Pascal 180 External Reference Specification

M R Renfro

DISCLAIMER:

This document is an internal working paper and does not represent any intent on the part of Control Data Corporation.

REVISION RECORD	
REVISION	DESCRIPTION
A 83-06-06	Preliminary version.
B 84-04-30	Update for Analysis and Design Phase.
C 84-10-31	Update to resolve comments to Revision B.
D 85-04-30	Update to resolve comments to Revision C.
E 85-10-22	Update for Release 1.1.4. Chiefly structured constants and revised diagnostics.

Table of Contents

1.0 INTRODUCTION AND OBJECTIVES

This document is the External Reference Specification for Pascal 180. It specifies the language implemented, the user interface to the compiler and the diagnostics produced by the compiler.

The Pascal referance manual [reference 8, below], which exists in draft form, presents additional detail, and cases, explanations and exceptions of Pascal which were felt to be more appropriate to place in the manual rather than in the ERS.

2.0 REFERENCES

2.0 REFERENCES

1. Specification for Computer Programming Language Pascal, ISO 7185, 1983.

2. American National Standard Pascal Computer Programming Language ANSI/IEEE770X3.97-1983.

3. Pascal 180 Project Plan, DCS S4407.

4. Cyber 180 System Interface Standard [SIS]. DCS S2196.

5. Pascal Version 1 Reference Manual. CDC Publication 60497700, Revision A, 1983.

6. Pascal 180 DR, DCS S4647.

7. ANSI/IEEE Joint Pascal Committee documents:

   a. X3J9/JPC/80-189R. Otherwise Clause in Case Statement.

   b. X3J9/JPC/80-150. Variant Part Completer .

   c. X3J9/JPC/82-072R. Underscore Character in Identifiers

   d. X3J9/JPC/82-025. Relaxation of Oder of Declaration.

   e. X3J9/JPC/84-025. Variable Length Strings.

   f. X3J9/JPC/84-032. Value Initialization.

   g. X3J9/JPC/84-080. Index String Function.

   h. X3J9/JPC/84-081. Substr String Function.

   i. X3J9/JPC/84-086. Dynamic Strings.

   j. X3J9/JPC/83-076. Ranges in Case Statement Constant Lists.

   k. X3J9/JPC/83-076. Ranges in Case Variant Constant Lists.

   l. X3J9/JPC/85-036. Structured Constants.

8. Pascal for NOS/VE Usage, 60485613 (draft).

9. Pascal 180 1.1.4 Project Plan, DCS S4989.

3.0 FEATURE DESCRIPTION

3.0 FEATURE DESCRIPTION

3.1 ABSTRACT

Pascal 180 implements the Programming Language Pascal, as described below.

3.2 DESCRIPTION

The initial release of Pascal 180 implements the Pascal language as defined by the ISO Standard and the ANSI Standard, with exceptions noted below. The ISO Standard provides the base language definition. Certain extensions, detailed below, are provided. These extensions to the above standards can be flagged at the user's request.

3.2.1 EXTENSIONS

Extensions to the ISO Standard are listed below. To obtain a complete language description, the text of the ISO Standard is modified as noted.

3.2.1.1 Non-alphanumeric Characters in Identifiers

Pascal identifers are extended to allow the underscore '_' and the currency symbol '$' as part of an identifier in the same manner as a digit.

Modify definition of identifier [p7, 6.1.3 of ISO Standard] as follows:

```
identifier = letter {letter|digit|letter-symbol} .
letter-symbol = "_" | "$" .
```

3.2.1.2 VALUE Initialization Part

The implementation of value initializations is defined as:

Modify word-symbol [p7 of ISO Standard] as follows:

```
Insert after "until", | "value"
```

Modify definition of block [p9 of ISO Standard] as follows:

```
Insert after 'variable-declaration-part' in 'block =':
     value-initialization-part

Insert after 'variable-declaration-part = ...':
     value-initialization-part =
          ["value" value-initialization ";"
```

```
            {value-initialization ";"} ] .
```

Insert after variable-declaration [p21-24 of ISO Standard] the following definitions:

```
     value-initialization =
          variable-identifier (":="|"=") value-specification .
          value-specification = constant | "nil" | set-value .
          set-value = set-constructor .
```

NOTE: The two forms of operator provided (':=' and '='). The former is the choice of the ANSI extension which is currently in progress, the later is the Pascal 170 form. The former is the preferred form.

