needs them, until an end-of-file condition is detected. It enforces the rules of the "standard" syntax. To do this, it employs a data driven state machine in the manner of classical lexical analysis.

ParseChPool uses ExerciseParseEngine to parse the single "pool" of characters it is given. That is the given "pool" is simply passed on to ExerciseParseEngine and AlwaysEof is utilized as the routine to read the next pool when the first is exhausted.

ParseCommand transforms the list of words given to this process by Shell / Spawn into the three lists of inputs, output, and switches. Note that the first word in the list of words is ignored.

WordifyPool transforms the given unbounded "pool" of characters into a list of words suitable for passing to Spawn.

#### 10.2.5. Word identification routines

Four routines are provided to support the classification of word-prefixes.

A Word Search Table is the underlying data structure for these identification routines. A Word Search Table is an array of pointers to linked lists of words to be identified. Each linked list contains those words with the same first character. Thus the common case of classifying a word whose first character is sufficient for uniqueness is fast.

Regarding timings, note that initializing a word search table is about ten times *slower* and that searching is about thirty times

faster than the older methods utilized in POS and prior Accent releases. Initialization plus one search is slightly faster than one search the old way. This means utilities should delay initialization of their switch word search table until they realize that they actually have a switch to deal with.

InitWordSearchTable allocates a search table and initializes the word list banner nodes to nil. The caller may indicate that the table being created is to contain words whose capitalization are to be ignored for search purposes.

AddSearchWord merges a word into an existing search table. The caller specifies (in addition to the text of the word) a numeric key by which the word is to be known. This key is returned by the identification routine. The key may be any arbitrary non-negative integer. The negative numbers are reserved by the identification and parsing routines to indicate errors.

DestroySearchTree deallocates the memory of a search table.

UniqueWordIndex is the word identification routine. It returns the word-key for the given prefix if and only if the prefix is uniquely found in the given table; otherwise an error is indicated. The entire text of the found word is also returned.

#### **10.2.6. Miscellaneous pool manipulators**

Routines are provided to translate a PERQ character string to / from an unbounded pool of characters as used by the parsing routines.

Also provided is a routine to deallocate the memory of an unbounded character pool.

## 10.3. Usage of the ExtraCmdParse Module

### 10.3.1. Routines GetCmd and GetShellCmd

These routines are intended for those utilities that expect their first input word to be a verb specifying an action to the utility. Two such utilities are FLOPPY and FTP. GetCmd and GetShellCmd also automatically nest command files should a command file reference appear as the only input. Because of these assumptions of the meaning of the first input word, GetCmd and GetShellCmd are probably not appropriate for utilities that do not accept verbs.

GetCmd performs the following:

- Prompts the user;
- Reads input;
- Parses the input.
- If the first (and only) input word is a command file reference then set up to accept further inputs from that file (via OpenCmdFile) and return a null input indication to the caller. The next GetCmd will automatically read the command file for input.
- Uses the word search routine to attempt to identify the first input word.
- Returns the parsed data (sans the first word) and identification results to the caller.

GetShellCmd performs basically the same processing as GetCmd. Except that GetShellCmd does not prompt the user for input, but rather utilizes the word list information passed to the utility by the Shell.

A short example of GetCmd / GetShellCmd:

```
InitCmdFile(Input_File_List);
InitCommandParse;

(* get the first input from the shell *)

Verb_Word_Key := GetShellCmd(Verb_Word_Search_Table
                           Verb_String,
                           Input_File_List,
                           Inputs, Outputs, Switches,
                           GR);

(* command file nesting handled transparently to the caller *)

while The.Utility_Has_Something_To_Do do
begin
  case Verb_Word_Key of
    Cmd_SomeError:
      GRWriteStdError(GR, GR_whatever, '');
    Cmd_NotIns_MaybeSwitches:
      doSwitches(Switches);
    Cmd_EmptyCmdLine:
      (* do nothing *)
    Cmd_NotUnique:
      GRWriteStdError(ErCmdNotUnique,
                      GR_whateverVerb_String);
    Cmd_NotFound:
      GRWriteStdError(ErBadCmd,
                      GR_whatever, Verb_String);
    some_cmd_key:
      do_some_cmd (whatever_the_command_needs,.....
                  Inputs, Outputs, Switches);

      (* deal with all inputs, outputs, *)
      (* and switches as is appropriate *)
      (* for the given command verb *)

    some_other_cmd_key:
      do_some_other_cmd(whatever_that
                        _command_needs,.....
                        Inputs, Outputs, Switches);

      (* deal with all inputs, outputs, *)
      (* and switches as is appropriate *)
      (* for the given command verb *)
  :
  :
  end;

(* get the next input from the user *)

Verb_Word_Key := GetCmd('utility name',
                        Verb_Word_Search_Table
                        Verb_String,
```

```
Input_File_List.  
Inputs. Outputs. Switches,  
CR);  
  
(* command file nesting handled *)  
(* transparently to the caller *)  
  
end;
```

### 10.3.2. Routine GetParsedUserInput

GetParsedUserInput is a general routine used to prompt the user for input and then return parsed data to the caller. It makes no assumptions about the meaning of the words it reads (other than the normal syntactic rules). Thus it does not automatically nest command files nor does it attempt to identify the first input word. This routine should be used by those utilities which desire to prompt the user for an entire command input and are therefore willing to deal with command file nesting themselves. Two such utilities are the COMPILER and the LINKer which prompt for a new, full input command line upon discovery of an error in the Shell command line.

### 10.3.3. Routine GetConfirm

GetConfirm is used to obtain from the user the answer to a Yes/No confirmation question. It will enforce the semantic rules of such confirmations:

- no outputs allowed (e.g., ~ is illegal)
- no command files allowed (e.g., @ is illegal)
- only zero or one inputs allowed
- if zero inputs then either switch(es) must exist or the line must be empty
- if one input then NO switch(es) must exist and it must be either Yes or No

#### 10.3.4. Routine GetChPool

GetChPool is used to obtain from the user a raw hunk of characters. GetChPool makes no assumptions about the meaning of the characters nor does it attempt any processing of the characters.

Note that since GetChPool attempts no interpretation of the characters it reads, it will not recognize quoted end-of-line characters and therefore it will not read continuation lines.

## 10.4. CommandDefs

### Abstract

Definitions for the command structure passed between Accent programs.

```
{}
{ Error General Return values
}

const
    CmdParse_Error_Base = 4200;

    ErBadSwitch      = CmdParse_Error_Base + 1;
    ErBadCmd         = CmdParse_Error_Base + 2;
    ErNoSwParam     = CmdParse_Error_Base + 3;
    ErNoCmdParam    = CmdParse_Error_Base + 4;
    ErSwParam        = CmdParse_Error_Base + 5;
    ErCmdParam       = CmdParse_Error_Base + 6;
    ErSwNotUnique   = CmdParse_Error_Base + 7;
    ErCmdNotUnique  = CmdParse_Error_Base + 8;
    ErNoOutFile     = CmdParse_Error_Base + 9;
    ErOneInput       = CmdParse_Error_Base + 10;
    ErOneOutput      = CmdParse_Error_Base + 11;
    ErIllCharAfter   = CmdParse_Error_Base + 12;
    ErBadQuote       = CmdParse_Error_Base + 13;
    ErAnyError       = CmdParse_Error_Base + 14;

    ParseInternalFault = CmdParse_Error_Base + 15;
    ParseWordTooLong  = CmdParse_Error_Base + 16;
    ParseIllegalCharInSwName = CmdParse_Error_Base + 17;
    ParseIllegalCharInSwVal  = CmdParse_Error_Base + 18;
    ParseIllegalCharInEnvNam = CmdParse_Error_Base + 19;
    ParseIllegalCharInQuoted = CmdParse_Error_Base + 20;
    ParseOnlyCmdAllowed = CmdParse_Error_Base + 21;
    ParseIllegalCharInInRed = CmdParse_Error_Base + 22;
    ParseIllegalCharInOutRed = CmdParse_Error_Base + 23;
    ParseIllegalCharInShPara = CmdParse_Error_Base + 24;

type
    Character_Pool = packed array [0..0] of char;
                    {* an unbounded chunk of characters *}

    pCharacter_Pool = ^Character_Pool;

    Char_Pool_Index = long;

    CommandBlock = record
        WordCount      : long;
                        { number of words }
        WordDirIndex   : Char_Pool_Index;
                        { Byte index to word
                          dictionary }
        WordArrayPtr   : pCharacter_Pool;
    end;
```

```
WordArray_Cnt : Char_Pool_Index;  
end;
```

## 10.5. Function Null\_CommandBlock

```
Function Null_CommandBlock: CommandBlock;
```

### Abstract:

Null\_CommandBlock is used to get a new, empty CommandBlock.

### Returns:

A new empty CommandBlock

## 10.6. CommandParse

### 10.6.1. Procedure InitCmdFile

```
Procedure InitCmdFile(var inF: pCommand_File_List);
```

#### Abstract:

Initializes inF to be a valid Text File corresponding to the keyboard. This must be called before any other command file routines. The application should then read from inF^.cmdFile. E.g. ReadLn(inFile^.cmdFile, s); or while not eof(inFile^.cmdFile) do ... Use popup only if inF^.next = NIL (means no cmd File). Is a fileSystem file if not inF^.isCharDevice. InF will never be NIL. The user should not modify the pCommand\_File\_List pointers; use the procedures provided.

#### Parameters:

InF        Set to the new command list

### 10.6.2. Function OpenCmdFile

```
Function OpenCmdFile(FileName: pWord_String;  
                  var inF: pCommand_File_List): GeneralReturn;
```

#### Abstract:

This function will prepare a command file for use by the standard Pascal I/O routines. The user should give OpenCmdFile the word as parsed by one of the parsing routines contained in this CommandParse module. The application is expected to ensure that the command file has appeared in a correct context for that application. These checks might include ensuring that no other words appeared within the input line containing the command file. This function maintains a stack of command files so that command files may contain other command files. Be sure to call InitCmdFile before calling this procedure.

#### Parameters:

FileName    Name of the command file to be opened. The application is responsible for ensuring that the file appears in a correct context.

inF        List of command files. This was originally created by InitCmdFile and is maintained by these routines. If FileName is a valid file, a new entry is put on the front of inF describing it. If there is an

error, then inF is not changed. In any case, inF will always be valid.

**Returns:**

A GeneralReturn code describing the success / failure of the Open

### **10.6.3. Procedure ExitCmdFile**

```
Procedure ExitCmdFile(var inF: pCommand_File_List);
```

**Abstract:**

Remove top command file from list. Call this whenever the end of a command file is reached.

**Suggested Use:**

```
While EOF(inF^.cmdFile) do ExitCmdFile(inF);
```

**Parameters:**

inF      List of command files. It must never be NIL. The top entry is removed; except when attempting to remove last command file, when it is simply re-initialized to be the backstop default input file. It is OK to call this routine even when at the last entry of the list.

**Returns:**

One element is popped off of the command stack, inF.

### **10.6.4. Procedure ExitAllCmdFiles**

```
Procedure ExitAllCmdFiles(var inF: pCommand_File_List);
```

**Abstract:**

Removes all command files from the given list. Use when a fatal error has been found or upon receipt of either a SigLevel2Abort or a SigLevel3Abort in order to reset all command files.

**Parameters:**

inF      List of command files. It must never be NIL. All entries but the last are removed.

**Returns:**

inF        Reset to a single list node attached to the backstop default input file

### 10.6.5. Procedure DstryCmdFiles

```
Procedure DstryCmdFiles(var inF: pCommand_File_List);
```

Abstract:

Removes all command files from the given list. All entries are removed and inF is set to NIL. InitCmdFile must be called again before any command file stack routines may be used.

Parameters:

inF        List of command files to be released

Returns:

inF        Set to NIL

### 10.6.6. Procedure InitCommandParse

```
Procedure InitCommandParse;
```

Abstract:

Initializes the parser by marking the parsing DFA tables as uninitialized. The Parser, when first invoked, will notice that the parsing table need to be initialized and will do so (thus delaying a possibly lengthy initialization process until the data is actually needed).

### 10.6.7. Procedure DestroyCommandParse

```
Procedure DestroyCommandParse;
```

Abstract:

Deallocates the parsing DFA table.

### 10.6.8. Function AllocCommandNode

```
Function AllocCommandNode(WordClass: Word_Type;  
                         WordString: Cmnd_String): pCommand_Word_List;
```

#### Abstract:

Allocates a new node to be inserted into the parsed data structures. User is responsible for establishing the correct linkages. This routine merely creates a node with the correct string text value for the word.

#### Parameters:

WordClass Class desired for the new node

WordString Text of the new word

#### Returns:

Returns a pointer to the newly allocated node

### 10.6.9. Procedure DestroyCommandList

```
Procedure DestroyCommandList(var argList:  
                           pCommand_Word_List);
```

#### Abstract:

Deallocates a parsed data structure.

#### Parameters:

argList A pointer to the structure to be released

#### Returns:

argList Set to NIL

### 10.6.10. Procedure AlwaysEof

```
Procedure AlwaysEof(var ChPool: pCharacter_Pool;  
                     var PoolLength: Char_Pool_Index);
```

#### Abstract:

Support routine for callers of ExerciseParseEngine. This routine sets up a character pool which contains an end-of-file marker. This will signal the parsing activity to stop.

#### Parameters:

**ChPool** Pointer to the pool to contain the end-of-file marker

**PoolLength**

length of the eof buffer (always set to 1 by this routine)

**Returns:**

Returns an unbounded character pool containing an end-of-file marker.

### 10.6.11. Function ExerciseParseEngine

```
Function ExerciseParseEngine(ChPool: pCharacter_Pool;
                           PoolLength: Char_Pool_Index;
                           procedure ReadPool(var Pool:
                                   pCharacter_Pool;
                                   var PLen:
                                   Char_Pool_Index);
                           var inputs: pCommand_Word_List;
                           var outputs: pCommand_Word_List;
                           var switches: pCommand_Word_List);
                           GeneralReturn*);
```

**Abstract:**

This routine is the Parser. It is called by all other routines which perform parsing. It scans the given character pool (augmented if necessary by reading another pool) and constructs the parsed data lists of inputs, outputs, and switches.

**Parameters:**

**ChPool** First hunk of characters to be parsed. This parameter may be NIL; in which case NIL parse structures will be Returned.

**PoolLength**

Length of the first pool of characters. This parameter may be zero; in which case NIL parse structures will be Returned.

**ReadPool** A procedure to read successive pools of characters, if required by the parsing actions

**inputs** The list to contain the words recognized as inputs

**outputs** The list to contain the words recognized as outputs

**switches** The list to contain the words recognized as switches

**Returns:**

Places the parsed lists into the parameters and returns a GeneralReturn code of either Success if all went well or an error code indicative of the error.

**10.6.12. Function ParseChPool**

```
Function ParseChPool(ChPool: pCharacter_Pool;  
                     PoolLength: Char_Pool_Index;  
                     var inputs: pCommand_Word_List;  
                     var outputs: pCommand_Word_List;  
                     var switches: pCommand_Word_List):  
                     GeneralReturn;
```

**Abstract:**

Parses the given unbounded pool of characters.

**Parameters:**

ChPool      Pointer to the beginning of the pool to be parsed

PoolLength

              Number of characters in the pool

inputs      List to contain the input words

outputs     List to contain the output words

switches    List to contain the switches

**Returns:**

Sets the three lists and returns a GeneralReturn code indicative of the parse results.

**10.6.13. Function ParseCommand**

```
Function ParseCommand(var inputs: pCommand_Word_List;  
                     var outputs: pCommand_Word_List;  
                     var switches: pCommand_Word_List):  
                     GeneralReturn;
```

**Abstract:**

Transforms the word list passed into the program into the parsed data lists of input, outputs, and switches.

The program name is normally passed to the program as the first word in the

list. This word is stripped and ignored by ParseCommand.

**Parameters:**

inputs      List to contain the input words  
outputs     List to contain the output words  
switches    List to contain the switches

**Returns:**

Sets the three lists and returns a GeneralReturn code indicative of the parse results.

### **10.6.14. Function WordifyPool**

```
Function WordifyPool(ChPool: pCharacter_Pool;  
                  PoolLength: Char_Pool_Index;  
                  var WordStruct: CommandBlock): GeneralReturn;
```

**Abstract:**

Transforms the given unbounded pool of characters into a word list suitable for passing to a Spawn'd child process.

**Parameters:**

ChPool      Pointer to the beginning of the pool to be 'wordified'  
PoolLength

Number of characters in the pool

WordStruct A buffer to contain the word list description

**Returns:**

Sets the WordStruct and returns an GeneralReturn code indicative of the wordification results.

## 10.6.15. Procedure InitWordSearchTable

```
Procedure InitWordSearchTable(var table: pWord_Search_Table;  
                           CaseSensitive: boolean);
```

### Abstract:

Creates an empty search table for use in word identification.

### Parameters:

table      Will be set to the address of the new search table

CaseSensitive

TRUE if the words in this table should retain their capitalization  
during the identification processing

### Returns:

The address of a new search table is returned in 'table'.

## 10.6.16. Procedure AddSearchWord

```
Procedure AddSearchWord(table: pWord_Search_Table;  
                      WordKey: integer;  
                      WordString: Cmnd_String);
```

### Abstract:

Merges the given word into the search table under the given key.

### Parameters:

table      POINTER to a search table

WordKey    Identification of the word

WordString

Text of the word

### 10.6.17. Procedure DeleteSearchWord

```
Procedure DeleteSearchWord(table: pWord_Search_Table;  
                           WordString: Cmnd_String);
```

Abstract:

Expunges the given word from the search table.

Parameters:

table      POINTER to a search table

WordString

Text of the word to be deleted

### 10.6.18. Procedure DestroySearchTreeTable

```
Procedure DestroySearchTreeTable(var table: pWord_Search_Table);
```

Abstract:

Expunges the data for the given search table.

Parameters:

table      POINTER to the search table to be destroyed

### 10.6.19. Function UniqueWordIndex

```
Function UniqueWordIndex(table: pWord_Search_Table;  
                        ptrWordString: pWord_String;  
                        var WordText: Cmnd_String): integer;
```

Abstract:

Locates the given word in the given search table. Returns both the key and the text of the word found.