3.2.1.3 OTHERWISE Clause in CASE Statement

The implementation of the otherwise clause in the case statement is defined as:

Modify definition of case statement [p45 of ISO Standard] to read:

```
     case statement =
          "case" case-index "of"
          case-list-element {";" case-list-element}
          case-statement-tail .
```

Insert following definition after 'case-index':

```
     case-statement-tail = ["otherwise" statement] [;] "end".
```

3.2.1.4 OTHERWISE Clause in Variant Record

The implementation of the otherwise clause in variant records is defined as:

In section 6.4.3.3 [p15 of ISO standard], replace the production for "variant-part" with the following:

```
     variant-part =
          "case" variant-selector "of"
          variant-list-element {";" variant-list-element}
          [ [";"] variant-part-completer ] .
     variant-list-element =
          case-constant-list ":" variant-denoter .
     variant-part-completer =
          "otherwise" variant-denoter .
     variant-denoter =
```

"(" field-list ")" .

3.2.1.5 Relaxation of Ordering of Declarative Parts

Pascal declarative parts are allowed to occur in any order, and may be repeated. The earlier restrictions to definition before use apply, in particular, a forward reference pointer type declaration must have the base type defined by the end of the type-definition-part in which it occurs. The extension is defined by:

In section 6.2 [p9 of ISO standard] modify the production for block as follows:

```
block = {label-declaration-part | constant-definition-part |
         type-definition-part | variable-declaration-part |
         value-declaration-part |
         procedure-and-function-declaration-part }
         statement-part .
```

NOTE: This extension takes into account the value extension in 3.2.1.2 of this document.

3.2.1.6 String Extensions

Pascal string extensions include: variable length strings, relaxation of operations on fixed length strings, dynamic strings and the inclusion of new predefined routines to facilitate string operations. The string extensions are defined as follows:

Replace the second sentence of the first paragraph of section 6.1.7 [p8, ISO standard] with the following:

> A character-string containing more than one string-element shall denote a value of a string-type with a length equal to the number of string-elements contained in the character-string. A character-string containing zero elements shall denote the null-string.

Replace the production for character-string [p8, ISO standard] with:

```
character-string = "'" {string-element} "'" .
```

Replace section 6.2.2.10 [p10, ISO standard] with:

> Identifiers that denote the required constants, types, schema, procedures and functions shall be used as if their

defining-points have a region enclosing the program.

Insert before the NOTE in section 6.4.2.2 [p13, ISO standard]:

The length of a char-type value shall be 1. The maximum-length of the char-type shall be 1.

NOTE: A char-type value may be used as a string-type value of length 1.

Replace the first sentence of 6.4.3.1 [p14, ISO standard] with:

A new-structured-type shall be classified as an array-type, record-type, set-type, file-type, or variable-string-type according to the unpacked-structured-type or variable-string-type closest-contained by the new-structured-type.

Replace the production for new-structured-type in 6.4.3.1 [p14, ISO standard] with:

```
new-structured-type = ["packed"] unpacked-structured-type
                    | variable-string-type .
```

Replace the last three paragraphs of 6.4.3.2 [p15, ISO standard to form the new section between 6.4.3.2 and 6.4.3.3:

6.4.3.x String Types

6.4.3.x.1 General. A string-type shall be denoted by either a fixed-string-type or a variable-string-type. The values of a string-type shall be structured as a mapping from each value of an index-domain onto a distinct component. The index-domain of a string-type value shall either be empty (that is, has no value) or shall be an integer subrange-type with a smallest value of 1 and a largest value of greater than or equal to 1. Each component of a string-type value shall be a value of the char-type.

The length of a string-type value shall be zero if the index-domain is empty; otherwise it shall be the largest value of the index-domain. The string-type value with length zero is designated the null-string.

The correspondence of character-strings to values of string-types is obtained by relating the individual string-elements of the character-string, taken in textual order, to the components of the values of the string-type in

order of increasing index.

NOTES: (1) String-types possess properties which allow accessing a substring and reading from a textfile. String-type values may be used as the actual-parameter correspond-ing to a value-parameter possessing a string-type written to a textfile and used with the relational operators and with the string concatenation operator.

(2) Char-type values possess properties that allow them to be used identically to string-type values of length 1. In particular, char-type values may be used as the actual-parameter corresponding to a value-parameter possessing a string-type, assigned to a variable possessing a string-type, written to a textfile and used with the relational-operators and with the string concatenation operator.

6.4.3.x.2 Fixed-string-types. Any type designated packed and denoted by an array-type having as its index-type a denotation of a subrange-type specifying a smallest value of 1 and a largest value of greater than or equal to one, and having as its component-type a denotation of the char-type, shall be designated a fixed-string-type.

The maximum-length of a fixed-string-type shall be the largest value of the index-type.

NOTES: (1) A fixed-string-type possesses the properties of both an array-type and a string-type.

(2) The length of all values of a particular fixed-string-type is equal to the maximum-length of the fixed-string-type.

6.4.3.x.3 Variable-string-types. The maximum-length of a variable-string-type shall be greater than or equal to 1. Each value of a variable-string-type shall be a string-type value with a length less than or equal to the maxi-mum-length of the variable-string-type.

There shall be a schema that is denoted by the required schema-identifier STRING. The scheme-identifier in a variable-string-type shall denote the required scheme STRING.

variable-string-type = schema-identifier
"(" maximum-length ")" .
schema-identifier = identifier .

maximum-length = constant .

Example:

var str: string (6);

NOTE: A variable-string-type possesses the properties of a string-type. The individual components of a variable-string-type can be obtained by indexing it as an array.

Replace the production for domain-type in section 6.4.4 [p19, ISO standard] as follows:

domain-type = type-identifier | indefinite-string-type

Replace rule d of section 6.4.5 [p20, ISO standard] with:

(d) T1 and T2 are char-types or string-types.

Replace rule e of section 6.4.6 [p20, ISO standard] with:

(e) T1 and T2 are compatible, T1 is a string-type, and the length of the value of T2 is less than or equal to the maximum-length of T1.

Add to the end of section 6.4.6 [p20, ISO standard]:

(3) It shall be an error of T1 and T2 are compatible, T1 is a string-type, and the length of the value of T2 is greater than the maximum-length of T1.

Modify the last sentence of the second paragraph of section 6.5.1 [p21, ISO standard] to read:

A variable-access, according to whether it is an entire-variable, a component-variable, an identified-variable, a buffer-variable, or a substring-variable, shall denote either a declared variable, a component of a variable, a variable which is identified by a pointer value, a buffer-variable or a substring-variable, respectively.

Replace the production for variable-access, section 6.5.1 [p21, ISO standard] with:

variable-access = entire-variable | component-variable | identified-variable | buffer-variable | substring-variable .

3.0 FEATURE DESCRIPTION
3.2.1.6 String Extensions

Replace the production for indexed-variable, section 6.5.3.2 [p22, ISO standard] with:

```
indexed-variable = array-variable "[" index-expression
                     {"," index-expression } "]" |
                   string-variable "[" index-expression "]" .
string-variable = variable-access .
```

Replace the second paragraph of 6.5.3.2 [p22, ISO standard] with:

An array-variable shall be a variable-access that denotes a variable possessing an array-type. A string-variable shall be a variable-access that denotes a variable possess-ing a string-type. The string-variable of an indexed-variable shall denote a variable possessing variable-string-type.

NOTE: Variables possessing a fixed-string-type are indexed using array-type properties.

For an indexed-variable closest-containing an array-variable and a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type.

For an indexed-variable closest-containing a string-variable and index-expression, the index-expression shall possess the integer type. It shall be an error if the vale of the index-expression in an indexed-variable closest-containing a string-variable is less than one or greater than the length of the value of the string-variable. It shall be an error to alter the length of the value of a string-variable when a reference to a component of the string-variable exists.

The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-variable or string-variable.

Add the following new section, after 6.5.5 [p24, ISO standard]:

6.5.6 Substring-variables. A substring-variable shall denote a variable possessing a new fixed-string-type.

```
substring-variable = string-variable "[" index-expression
```

"." index-expression "]" .

The index-expression in a substring-variable shall possess the integer-type. It shall be an error if the values of either index-expression in a substring-varible is less than 1 or greater than the length of the value of the string-variable of the substring-variable or if the value of the leftmost index-expression is greater than the value of the rightmost index-expression. The maximum-length of the fixed-string-type possessed by the variable denoted by the substring-variable shall be equal to one plus the value of the rightmost index-expression minus the value of the leftmost index-expression. The components of the variable denoted by the substring-variable shall be, in order of increasing index, the contiguous components of the string-variable from the component that corresponds to the value of the leftmost index-expression through the component that corresponds to the value of the rightmost index-expression.

The order of evaluation of the index-expressions of a substring-variable shall be implementation-dependent.

It shall be an error to alter the length of the value of a string-variable when a reference to a substring of the string-variable exists. A reference or access to a substring of a variable shall constitute a reference or access, respectively, to the variable.

Replace the production for variable-parameter-specification in section 6.6.3.1 [p28, ISO standard] with:

```
variable-parameter-specification =
                "var" identifer-list ":" (type-identifier |
                                          indefinite-string-type ) .
indefinite-string-type = schema-identifier .
```

Add the following paragraph to the end of section 6.6.3.2 [p29, ISO standard]:

If the formal parameter possesses a fixed-string-type and its maximum-length is greater than the length of the value of the expression then the value attributed to the variable denoted by the formal parameter shall be a value of the fixed-string-type whose components in order of increasing index shall be the components of the value of the expression in order of increasing index or the char-type value of the expression, followed by spaces.