Parameters:

table      Search table in which to attempt the identification

ptrWordString

A pointer to the text of the word-prefix to be found

WordText   Text of the found word

Returns:

Returns the word-key and the entire text of the identified word if and only if the given word-prefix is in the table and is unique. Will return a WS\_NotFound if the word-prefix is not found in the table or a WS\_NotUnique if the word-prefix is not unique. In either of the latter error conditions WordText will contain the erroneous word-prefix.

### 10.6.20. Procedure ConvertStringToPool

```
Procedure ConvertStringToPool(CnvStr: Cmnd_String;
                             var ChPool: pCharacter_Pool;
                             var PoolLength: Char_Pool_Index);
```

#### Abstract:

Transforms a PERQ Pascal String into an unbounded pool of characters suitable for use by the ParseChPool routine. Is symetrical with ConvertPoolToString.

#### Parameters:

CnvStr      PERQ Pascal String to be converted

ChPool      Set to the address of the created pool

PoolLength

Number of characters in the resultant pool

#### Returns:

Sets the parameters 'ChPool' and 'PoolLength' to describe the new unbounded hunk of characters.

### 10.6.21. Function ConvertPoolToString

```
Function ConvertPoolToString(ChPool: pCharacter_Pool;
                            FirstChar: Char_Pool_Index;
                            StringLength: Char_Pool_Index):  
                           Cmnd_String;
```

#### Abstract:

Transforms an arbitrary portion of the given unbounded pool of characters into a PERQ Pascal String.

#### Parameters:

ChPool      Pointer to the unbounded hunk of characters.

**FirstChar** The beginning position (zero based) within ChPool of the desired string.

**StringLength**  
The number of characters to be moved from the pool to the string.

**Returns:**

Returns a PERQ Pascal String.

### 10.6.22. Procedure DestroyChPool

```
Procedure DestroyChPool(var ChPool: pCharacter_Pool;  
                        var PoolLength: Char_Pool_Index);
```

**Abstract:**

Deallocates storage of a given unbounded pool of characters.

**Parameters:**

**ChPool** Pointer to the pool to be released; is set to NIL  
**PoolLength**

Number of characters to be released; is zeroed

**Returns:**

Both ChPool and PoolLength are modified to describe an empty pool.

### 10.6.23. Function GetIthWordPtr

```
Function GetIthWordPtr(i: long; CmndBlock: CommandBlock):  
                    pWord_String;
```

**Abstract:**

Retrieves a pointer to the desired word from the given word list.

**Parameters:**

**i** Number (one-based) of the desired word  
**CmndBlock**

Block from which to fetch the word

**Returns:**

Returns a pointer to the requested word or NIL if the desired number is out of

bounds.

## 10.7. ExtraCmdParse

### 10.7.1. Function GetCmd

```
Function GetCmd(prompt: Cmnd_String;  
                SearchTable: pWord_Search_Table;  
                var CmdName: Cmnd_String;  
                var inF: pCommand_File_List;  
                var inputs: pCommand_Word_List;  
                var outputs: pCommand_Word_List;  
                var switches: pCommand_Word_List;  
                var ErrorGR: GeneralReturn): integer;
```

#### Abstract:

GetCmd will obtain a command input from the top file on the given file list. It will then process that command in several ways. First, it will parse the command. Next it will examine the parsed structures to determine if a command file reference was the first (and only) item in the command. If a command file is found, that file is prepended to the command file list and a null command is signalled to the caller. If no command file is found, then an attempt to locate the first input argument in the given search table is made. The results of that search are returned to the caller along with the text of the first argument, and the lists of inputs (sans first argument), outputs, and switches.

#### Parameters:

**Prompt**      Prompt string to print for the user. Do not put the prompt separator (>) on the end of the prompt; GetCmd will do that for you. If reading from a command file, GetCmd changes the prompt appropriately. Prompt is always displayed on the current default output file.

**SearchTable**

The word search in which to attempt to locate the user's first input

**CmdName**

Buffer to contain the text of the user's first input word

**inF**      Command file list from which to read the user's command and on which to nest command files (if any)

**inputs**      List to contain the user's input words

**outputs**      List to contain the user's output words

**switches** List to contain the user's switch selections

**ErrorGR** GeneralReturn code indicating status of the parse processing

**Returns:**

Returns the UniqueWordIndex of WordKey for CmdName in SearchTable or:

**Cmd\_NotFound**

First input word not found in SearchTable

**Cmd\_NotUnique**

First input word not unique

**Cmd\_EmptyCmdLine**

Command line was empty

**Cmd\_NotInsMaybeSwitches**

No inputs appeared on the line - but there may be switches

**Cmd\_SomeError**

Error was discovered (ErrorGR contains error code)

### 10.7.2. Function GetShellCmd

```
Function GetShellCmd(SearchTable: pWord_Search_Table;
                     var CmdName: Cmnd_String;
                     var inF: pCommand_File_List;
                     var inputs: pCommand_Word_List;
                     var outputs: pCommand_Word_List;
                     var switches: pCommand_Word_List;
                     var ErrorGR: GeneralReturn): integer;
```

**Abstract:**

This routine is similar to GetCmd except that it works on the command line specified to the Shell. It is should be used by programs that use GetCmd so that the Shell command line may be parsed in a similar manner. Command files are handled by GetShellCmd in a manner like GetCmd.

**Parameters:**

**SearchTable**

The word search in which to attempt to locate the user's first input

**CmdName**

Buffer to contain the text of the user's first input

**inF** Command file list on which to nest command files (if any)

**inputs** List to contain the user's input words  
**outputs** List to contain the user's output words  
**switches** List to contain the user's switch selections  
**ErrorGR** GeneralReturn code indicating status of the parse processing

**Returns:**

**Identical** to GetCmd. Viz: Returns the UniqueWordIndex of WordKey for **CmdName** in SearchTable or:

**Cmd\_NotFound**

First input word not found in SearchTable

**Cmd\_NotUnique**

First input word not unique

**Cmd\_EmptyCmdLine**

Command line was empty

**Cmd\_NotInsMaybeSwitches**

No inputs appeared on the line - but there may be switches

**Cmd\_SomeError**

Error was discovered (ErrorGR contains code)

### 10.7.3. Function GetParsedUserInput

```
Function GetParsedUserInput(prompt: Cmnd_String;
                           var inF: pCommand_File_List;
                           var inputs: pCommand_Word_List;
                           var outputs: pCommand_Word_List;
                           var switches: pCommand_Word_List);
                           GeneralReturn;
```

**Abstract:**

GetParsedUserInput will obtain a command input from the top file on the given command list. It will then parse that command and return to the caller the lists of inputs, outputs, and switches. This routine is intended for those applications in which the rules for parsing user input into words is desired but for which the application desires to perform all processing of the parsed words. (Use this if you don't want the command-file and word-identification effects of the GetCmd routine).

**Parameters:**

**Prompt**      Prompt string to print for the user. Prompt string is displayed as is; no modifications are made. Prompt is always displayed on the current default output file.

**inF**      Command file list from which to read the user's command

**inputs**      List to contain the user's input words

**outputs**      List to contain the user's output words

**switches**      List to contain the user's switch selections

**Returns:**

Returns a GeneralReturn code indicating either Success if all went well or an appropriate error code.

#### 10.7.4. Function GetConfirm

```
Function GetConfirm(prompt: Cmnd_String;
                    def: integer;
                    var switches: pCommand_Word_List): integer;
```

**Abstract:**

Handles a question that is to be answered Yes or No where the answer should come from the keyboard. Prompt followed by default (if any) is printed. Prompt may be null. If illegal input is typed, GetConfirm re-asks but doesn't use prompt.

**Parameters:**

**prompt**      Prompt to display for question. Prompt is always displayed on the current default output file.

**default**      Index of the default answer:  
                  Confirm\_YES = true or yes;  
                  Confirm\_NO = false or no;  
                  other numbers mean no default

**switches**      Set to NIL or a list of switches specified. Be sure to handle the switches first since one might be HELP.

**Returns:**

**Confirm\_YES**

If true or yes

**Confirm\_NO**

If false or no

Confirm\_Switches if naked return when no default and switches <> NIL. This means that there was no argument but a switch was entered. If an answer is still needed, the application should re-call GetConfirm.

### 10.7.5. Procedure GetCharacterPool

```
Procedure GetCharacterPool(prompt: Cmnd_String;
                           var InputFile: Text;
                           var ChPool: pCharacter_Pool;
                           var PoolLength: Char_Pool_Index);
```

#### Abstract:

GetCharacterPool should be used by those applications which wish to perform their own command input line parsing. This routine will interact with the user to obtain an unbounded raw pool of characters.

#### Parameters:

**Prompt**      Prompt string to print for the user. GetCharacterPool displays this string exactly as given; no changes are made. Prompt is always displayed on the current default output file.

**InputFile**      File from which to read the pool of chars

**ChPool**      A pointer to the pool read

**PoolLength**

Number of characters read

#### Returns:

All results returned via the parameters.

## **11. Configuration**

Module Configuration is used to provide information about the hardware configuration of the current machine. To use this module a process must have access to physical memory.

```
type
  Cf_IOBoardType = (Cf_CIO,      { Perq1 }
                     Cf_EIO);     { Perq2 }

  Cf_MonitorType = (Cf_Landscape,
                     Cf_Portrait);

  Cf_NetworkType = (Cf_CMUNet,
                     Cf_10MBitNet);
```

### **11.1. Function CF\_IOBoard**

```
Function CF_IOBoard: Cf_IOBoardType;
```

#### Abstract:

This function is used to obtain the type of the I/O board.

#### Returns:

Return Cf\_CIO if this is a PERQ I/O board, Cf\_EIO if it is a PERQ2.

### **11.2. Function Cf\_Monitor**

```
Function Cf_Monitor: Cf_MonitorType;
```

#### Abstract:

This function is used to obtain the type of display that is on the machine.

#### Returns:

Cf\_Landscape if the display is landscape, Cf\_Portrait otherwise

### 11.3. Function Cf\_OldZ80

```
Function Cf_OldZ80: boolean;
```

Abstract:

This function is used to determine the protocol that is to be used to communicate with the I/O board Z80. Accent no longer supports the "Old" Z80 protocols.

Returns:

Return True if the protocol is old Z80. This procedure should always return False.

### 11.4. Function Cf\_Network

```
Function Cf_Network: Cf_NetworkType;
```

Abstract:

This function provides information about the type of network that is in use.

Returns:

Return Cf\_10MBitNet if there is a standard Ethernet or if there is no network. If the machine is running in the Carnegie-Mellon University network environment return Cf\_CMUNet.

## 12. ControlStore

---

Module ControlStore in ControlStore.Pas exports types defining the format of PERQ micro-instructions and procedures to load and call routines in the control-store. You must have physical memory privileges to use these procedures.

```
type MicroInstruction = { The format of a micro-instruction
                           as produced by the micro-assembler. }
  packed record case integer of
    0: (Word1: integer;
        Word2: integer;
        Word3: integer);
    1: (Jmp: 0..15;
        Cnd: 0..15;
        Z: 0..255;
        SF: 0..15;
        F: 0..3;
        ALU: 0..15;
        H: 0..1;
        W: 0..1;
        B: 0..1;
        A: 0..7;
        Y: 0..255;
        X: 0..255);
    2: (JmpCnd: 0..255;
        Fill1: 0..255;
        SFF: 0..63;
        ALU0: 0..1;
        ALU1: 0..1;
        ALU23: 0..3)
  end;

  MicroBinary = { The format of a micro-instruction
                  and its address as produced by
                  the micro-assembler. }
  record
    Adrs: integer;
    MI: MicroInstruction
  end;

  TransMicro = { The format of a micro-instruction
                 as needed by the WCS QCode. }
  packed record case integer of
    0: (Word1: integer;
        Word2: integer;
        Word3: integer);
    1: (ALU23: 0..3;
```

```
ALU0: 0..1;
W:    0..1;
ALU1: 0..1;
A:    0..7;
Z:    0..255;
SFF:  0..63;
H:    0..1;
B:    0..1;
JmpCnd:0..255)
end;

MicroFile = file of MicroBinary; { A file of
                                micro-instructions. }
```

## **12.1. Procedure LoadControlStore**

```
procedure LoadControlStore( var F: MicroFile );
```

### **Abstract:**

Loads the contents of a MicroFile into the PERQ control-store. The file should be opened (with Reset) before calling LoadControlStore. It is read to EOF but not closed; thus it should be closed after calling LoadControlStore.

### **Parameters:**

F        MicroFile that contains the micro-instructions to be loaded

## **12.2. LoadMicroInstruction**

```
Procedure LoadMicroInstruction( Adrs: integer;
                               MI: MicroInstruction );
```

### **Abstract:**

Loads a single micro-instruction into the PERQ control-store.

### **Parameters:**

Adrs      Control store address to be loaded

MI        Micro-instruction to be loaded.

## 12.3. Procedure JumpControlStore

```
Procedure JumpControlStore( Adrs: integer );
```

### Abstract:

Transfers control of the PERQ micro engine to a particular address in the control-store.

Note 1: Values may not be loaded onto the expression stack before calling JumpControlStore. If you wish to pass values through the expression stack, the following code whould be used rather than calling LoadControlStore.

```
LoadExpr( Rotate(Adrs,8) );
InLineByte( JCS );
```

Note 2: Microcode called by JumpControlStore should terminate with a "NextInst(0)" microcode jump instruction if the code is in bank 0. If the microcode is in an upper bank, use "goto(NextInstLoc)." NextInstLoc is defined in Perq.High.Dfs.

### Parameters:

Adrs      Address to jump to

PERQ Systems Corporation  
Accent Operating System

Pascal Library  
Coreload

## **13. Coreload**

---

Module Coreload in Coreload.Pas contains the routines for reading Accent Lisp core files.

### **13.1. Function CoreRunLoad**

```
Function CoreRunLoad(FileName:Path_Name; P:pointer;
    filesize:long;
    Process:port; LoadDebug:boolean) :
    GeneralReturn;
```

#### **Abstract:**

Loads a core file into an existing process. The file can be given as either a filename or a pointer. If it is given as a pointer and the data is not in core format, the data is not disturbed.

#### **Parameters:**

FileName Name of a corefile to read

P Pointer to a core file which has already been read, or NIL. If P<>NIL, then FileName will be ignored.

Filesize Number of bytes in the core file pointed to by P. Ignored if p is nil.

Process Kernel port of the target process. The process is assumed to have no valid memory.

LoadDebug Ignored

#### **Returns:**

SUCCESS Core file was loaded

NotACoreFile

File was not loaded because it was not in core format

FAILURE Core file was not loaded for some other reason

#### **Side Effects:**

If P is provided, and the return is either SUCCESS or FAILURE, then the

pages pointed to by P are invalidated.

## 14. DiskUtils

---

Module DiskUtils in DiskUtils.Pas contains definitions about such things as the kinds of disks and parameters for disks. It also contains a number of procedures of use to programs wishing to call DirectIO.

### EXPORTS

```
imports accenttype      from accenttype;
imports ifiledefs       from ifiledefs;
Imports SesDisk          from SesDiskUser;

Const DISKBITS =      #300000000000;
      DIBAddress =   #300000000000; { Disk address of the DIB }
      RECORDIOBITS = #140000;        { VirtualAddress upper 16 bits
                                      of disk}
      BootDisk    = 0;             { BootDisk must be 0 }
```

### Type

```
DiskAddress = Long;           { has the disk bits set }
DiskBlockNumber = long;

intblock = record
            IntArray : array [0..PAGEWORDSIZE-1]
            of integer;
          end;

DiskBlock = packed record
            case integer of
              1: (
                  Addr : array [0..(PAGEWORDSIZE div 2)-1]
                  of Long
                  );
              2: (
                  IntData : array [0..PAGEWORDSIZE-1]
                  of integer
                  );
              3: (
                  ByteData : packed array
                  [0..PAGEBYTESIZE-1] of Bit8
                  );
              4: (
                  {4 is format of the FileInformationBlock; the FIB has Logical
                   Block -1 }
                  );
            end;
```

```
FSData      : FSDataEntry;

{The Random Index is a hint of the
DiskAddresses of the blocks that
form the file. It has three parts
as noted above. Notice that all
three parts are always there, so
that even in a very large file,
the first DIRECTSIZE blocks can
be located quickly. The blocks in
the Random index have logical
block numbers that are negative.
The logical block number of
Indirect[0] is -2 (the FIB is -1).
The last possible block's number
is -(INDSIZE+DBLINBDSIZE+1)}

Direct     : array [0..DIRECTSIZE-1]
            of DiskAddress;
Indirect   : array [0..INDSIZE-1]
            of DiskAddress;
DblInd    : array [0..DBLINDSIZE-1]
            of DiskAddress;

SegKind    : SpiceSegKind;

NumBlksInUse : integer;
              {segments can have
               gaps: block n may
               exist when block
               n-1 has never been
               allocated.
               NumBlksInUse says
               how many data
               blocks are
               actually used
               by the segment}

LastBlk    : integer;
              {Logical Block
               Number of
               largest block
               allocated}

LastAddr   : DiskAddress;
              {DiskAddress of
               LastBlk }

LastNegBlk : integer;
              {Logical Block
               Number of
               largest pointer
               block allocated}

LastNegAddr: DiskAddress;
              {Block number of
               LastNegBlk}

};

{5 is the format of a DiskInformationBlock or a
PartitionInformationBlock}
```

```

5: (
      {The Free List is a chain of free
       blocks linked by their headers }

      FreeHead    : DiskAddress;
                  {Hint of Block
                   Number of the
                   head of the
                   free list}
      FreeTail    : DiskAddress;
                  {Hint of Block
                   Number of the
                   tail of the
                   free list}
      NumFree     : DiskAddress;
                  {Hint of how many
                   blocks are on
                   the free list}
      RootDirID   : DiskAddress;
                  {where to find the
                   Root Directory}
      BadSegID    : DiskAddress;
                  {where the bad
                   segment is}

      {when booting, the boot character
       is indexed into the following
       tables to find where code to
       be boot loaded is found }

      BootTable   : array [0..25] of
                    PhysicalAddress;
                    {qcode}
      InterpTable: array [0..25] of
                    PhysicalAddress;
                    {microcode}
      PartName    : packed array [1..8]
                    of char;
      PartStart   : DiskAddress;
      PartEnd     : DiskAddress;
      SubParts    : array [0..63] of
                    DiskAddress;
      PartRoot    : DiskAddress;
      PartKind    : PartitionType;
      PartDevice  : DiskType
    );
{6 is the format of a block of a Directory}
(***)*
6: (
      Entry      : array [0..FILESPERDIRBLK-1]
      of DirEntry
    );****)
7: ( Filling : Packed Array[0..9] of integer;
      IsPartRelative : Boolean);
8: (

```

```
        intrec : intblock
      );
9: (
      Params: DiskParams
    );
10: (
      Data: Diskbuffer
    )
  end;

ptrDiskBlock = ^DiskBlock;

ptrDiskHeader = ^DiskHeader;

inthdr = record
  IntArray : array [0..7] of integer;
end;

DiskHeader = Packed Record Case Integer of
  0: (SerialNumber: DiskAddress;
       LogBlock: integer;
       Filler : Integer;
       PrevAddr: DiskAddress;
       NextAddr : DiskAddress);
  1: (intrec : inthdr)
End;

DeviceRecord = packed record
  InfoBlk: DiskAddress; {where the
                        DiskInfoBlock is}
  InUse : boolean; {this DeviceTable
                    entry is valid}
  RootPartition: PartString
                  {name of this disk}
end;

PartRecord = record
  PartHeadFree : DiskAddress; {pointer to Head
                               of Free List}
  PartTailFree : DiskAddress; {pointer to tail
                               of Free List}
  PartInfoBlk : DiskAddress; {pointer to
                            PartInfoBlock}
  PartRootDir : DiskAddress; {pointer to Root
                             Directory}
  PartNumOps : integer; {how many
                        operations
                        done since last}
```

```
                                update of
                                PartInfoBlock}
PartNumFree   : Bit32;      {HINT of how many
                           free pages}
PartInUse     : boolean;    {this entry in
                           PartTable is
                           valid}
PartMounted   : boolean;    {this partition is
                           mounted}
PartDevice    : integer;    {which disk this
                           partition is
                           in}
PartStart     : DiskAddress;
                           {Disk Address of
                           1st page}
PartEnd       : DiskAddress;
                           {Disk Address of
                           last page}
PartKind      : PartitionType;
                           {Root or Leaf}
PartName      : PartString;
                           {name of this
                           partition}
PartExUse    : boolean;    { Opened
                           exclusively }
PartPort      : Port
end;
```

## 14.1. Function AddrToBlkNum

```
function AddrToBlkNum(Addr: DiskAddress): DiskBlockNumber;
```

### Abstract:

Converts a diskaddr to a logical block number.