3.0 FEATURE DESCRIPTION
3.2.1.6 String Extensions

Replace the second sentence of section 6.6.3.3 [p29, ISO standard] with:

> The schema-identifier in an indefinite-string-type shall denote the required schema string. The actual-parameters corresponding to formal parameters that occur in a single variable-parameter-specification containing an indefinite-string-type shall all possess the same variable-string-type. The formal parameters shall possess a variable-string-type that shall be distinct from any other type, and which shall have a maximum-length equal to the maximum-length of the variable-string-type possessed by the actual-parameters. Otherwise, the type possessed by the actual-parameters shall be the same as that denoted by the type-identifier of the variable-parameter-specification, and the formal parameters shall also possess that type.

Add to the end of the last paragraph of section 6.6.3.3 [p29, ISO standard] the following:

> An actual variable parameter shall not denote a component of a string-type.
>
> NOTE: An actual variable parameter cannot denote a substring-variable because the type of a substring-variable is a new fixed-string-type different from every named type.

See section 3.2.1.8, this document for new predefined functions index, length, maxlength, and substr descriptions.

In section 6.6.5.3 [p34, ISO standard], after the first sentence for both paragraphs "new(p)" and "new(p,c1,...,cn)" insert the following sentence:

> "The domain-type of the pointer-type possessed by p shall be a type-identifier."

Add after the second paragraph in section 6.6.5.3 [p35, ISO standard] the following paragraph:

> new(p,i) shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The domain-type of the

pointer-type possessed by p shall be an indefinite-string type. The created variable shall possess a new variable-string-type which shall have a maximum-length specified by the value of the expression i which shall be of integer-type. It shall be an error if the value of i is not greater than zero.

Replace the second to last paragraph of section 6.7.2.5 [p41, ISO standard] with:

When the relational operators =, <>, <, >, <=, and >= are used to compare operands of compatible char-types or string-types, they denote the lexicographic relations defined below. Lexicographic ordering imposes a total ordering on values of a char-type or string-type.

Let s1 and s2 be two values of compatible char-types or string-types where the length of s1 is less than or equal to the length of s2, and let n1 be the length of s1, and let n2 be the length of s2; then

```
s1 = s2 iff (for all i in [1..n1]: s1[i] = s2[i])
        and (for all i in [n1+1..n2]: ' ' = s2[i])

s1 < s2 iff (there exists a p in [1..n]:
            (for all i in [1..p-1]: s1[i] = s2[i])
             and s1[p] < s2[p] )
        or ( (for all i in [1..n1]: s1[i] = s2[i])
        and (there exists p in [n1+1..n2]:
            (for all i in [n1+1..p-1]: ' ' = s2[i])
             and ' ' < s2[p]) )
```

Add the following new section, before 6.7.3 [p41, ISO standard]

6.7.2.6 String operator. The types of operands and results for the string operator shall be as shown in table 6.

Table 6
String Operator

operator	operation	type of operands	type of result
+	string concatenation	any char-type or string-type	variable-string-type

Let a and b be operands possessing any char-type or string-type, then a + b shall denote a string-type value whose length shall be equal to the sum of the length of a and the length of b. The value of the components of a + b in order of increasing index shall be the values of the components of a in order of increasing index or the char-type value of a followed by the values of the components of b in order of increasing index or the char-type value of b.

Add the following after the first paragraph of section 6.8.2.2 [p42, ISO standard]:

If the variable denoted by the variable-access of the assignment-statement or the activation result that is denoted by the function-identifier of the assignment-statement possesses a fixed-string-type and its maximum-length is greater than the length of the value of the expression of the assignment-statement, then the value attributed to the variable-access or activation result shall be a value of the fixed-string-type whose components in order of increasing index shall be the components of the value of the expression in order of increasing index or the char-type value of the expression, followed by spaces.

NOTE: This applies to substring-variables as well, since they possess a fixed-string-type.

Replace in the last sentence of p48, ISO standard:

"or the real-type)."

with:

"the real-type, or a string-type)."

Insert between paragraphs (b) and (c) of section 6.9.1 [p49, ISO standard] the following:

(b.1) If v is a variable-access possessing a fixed-string-type, read(f,v) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall cause the reading from the referenced textfile variable of a sequence of characters. Reading shall cease as soon as either the number

of characters read from the buffer-variable of the referenced textfile attributed to the buffer-variable variable is an end-of-line. The value attributed to the variable v shall be the value of the fixed-string-type whose components in order of increasing index consist of the sequence of characters read from the buffer-variable followed by zero or more spaces.

NOTE: If the value of the buffer-variable is initially an end-of-line, then no characters are read and the value of each component of v is a space.

(h.2) If v is a variable-access possessing a variable-string-type, read(f,v) shall access the textfile variable and establish a reference to that textfile variable for the remaining execution of the statement. The remaining execution of the statement shall cause the reading from the referenced textfile variable of a sequence of characters. Reading shall cease as soon as either the number of characters read from the buffer-variable of the referenced textfile equals the maximum-length of the variable-string-type or the component attributed to the buffer-variable variable is an end-of-line. The value attributed to the variable v shall be the value of the variable-string-type whose length is equal to the number of characters read from the buffer-variable and whose components in order of increasing index consist of the sequence of characters read from the buffer-variable.

NOTE: If the value of the buffer-variable is initially an end-of-line, then no characters are read and the value of v is the null-string.

Replace the first sentence of section 6.9.3.6 [p52, ISO standard] with:

If the value of e is a string-type value with a length of n, the default value of TotalWidth shall be n.

3.2.1.7 Ranges in CASE Statement and Variant Constant Lists

Ranges are allowed in case statement constant lists and case variant constant lists. The extension is defined as follows:

In section 6.4.3.3 [p15 of ISO standard] the production for case-constant-list is changed as follows:

case-constant-list = case-range { "," case-range } .

3.0 FEATURE DESCRIPTION
3.2.1.7 Ranges in CASE Statement and Variant Constant Lists

Add the production for case-range:

case-range = case-constant { ".." case-constant} .

Add after the end of the first paragraph after the first NOTE in 6.4.3.3 [p16 ISO standard] the following paragraph:

The value denoted by a case-constant of a case-range shall be distinct from the value denoted by any other case-constant of the case-range.

Add to the end of the first paragraph after the first NOTE in 6.4.3.3 [p16 ISO standard]:

A case-range shall denote the set of values consisting of the values denoted by the case-constants of the case-range and, if two case-constants are specified, the values between the values denoted by the case-constants. If present, the second case-constant of the case-range shall denote a value greater then or equal to the value denoted by the first case-constant of the case-range.

Replace the second paragraph after the NOTE with:

The type of each case-constant of a case-constant-list of a variant-list_element of a variant-part shall be compatible with the type denoted by the tag-type of the variant-selector of the variant-part, and the value denoted by each such case-constant shall be a member of the set of values determined by that type.

Replace the first paragraph of 6.8.3.5 [p44 ISO standard] with:

The values denoted by the case-ranges of the case-constant-list of the case-list_elements of a case-statement shall be distinct and of the same ordinal-type as the expression of the case-statement. A case-range shall not overlap the range of an already specified value. On execution of the case-statement the case-index shall be evaluated. That value shall then specify execution of the statement of the case-list-element closest-containing the case-range denoting the set of values that contains that value.

It shall be an error is none of the case-ranges is equal to or contains in its range the value of the case-index upon entrance to the case-statement.

3.2.1.8 Structured Constants

Structured constant constructures shall be allowed in constant definition parts, and those constants so defined shall be allowed as defined below:

In 6.2.2.3 [p10 ISO standard] change the reference to 6.3 to refer to 6.3.1 and add references 6.3.2 and 6.3.4.2 .

Between the heading and text of section 6.3[p11 ISO standard], add the heading "6.3.1 General.". In the subsection thus created, change the production rule for constant to be as follows:

```
constant = [sign] ( unsigned-number ¦ constant-identifier )
           ¦ character-string ¦ "nil" ¦ structured-constant .
```

Following the above section 6.3.1 [p11-12 ISO standard], add the following new sections 6.3.2, 6.3.3, and 6.3.4:

6.3.2 Structured-constants

6.3.2.1 General. A structured-constant shall denote a value of the type of the structure-constant. That type shall be a type that is permissible as the component-type of a file-type (see 6.4.3.5).

```
structured-constant =
           array-type-identifier array-value ¦
           record-type-identifier record-value ¦
           set-type-identifier set-value .

component-value = constant ¦ array-value ¦ record-value
                  ¦ set-value .
```

The type of a structured-constant shall be the type denoted by the array-type-identifier, record-type-identifier, or set-type-identifier of the structured-constant. The type of an array-value, a record-value, or a set-value of either a structured-constant or a component-value shall be the type of the structured-constant or the component-value, respectively. The value denoted by a constant in a component-value shall be assignment-compatible with the type of the component-value. The structure of a value possessing a structured type shall be the structure of the structured type.

6.3.2.2 Array-values. The type of an array-value shall be an array type, and the array-value shall denote a value of that type.

```
array-value = "[" [ array-value-element
                  { ";" array-value-element } [ ";" ] ]
```

```
                        [ array-value-completer [ ";" ] ] "]" .

     array-value-completer = "otherwise" component-value .

     array-value-element = case-constant-list ":" component-value .
```

The type of a component-value of either an array-value-element or an array-value-completer of an array-value shall be the component type of the array type of the array-value.

The values denoted by the case-ranges of the case-constant-lists of the array-value-elements of an array-value shall be distinct and shall belong to the set of value determined by the index type of the array type possessed by the array-value. Every component of an array-value shall be a value, as specified by one of the following two statements:

(a) The component mapped to be each value denoted by a case-range of a case-constant-list of an array-value-element of the array-value shall be the value denoted by the component-value of the array-value-element.