### Parameters:

Addr      A disk address.

### Returns:

The block num by stripping the top 4 bits and shifting.

## 14.2. Function BlkNumToAddr

```
function BlkNumToAddr(Disk : Integer; Blk : DiskBlockNumber):  
    DiskAddress;
```

## 14.3. Function MapAddr

```
Function MapAddr(LogAddr: DiskAddress; var Disk: integer):  
    DiskAddress;
```

### Abstract:

This function is used to convert a logical address into a disk specific Physical Address.

### Parameters:

LogAddr    Logical address to convert. Disk will be set to the disk index for the disk referenced by LogAddr.

### Returns:

The physical adddress.

## 14.4. Function UnMapAddr

```
Function UnMapAddr(Disk: integer; PhyAddr: DiskAddress):  
    DiskAddress;
```

### Abstract:

This function is used to convert a disk specific physical address into a logical address.

### Parameters:

Disk        Index of the disk that is to be used when calculating the logical address

PhyAddr     Physical address that is to be converted.

### Returns:

A logical address

## 14.5. Procedure InitDiskUtils

```
Procedure InitDiskUtils(Disk: integer);
```

### Abstract:

Do initialization needed to touch the disk. This procedure must be called before other procedures in this module are used.

### Parameters:

Disk      Disk number of the disk that is to be accessed

## 14.6. Procedure FinishDiskUtils

```
Procedure FinishDiskUtils(Disk: integer);
```

### Abstract:

Cleans up the DiskUtils module.

### Parameters:

Disk      Disk number of the disk to be accessed

## 14.7. Procedure GetDiskInfo

```
Procedure GetDiskInfo(UnitNumber: Integer;  
                  Var NHeads, NCyls, NSectors,  
                  Bootsize: Integer; Var DType: DiskType);
```

### Abstract:

Gets the disk parameters for the disk specified.

### Parameters (all but UnitNumber are OUT parameters):

#### UnitNumber

Unit number of the disk

NHeads    Number of heads

NCyls     Number of cylinders

NSectors   Number of sectors per track

BootSize   Number of blocks reserved for boot area

DType     Disk Type code

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**Dynamic**

## **15. Dynamic**

Module Dynamic in Dynamic.Pas implements Pascal dynamic allocation - New and Dispose. It also provides routines CreateHeap, ResetHeap, and DestroyHeap to make, empty, and get rid of heaps. In general, calls to these routines should only be generated by the compiler.

### **15.1. Procedure InitDynamic**

Procedure InitDynamic;

**Abstract:**

Initializes Pascal dynamic memory allocation. Must be called only once and before any call to any other routine in this module.

### **15.2. Function CreateHeap**

Function CreateHeap : HeapNumber

**Abstract:**

Validates memory for a heap and initializes its free list.

**Returns:**

TooManyHeaps

If the process is already using MaxHeaps number of heaps

### **15.3. Procedure ResetHeap**

Procedure ResetHeap(S : HeapNumber)

**Abstract:**

Returns all storage in heap S to free list.

**Parameters:**

S              HeapNumber of the heap that is to be initialized

## 15.4. Procedure DestroyHeap

```
Procedure DestroyHeap(S : HeapNumber);
```

### Abstract:

Releases storage for heap S.

### Parameters:

S            HeapNumber of heap to destroy

## 15.5. Procedure DisposeP

```
Procedure DisposeP(
    var Where      : pointer;
    Len            : integer);
```

### Abstract:

Deallocates memory.

### Parameters:

Where        Pointer to the record to release

Len           Length in words

## 15.6. Procedure NewP

```
Procedure NewP(
    S            : HeapNumber;
    A            : integer;
    var Where    : pointer;
    L            : integer );
```

### Abstract:

Allocates memory.

### Parameters:

S            Number of heap in which to allocate. 0 means the default data heap.

A            Alignment of node in words relative to beginning of segment.  
Must be a power of 2. 0 represents  $2^{16}$ , the maximum value.

Where       Set to point to the memory that was allocated. If the data segment is full and cannot be increased, P is set to nil.

L            Length in words. 0 represents  $2^{16}$ , the maximum. Negative

values are interpreted as (L+2<sup>16</sup>).

## 15.7. Procedure CheckHeap

Procedure CheckHeap(S: HeapNumber);

### Abstract:

Raises BadHeap if heap is broken.



## **16. EnvMgrDefs / EnvMgrUser**

---

Modules EnvMgrDefs in EnvMgrDefs.Pas and EnvMgr in EnvMgrUser.Pas contain the definitions and routines used by the environment manager calls. The definitions and routines are described in the document "The Environment Manager" in the *Accent Programming Manual*.

PERQ Systems Corporation  
Accent Operating System

Pascal Library  
Except

## **17. Except**

### **Abstract:**

Module Except in Except.Pas provides the following things:

1. Definitions of the microcode generated exceptions;
2. A procedure to tell the microcode which segment number these exceptions are defined in;
3. The default handler of all exceptions. The compiler enables this handler in every main program;
4. A Pascal routine to search the stack in when an exception is raised.

```
exception Abort( Message: String );
exception Dump( Message: String );
exception DivZero;    { division by zero }
exception MulOvfl;   { overflow in multiplication }
exception StrIndx;   { string index out of range }
exception StrLong;   { string to be assigned is too long }
exception InxCase;   { array index or case expression out
                      of range }
exception UndfQcd;   { execution of an undefined Q-code }
exception UndfInt;   { undefined device interrupt detected }
exception MParity;   { memory parity error }
exception EStack;    { E-stack wasn't empty at INCDDS }
exception Ovflli;    { Overflow in conversion to integer
                      from Long Integer }
exception NormMsg;   { normal IPC msg pending }
exception EmergMsg;  { emergency IPC msg pending }
exception UndReal;   { floating point underflow }
exception OvrReal;   { floating point overflow }
exception RtoiOvfl;   { floating point truncate error }
exception RealDivZero; { floating point divide by zero }
exception NextOp;    { NextOp across a page boundary }
exception BadExit;   { EXITT or EXGO falling off stack }
```

## 17.1. Procedure InitExceptions

```
Procedure InitExceptions;
```

### Abstract:

InitExceptions tells the microcode what segment number to use when raising its own exceptions. The segment number is the one that the system assigns to this module.

## 17.2. Procedure RaiseP

```
Procedure RaiseP( ES, ER, PStart, PEnd: Integer );
```

### Abstract:

RaiseP is called to raise an exception. The compiler generates a call to RaiseP in response to

```
raise SomeException  
( original parameters )
```

in the following way:

```
Push original parameters onto  
the MStack.  
RAISE SegmentNumber(SomeException)  
RoutineNumber (SomeException)  
parametersize
```

The microcode calls RaiseP in the following way:

```
Push parameters onto the MStack  
if appropriate.  
Parametersize := WordsOfParameters.  
Error := ErrorNumber. Goto(CallRaise).
```

where CallRaise does the following:

```
SaveTP := TP.  
Push ExcSeg onto the MStack.  
Push Error onto the MStack.  
Push SaveTP-Parametersize+1  
onto the MStack.  
Push SaveTP+1 onto the MStack.  
call RaiseP.
```

### Parameters:

ER      Routine number of the exception to be raised

- ES** Segment number of the exception to be raised
- PStart** Pointer to the original parameters (as an offset from the base of the stack)
- PEnd** Pointer to the first word after the original parameters (as an offset from the base of the stack)



## **18. IFileDefs**

Module IFileDefs in IFileDefs.Pas contains the internal definitions for the file system. See the document "The File System" in the *Accent Programming Manual*.



## **19. IODefs / IOUser**

Modules IODefs in IODefs.Pas and IO in IOUser.Pas contain the definitions and routines for the IO system. See the document "The IO System" in the *Accent Programming Manual*.

**PERQ Systems Corporation**  
**Accent Operating System**

Pascal Library  
IPCRecordIO

## **20. IPCRecordIO**

---

Module IPCRecordIO in IPCRecordIO.Pas contains routines for sending and receiving simple Pascal records.

### **20.1. Function SendRecord**

```
Function SendRecord(
    localport      : Port;
    remoteport     : Port;
    id             : long;
    MsgType        : long;
    recptr         : Pointer;
    recsize        : integer)
:GeneralReturn;
```

#### **Abstract:**

Sends a Pascal record (which does not contain a port reference) to a port.

#### **Parameters:**

**localport** A port in the current process (usually used for a reply to the sent message, but may be nothing)

**remoteport**

Port to send the message to

**id** ID to use for the message (a 32 bit number)

**msgtype** NORMALMSG or EMERGENCYMSG

**recptr** Pointer to a record to send

**recsize** Type size IN BITS of the record

#### **Returns:**

The return of the send operation

## 20.2. Function RecRecord

```
Function RecRecord(
    var localport    : Port;
    var remoteport   : Port;
    var id           : long;
    var MsgType      : long;
    var recptr       : Pointer;
    var recsize      : integer)
: GeneralReturn;
```

### Abstract:

Receives a Pascal record from a port.

### Parameters:

**localport** A port in the current process on which the message was received

**remoteport** Reply port, if any

**id** ID of the message (a 32 bit number)

**msgtype** NORMALMSG or EMERGENCYMSG

**recptr** Pointer to the received record

**recsize** Type size IN BITS of the record

### Returns:

Same as for Receive plus BadMsgType.

## **21. KeyTran / KeyTranDefs**

---

Modules KeyTran in KeyTran.Pas and KeyTranDefs in KeyTranDefs.Pas are used for keytranslation table manipulations. The routines are described in the document "The Window Manager" in the *Accent Programming Manual*.



## **22. ModGetEvent**

---

Module ModGetEvent in ModGetEvent.Pas is used to allow a client process to receive asynchronous events from more than one window. The code is derived from SapphUser.

### **22.1. Procedure GetEventPort**

```
Procedure GetEventPort(
  ServPort    : Window;
  howWait     : KeyHowWait;
  retPort     : Port
);
```

### **22.2. Function ExtractEvent**

```
Function ExtractEvent(
  repMsg      : Pointer;
  var w        : ViewPort;
  var e        : EventRec
): Boolean;
```



## **23. MsgNUser**

---

Module MsgNUser in MsgNUser.Pas provides routines to manipulate the network-wide name / port mapping. The name server is explained in the document "The Name Server" in the *Accent Programming Manual*.



## **24. NameErrors**

---

Module NameErrors in NameErrors.Pas exports the MsgServer (NameServer) errors. For more information, see the document "The Name Server" in the *Accent Programming Manual*.



## **25. Net10MBDefs / Net10MBUser**

Modules Net10MBDefs in Net10MBDefs.Pas and Net10MBUser in Net10MBUser.Pas, as well as NSTypes in NSTypes.Pas, are used with the network server. See the document "The Network Server" in the *Accent Programming Manual*.

PERQ Systems Corporation  
Accent Operating System

Pascal Library  
NoEchoInput

## **26. NoEchoInput**

---

Module NoEchoInput in NoEchoInput.Pas provides routines to obtain input from the user window without echoing to the screen. It is used to read passwords. It assumes that stream IO is through the standard Input and Output files and that they go through UserWindow.

`Function NoEchoRead: PString;`

### **Abstract:**

This function reads a string without displaying it. It processes BACKSPACE, OOPS, CTRL-H, CTRL-U, and RETURN. It ignores characters longer than the maximum length of a string.

### **Returns:**

The string read in

**PERQ Systems Corporation**  
**Accent Operating System**

Pascal Library  
NSTypes

## **27. NSTypes**

Module NSTypes in NSTypes.Pas defines the message format used by the Net Server. This is the message received on the port given to the Net Server for a particular filter, containing a pointer to the text of the received packet.

The routines in this module, as well as those in Net10MBDefs in Net10MBDefs.Pas and Net10MBUser in Net10MBUser.Pas, are explained in the document "The Network Server" in the *Accent Programming Manual*.



## **28. OldTimeStamp**

---

Module OldTimeStamp in OldTimeStamp.Pas is a compatibility module for converting between new time values and POS timestamps.

### **28.1. Exported Types**

```
type TimeStamp = packed record
  { the fields in this record are ordered this way to }
  { optimize bits}
  Hour: 0..23;
  Day: 1..31;
  Second: 0..59;
  Minute: 0..59;
  Month: 1..12;
  Year: 0..63; {year since 1980}
end;
```

### **28.2. Function OldCurrentTime**

```
Function OldCurrentTime: TimeStamp;
```

**Abstract:**

Gets the current time as an old style time stamp.

**Returns:**

TimeStamp for time, in system time zone

### **28.3. Function NewToOldTime**

```
Function NewToOldTime(NewTime: Internal_Time):TimeStamp;
```

**Abstract:**

Converts a new internal time record to an old time stamp.

**Returns:**

Time stamp representing NewTime, in system time zone

## 28.4. Function OldToNewTime

```
Function OldToNewTime(OldTime: TimeStamp): Internal_Time;
```

### Abstract:

Converts an old time stamp to a new internal time record.

### Returns:

Internal time representing OldTime

## 29. PascalInit

Module PascalInit in PascalInit.Pas is used to complete the creation of a new process. It contains the first routine invoked in a process before the main program is called. It initializes the server environments and state variables for a process.

```
TYPE
  (---- Definitions shared with spawn ----)

{See the comment in InitPascal on how to modify this message}
{InitMsgType --> Definition of InitialMessage. Sent to a }
{  spawned process }

  InitMsgType = RECORD
    Head : Msg;
    DefaultType : TypeType;
    DefInName : STRING[ 255];
    DefOutName : STRING[ 255];
    PassPortType : TypeType;
    PassedPorts : PtrPortArray;
    UWinSharedType : TypeType;
    UWinShared : Boolean;
    EMTYPE : TypeType;
    EMPort : Port;
    WindowType : TypeType;
    UserWindow : Port;
    TSType : TypeType;
    UserTS : Port;
    WordCountType : TypeType;
    WordCount : long;
    WordDirIndexType : TypeType;
    WordDirIndex : long;
    WordArrayPtrType : TypeType
      {long form}
    WordArrayPtrEltSize : integer;
    WordArrayPtrTName : integer;
    WordArray_Cnt : long;
    WordArrayPtr : pCharacter_Pool;
    ComputeEnvType : TypeType;
    ComputingEnvironment: Long;
  END;

  CONST
    InitMsgSize = WORDSIZE( InitMsgType ) * 2;
```

## 29.1. Procedure InitPascal

### Abstract:

This is the first routine to run in an empty address space. It is called implicitly by ALoad and should not ever be called by a user program.

## 29.2. Procedure InitProcess

```
Procedure InitProcess(AmIClone: BOOLEAN);
```

### Abstract:

Is used to provide initialization for Pascal processes. This procedure should NEVER be called by a user process.

## 29.3. Function DisablePrivils

```
Function DisablePrivils(Proc: PORT): GeneralReturn;
```

### Abstract:

Places the process in User mode.

Supervisor mode allows access to physical memory and I/O device registers, while User mode does not. This procedure will only work if Proc is KERNELPORT (i.e., you are modifying your own privileges), or if the process to be affected is suspended. It sets bits in a processor register.

### Parameters:

Proc      Kernel port of the process to affect

### Returns:

Success    Process is in User mode

Failure    Privileges not affected

## 29.4. Function EnablePrivs

```
Function EnablePrivs(Proc: PORT): GeneralReturn;
```

### Abstract:

Places the process in Supervisor mode. Supervisor mode allows access to physical memory and I/O device registers, while User mode does not. This procedure will only work if Proc is KERNELPORT (i.e., you are modifying your own privileges), or if the process to be affected is suspended. It clears bits in a processor register.