(b) Any component not mapped to by a value denoted by a case-range of a case-constant-list of an array-value-element of the array-value shall be the value denoted by the component-value of the array-value-completer of the array-value; there shall be at least one such component if and only if there is an array-value-completer in the array-value.

6.3.2.3 Record-values. The type of a record-value shall be a record type, and the record-value shall denote a value of that type.

```
     record-value = "[" field-list-value "]" .

     field-list-value =
               [ ( fixed-part-value [ ";" variant-part-value ]
                 ! variant-part-value ) [ ";" ] ] .

     fixed-part-value = field-value { ";" field-value } .

     field-value = field-identifier { "," field-identifier }
                   "=" component-value .

     variant-part-value = "case" [ tag-field-identifier "=" ]
                          tag-value ":" "[" field-list-value "]" .

     tag-value = constant .
```

3.0 FEATURE DESCRIPTION
3.2.1.8 Structured Constants

tag-field-identifier = field-identifier .

The occurrence of a record-value shall constitute the defining-point of each of the field-identifiers of the record type of the record-value as field-identifiers associated with the components of the record-value for each region that is a field-identifier closest-contained by the record-value. The component associated with each field-identifier in a field-value shall be the value denoted by the component-value of that field-value. The type of the component-value of a field-value shall be the type of each of the components that are components of the record type of the record-value closest-containing the field-value, and that are associated with the field-identifiers of the field-value.

NOTE: Consequently, all field-identifiers in a field-value must have been declared to have the same type.

Each field-identifier in a field-value of a fixed-part-value of a field-list-value that corresponds to a field-list shall denote a field of the field-list. The field-list-value of a record-value shall correspond to the field-list of the record type possessed by the record-value. The fixed-part-value or variant-part-value of a field-list-value shall correspond to the fixed-part or variant-part respectively, of the field-list corresponding to the field-list-value. The constant of a tag-value of a variant-part-value shall denote a value belonging to the set of values determined by the type denoted by the tag-type of the variant-selector of the variant-part corresponding to the variant-part-value. The field-list-value of a variant-part-value shall correspond to the field-list of the variant corresponding to the value of the constant of the tag-value of the variant-part-value; the selector component of the variant-part-value shall be a value that is associated with that variant. A tag-field-identifier in a variant-part-value shall be the field-identifier associated with the selector of the variant-part corresponding to the variant-part-value; the component of the variant-part-value associated with the field-identifier shall be the selector of the variant-part and shall be the value denoted by the tag-value of the variant-part-value. The field-identifier, if any, associated with the selector of a variant-part shall have an applied occurrence in the tag-field-identifier of each variant-part-value corresponding to the variant-part.

Each field-identifier associated with a component of a field-list shall have exactly one applied occurrence as a field-identifier for each field-list-value that corresponds to the field-list, and the field-list-value shall closest-contain the field-identifier.

NOTE: Consequently, every component of the record-value, including each active variant, must be defined. Also, a field-identifier cannot be specified more than once in a record-value.

6.3.2.4 Set-values. The type of a set-value shall be a set type, and the set-value shall denote a value of that type.

```
set-value = set-constructor .
```

An expression in a member-designator of the set-constructor (see 6.7.1) of a set-value shall contain only unsigned-constants and signs. The type of the set-constructor of a set-value shall be compatible with the type of the set-value.

6.3.3 Example of constant-definition-part

NOTE: The type-identifiers sieve, vector, quiver, PunchedCard, and and Polar are defined in 6.4.7.

```
const
   unity = 1.0;
   SmallPrimes = sieve[2,3,5,7,11,13,17,19];
   ZeroVector = vector[1..limit: 0.0];
   UnitVector = vector[1:unity otherwise 0];
   ZeroQuiver = quiver[otherwise ZeroVector];
   BlankCard = PunchedCard[1..80: ' '];
   blank = ' ';
   Unit = polar[r=1; theta=0];
   Origin = polar[r,theta=0.0];
   thrust = 5.3; theta = -2.0;
   warp = polar[r=thrust; theta=theta];
```

6.3.4 Component-constants

6.3.4.1 General. A component-constant shall denote a component of a value.

```
component-constant = indexed-constant
                   ! field-designated-constant .

constant-access = component-constant ! constant-identifier .
```

The value and type of a constant-access shall be the value and type respectively, of the component-constant or the constant-identifier of the constant-access.

NOTE: Field-designated and indexed constants are not defined to be constants themselves. Hence they cannot be used in defining other

constants.

6.3.4.2 Indexed-constants. A component of a value possessing an array type shall be denoted by an indexed-constant.

```
Indexed-constant = array-constant "[" index-expression
                                  { "," index-expression } "]" .

array-constant = constant-access .
```

An array-constant shall be a constant-access possessing an array type. For an indexed-constant closest-containing a single index-expression (see 6.5.3.2), the value of the index-expression shall be assignment-compatible with the index type of the array type. The component denoted by the indexed-constant shall be the component that corresponds to the value of the index-expression by the mapping of the array type (see 6.4.3.2).

If the array-constant is itself an indexed-constant an abbreviation may be used. In the abbreviated form, a single comma shall replace the sequence][that occurs in the full form. The abbreviated form and the full form shall be equivalent.

The order of evaluation of the index-expressions of an indexed-constant shall be implementation-dependent.

Examples
```
UnitVector[limit]
BlankCard[1]
```

6.3.4.3 Field-designated-constants. A field-designated-constant either shall denote a component of the value denoted by the record-constant of the field-designated-constant, and the component shall be the one associated with the field-identifier of the field-specifier (see 6.5.3.3) of the field-designated-constant by the record-type possessed by the record-constant; or else shall denote the value denoted by the constant-field-identifier (see 6.8.3.10) of the field-designated-constant.

The occurrence of a record-constant in a field-designated-constant shall constitute the defining-point of the field-identifiers associated with components of the record type possessed by the record-constant, for the region that is the field-specifier of the field-designated-constant.

```
field-designated-constant =
            record-constant "." field-specifier
          ! constant-field-identifier .
```

```
record-constant = constant-access .
```

A record-constant shall be a constant-access possessing a record type.

It shall be an error to denote a component of a variant, unless the variant is active.

Examples
```
origin.r
origin.theta
unit.theta
```

In section 6.4.1 [p12 ISO standard] replace the production rule for structured-type-identifier with the following:

```
structured-type-identifier = array-type-identifier |
                             record-type-identifier |
                             set-type-identifier |
                             file-type-identifier .

array-type-identifier = type-identifier .

record-type-identifier = type-identifier .

set-type-identifier = type-identifier .

file-type-identifier = type-identifier .
```

In section 6.4.1, replace the phrase "a structured-type-identifier" "an array-type-identifier, a record-type-identifier, a set-type-identifier, a file-type-identifier".

In section 6.4.7, [p20 ISO standard] add the following definitions:

```
quiver = array[1..10] of vector;
sieve = set of 1..20;
```

In section 6.7.1 [p37 ISO standard], change the production for factor to be as follows:

```
factor = variable-access | unsigned-constant | set-constructor
       | function-designator | "(" expression ")"
       | "not" factor | component-constant .
```

In section 6.8.3.10 [p48 ISO standard], replace the production rules and first paragraph with the following:

```
with-statement = "with" record-list "do" statement .

record-list = record { "," record } .

record = record-variable ! record-constant .

field-designator-identifier = identifier .

constant-field-identifier = identifier .
```

A with-statement shall specify the execution of the statement of the with-statement. The occurrence of a record-variable or record-constant as the only record in the record-list of a with-statement shall constitute the defining-point of each of the field-identifiers associated with components of the record-type possessed by the record-variable or record-constant as a field-designator-identifier or constant-field-identifier, respectively, for the region that is the statement of the with-statement; each applied occurrence of a field designator-identifier or constant-field-identifier shall denote that component of the record-variable or record-constant, respectively, that is associated with the field-identifier by the record type. The record-variable, if any, shall be accessed before the statement of the with-statement is executed, and that access shall establish a reference to the variable during the entire execution of the statement of the with-statement.

3.2.1.9 Additional Directive

In section 6.1.4 [p9 of ISO standard] after the production for directive, add the sentence:

> Pascal provides one additional directive, EXTERNAL. This directive indicates that the procedure or function is external to the Pascal program.

3.2.1.10 Additional Predefined Routines

In addition to those predefined routines required by the standard, Pascal will supply the following:

Routine Name	Argument Type	Result Type	Description
CARD(s) function	Set	integer	Returns number of elements present in set.

3.0 FEATURE DESCRIPTION
3.2.1.10 Additional Predefined Routines

CLOCK function	none	integer	Returns current elapsed time in milliseconds.
DATE(a) procedure	string len=8	none	Returns current date in (a). Format: mm/dd/yy .
HALT(a) procedure	string	none	Terminates the program, copies (a) to job_log.
INDEX(a,b) function	string or char, string or char	integer	Returns the index of the position in expression (a) that contains expression (b) as a substring. If (b) is not contained in a, INDEX returns 0. If b is the null string, INDEX returns 1.
LENGTH(a) function	string or char	integer	Returns the length of expression a.
MAXLENGTH(a) function	string or char	integer	Returns the maximum length of variable access a.
MESSAGE(a) procedure	string	none	Copies (a) to job_log.
SUBSTR(a,b,c) (a,b[,c]) function	string or char, integer expr, integer expr	string	Returns a variable string as follows: If c=0, then the null string. If c is > 0 then the string returned is defined to be: a[b..(b+c-1)]. If c is omitted, the string returned is defined to be: a[b..LENGTH(a)]. If LENGTH(a)< b+c-1,

			it shall be an error.
TIME(a) procedure	string len=8	none	Returns current time in (a). Format: hh:mm:ss .