### Parameters:

Proc        Kernel port of the process to affect

### Returns:

Success     The process is in Supervisor mode

Failure     Privileges not affected

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**PasLong**

## **30. PasLong**

Module PasLong in PasLong.Pas is the module for the Stream package, used to process 32-bit long quantities. Its routines are called by code generated by the Pascal compiler in response to variations on Read, Readln, Write and Writeln statements. It is one level above Module Stream and uses Stream's lower-level input routines.

### **30.1. Procedure ReadD**

```
procedure ReadD( var F: FileType; var X: long;  
                 B: integer );
```

#### Abstract:

Reads an double integer in free format with base B. B may be any integer between 2 and 36, inclusive.

Number is read into the low order word of two double precision integers to avoid overflow.

#### Parameters:

**X**      The double to be read

**F**      The file from which X is to be read

**B**      The base of X. It may be any integer between 2 and 36, inclusive. If B is less than zero and the user does not type an explicit plus or minus sign, X is read as an unsigned number.

#### Returns:

**PastEof**   An attempt was made to read F past the Eof

**NotNumber** Non-numeric input was encountered in the file

**LargeNumber**

Number is not in the range -2^31..2^32-1

**BadBase**   Base is not in the range 2..36

## 30.2. Procedure WriteD

```
Procedure WriteD( var F: FileType; X: long;  
                  Field, B: integer );
```

### Abstract:

Writes an double integer in fixed format with base B. Value written from two double precision words to avoid overflow.

### Parameters:

- X Double to be written.
- F File into which X is to be written.
- Field Size of the field into which X is to be written.
- B Base of X. It is an integer whose absolute value must be between 2 and 36, inclusive. If B is less than zero, X is written as an unsigned number.

### Returns:

- BadBase Base is not in the range 2..36

## **31. PasReal**

---

Module PasReal in PasReal.Pas is the module for the Stream package used to process real numbers. Its routines are called by code generated by the Pascal compiler in response to variations on Read, Readln, Write and Writeln statements. It is one level above Module Stream and uses Stream's lower-level input routines.

### **31.1. Procedure ReadR**

```
Procedure ReadR(var F: FileType;  
                 var value: real);
```

#### **Abstract:**

This procedure reads a real number from the file F and returns its value.

#### **Parameters:**

F        File from which to read

value      A return parameter returning the value of the real number read from the file F.

#### **Side Effects:**

This procedure reads characters from the external file F, until it receives a character which cannot be part of the real number, and it leaves this character in the file buffer.

There has been only minimal care taken with the file buffer, so if an exception is raised during the read, there is no guarantee about its contents.

#### **Exceptions:**

PastEof      Raised if an attempt is made to read beyond the end of file

NotReal      Raised if the stream of characters in file F does not correspond to a real number

SmallReal      Raised if the number read is too small to be represented by a 32-bit IEEE-Standard number

**LargeReal** Raised if the number read is too large to be represented by a 32-bit IEEE-Standard real number

## 31.2. Procedure WriteR

```
procedure WriteR(var F: FileType;  
                  e: real;  
                  TotalWidth: integer;  
                  FracDigits: integer;  
                  format: integer);
```

### Abstract:

This procedure writes a real number to a file F, under the given format specifications.

### Parameters:

F        File to which to write

e        Value to write

TotalWidth Minimum number of characters to write

FracDigits Number of characters in the fractional part (used for fixed format only).

format    Indicates either fixed or floating format

## **32. PathName**

---

The routines in Module PathName in PathName.Pas provide a non-primitive interface to the file server. These routines make use of both environment manager functions and name server functions. All pathnames should be handled by routines at this level.

The routines in PathName are covered in the document "The File System" in the *Accent Programming Manual*.

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**PMatch**

## **33. PMatch**

Module PMatch in PMatch.Pas provides facilities that are used to do pattern matching on strings. Valid pattern matching characters are \*, which matches zero or more characters, and ?, which matches exactly one character.

### **33.1. Procedure PattDebug**

```
Procedure PattDebug(v : boolean)
```

**Abstract:**

Sets the global debug flag.

**Parameters:**

v        Value to set debug to

**Returns:**

Changes debug value.

### **33.2. Function IsPattern**

```
Function IsPattern(
      var str : pms255)
      : boolean;
```

**Abstract:**

Tests to see whether str contains any pattern matching characters.

**Parameters:**

str        String to test

**Returns:**

true        If str contains any pattern matching characters; false otherwise

### 33.3. Function PattMatch

```
Function PattMatch(
    var str, pattern : pms255;
        fold          : boolean)
    :boolean;
```

#### Abstract:

Compares str against pattern.

#### Parameters:

str        Full string to compare against pattern  
pattern    Pattern to compare against. It can have special characters in it.  
fold       If true, upper and lower cases match; otherwise they are distinct.

#### Returns:

true       If string matches pattern; false otherwise

### 33.4. Function PattMap

```
Function PattMap(
    var str,inpatt,outpatt,outstr : pms255;
        fold                  : boolean)
    :boolean;
```

#### Abstract:

Compares str against inpatt. Puts the parts of str that match inpatt into the corresponding places in outpatt and returns the result.

#### Parameters:

str        Full string to compare against pattern  
inpatt     Pattern to compare against. It can have special characters in it.  
outpatt    Pattern to put the parts of str into; it must have the same special characters in the same order as in inpatt.  
outstr     The resulting string if PattMap returns true  
fold       If true, upper and lower cases match; otherwise they are distinct.

#### Returns:

true       if string matches pattern; false otherwise

**Errors:**

Raises BadPatterns exception if outpatt and inpatt do not have the same patterns in the same order.

### 33.5. Function NextCh

```
Function NextCh(var s: pms255;  
                 var i: integer; var sp: boolean);  
                 char;
```

**Abstract:**

Returns the next character from s starting at i; decodes special characters, quoted characters, and case folds if necessary.

**Parameters:**

s        String to look at  
i        Index of where to start looking  
sp      Returns true if got a pattern matching character, else false

**Returns:**

The character found

### 33.6. Function UpCh

```
Function UpCh(var s: pms255; i: integer): char;
```

**Abstract:**

Returns uppercase of s[i].

**Parameters:**

s        String to look at  
i        Index of char wanted

**Returns:**

If i > length(s) then space; else if caseFold then UpperCase of the character; else the character itself

### 33.7. Function PattCheck

```
Function PattCheck(var p1,p2: pms255): boolean;
```

Abstract:

Checks to see that two strings have the same pattern match characters in the same order.

Parameters:

p1 and p2 Patterns to compare

Returns:

True if have the same patterns in the same places; false otherwise

## **34. ProcMgrDefs / ProcMgrUser**

---

Modules ProcMgrDefs in ProcMgrDefs.Pas and ProcMgrUser in ProcMgrUser.Pas contain the type definitions and routines for the process manager. They are described in the document "The Process Manager" in the *Accent Programming Manual*.



## 35. QMapDefs

Module QMapDefs in QMapDefs.Pas contains the constants and types used to map from Q-code to source code.

```
Exports
{$IFC UnderAccent THEN}
imports SesameDefs    from SesameDefs;
{$ELSEC}
imports FileSystem from FileSystem;
{$ENDC}

Const
  Max_QmapRoutines = 251;

type
  QMapDict = Record
{$ifc UnderAccent then}
  CompID: Internal_Time;
  SourceFileBlock: Integer;
  Routines: array[0..Max_QmapRoutines] of integer;
{$elsec}
  CompID: TimeStamp;
  SourceFileBlock: Integer;
  Routines: array[0..252] of integer;
{$endc}
  End;

  QMapInfoRecord =
  packed record
    QCodeNumber: integer;           {a procedure relative offset
                                     of the first qcode for
                                     this statement}
    SourceLineNumber: integer;      {count of the number of carriage
                                     returns in text file before this
                                     statement}
    BlockNumber: integer;          {block offset in text file of
                                     start of statement}
    Offset: 0..511;                {byte offset in that block of the
                                     last character of the first id
                                     of the statement}
    SourceFileNumber: 0..127;       {offset in the SourceMap of the
                                     appropriate source file}
  end;

{$ifc 256 mod WordSize(QMapInfoRecord) <> 0 then}
{$Message ***** QMapInfoRecord is not evenly divisible *****}
{$endc}

  SourceMapRecord =
```

```
record
{$ifc UnderAccent then}
  FileName: APath_Name;
  CompID: Internal_Time;
  Filler: array[0..124] of integer;
{$elsec}
  FileName: PathName;
  fileBlocks, fileBits: Integer; { this may eventually be replaced
                                  by CompID: TimeStamp;
  }
{$endc}
end;

const
  WSQMapInfoRecord = WordSize(QMapInfoRecord);
  QMap_entries_per_block = 256 div WSQMapInfoRecord;

  WSSourceMapRecord = WordSize(SourceMapRecord);
  Source_entries_per_block = 256 div WSSourceMapRecord;

type
  QCodeMap = Array[0..QMap_entries_per_block - 1] of
    QMapInfoRecord;
  SourceMap = Array[0..Source_entries_per_block - 1] of
    SourceMapRecord;
  QMap_Buffer = array [0..PageWordSize-1] of integer;
  QMap_ByteBuffer = array [0..PageByteSize-1] of bit8;

  pQMap = ^QCodeMap;
  pSourceMap = ^SourceMap;
  pQMap_Buffer = ^QMap_Buffer;
  pQMap_ByteBuffer = ^QMap_ByteBuffer;
  pQMapDict = ^QMapDict;

  QMap_ptr_type =
    record case integer of
{$ifc UnderAccent then}
      0: (P: POINTER);
      1: (Buffer: pQMap_Buffer);
      2: (ByteBuffer: pQMap_ByteBuffer);
{$elsec}
      2: (p: pDirBlk);
{$endc}
      2: (pDict: pQMapDict); {blocks 0 .. 1 of file}
      3: (pMap: pQMap); {blocks 2 .. N-1}
      4: (pSource: pSourceMap); {blocks N .. EOF}
    end;
```

## **36. RealFunctions**

Module RealFunctions in RealFunctions.Pas implements many of the standard functions whose domain and / or range is the set of real numbers. The implementation of these functions was guided by the book *Software Manual for the Elementary Functions*, by William J. Cody, Jr., and William Waite, (C) 1980, Prentice-Hall, Inc.

The domain (inputs) and range (outputs) of the functions are given in their abstract. The following notation is used.

Parentheses ( ) are used for open intervals (those that do not include the endpoints), and brackets [ ] are used for closed intervals (those that do include their endpoints). The closed interval [RealMLargest, RealPLargest] is used to mean all real numbers, and the closed interval [-32768,32767] is used to mean all integer numbers.

### **36.1. Function Sqrt**

```
Function Sqrt( X: Real ): Real;
```

#### Abstract:

Computes the square-root of a number. Domain = [0.0,RealPLargest]. Range = [0.0,Sqrt(RealPLargest)].

#### Returns:

Square-root of X

## 36.2. Function Ln

```
Function Ln( X : Real ): Real;
```

Abstract:

Computes the natural log of a number. Domain = [0.0,RealPLargest]. Range = [RealMLargest,Ln(RealPLargest)].

Returns:

Natural log of X

## 36.3. Function Log10

```
Function Log10( X : Real ): Real;
```

Abstract:

Computes the log to the base 10 of a number. Domain = [0.0,RealPLargest]. Range = [RealMLargest,Log10(RealPLargest)].

Returns:

Log to the base 10 of X

## 36.4. Function Exp

```
Function Exp( X : Real ): Real;
```

Abstract:

Computes the exponential function. Domain = [-85.0,87.0]. Range = (0.0, RealPLargest].

## 36.5. Function Power

```
Function Power( X, Y : Real ): Real;
```

Abstract:

Computes the result of an arbitrary number raised to an arbitrary power. DomainX = [0.0,RealPLargest]. DomainY = [RealMLargest, RealPLargest]. Range = [0.0, RealPLargest]. Restrictions: 1) if X is zero, Y must be greater than zero; 2) X raised to the Y is a representable real number.

Returns:

e              Raised to the X power

## 36.6. Function PowerI

```
Function PowerI(
      X      : Real;
      Y      : Integer )
      : Real;
```

### Abstract:

Computes the result of an arbitrary number raised to an arbitrary integer power. The difference between Power and PowerI is that negative values of X may be passed to PowerI. DomainX = [RealMLargest, RealPLargest]. DomainY = [-32768, 32767]. Range = [RealMLargest, RealPLargest]. With the restrictions that 1) if X is zero, Y must be non-zero; 2) X raised to the Y is a representable real number.

### Returns:

X              Raised to the Y power

## 36.7. Function Sin

```
Function Sin( X : Real ): Real;
```

### Abstract:

Computes the sin of a number. Domain = [-1E5,1E5]. Range = [-1.0, 1.0].

### Returns:

Sin of X

## 36.8. Function Cos

```
Function Cos( X : Real ): Real;
```

### Abstract:

Computes the cosin of a number. Domain = [-1E5,1E5]. Range = [-1.0, 1.0].

### Returns:

Cos of X

## 36.9. Function Tan

```
Function Tan( X: Real ): Real;
```

Abstract:

Computes the tangent of a number. Domain = [-6433.0, 6433.0]. Range = [RealMinfinity, RealPInfinity].

Returns:

Tangent of X

## 36.10. Function CoTan

```
Function CoTan( X : Real ): Real;
```

Abstract:

Computes the cotangent of a number. Domain = [-6433.0, 6433.0]. Range = [RealMinfinity, RealPInfinity].

Returns:

Cotangent of X

## 36.11. Function ArcSin

```
Function ArcSin( X : Real ): Real;
```

Abstract:

Computes the arcsin of a number. Domain = [-1.0, 1.0]. Range = [-Pi / 2, Pi / 2). It seems that the Domain and Range should be closed intervals; however this implementation apparently returns a number very close to zero when X is 1.0, rather than returning Pi / 2 as it should.

Returns:

Arcsin of X

## 36.12. Function ArcCos

```
Function ArcCos( X : Real ): Real;
```

### Abstract:

Computes the arccosine of a number. Domain = (-1.0, 1.0]. Range = (-Pi/2, Pi/2]. It seems that the Domain and Range should be closed intervals; however, this implementation apparently returns a number very close to zero when X is -1.0, rather than returning -Pi/2 as it should.

### Returns:

Arccosin of X

## 36.13. Function Arctan

```
Function Arctan( X : Real ): Real;
```

### Abstract:

Computes the arctangent of a number. Domain = [RealMLargest, RealPLargest]. Range = (-Pi/2, Pi/2). Works fine except for very large numbers.

### Returns:

Arctangent of X

## 36.14. Function ArcTan2

```
Function ArcTan2( Y, X : Real ): Real;
```

### Abstract:

Computes the arctangent of the quotient of two numbers. Seems fine except for very large Y/X. One interpretation is that the parameters represent the cartesian coordinate (X,Y) and ArcTan2(Y,X) is the angle formed by (X,Y), (0,0), and (1,0). DomainY = [RealMLargest, RealPLargest]. DomainX = [RealMLargest, RealPLargest]. Range = [-Pi,Pi].

### Returns:

Arctangent of Y/X

### 36.15. Function SinH

```
Function SinH( x:real ) : real;
```

Abstract:

Computes the hyperbolic sin of a number.

Returns:

Hyperbolic sin of x

### 36.16. Function CosH

```
Function Cosh( x:real ) : real;
```

Abstract:

Computes the hyperbolic cosin of a number.

Returns:

Hyperbolic cosin of x

### 36.17. Function TanH

```
function TanH( x:real ) : real;
```

Abstract:

Computes the tangent of a number.

Returns:

Hyperbolic tangent of x

## **37. RunDefs**

Module RunDefs in RunDefs.Pas defines the form of Pascal Run files.

```
CONST
  ModNameLength = 19;
  RMAXFILE      = 255;
  RMAXIMPORT    = 4095;
  MAINKLUDGE    = 255;

  { To make library files work. See note below. }
  RFDELAYEDMAIN = 255;
  RMAXSEG       = RFDELAYEDMAIN - 1;

  RFFORMAT      = 5; { Run File Format Version Number }

  RUNHEADLOC   = PageWordSize; { MinUser }

TYPE
  ModString     = String[ModNameLength];

  SegIndex      = 0..RFDELAYEDMAIN;
  FileIndex     = 0..RMAXFILE;
  ImpIndex      = 0..RMAXIMPORT;

  ImpArray = packed array [0..4096] of SegIndex;
  pImpArray = ^ImpArray;

  LinkFileType = (SegFile, RunFile, DataFile,
                  SymfsFileType, QMapFileType);

  FileEntry =
    record
      FileType:     LinkFileType;
                    { if Data, Location is offset in .RUN }
      FileLocation: long;
                    { address if mapped process, 0 if not }
      FileBlocks:   long;
                    { length in blocks }
      FileName:     APath_Name;
                    { full path name }
      WriteDate:    Internal_Time;
                    { Write date at time of link }
      Version:      long;
                    { version number (run files only) }
    end;
  FEarray = array [ FileIndex ] of FileEntry;
  pFEarray = ^FEarray;
```

```
SegEntry =
record
  ModuleName: ModString;
  GDBSize: long; { size of Global Data Block }
  GDBLocation: long; { can be assigned by
                      the loader}

  SegFile: FileIndex; { index of file containing
                       code }
  SegHdrOffset: long; { offset of .SEG hdr in that
                       file }
  SegHdrSize: long; { size of .SEG header info }

  CodeOffset: long; { offset of code in the file }
  CodeSize: long; { size of code (incl. RDict) }

  ImportsOffset: long; { offset of imports list }
  ImportsSize: long; { length of Imports info }

  ProcNamesOffset: long; { offset of procedure names }
  ProcNamesSize: long; { length of proc names entries}

  SymsFile: FileIndex; { index of file containing
                        .SYMS info }
  SymsOffset: long; { offset of .SYMS info }
  SymsSize: long; { length of .SYMS info }

  QMapFile: FileIndex; { index of file containing
                        .QMAP info }
  QMapOffset: long; { offset of .QMAP info }
  QMapSize: long; { length of QMAP info }

  FirstImport: ImpIndex; { place in ImpArray where
                         imports start }
  NumImports: integer; { number of files imported }
end;
SEarray = array [ SegIndex ] of SegEntry;
pSEarray = ^SEarray;

RunHead =
record
  RFormat: integer;

  InitSeg: SegIndex;
    { index of Pascal initializer }
  MainSeg: SegIndex;
    { index of user Main program }

  LinkVersion: string[20];
  LinkLocation: string[20];
  LinkDate: Internal_Time; { when created }
  LinkCommand: string;

  Version: long; { version of this file }
```

```
    SegEntryOffset: long;      { offset of SEarray in
                                .RUN file }
    NumSegs:           SegIndex; { number of SegEntries
                                0..NS-1 }
    ImpIndexOffset:   long;     { offset of ImpArray in
                                .RUN file }
    NumImports:        ImpIndex; { number of entries in
                                ImpArray 0..NI-1 }
    FileEntryOffset:  long;     { offset of FEarray in
                                .RUN file }
    NumFiles:          FileIndex; { number of FileEntries
                                0..NF-1 }
  end;
pRunHead = ^RunHead;
```

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**SaltError**

## **38. SaltError**

**Module** SaltError in SaltError.Pas is the standard system error module. It provides a number of facilities for generating and printing error messages.

**There are three classes of errors:**

```
type GR_Error_Type = (GR.Warning, GR.Error,  
                      GR.FatalError);
```

1. Warnings are only informative. Warnings are preceded by a single "\*\*\*".
2. Errors indicate that there was a true error in the processing. They are preceded by a "\*\*\*". It may be possible to recover from an Error. The recovery is left to the application program. Fatal Errors are really bad. Like Errors, they are preceded by a "\*\*\*". There is no recovery from a fatal error.

### **38.1. Procedure GRWriteStdError**

```
Procedure GRWriteStdError(GR :GeneralReturn;  
                         ER_Type :GR_Error_Type;  
                         InMsg :PString);
```

#### **Abstract:**

This procedure is used to print an error message given a general return value. It is assumed that this routine will be used when an error occurs as a direct result of a user action.

#### **Parameters:**

**Gr** General return that is to be translated to a text message

**Er\_Type** Class of error

**InMsg** An optional message that will be printed

## 38.2. Procedure GRStdError

```
Procedure GRStdError(GR      :GeneralReturn;
                      ER_Type   :GR_Error_Type;
                      InMsg     :PString;
                      var OutMsg  :PString);
```

### Abstract:

This procedure is used to format an error message string given a general return value. It is assumed that this routine will be used when an error occurs as a direct result of a user action.

### Parameters:

Gr        General return that is to be translated to a text message  
Er\_Type    Class of error  
InMsg     An optional message that will be printed  
OutMsg    Will be set to contain the formatted error message

## 38.3. Procedure GRWriteErrorMsg

```
Procedure GRWriteErrorMsg(GR    :GeneralReturn;
                           ER_Type :GR_Error_Type;
                           ProgName: String;
                           InMsg   :PString);
```

### Abstract:

This procedure is used to print an error message given a general return value. It is assumed that this routine will be used for errors that may not be a direct result of a user action. Because the error may not have been caused by a direct user action a file is provided that can be used to provide Program / Module information.

### Parameters:

Gr        General return that is to be translated to a text message  
Er\_Type    Class of error  
ProgName  Used to provide information about where in the system, program, and module the error occurred  
InMsg     An optional message that will be printed

### 38.4. Procedure GSErrorMsg

```
Procedure GSErrorMsg(GR      :GeneralReturn;
                      ER_Type   :GR_Error_Type;
                      ProgName  :String;
                      InMsg     :PString;
                      var OutMsg :PString);
```

#### Abstract:

This procedure is used to format an error message string given a general return value. It is assumed that this routine will be used for errors that may not be a direct result of a user action. Because the error may not have been caused by a direct user action a field is provided that can be used to provide Program / Module information.

#### Parameters:

Gr        General return that is to be translated to a text message

Er\_Type   Class of error

ProgName Used to provide information about where in the system, program, and module the error occurred

InMsg      An optional message that will be printed

OutMsg     Will be set to contain the formatted error message

### 38.5. Procedure ErrorMsgPMBroadcast

```
Procedure ErrorMsgPMBroadcast
                      (GR      :GeneralReturn;
                       ER_Type   :GR_Error_Type;
                       ProgName  :String;
                       InMsg     :PString);
```

#### Abstract:

This procedure is used by processes that do not have access to a window in which to display error information. System Servers fall into this class. The error message will be printed in the process manager's window.

#### Parameters:

Gr        General return that is to be translated to a text message

Er\_Type   Class of error

ProgName Used to provide information about where in the system, program,

and module the error occurred

InMsg An optional message that will be printed

## 38.6. Function GRStdErr

```
function GRStdErr(GR      :GeneralReturn; { The return code to
                                         translate   }
                  ER_Type  :GR_Error_Type; { Warning, Error or
                                         Fatal Error  }
                  InMsg    :PString;        { A filename, etc.
                                         to display   }
                  var OutMsg  :PString):boolean;
                                         { The translated
                                         error msg   }
```

### Abstract:

Takes the GR value, finds the message, and returns it in OutMsg.

### Returns:

Returns TRUE if a message has been found; otherwise FALSE.

If warning, preceeds message with one \*.

If Error or FatalError, preceeds it with two \*\*.

If FatalError, exits program- you probably don't want to do this since you'll lose outmsg.

## **39. Sapphire-Related Modules**

---

The following modules are used in connection with the window manager (which is referred to in the modules by its internal name, Sapphire):

- SaphEmrExceptions in SaphEmrExceptions.Pas - Contains the exceptions raised by the window manager (server side);
- SaphEmrServer in SaphEmrServer.Pas - Contains a procedure that programs should call to have the Exception raised;
- SapphDefs in SapphDefs.Pas - Contains the exported type definitions. This includes the exports for both the window manager layer and the viewport layer. This type module is imported by both the server and the user side of the interface.
- SapphFileDefs in SapphFileDefs.Pas - Contains the type definitions for viewports, available to applications;
- SapphLoadFile in SapphLoadFile.Pas - Contains routines for obtaining information to pass on to the window manager, such as cursor data files, font data files, and Picture files;
- SapphUser in SapphUser.Pas - Contains the routines used by applications;
- ViewPt in ViewPtUser.Pas - Contains the routines used by applications in connection with viewports;

- KeyTran in KeyTran.Pas and KeyTranDefs in KeyTranDefs.Pas - Contain the routines used for keytranslation table manipulations.

The definitions and routines are described in the document "The Window Manager" in the *Accent Programming Manual*.

Also see WindowUtils in this document for routines that manipulate the window associated with the current process.

## 40. SegDefs

Module SegDefs in SegDefs.Pas defines the constants and types used in the .SEG file output of the PERQ Pascal compiler.

```
*****  
{* .SEG file format  
{* as generated by Perq Pascal compiler:  
{*  
{* Block 0:           SegBlock case 0  
{* Block 1:           SegBlock case 1. code  
{* Block 2..n:        code  
{* Block ImportBlock: array of CImpInfo, array of ProcNames  
{*  
{*}  
  
const  
  QCodeVersion = 5;          { Current QCode Version Number }  
  SegLength    = 8;          { max chars in a segment name }  
  CommentLen   = 80;         { the length of comment and version  
                           str in seg}  
  FileLength   = 100;  
  
type  
  SNAArray = packed array[1..SegLength] of Char;  { segment name }  
  
  FNString = String[FileLength];  { file name }  
  
  QVerRange = 0..255;    { range of QCode version numbers }  
  
  Language = (Pascal, Fortran, Imp);  
  
  SegBlock = packed  
  record  
    case integer of          { .SEG file definition }  
      { zeroth (header) block: }  
    0:  
    {  
      ProgramSegment: boolean;  
      LongIds       : boolean;  
      DbgInfoExists : boolean;  
      OptimizedCode : boolean;  
      ThirtyChIds  : boolean;  
      SegBlkFiller  : 0..7;  
      QVersion     : QVerRange;  
      ModuleName   : SNAArray;  
      FileName     : FNString;  
      NumSeg       : integer;  
      ImportBlock  : integer;  
      GDBSSize     : integer;
```

```
Version      : String[CommentLen];
Comment      : String[CommentLen];
Source       : Language;
PreLinkBlock : integer;
RoutDescBlock: integer;
DiagBlock    : integer;
QMapFileName: FNString;
SymFileName  : FNString;
CompId       : TimeStamp
);

{ first block: }
l:
(
  OffsetRD    : integer;
  RoutsThisSeg: integer
);

2:
(
  Block: array[0..255] of integer
)
end;
pSegBlock = ^SegBlock;

CImpInfo =
record
  case boolean of { Import List Info - as generated
                    by compiler }
    true:
    (
      ModuleName: SNAArray; { module identifier }
      FileName:  FNString { file name }
    );
    false:
    (
      Ary: array [0..0] of integer
    )
  end;
pImpInfo = ^CImpInfo;

SegFileType = file of SegBlock;
```

## **41. Sesame-Related Modules**

Modules SesameDefs in SesameDefs.Pas and Sesame in SesameUser.Pas contain the common definitions and routines used by the file system. Modules SesDiskDefs and SesDisk in files SesDiskDefs and SesDiskUser contain definitions and routines used in disk mounting and certain other activities. (In the modules the file system is referred to by its internal name, Sesame.)

The definitions and routines are described in the document "The File System" in the *Accent Programming Manual*.

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**Spawn**

## 42. Spawn

Module Spawn in Spawn.Pas creates a new process and notifies the Process Manager of its existence. An initialization message containing state and system server ports is sent to be received in the new process (by InitProcess in PascalInit). Spawn is used to handle the general case of process creation while Exec and Split are implemented by calling Spawn with specific arguments.

### **42.1. Function Exec**

```
Function Exec(
    VAR ChildKPort      : Port;
    VAR ChildDPort      : Port;
    ProcessName        : APath_Name;
    HisCommand         : CommandBlock;
    : GeneralReturn;
```

#### Abstract:

Creates a new process running a new program (this is the most simple way). The new process will share the caller's window and typescript and copy the caller's Environment Manager connection. The process will be registered with the Process Manager under its run file name (i.e., ProcessName). It maintains the privilege mode of the parent (Supervisor / User). It won't startup in the debugger.

#### Parameters:

ChildKPort

Set to the new process's kernel port

ChildDPort

Set to the new process's data port

ProcessName

Name of the file to execute and the name that will be registered with the Process Manager

**HisCommand**

Used to set the new process's UserCommand

**Returns:**

Success    Process was created and loaded

Failure    No new process was created

## 42.2. Function Split

```
Function Split(
    var ChildKPort : Port;
    var ChildDPort : Port)
: GeneralReturn;
```

**Abstract:**

Creates a copy of the current process. This is the most simple way to "fork" and have all the standard initialization done. The caller's window and typescript will be shared. The caller's Environment Manager state will be duplicated. It maintains the privilege mode the parent (Supervisor/User).

**Parameters:**

ChildKPort Set to the new process's kernel port

ChildDPort Set to the new process's data port

**Returns:**

IsParent    The original process

IsChild    The copy

## 42.3. Function Spawn

```
Function Spawn(
    var ChildKPort : Port;
    var ChildDPort : Port;
    ProgName : APath_Name;
    ProcName : string;
    HisCommand : CommandBlock;
    DebugIt : boolean;
    ProtectChild : boolean;
    SapphConn : ConnectionInheritance;
    pWindow : Port;
    pTypeScript : Port;
    EMConn : ConnectionInheritance;
    pEMPort : Port;
    PassedPorts : ptrPortArray;
```

```
    NPorts      : long;
    LoaderDebug : boolean)
: GeneralReturn;
```

**Abstract:**

This is a general case of process creation, which can exec as well as fork.

The Spice kernel calls to create processes, Fork and CreateProcess, do not pass state or ports to the new process. Spawn gets around this restriction by passing a message to the new process containing this information. InitProcess receives this message in the new process.

Basically, Spawn just passes the system ports except if SapphConn or EMConn is "NewOne." In this case, it must create a new Sapphire and Environment connection. The Process Manager is notified (PMRegisterProcess) of the new process, its window, environment, and ProcName. Supervisor mode privileges may be given up. At present, these are the rights to access physical memory and I/O device registers. The new process will be resumed or left for debugging (PMMakeDebugProcess) depending on the debugit parameter.

**NOTE:**

If the ProgName string is not empty, Spawn performs the Exec function by creating a new process and Aloading into it.

If you don't specify "Newone" as the SapphConn parameter, you will have to do a GetKeyTab on the window yourself.

**Parameters**

ChildKPort

Will be set to the Child's KernelPort in the Parent

ChildDPort

Will be set to the Child's DataPort in the Parent

**ProgName**

Name of the .RUN file that you want loaded and executed. If this is null, a fork is done and the new process continues to run the existing program.

**ProcName**

Name of the child process as it will be registered with the Process Manager. This name will appear in the Process Manager window in response to the SYSTAT command. Usually the name is the same as ProgName.

**HisCommand**

Used to set the child's UserControl

**DebugIt** If true, the new process is suspended and the debugger is invoked on it before it gets control.

**ProtectChild**

If true, the child process will be in User mode; if false, the child will inherit the parent's rights.

**SapphConn**

If "Given" the child's window and typescript will be taken from the pWindow and pTypeScript arguments. They will be shared (i.e., WindowShared = true) with these ports. "GivenReg" has the semantics above and ownership rights for these two ports are passed to the child. If "NewOne" Sapphire will be asked to create a new window and TypeScript will be initialized to this window. These new ports are passed with ownership rights.

**pWindow** A window to share with the new process

**TypeScript**

A TypeScript to share with the new process

**EMConn** Controls access to the environment manager. If "Given" then the connection will be taken from "pEMPort." "GivenReg" is the same as "Given" except that ownership rights to the port are passed. If "Newone" then a new Environment Manager connection will be made, copying the state of the current connection.

**pEMPort** Parent's Environment Manager port

**PassedPorts** An array of ports to pass to the child. The array should contain

only ports that the parent wants to pass to the child. The system service ports are passed automatically. The former ports will be available in the InPorts array, with the same indexes as in PassedPorts.

**LoaderDebug**

Turns on Spawn and loader debugging

**Returns:**

Success      A new process was exec'ed

IsParent      Returned in the parent image of the fork

IsChild      Returned in the child process image of the fork

Failure      Process exec or spawn failed



## **43. SpawnInitFlags**

---

Module SpawnInitFlags in SpawnInitFlags.Pas differentiates between starting processes from the boot file and starting all other processes after that. There is a version of this file used by 'system' and a version used by 'pascalinit'. This module affects 'pascalinit,' 'spawn,' and 'typescript.' This module is normally used by builders of the kernel and the Pascal library.

**PERQ Systems Corporation**  
**Accent Operating System**

**Pascal Library**  
**Spice\_String**

## **44. Spice String**

---

Module Spice\_String in Spice\_String.Pas implements string hacking routines for PERQ Pascal. It is a complete replacement for PERQ\_String and Sail\_String.

Strings in PERQ Pascal are stored a single character per byte with the byte indexed by 0 being the length of the string.

### **44.1. Procedure Adjust**

```
Procedure Adjust(
      var Str    : PString;
      Len     : Integer);
```

#### Abstract:

Used to change the dynamic length of a string.

#### Parameters:

Str        String that is to have the length changed

Len        New length of the string. Must be no greater than MaxPStringSize (which is 255).

#### Returns:

Does not return a value.

### **44.2. Procedure AppendChar**

```
Procedure AppendChar(
      var Str    : PString;
      c        : Char);
```

#### Abstract:

Puts c on the end of Str.

#### Parameters:

Str        String to place character C on the end of

c            Character to place at the end of string Str

### 44.3. Procedure AppendString

```
Procedure AppendString(
    var Str1 : PString;
    Str2 : PString);
```

Abstract:

Puts Str2 on the end of Str1.

Parameters:

Str1        String to place string Str2 on the end of  
Str2        String to place at the end of string Str1

### 44.4. Function Cat3

```
Function Cat3(
    Str1,Str2,Str3 : PString)
    : PString;
```

Abstract:

Concatenates three strings together. Uses ConCat to combine the arguments into a temporary.

Parameters:

Str1, Str2, Str3  
                Strings to concatenate

Returns:

Returns a single string composed of the parameters.

### 44.5. Function Cat4

```
Function Cat4(
    Str1,Str2,Str3,Str4 : PString)
    : PString;
```

Abstract:

Concatenates four strings together. Uses ConCat to combine the arguments into a temporary.

Parameters:

Str1, Str2, Str3, Str4  
strings to concatenate

**Returns:**

Returns a single string composed of the parameters.

#### **44.6. Function Cat5**

```
Function Cat5(
    Str1,Str2,Str3,Str4,Str5 : PString)
    : PString;
```

**Abstract:**

Concatenates five strings together. Uses ConCat to combine the arguments into a temporary.

**Parameters:**

Str1, Str2, Str3, Str4, Str5  
Strings to concatenate

**Returns:**

Returns a single string composed of the parameters.

#### **44.7. Function Cat6**

```
Function Cat6(
    Str1,Str2,Str3,Str4,Str5,Str6 : PString)
    : PString;
```

**Abstract:**

Concatenates six strings together. Uses ConCat to combine the arguments into a temporary.

**Parameters:**

Str1, Str2, Str3, Str4, Str5, Str6  
Strings to concatenate

**Returns:**

Returns a single string composed of the parameters.

## 44.8. Function Concat

```
Function Concat(
    Str1,Str2      : PString)
    : PString;
```

### Abstract:

Concatenates two strings together. Uses MVBW (Move bytes Q-code) to copy Str2 onto the end of Str1.

### Parameters:

Str1, Str2 The two strings that are to be concatenated

### Returns:

Returns a single string as described by the parameters.

## 44.9. Procedure ConvUpper

```
procedure ConvUpper(
    var Str      : PString);
```

### Abstract:

Converts Str to all upper case. Uses character compares to test whether character is lower case.

### Parameters:

Str String to be converted

## 44.10. Function CVD

```
Function CVD(
    Str      : PString)
    : integer;
```

### Abstract:

Converts a decimal string to an integer. Conversion stops at the first character which is not legal in the decimal radix. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored.

### Parameters:

Str String to be converted

### Returns:

An integer containing the value

#### 44.11. Function CVH

```
Function CVH(  
           Str          :PString)  
           :integer;
```

##### Abstract:

Converts a hexadecimal string to an integer. Conversion stops at the first character which is not legal in the hexadecimal radix. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. Lower case "a".."f" are the same as upper case "A".."F".

##### Parameters:

Str        String to be converted

##### Returns:

An integer containing the value

#### 44.12. Function CVHS

```
Function CVHS(  
           I      :integer)  
           :PString;
```

##### Abstract:

Converts an integer to a string using a hexadecimal radix.

##### Parameters:

I        Integer to be converted

##### Returns:

A string containing the character representation; the string will represent the value expressed in hexadecimal.

## 44.13. Function CVHSS

```
Function CVHSS(
    I      :integer;
    W      :integer)
    :Pstring;
```

### Abstract:

Converts an integer to a string of width W. The return string is padded on the left with spaces if necessary to fill out the width; the radix will be hexadecimal.

### Parameters:

I        Integer to be converted  
W        Minimum field width to be produced

### Returns:

A string containing the character representation; the string will be of at least width W and be filled on the left with spaces and the conversion will be done in hexadecimal radix.

## 44.14. Function CvInt

```
Function CvInt(
    Str      : PString;
    R        : integer)
    : integer;
```

### Abstract:

Converts a string to an integer according to base R (radix). Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. A sign is permitted. Lower case "a".."z" are the same as upper case "A".."Z". Errors: Overflow is possible but not checked for.

### Parameters:

Str       String to be converted  
R        Radix to use

### Returns:

A integer containing the value

#### 44.15. Function CvL

```
Function CvL(
    Str      : PString;
    Radix   : integer)
: long;
```

##### Abstract:

Converts a string to a long integer according to base Radix. Conversion stops at the first character which is not legal in the radix used. Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored. A sign is permitted. Lower case "a".."z" are the same as upper case "A".."Z". Errors: Overflow is possible but not checked for.

##### Parameters:

Str        String to be converted  
Radix      Radix to use

##### Returns:

A long integer containing the value

#### 44.16. Function CVN

```
Function CVN(
    I      : integer;
    W      : integer;
    B      : integer;
    Fill  : Pstring)
:Pstring;
```

##### Abstract:

Exactly same as CVLS, but for integers and B corresponds to Radix.

## 44.17. Function CVLS

```
Function CVLS(
    I      : long;
    W      : integer;
    Radix : integer;
    Fill   : Pstring)
    :Pstring;
```

### Abstract:

Converts a long integer to a string of width W, padding on the left with "Fill" if necessary to fill out the width. The base for the conversion is Radix. If Radix>10, the letters "A".."Z" will be used to compute the character for the representation. Using a base > 36 will produce bogus results. A negative base will force an unsigned result.

### Parameters:

I        Long integer to be converted  
W        Minimum field width to be produced  
Radix    Base to use (2..36)  
Fill     A one-character string to fill on the left

### Returns:

A string containing the character representation; the string will be of at least width W, be filled on the left with Fill and converted according to radix Radix.

## 44.18. Function CVO

```
Function CVO(
    Str     :PString)
    :integer;
```

### Abstract:

Converts an octal string to an integer. Conversion stops at the first character which is not legal in the octal radix Characters whose ordinal value is less than or equal to 32 (a space) which precede the value are ignored.

### Parameters:

Str     String to be converted

### Returns:

An integer containing the value

## 44.19. Function CVOS

```
Function CVOS(  
    I      :integer)  
    :Pstring;
```

### Abstract:

Converts an integer to a string using octal radix.

### Parameters:

I        Integer to be converted

### Returns:

A string containing the character representation; the string will represent the value expressed in octal.

## 44.20. Function CVOSS

```
Function CVOSS(  
    I      :integer;  
    W      :integer)  
    :Pstring;
```

### Abstract:

Converts an integer to a string of width W, padding on the left with spaces if necessary to fill out the width. Radix will be octal.

### Parameters:

I        Integer to be converted

W        Minimum field width to be produced

### Returns:

A string containing the character representation. The string will be of at least width W and be filled on the left with spaces, and the conversion will be done in octal radix.

## 44.21. Function CVS

```
Function CVS(
    I      :integer)
    :Pstring;
```

### Abstract:

Converts an integer to a string using decimal radix.

### Parameters:

I        Integer to be converted

### Returns:

A string containing the character representation; the string will represent the value expressed in decimal.

## 44.22. Function CVSS

```
Function CVSS(
    I      :integer;
    W      :integer)
    :Pstring;
```

### Abstract:

Converts an integer to a string of width W, padding on the left with spaces if necessary to fill out the width. Radix will be decimal.

### Parameters:

I        Integer to be converted

W        Minimum field width to be produced

### Returns:

A string containing the character representation. The string will be of at least width W and be filled on the left with spaces, and the conversion will be done in decimal radix.

## 44.23. Function CvUp

```
Function CvUp(
    Str      :PString)
    :PString;
```

### Abstract:

Returns a copy of Str with lower case replaced by upper. Uses character compares to test whether character is lower case.

### Parameters:

Str      String to be converted

### Returns:

A copy of Str with all alphabetic characters converted to upper case.

## 44.24. Procedure DeleteChars

```
procedure DeleteChars(
    var Str      :PString;
    Index, Size  :Integer);
```

### Abstract:

Removes characters from a string. Will change Str; uses MVBW (Move Bytes Q-Code) to copy Str back onto itself.

### Parameters:

Str      String that is to be changed. Characters will be removed from this string.

Index     Starting position for the delete

Size     Number of characters that are to be removed. Size characters will be removed from Str starting at index.

### Returns:

Does not return a value.

## 44.25. Function GetBreak

```
Function GetBreak
    : BreakTable;
```

### Abstract:

Allocates and clears a break table.

### Returns:

A new, empty break table

## 44.26. Function Initial

```
Function Initial(
    Str1,Str2      : Pstring)
    :boolean;
```

### Abstract:

Returns true if Str2 is an initial string of Str1. The comparison is case-sensitive. A null string is an initial substring of any string. Uses EQUBYT QCode to test if the first Length (Str2) chars are equivalent.

### Parameters:

Str1        String to be tested

Str2        String which is the initial substring to test for

### Returns:

true        If Str2 is an initial substring of Str1

## 44.27. Procedure InsertChars

```
procedure InsertChars(
    Source      :Pstring;
    var Dest     :Pstring;
    Index       :Integer);
```

### Abstract:

Inserts a string into the middle of another string. Will insert Source in Dest starting at location Index.

### Parameters:

Source      String that is to be inserted

Dest      String into which the insertion is to be made  
Index     Starting position in Dest for insertion

Returns:

Does not return a value.

## 44.28. Function Lop

```
Function Lop(
    var Str           : PString)
    : PString;
```

Abstract:

Removes the first character from Str and returns it as the value of the function. Str no longer contains the character. Modifies Str, removes the first character. Uses MVBW to copy the string back onto itself.

Parameters:

str      String from which a character will be lopped off

Returns:

Returns the first character of Str.

## 44.29. Function Pad

```
Function Pad(
    Str       : PString;
    TotalLen : integer;
    PadCh    : char;
    Where    : integer)
    : PString;
```

Abstract:

Adds padding characters to a string. Produces a copy of Str adjusted to have length TotalLen by inserting sufficient copies of PadCh just after location Where.

Parameters:

Str      Original string

TotalLen   Desired length for result string

PadCh    Character to insert to achieve desired length

Where      Where to insert characters. Use 0 for padding on left and Strlntf (or length(Str)) for padding on the right. If some other value, padding will be done just after the character in that position.

Returns:

A string of length TotalLen consisting of a copy of Str with sufficient copies of PadCh inserted just after position Where.

#### 44.30. Function PosC

```
Function PosC(
    Str      :Pstring;
    C        :char)
    :integer;
```

Abstract:

Finds the position of C in Str. Returns 0 if absent. If supported, use BSCAN QCode.

Parameters:

Str      String that is to be searched  
C      Character we are looking for

Returns:

If C occurs in Str then the index into Str of the first character matching C will be returned. If C was not found (even if Str is empty) then return 0.

#### 44.31. Function PosString

```
Function PosString(
    Source, Mask      :Pstring)
    :integer;
```

Abstract:

Finds the position of Mask in the substring of S. Scans for first character of mask. When found, uses EQUBYT (byte string compare Q-code) to test rest of Mask. The Source is temporarily modified during the search.

Parameters:

Source      String that is to be searched  
Mask      Pattern that we are looking for

Returns:

If Mask occurred in Source, then the index into (the original) Source of the first character matching the Mask will be returned. If Mask was not found then return 0. If Mask is empty, return 1.

#### 44.32. Procedure ReplaceChars

```
procedure ReplaceChars(
    var Str      :Pstring;
    NewS     :Pstring;
    Index    :integer);
```

Abstract:

Replaces a substring of Str with another string. Use MVBW QCode to copy NewS over Str.

Parameters:

Str      String into which the replacement is to be made

Index     Starting position in Str for NewS. If it is greater than Length (Str), no replacement will be made. If it is less than zero, start at one.

NewS     String that is to replace the deleted segment

Returns:

Returns a string of the same length with the appropriate replacement.

#### 44.33. Function RevPosC

```
Function RevPosC(
    Str      :PString;
    c        :char)
: integer;
```

Abstract:

Tests if c is a member of Str.

Parameters:

Str      String to test for C member of

c        Any character

Returns:

Index of C in Str (from end of string) or zero if not there

#### 44.34. Function RevPosString

```
Function RevPosString(  
    Source, Mask      :Pstring;  
    :integer;
```

##### Abstract:

Finds the position of Mask in the substring of Source. Scans for first character of Mask. When found, uses EQUBYT (byte string compare Qcode) to test rest of Mask. The Source is temporarily modified during the search.

##### Parameters:

Source      String that is to be searched

Mask        Pattern that we are looking for

##### Returns:

If Mask occurred in Source, then the index (from the end) into (the original) Source of the first character matching the Mask will be returned. If Mask was not found then return 0. If Mask is " " return 1.

#### 44.35. Function Scan

```
Function Scan(  
    var S          : Pstring;  
    BT           : BreakTable;  
    var BRK       : Pstring)  
    :Pstring;
```

##### Abstract:

Scans the string S according to the breakable specifications of BT.

##### Parameters:

S           String to be scanned

BT          Breaktable initialized by SetBreak

BRK        Break character

##### Returns:

The initial substring determined by the breaktable is removed from Sd returned as the value of the function. The BRK variable contains the string (character)

which caused the scan to stop, or the null string if the string was exhausted.

## 44.36. Procedure SetBreak

```
procedure SetBreak(  
    var BT           : BreakTable;  
    Break, Omit     : PString;  
    Options        : BreakKind);
```

### Abstract:

Initializes a breaktable according to the specifications of Break, Omit, and Options.

### Parameters:

BT	Breaktable to be initialized
Break	Set of characters (as a string) on which a scanning break will occur
Omit	Set of characters which will be removed from the string
Options	Allows specification of one option from each of the following groups:

#### Group A

Inclusive	the Break set is the set of characters on which a break will occur.
Exclusive	the Break set is the set of characters on which a break will not occur.

If no option is specified from the group, "Inclusive" is assumed.

#### Group B

Skip	Upon return, the break character will be in the break variable. The result of the scan will be all characters up to the break character, and the input string is modified to start immediately after the break character.
Append	Upon return, the break character will be in the break variable. The result of the scan will be all characters up to and including the break character, and the input string is modified to start immediately after the break character.
Retain	Upon return, the break character will be in the break variable. The result will be all characters up to the

break character, and the input string is modified to start at the break character.

If no option is specified from this group, "Skip" is assumed.

Group C

**FoldUp** Before anything else is done, each character which is alphabetic is folded to uppercase. Note that break sets are case sensitive, but this is done before the break test. This folding proceeds until the break condition is reached.

**FoldDown** Similar to FoldUp, except uppercase alphabetic characters are made lowercase.

If no option is specified from this group, no case folding is done.

NOTE - No guarantees about behavior are made if more than one option is selected from each set group.

#### 44.37. Function ShowBreak

```
Function ShowBreak(
    BT           : BreakTable
    : PString;
```

Abstract:

Creates a string representation of the BreakTable for debugging.

Parameters:

BT        Breakable initialized by SetBreak

Returns:

An informative string of the breaktable specifications

#### 44.38. Function Squeeze

```
Function Squeeze(
    Str          : PString)
    : PString;
```

Abstract:

Removes all spaces and tabs from a string.

Parameters:

Str        String to be squeezed

Returns:

A string which has all spaces and tabs removed

#### 44.39. Function Str

```
Function Str(
      Ch              :char)
      :PString;
```

Abstract:

Coerces a character to a string.

Parameters:

Ch        Character to be coerced to a string

Returns:

A string value for a one-character string containing the character

#### 44.40. Function Strip

```
Function Strip(
      Str              :PString)
      :PString;
```

Abstract:

Converts sequences of spaces, tabs, CR, and LF to a single space.

Parameters:

Str        String to be squeezed

Returns:

A string which has all sequences of spaces, tabs, CR, and LF changed to a single space

#### 44.41. Function SubStrFor

```
Function SubStrFor(
    Source      :PString;
    Index, Size :Integer)
    :PString;
```

##### Abstract:

Returns a sub-porton of the string passed as a parameter.

##### Parameters:

Source      String that we are to take a portion of  
Index      Starting position in Source of the substring  
Size      Size of the substring that we are to take

##### Returns:

Returns a substring as described by the parameter list. If Index+Size exceed the dynamic length of the string, return Index to DynamicLength. No error message is generated.

#### 44.42. Function SubStrTo

```
Function SubStrTo(
    Source      :PString;
    Index, EndIndex :Integer)
    :PString;
```

##### Abstract:

Returns a sub-porton of the string passed as a parameter.

##### Parameters:

Source      String that we are to take a portion of  
Index      Starting position in Source of the substring  
EndIndex   Ending position in the source of the substring

##### Returns:

Returns a substring as described by the parameter list. If Index or EndIndex exceed the dynamic length of the string, return Index to DynamicLength. No error message is generated.

### 44.43. Function Trim

```
Function Trim(
    Str          : PString)
: PString;
```

Abstract:

Deletes leading and trailing spaces and tabs from a string.

Parameters:

Str        String to be squeezed

Returns:

String which has all leading and trailing spaces and tabs removed

### 44.44. Function ULInitial

```
Function ULInitial(
    Str1,Str2          : PString)
: boolean;
```

Abstract:

Returns true if Str2 is an initial string of Str1.

Parameters:

Str1        String to be tested

Str2        String which is the initial substring to test for

Returns:

Returns true if Str2 is an initial string of Str1. The comparison is case-insensitive. A null string is an initial substring of any string.

### 44.45. Function ULPosString

```
Function ULPosString(
    Source, Mask          : PString)
: Integer;
```

Abstract:

Finds the position of a pattern in a given string without case sensitivity.

Parameters:

Source      String that is to be searched

**Mask** String to search for

**Returns:**

If Mask occurred in Source, then the index into Source of the first character of Mask will be returned. If Mask was not found, then return 0.

#### 44.46. Function UpChar

```
Function UpChar(  
    C      :Char)  
    :char;
```

**Abstract:**

Converts C to upper case. Uses character compares to test whether character is lower case.

**Parameters:**

C Character to be converted

**Returns:**

Returns the upper-case character value of the character C.

#### 44.47. Function UpEQU

```
Function UpEQU(  
    Str1     : PString;  
    Str2     : PString)  
    :boolean;
```

**Abstract:**

Compares two strings for case-independent equality.

**Parameters:**

Str1, Str2 Strings to be compared

**Returns:**

True if the strings are equal; false if they are not. Independent of case.

## **45. Stream**

---

Module Stream in Stream.Pas implements the low-level Pascal I/O. It is not intended for use directly by user programs, but rather the compiler generates calls to these routines when a Reset, Rewrite, Get, or Put is encountered. Higher-level character I/O functions (Read and Write) are implemented by the two modules Reader and Writer.

In this module, the term "file buffer variable" refers to F up-arrow for a file variable F. Files: stream.pas

### **45.1. Procedure StreamInit**

```
procedure StreamInit( var F: FileType;  
                      WordSize, BitSize: integer;  
                      CharFile: boolean );
```

#### Abstract:

Initializes but does not open the file variable F. Automatically called upon entry to the block in which the file is declared.

#### Parameters:

F        File variable to be initialized

WordSize and BitSize

Size of an element of the file

CharFile    Determines whether or not the file is of characters

## 45.2. Procedure StreamClose

```
procedure StreamClose(  
    var F           : FileType );
```

### Abstract:

Closes the file variable F.

### Parameters:

F        File variable to be closed

## 45.3. Procedure StreamOpen

```
procedure StreamOpen(  
    var F           : FileType;  
    var Name        : SName;  
    WordSize, BitSize : integer;  
    CharFile       : boolean;  
    OpenWrite      : boolean );
```

### Abstract:

Opens the file variable F. This procedure corresponds to both Reset and Rewrite.

### Parameters:

F        File variable to be opened

Name      Filename

WordSize   Number of words in an element of the file (0 indicates a packed file)

BitSize    Number of bits in an element of the file (for packed files)

CharFile   True if the file is a character file

OpenWrite

True if the file is to be opened for writing (otherwise it is opened for reading)

## 45.4. Procedure GetB

```
procedure GetB(  
    var F           : Filetype);
```

### Abstract:

Advances to the next element of a block-structured file and gets it into the file buffer variable.

### Parameters:

F            File to be advanced

## 45.5. Procedure GetC

```
procedure GetC(  
    var F           : Filetype);
```

### Abstract:

Gets the next character of a file. Advances to the next element of a character structured file and gets it into the file buffer variable.

### Parameters:

NotOpen    If F is not open

NotReset   If F has not been reset

PastEof     If an attempt is made to read F past Eof

TimeOutError

    If RS: or RSX: times out

UnitIOError If IOCRread doesn't return IOEIOC or IOEIOB

UndfDevice

    If F is open, but the device number is bad

## 45.6. Procedure PutB

```
procedure PutB(
    var F      : Filetype );
```

### Abstract:

Writes the value of the file buffer variable to the block-structured file and advances the file.

### Parameters:

F        File to be advanced

## 45.7. Procedure PutC

```
procedure PutC(
    var F      : FileType );
```

### Abstract:

Puts the next character in a file. Writes the value of the file buffer variable to the character-structured file and advances the file.

### Parameters:

F        File to be advanced

## 45.8. Procedure PReadIn

```
procedure PReadIn(
    var F      : Filetype );
```

### Abstract:

Advances to the first character following an end-of-line.

### Parameters:

F        File to be advanced

## 45.9. Procedure PWriteln

```
procedure PWriteln(
    var F           : Filetype );
```

### Abstract:

Writes an end-of-line.

### Parameters:

F        File to which an end-of-line is written

## 45.10. Procedure KBFlushBoardOutput

```
procedure KBFlushBoardOutput(
    var F           : Filetype );
```

### Parameters:

F        File to be flushed

## 45.11. Procedure StreamKeyBoardReset

```
procedure StreamKeyBoardReset(
    var F           : Text );
```

### Abstract:

Clears the keyboard input buffer and the file variable F so that all input typed up to this point will be ignored.

### Parameters:

F        File to be cleared

## 45.12. Procedure InitStream

```
procedure InitStream;
```

### Abstract:

Initializes the stream package; called by System.

### 45.13. Function FullLn

```
Function FullLn(
      var F           : Text )
      : Boolean;
```

#### Abstract:

Determines if there is a full line in the keyboard input buffer. This is the case if a carriage return has been typed. This function is provided in order that a program may continue to do other things while waiting for keyboard input. If the file is not open to the console, FullLn is always true.

#### Parameters:

F        File to be checked

#### Returns:

True if a full line has been typed

### 45.14. Function StreamName

```
Function StreamName(
      var F           : FileType )
      : SName;
```

#### Abstract:

Returns the file name associated with the file variable F.

#### Parameters:

F        File whose name is to be returned

### 45.15. Procedure WriteNChars

```
procedure WriteNChars(
      var F           : FileType;
      c               : char;
      N               : Integer);
```

#### Abstract:

Writes some number of characters to the file.

#### Parameters:

F        File to write to

c        Character to duplicate

N            Number of characters to write

### 45.16. Procedure WriteChars

```
procedure WriteChars(
    var F : FileType;
    var S : String);
```

#### Abstract:

Writes some number of spaces to the file.

#### Parameters:

F            File to write to  
S            String to write

### 45.17. Function IsStreamDevice

```
Function IsStreamDevice(
    S : SName)
    : integer;
```

#### Abstract:

Indicates whether a name is one of the special devices that the Stream package uses.

#### Parameters:

S            String containing name to check

### 45.18. Procedure StreamFlushOutput

```
procedure StreamFlushOutput(var F: Text);
```

#### Abstract:

Flushes all queued Typescript output. Use this to make sure that all Typescript output is actually written before getting input from other than Typescript Input.

#### Parameters:

F            File to be flushed



## 46. SymDefs

Module SymDefs in SymDefs.Pas is the definitions module for the Symbol table file produced by the Pascal compiler. This module is used by the debugger and the compiler.

```
EXPORTS
{$IFC not UnderAccent THEN}
Imports FileSystem from PFileSys;
{$ENDC}
Imports TimeDefs    from TimeDefs;

Type  BaseType =
      (T_Unknown, T_Char, T_Boolean, T_Integer, T_Enumerated,
       T_Real, T_Long, T_Pointer, T_Var, T_Routine, T_File,
       T_String, T_Set, T_Record, T_Array, T_Misc);

{Note: Subranges use the base type. T_Var is for
 Var parameters to routines. T_Routine is used for
 Routine parameters to routines. T_Unknown is used
 when the compiler is unable to provide the
 information for some reason or if there is no
 information to supply.}

{Note: T_Misc in conjunction with the Etc field
 specifies symbols which are either in the registers
 or are compiler generated temporaries.}

Const   { special values for Etc when mainType is T_Misc }
      MiscEtc_CompilerTemp = 0;
             { compiler generated temporary }
      MiscEtc_Register_16Bit = 1;
             { 16-bit, integer, which resides in a register }
      MiscEtc_Register_32Bit = 2;
             { 32-bit, long, which resides in a register }

Const
      Max_SymRoutines = 251;

Type  VarDictEntry = Record  {this is block zero of the file}
      CompID: Internal_Time;
      Globals: Integer;
             {the globals of the module or program}
      Routines: Array[0..Max_SymRoutines] of Integer;
End;

{$ifc WordSize(VarDictEntry) < 256 then}
{$Message *****VarDictEntry must be 256 words long *****}
```

```
{$endc}

pVarDictEntry = ^VarDictEntry;

Type VarNameArray = Record
    TotNumChars: integer;
    VarNames: Packed Array[0..0] of Char;
End;

{The VarNameArray type is actually never used to
access the variable character information. It is
included here for documentation purposes only.}

VarDescriptor = Packed Record
    mainType: BaseType; {the type of the variable}

    subType: BaseType; {if Array, Pointer, File
                        or Set, this field specifies
                        the type of thing aggregated
                        over}

    etc: 0..255;         {if mainType = record or set
                        then is number of words
                        necessary to store mainType.
                        else if mainType = string,
                        then is number of characters
                        in the string. else if
                        mainType = array then is size
                        of SubType (the component of
                        the array) else 1. Except:
                        If mainType is a packed array
                        then 0. If the etc field would
                        be filled with a number > 255
                        then 0. Note etc has special
                        values when mainType is
                        T_Misc.}

End;

Type Var_CharAr = Packed Array[0..511] of Char;
pVar_CharAr = ^Var_CharAr;

Var_IntAr = Array[0..255] of Integer;
pVar_IntAr = ^Var_IntAr;

Type var_ptr = Record Case Integer Of
{$ifc UnderAccent then}
    0: (P: POINTER);
{$elsec}
    1: (p: pDirBlk);
{$endc}
    2: (dict: pVarDictEntry);
    3: (chars: pVar_CharAr);

    4: (int: pVar_IntAr);
End;
```

## **47. TimeDefs / TimeUser**

Modules TimeDefs in TimeDefs.Pas and Time in TimeUser.Pas contain the definitions and routines for the time server. They are described in the document "The Time Server" in the *Accent Programming Manual*.



## **48. TSDefs / TSUser**

Modules TSDefs in TSDefs.Pas and TS in TSUser.Pas contain the definitions and routines for the typescript manager. They are described in the document "The Typescript Manager" in the *Accent Programming Manual*.

PERQ Systems Corporation  
Accent Operating System

Pascal Library  
ViewKern

## 49. ViewKern

---

Module ViewKern in ViewKern.Pas attempts to call the kernel protected graphics operations. If those operations fail, then it calls the window manager's graphics operations instead.

The module contains the following routines, which are documented in "The Window Manager" in the *Accent Programming Manual*:

- procedure VPROP
- procedure VPColorRect
- procedure VPSscroll
- procedure VPLine
- procedure VPString
- procedure VPChArray
- procedure VPChar
- procedure VPPutString
- procedure VPPutChArray
- procedure VPPutChar



## **50. ViewPtUser**

---

Module ViewPt in ViewPtUser.Pas contains the routines for viewports. They are described in the document "The Window Manager" in the *Accent Programming Manual*.



## **51. WindowUtils**

---

Module WindowUtils in WindowUtils.Pas provides a number of useful functions for manipulating the window associated with the current process. These routines should be used in place of the window manager routines.

### **51.1. Procedure ShowPathAndTitle**

```
Procedure ShowPathAndTitle(S: TitStr);
```

**Abstract:**

This procedure is called to display the current path and the given string in the title line.

### **51.2. Procedure ShowWindowErrorFlag**

```
Procedure ShowWindowErrorFlag;
```

**Abstract:**

Turn on the error Icon flag in the window associated with this process.

### **51.3. Procedure RemoveWindowErrorFlag**

```
Procedure RemoveWindowErrorFlag;
```

**Abstract:**

Turn off the error Icon flag in the window associated with this process.

### **51.4. Procedure ShowWindowRequestFlag**

```
Procedure ShowWindowRequestFlag;
```

**Abstract:**

Turn on the request Icon flag in the window associated with this process.

## 51.5. Procedure RemoveWindowRequestFlag

Procedure RemoveWindowRequestFlag;

Abstract:

Turn off the request Icon flag in the window associated with this process.

## 51.6. Procedure ShowWindowAttentionFlag

Procedure ShowWindowAttentionFlag;

Abstract:

Turn on the attention Icon flag in the window associated with this process.

## 51.7. Procedure RemoveWindowAttentionFlag

Procedure RemoveWindowAttentionFlag;

Abstract:

Turn off the attention Icon flag in the window associated with this process.

## 51.8. Procedure StreamProgress

Procedure StreamProgress(var F: File);

Abstract:

Shows progress in reading a Pascal File in the title-line progress bar.

Parameters:

F        File being read. It must have been Reset and not Closed.

## 51.9. Procedure ComputeProgress

Procedure ComputeProgress(Current, Max: Long);

Abstract:

Shows progress in the title-line progress bar, as an amount of a total.

Parameters:

Current     How far the operation has gotten

Max        Total amount for the operation

## 51.10. Procedure RandomProgress

```
Procedure RandomProgress;
```

Abstract:

Shows random progress (something is happening, but we're not sure how much) in the title-line progress bar.

## 51.11. Procedure QuitProgress

```
Procedure QuitProgress;
```

Abstract:

Turns off the title-line progress bar.

## 51.12. Procedure MultiLevelProgress

```
Procedure MultiLevelProgress(Level: integer; Current, Max: Long);
```

Abstract:

Shows progress in the selected progress bar, as an amount of a total.

Parameters:

Level      Which progress bar to use

Current     How far the operation has gotten

Max        Total amount for the operation

## 51.13. Procedure MultiStreamProgress

```
Procedure MultiStreamProgress(Level: integer; var F: File);
```

Abstract:

Shows progress in reading a Pascal file in the selected progress bar.

Shows progress in reading a Pascal File in the selected progress bar.

Parameters:

Level      Progress bar to show progress in

F           File being read. It must have been Reset and not Closed.

## 51.14. Procedure QuitMultiProgress

```
Procedure QuitMultiProgress(Level: integer);
```

Abstract:

Turns off the selected progress bar.

Parameters:

Level      Which progress bar to use

}

# **PASMAC: A PASCAL MACRO-PROCESSOR**

**December 7, 1984**

Copyright © 1984 PERQ Systems Corporation  
2600 Liberty Avenue  
P. O. Box 2600  
Pittsburgh, PA 15230  
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

This document is adapted from the paper by A. L. Lansky,  
*PASMAC -- A Macro-Processor for Pascal*. Carnegie-Mellon University, Pittsburgh, PA, 1984.

This document is not to be reproduced in any form or transmitted in whole or in part, without the prior written authorization of PERQ Systems Corporation.

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.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

<u>Table of Contents</u>	<u>Page</u>
<u>1. Introduction</u>	<u>PM-1</u>
<u>2. How PasMac Processes a Program</u>	<u>PM-5</u>
<u>2.1. Stages</u>	<u>PM-5</u>
<u>2.2. Expansion and Evaluation</u>	<u>PM-5</u>
<u>3. Invoking PasMac</u>	<u>PM-7</u>
<u>4. Syntax</u>	<u>PM-9</u>
<u>5. Semantics</u>	<u>PM-11</u>
<u>5.1. Macro Definitions</u>	<u>PM-11</u>
<u>5.1.1. Scope and definition processing</u>	<u>PM-11</u>
<u>5.1.2. Macro blocks</u>	<u>PM-11</u>
<u>5.1.3. Macro expressions</u>	<u>PM-12</u>
<u>5.2. Macro Calls</u>	<u>PM-13</u>
<u>5.2.1. Scope and utilization</u>	<u>PM-13</u>
<u>5.2.2. Macro parameters</u>	<u>PM-13</u>
<u>5.2.3. Added notes</u>	<u>PM-16</u>
<u>5.3. Macro Variables (Mvars)</u>	<u>PM-16</u>

<b>5.3.1. Scope</b>	<b>PM-16</b>
<b>5.3.2. Typing</b>	<b>PM-17</b>
<b>5.3.3. Calls, assignments, and definition</b>	<b>PM-17</b>
<b>5.3.4. MSEQU and MLS Operators</b>	<b>PM-18</b>
<b>5.4. MINCLUDE, MEXP, and MIF Statements</b>	<b>PM-18</b>
<b>5.5. Pascal Comments</b>	<b>PM-19</b>
<b>5.6. Upper and Lower Case</b>	<b>PM-20</b>
<b>6. Examples</b>	<b>PM-21</b>
<b>6.1. Example 1</b>	<b>PM-21</b>
<b>6.1.1. The program</b>	<b>PM-21</b>
<b>6.1.2. The expansion</b>	<b>PM-22</b>
<b>6.2. Example 2</b>	<b>PM-22</b>
<b>6.2.1. The program</b>	<b>PM-22</b>
<b>6.2.2. The expansion</b>	<b>PM-24</b>
<b>6.2.3. The execution</b>	<b>PM-25</b>

## **1. Introduction**

---

PasMac is a macro pre-processor, expressly designed for and implemented in Pascal. A macro is a sequence of statements with a name. Once defined in a program, a macro may be used by simply writing its name and parameters. PasMac inserts the actual statements of the macro and substitutes the parameters. A macro differs from a routine in two ways:

- A call to a macro results in its code actually being copied into the program, whereas a call to a routine results in a linkage but no copy;
- The copy of the macro code is inserted before execution begins, whereas a routine is invoked during execution.

Pascal programs submitted to PasMac will be expanded to standard Pascal code, which may then be submitted to the Pascal compiler. PasMac is intended primarily for use with Pascal programs, but it can be used on any text obeying the PasMac rules (such as microcode).

PasMac utilizes the following:

- macro declarations
- mvar declarations (macro variables)
- macro calls
- MIF statements (conditional statements)
- MASSIGN statements (mvar assignments)

- MINCLUDE statements (include statements)
- MEXP statements (expand statements)
- loop statements
- macro expressions (strings of text occurring in the bodies of macros or in MIF statements; treated as normal source code)
- MEQU and MLS operators (Macro comparison operators that take two parameters. MEQU returns true if its parameters are equal; false, otherwise. MLS returns true if its first parameter is less than its second; false, otherwise.)

PasMac was designed to be consistent with the syntax, conventions, and philosophy of Pascal. One example of this is the strong typing of macro variables and macro parameters. Macros themselves are in many ways identical to Pascal procedures, with typed parameters (call by value) and local macro variables.

For ease of implementation and efficiency, parsing of Pascal source code was completely avoided. This had several ramifications, foremost being the lack of scoping of macro definitions. All macros and macro variables are global, except for macro variables local to a given macro.

Identical syntax is used for macro, procedure, and function calls. This makes in-line macro-time expansion of procedure calls more attractive.

Macros may be written to compensate for undesirable characteristics of Pascal. For example, a macro may be written for a "general" procedure which when expanded will have

variables of any type specified as a macro parameter. In this way, a procedure can be created for any size array, and the lack of variable length arrays can, at least artificially, be circumvented.



## **2. How PasMac Processes a Program**

### **2.1. Stages**

On a very basic level, PasMac moves through three stages of operation. The first stage simply searches for the end of the Pascal "program" statement. During this time, initial comments will be passed over and MINCLUDE statements will be expanded. After the end of the program statement, the processor enters stage two. In this stage macro and global mvar (macro variable) definitions are read in and placed into a global data structure. This stage will also handle any MIF, MASSIGN, and MINCLUDE statements that occur before stage three is entered.

The onset of stage three is triggered by recognition of the beginning of actual Pascal source code. From this point on, PasMac simply "expands" the text. This expansion entails processing of MIF, MINCLUDE, MASSIGN, and MEXP statements, as well as calling of macros and mvars, processing of the concatenation character (&), and stripping of name strings (strings enclosed by double quotes).

### **2.2. Expansion and Evaluation**

PasMac distinguishes between *expansion* and *evaluation* of text. When text is *expanded*, the following operations are performed: calling of macros and mvards, processing of MIF, MINCLUDE, MASSIGN, MEXP statements, stripping off of double quotes, and concatenation (&). After text has been expanded it may then be *evaluated*. This is done to determine the value of an MINTEGER or MBOOLEAN mvar or macro parameter.



### **3. Invoking PasMac**

Insert into your program the PasMac statements you wish to use (or place them in a separate file and include the file in your program). The statements you may use and the correct syntax are discussed in Chapters 4 and 5, and examples are given in Chapter 6.

The current version of PasMac resides in the Accent boot partition. To run the program, type *pasmac*. The program will ask for an input and output file. You can also give the input and output filenames on the command line. The input file should be the program input to PasMac, without a header page. The output file will be your expanded file.

If you do not specify an extension on the input filename, PasMac will assume the extension is .PASMAC. If no output filename is specified, PasMac will use the file's name with the extension .PAS.



## 4. Syntax

Shown below is the proper syntax to use for PasMac.

```
<pasmac program> ::= <predef stage> <def stage> <postdef stage>

<predef stage> ::= <predef stage> <minclude statement>
                  <predef stage>
 ::= Pascal Code Without Macro-Time Keywords

<def stage> ::= {<def stage statement>}
<def stage statement> ::= <macro declaration>
                         ::= <mvar declaration>
                         ::= <mif statement>
                         ::= <mvar assignment>
                         ::= <minclude statement>

<postdef stage> ::= <macro block> /* simply stuff that will be
                           expanded */

<macro declaration> ::= MACRO <macro id>; <macrobody>
                         ::= MACRO <macro id>(<par list>); <macrobody>

<macro id> ::= <identifier>

<par list> ::= <par group> {;<par group>}
<par group> ::= <identifier> .<identifier>:<mactype>

<mactype> ::= MSTRING|MINTEGER|MBOOLEAN

<macrobody> ::= MBEGIN <macro block> MEND
                  ::= <mvar declaration> MBEGIN <macro block> MEND

<mvar declaration> ::= MVAR <mvar def>{ <mvar def> }

<mvar def> ::= <mvar id>:<mactype> :=<mconst>;
                  /* blanks any place between
                     := and ; are significant */

<mvar id> ::= <identifier>

<mconst> ::= <mexpr>

<macro block> ::= <macro statement>{ <macro statement> }
<macro statement> ::= <macro expr>
                      ::= <mif statement>
                      ::= <mvar assignment>
```

```
 ::= <minclude statement>
 ::= <mexpand statement>

<mvar assignment> ::= MASSIGN (<mvar id>,<mconst>)

<mif statement> ::= MIF <mexpr> MTHEN <macro block> MFI
                   ::= MIF <mexpr> MTHEN <macro block>
                     MELSE <macro block> MFI

<macro call> ::= <macro id>
                  ::= <macro id>(<mac params>)

<mac params> ::= <mpar>{ .<mpar> }

<mvar call> ::= <mvar id>

<minclude statement> ::= MINCLUDE(<filename>)

<mexpand statement> ::= MEXP(<mpar>)

<mexpr> ::= <msimple expr>
            ::= <msimple expr> <mrel op> <msimple expr>

<mrel op> ::= =|<|=|<|=|=|>

<msimple expr> ::= <mterm>
                  ::= <msimple expr> [ +|-|OR | · mterm>

<mterm> ::= <mfactor>
            ::= <mterm> [ *|MOD|DIV|AND ] <mfactor>

<mfactor> ::= <unsigned integer>
              ::= <mstring>
              ::= <macro call>
              ::= <mvar call>
              ::= <mboolean>
              ::= ( <mexpr> )
              ::= NOT <mfactor>
              ::= MEQUAL(<mstring>,<mstring>)
              ::= MLS(<mstring>,<mstring>)

<mboolean> ::= TRUE | FALSE
<mstring> ::= <mpar>
```

## 5. Semantics

---

### **5.1. Macro Definitions**

#### **5.1.1. Scope and definition processing**

All macros are global to both source code and other macros and must be defined (along with any global mvars) all in one place within the source program, before their use. This is the <def stage> and may include macro and mvar declarations as well as MIF, MASSIGN, and MINCLUDE statements. The <predef stage> may include normal source text, but no macro time keywords except MINCLUDE statements. The <postdef stage> is text which is expanded.

It is illegal for two macros to be declared with the same macro name, or for a macro name to be the same as a global mvar name, macro-time, or Pascal keyword. All formal parameters must be declared to be of type MINTEGER, MBOOLEAN, or MSTRING. The passing convention will be by value.

#### **5.1.2. Macro blocks**

Macro blocks consist of a list of macro statements. These statements may be of various kinds: the conditional statement (MIF), an mvar assignment (MASSIGN), a loop statement, an include statement (MINCLUDE), an expand statement (MEXP), or simply a macro expression.

### 5.1.3. Macro expressions

A macro expression, which occurs in the bodies of macros as well as in MIF statements, is a string of text, treated by the macro processor as it would treat normal source code. Hence the following discussion is applicable to the <postdef stage> as well.

During expansion of macro expressions, macro calls and mvar calls are expanded. Note that this expansion will occur only when a macro is called, not at macro definition time. Only the appropriate <macro block> in an MIF statement will be inserted into expanded code. MASSIGN statements will cause the desired changes to mvars, but will not occur in expanded code. The effects of the MINCLUDE and MEXP statements are described in Section 5.4.

Within any string of expanded text, the concatenation character (&) may be used to concatenate two pieces of text. Hence if A is a macro variable with value 2, VAR&A will be expanded as VAR2. If you wish to include the character & in your text without any significance, use && which will be interpreted as a single &.

Within any string of text, a string may be double-quoted ("..."). The string within the double quotes will be inserted into the expanded code, but will not be expanded itself. If you wish to include the character " in your text without significance, use "" which will be interpreted as a single ".

Blanks and carriage return / line feeds are significant within macro blocks except as follows:

- Immediately before the key words  
MBEGIN,MIF,MELSE,MFI,MEND.
- Immediately after the key word MBEGIN

## 5.2. Macro Calls

### 5.2.1. Scope and utilization

A macro call may occur anywhere after its definition. Each time a macro is called, it is expanded and inserted in-line, replacing the text of its call. Note that rescanning is *not* performed. For example, if a macro forms the name of another macro when expanded, that second macro will not be called.

Example: MACRO mac; MBEGIN foo&1 MEND

If `foo1` is the name of a macro, it will *not* be called when a call of `Mac` is performed. If you want `foo1` to be called, write `MEXP(mac)`. `MEXP`, as described below, simply rescans its parameter and returns the result of this second expansion.

The character & is used as a break character which delimits tokens read by the macro processor. The resulting expansion (of, say, `foo&1`) will look like one token to the Pascal compiler, but not to the macro processor.

Note also that macro expansion is always inhibited inside of Pascal comments and within Pascal strings under the circumstances listed in Section 5.5.

### 5.2.2. Macro parameters

Actual parameters to macros are expanded in the same way and may include the same things as macro blocks. Usually, however, they will only be macro expressions. The characteristics unique to macro parameters are as follows:

- PasMac permits passing null parameters to macros in a natural way. For example, `mac(,)` is a macro call with two null parameters.
- Each macro parameter is delimited by either a comma

or right parenthesis. A nesting count is kept of all ( ) and [ ] pairs. This count prevents truncation of parameters meant for nested macro invocations. Any commas or ')'s occurring when the nesting count is greater than zero will not be considered delimiters of the parameter currently under expansion. Commas or ')'s occurring when the nesting count is zero will delimit the macro parameter currently under expansion. Thus, in order to include either a comma or right parenthesis within a macro parameter, it must be quoted.

Example:

```
MACRO str; MBEGIN This is a string MEND
MVAR v1:minteger = 25
```

The macro call mac( str&v1 "str" (,)"," ,str"" str b")" ) has two parameters. The first one is (square brackets excluded):

[ This is a string25 str (,), ]

and the second one is:

[This is a string" This is a string b) ].

Doubling insignificant double quotes ("") makes passing parameters like "|||||parameter|||||" (which you want to pass down three macro levels) a possible cause of confusion. The number of quotes required is 2\*(number of levels-1). Apply MEXP to the parameter each time it is passed down a further level.

Example:

```
MACRO levels( B:mstring);
MBEGIN
...
MIF not(MEQU(B,abc)) MTHEN
```

```
    levels(MEXP(B))  
MFI  
...  
MEND
```

If we call this macro as follows:

levels(""""abc""") then at first call B = ""abc"""  
,second call B = "abc" ,third call B = abc

The use of MEXP is necessary because double quotes are stripped off only when code is *expanded*, not when it is scanned. Hence, if we were to use the call LEVELS(B) within the macro LEVELS, the formal parameter B would be expanded to its current value and passed on to the next depth of macro call, but not expanded(stripped) again.

- Once expanded, a macro parameter may then be evaluated, depending on what type it is. If it is an MSTRING parameter, the expanded parameter is left as is. If it is an MINTEGER or MBOOLEAN parameter, it is evaluated as an integer or boolean expression and the actual parameter used will now be that new integer or boolean value.

Example:

```
MACRO eval(n:mininteger); MBEGIN n MEND  
MACRO noteval(m:mstring); MBEGIN m MEND  
noteval(5+3+eval(2+3)) = noteval(5+3+5)  
= 5+3+5
```

Note that the operands of the functional boolean operators, MEQU and MLS, are not treated as regular parameters. In other words, they are not re-expanded when the operators are invoked.

Example:

```
MACRO bool(b:mboolean);
```

```
MBEGIN b MEND
bool(mequ("abc", abc)) would be TRUE
bool(mequ("""abc""", abc)) would be FALSE
```

For further details on MLS and MEQU, see Section 5.3.4.

The results of expanding and then evaluating a macro parameter results in a string which is then used to replace all occurrences of a formal parameter within the macro body. Note that this occurs without reevaluation( or re-expansion ). Hence a parameter "<macro name>" cannot be used to invoke a macro call within the calling macro's macrobody. However, MEXP may be used to do this if you desire.

### 5.2.3. Added notes

Tokens within macro bodies are tested first for being formals or local mvars and then macros or global mvars. Hence, if a formal parameter has the same name as a macro, it will be interpreted as a formal. Moreover, the names of all formals and local mvars must be disjoint, and the same is true for global mvars and macros.

## 5.3. Macro Variables (Mvars)

### 5.3.1. Scope

Macro variables may be used locally within a macro definition or globally. Those declared along with macros, at the beginning of the program, are global to all source code and other macros. Those declared within a macro definition are local to that macro.

Macro variable names may not be duplicated on the same scope level. That is, it is illegal to declare two global mvars of the same name or two local mvars of the same name (local to the same macro definition), but it is all right to have a global and local mvar of the same name.

### 5.3.2. Typing

All mvars are typed as MBOOLEAN, MINTEGER, or MSTRING. When an mvar is declared, its name and type information is stored. Its initialized value as well as all values later assigned to it must be of the the correct type. <mconst>'s are dealt with the same way as macro parameters. In other words, first they are *expanded*, then *evaluated*. This is an important point to realize, or arithmetic and boolean expressions may not be evaluated in the way one expects. The processor expects that <mconst>'s (in particular, those of type MINTEGER or MBOOLEAN) will be of the correct type when expanded.

### 5.3.3. Calls, assignments, and definition

Mvar calls follow the exact same scope rules and conventions as those listed for macro calls (see Section 5.2).

Mvar assignment statements may occur anywhere and are not included in expanded text. An mvar must be declared before an assignment is made to it. If a local and global mvar have the same name, only the local mvar will be affected if assignment occurs within its scope of definition. Otherwise, the assignment affects only the global variable. Note, too, that the second parameter to MASSIGN will be *expanded*, then *evaluated*, as is done for macro parameters. The same is done for the initialization value of a macro variable at mvar definition time. All blanks between := and ; are considered significant within the mvar definition. Note, too, that any insignificant semi-colons (;) must be quoted.

### 5.3.4. MEQU and MLS Operators

MEQU and MLS are macro comparison operators that take two parameters. MEQU returns true if its parameters are equal; false, otherwise. MLS returns true if its first parameter is less than its second; false, otherwise.

MEQU and MLS are treated just as functional operators. Hence, they are not applied until a boolean expression is being evaluated. During the initial expansion of a boolean expression, the parameter strings to MLS and MEQU will also be expanded. But when MLS or MEQU is "applied," this expansion *does not* reoccur. Also, *the counting the balanced round and square brackets is not done. Hence all commas will be significant within the expanded string parameters.*

For example, MACRO m (a,b:mstring) MBEGIN a,b MEND  
MEQU(m(abc,def),"abc,def") will be false. The first parameter to MEQU will be 'abc' and the second will be 'def,abc,def'.  
MEQU(m(abc,abc)) will be true though. MEQU(abc,""abc"") will be false.

### 5.4. MINCLUDE, MEXP, and MIF Statements

MININCLUDE statements may occur anywhere and are replaced in expanded text by the contents of the file specified. The specified file is read by the macro processor as if it were part of the original file being read. That is, it is processed as a continuation of the original file and processing returns to the original file once the included file has been exhausted. The file name specified will be "expanded" as if it were a string parameter to a macro call.

Nested inclusions may occur only to a certain level. In the current implementation, eight files may be included at the same moment (within each other).

MEXP statements may also occur anywhere. The parameter to MEXP is expanded twice (expanded and then re-expanded) and then replaces the MEXP call.

MIF statements may also occur anywhere. The conditional switch of the call is handled in the straightforward way, i.e. expanded, and then evaluated as a boolean expression. Depending on the result of the conditional expression, the appropriate <macro block> is expanded and inserted into the source text.

## 5.5. Pascal Comments

Comments, when recognized, will not occur in macroprocessor expansions under the following circumstances (they will simply be ignored):

- Within an integer or boolean valued <mexpr> (i.e. the value of an integer or boolean MVAR or the boolean switch of an MIF statement)
- Within an MINCLUDE statement
- Within a MACRO definition (but allowed within <macrobody>)
- Within an MVAR definition (but allowed within <mconst>)
- Within an MEXP statement (but allowed within the parameter to MEXP).
- Within an MASSIGN statement (but allowed within the second parameter, <mconst>)
- Within an MLS or MEQU call (but allowed within the two mstring parameters. Note that comments included in such parameters will be checked for comparison as *part* of the parameter strings).

## 5.6. Upper and Lower Case

The current implementation admits upper and lower case characters. In fact, the entire ASCII character set is admissible. However, capitalization is *not* significant with respect to macro-time keywords, macro names, mvar names, or formal parameter names. Hence a macro macfoo may be invoked by MACFOO or MACfoo or macfoo.

## **6. Examples**

### **6.1. Example 1**

The following short example illustrates various aspects of expansion and evaluation.

#### **6.1.1. The program**

```
program tester;
MVAR a:MINTEGER := 5*6-5;
      b:MINTEGER := 3+(a DIV 2);
      c:MINTEGER := 3+((b+1) MOD 3)+20;
      d:MBOOLEAN := TRUE OR FALSE;
      e:MBOOLEAN := FALSE AND d OR TRUE;
      f:MBOOLEAN := 2 < 7*3;
      g:MBOOLEAN := (2+5 <= 7) AND (2-7 >= 7);
      h:MSTRING := a b c d e f g h;
      j:MBOOLEAN := MLS(abc,abcd);
      k:MBOOLEAN := MEQU(abcd,abcde);
      l:MSTRING :=string;
      m:MSTRING := "h" MLS(l,string) 'l';
      p:MBOOLEAN := MEQU("'"abc"',abc);

MACRO humbug(a:MINTEGER;b:MSTRING;c,d:MBOOLEAN);
MBEGIN
parameters : a,b,c,d
the global "h" : h
mvars "j,k,l,m,p" : j,k,l,m,p
MEND

MACRO doowop(a:MSTRING);
MBEGIN
a ;
MIF NOT (MEQU(a,ladida)) MTHEN
doowop(MEXP(a))
MFI
MEND

MACRO evalmls(b:MBOOLEAN); MBEGIN b MEND

MVAR n:MSTRING :=evalmls(MLS(l,string));

VAR
  I:INTEGER
BEGIN
humbug(1+(a),foobar,f,g)
a  massign(a,a*(2))    a
```

```
doowop("""ladida""")
n&boop
END.
```

### 6.1.2. The expansion

```
program tester;

VAR
  I:INTEGER
BEGIN
parameters : 26,foobar,TRUE,FALSE
the global h : 25 15 24 TRUE TRUE TRUE FALSE h
mvars j,k,l m,p : TRUE,FALSE,string,h MLS(string,string) '1',FALSE
25      50
"ladida" :    ladida ;
FALSEboop
END.
```

## 6.2. Example 2

The following longer example illustrates expansion, evaluation, and execution.

### 6.2.1. The program

The file Defs.Txt is as follows:

```
mvar symno:minteger := 1;
macro gensym;
mbegin
sym&symno massign(symno,symno+1)
mend

macro factorial(n:minteger);
mbegin
mif n=1 mthen
  1
melse
  n*factorial(n-1)
mfi
mend

macro condprint(message:mstring; trace:mboolean);
mbegin
mif trace mthen
  writeln(output,message)
mfi
mend

macro switchproc(mtype:mstring);
mbegin
procedure switch&mtype(var a,b:mtype);
```

```

var c:mtype;
begin
  c := a;
  a := b;
  b := c;
end;
mend

macro switchcode(a,b,c:mstring);
mbegin
  c := a;
  a := b;
  b := c;
mend

mvar fool:minteger := 1;
mvar foo:minteger := 2;

macro exptest;
mbegin "foo"&l mend

macro eval(x:minteger);
mbegin x mend

macro increment(a:minteger);
mvar c:minteger := a;
mbegin
  massign(c,c+1) c
mend

```

The main program is as follows:

```

program pmtest(input,output);
minclude(defs.txt)
type
  str=packed array[1..5] of char;
var
  gensym,gensym,gensym:integer;
  var1,var2,var3:integer;
  str1,str2,str3:str;

switchproc(integer)
switchproc(str)
begin
var1 := eval(factorial(3));
var2 := eval(factorial(4));
condprint('var1:','');
var1,"' var2:'","");
var2,true);
switchinteger(var1,var2);
condprint('var1:','');
var1,"' var2:'","");
var2,true);
switchcode(var1,var2,var3);
condprint('var1:','');
var1,"' var2:'","");
var2,true);
str1 := """gensym""";
str2 := """gensym""";
condprint('str1:','');
str1,"' str2:'","");
str2,true);
switchstr(str1,str2);

```

```
condprint('str1:","",str1,"' str2:","",str2,true);

var1 := foo&1;
condprint('var1:","",var1,true);
var1 := mexp(exptest);
condprint('var1:","",var1,true);
var1 := increment(10);
condprint('var1:","",var1,true);
end.
```

## 6.2.2. The expansion

```
program pmtest(input,output);
type
  str=packed array[1..5] of char;
var
  sym1 ,sym2 ,sym3 :integer;
  var1,var2,var3:integer;
  str1,str2,str3:str;

procedure switchinteger(var a,b:integer);
var c:integer;
begin
  c := a;
  a := b;
  b := c;
end;

procedure switchstr(var a,b:str);
var c:str;
begin
  c := a;
  a := b;
  b := c;
end;

begin
  var1 := 6;
  var2 := 24;
  writeln(output,'var1:',var1,' var2:',var2);
  switchinteger(var1,var2);
  writeln(output,'var1:',var1,' var2:',var2);
  var3 := var1;
  var1 := var2;
  var2 := var3;
  writeln(output,'var1:',var1,' var2:',var2);
  str1 := 'sym4 ';
  str2 := 'sym5 ';
  writeln(output,str1,str1,' str2:',str2);
  switchstr(str1,str2);
  writeln(output,str1,str1,' str2:',str2);

  var1 := 21;
  writeln(output,'var1:',var1);
  var1 := 1;
```

```
writeln(output,'var1:',var1);
var1 := 11;
writeln(output,'var1:',var1);
end.
```

### 6.2.3. The execution

```
VAR1:          6 VAR2:          24
VAR1:          24 VAR2:           6
VAR1:          6 VAR2:          24
STR1:SYM4  STR2:SYM5
STR1:SYM5  STR2:SYM4
VAR1:          21
VAR1:           1
VAR1:          11
```