3.2.2 IMPLEMENTATION DEFINED AND IMPLEMENTATION DEPENDENT FEATURES

Implementation defined and implementation dependent features of Pascal are addressed in both standards.

3.2.2.1 Implementation Defined Features

Implementation defined features may differ between processors, but will be defined by all processors. Below are the implementation defined features of Pascal and the Pascal 180 implementation:

Implementation Defined Feature	Pascal 180 Implementation
Subset of characters which can occur in char/string literal	The ASCII character set.
The ordinal representation of char values.	The ASCII ordinals.
Subset of real numbers allowed.	-4.8e1234 thru -5.2e-1232 0 4.8e-1234 thru 5.2e1232
Value of maxint.	9223372036854775807
Number of characters for exponent field on output.	6, [E+dddd or E-dddd].
Symbol denoting exponent on output ('e' or 'E')	'E', upper case.
Case for Boolean values on output.	Upper case, [TRUE or FALSE]
Default values for total width on integer, Boolean and real output.	Integer: 20 Boolean: 5 Real : 22 (applies only

	to floating point)
Point of actual action and checking of assertions on I/O routines.	Pre-assertions will be checked prior to execution of the code necessary to perform the specified I/O. Post-assertions will be true after the code necessary to perform the specified I/O action has been executed.

Effect of page on textfile.	page(f) is equivalent to:
	writeln(f);
	write(f,'1');

Effect of reset/rewrite to predefined files INPUT and OUTPUT.	Reset on INPUT has the effect described in both standards section 6.6.5.2 except that after reset on interactive files, the file buffer variable will not contain the first character from the interactive file. Rewrite on OUTPUT has no real effect, leaving the associated file $ASIS.
Binding of file type program parameters.	Bound at run time to external files. External file names present on the execution (lgo) statement are associated with the corresponding program parameter on the program statement. If no such argument exists on the lgo statement, the files will be referenced as named on the program statement. e.g., program xxx(output); ... lgo,stuff would associate the file STUFF with every access

	to OUTPUT in program xxx.

3.2.2.2 Implementation Dependent Features

Implementation dependent features may differ between processors and need not be defined by any given processor. Below are the implementation dependent features of Pascal and the Pascal 180 implementation:

Implementation Dependent Feature	Pascal 180 Implementation
Order of evaluation of index expressions.	Left to right.
Order of evaluation of set member designators.	Left to right.
Order of evaluation of operands of dyadic operators.	Left to right, complete evaluation.
Order of evaluation, access and binding of actual parameters to functions and procedures.	Order of evaluation: left to right. Access is from the stack. Binding is at runtime.
For assignment statements, order of: evaluate access of variable, evaluate expression.	Evaluate expression, then evaluate variable access.
Effect of inspecting a textfile to which page has been applied	The character '1' will appear in the first column of the line to which the procedure page has been applied.
Binding of non-file type program parameters.	Bound at run time. Actual parameters must be SCL values. Type of formal parameters are restricted to integer, Boolean and string. Formal program parameters will be treated in the program as global

	variables. Missing actual parameters will result in no initialization of the corresponding program parameter. The formal parameters are treated in an analogous manner to value parameters of a procedure or function.

3.2.3 ERRORS NOT DETECTED

The ISO Standard defines an error to be a violation of the standard (typically requiring execution of the program to detect) which the processor is not required to detect. Conformance to the standard does, however, require a documented list of errors not detected. This list will be found in section 5.9.

3.2.4 DIFFERENCES FROM THE PREDECESSOR PRODUCT

Following is a list of extensions available in the predecessor product (170 Pascal) which are not present in Pascal 180, with justifications:

3.2.4.1 Compiler_Directives

Compiler directives (pragmas) will not be supported at first release. It is not clear at this juncture what subset of the predecessor product's directives should be implemented, time constraints dictate not implementing the extension at this time, and the ANSI committee is currently producing an extension proposal covering such directives, and Pascal 180 should probably follow the syntax of the proposed new standard.

3.2.4.2 Segmented_File_Operations

The segmented file operations will not be implemented, as the type of extension does not readily translate into something useful on NOS/VE. The predefined routines GETSEG, PUTSEG and EOS will not be implemented, nor will the extended form of REWRITE, (e.g. REWRITE(f,n)).

3.2.4.3 Type_ALFA

Type ALFA will not be implemented. The type is machine dependent (allows a string type which is one Cyber 170 word), and would not allow programs to transfer from 170 and maintain the meaning.

3.2.4.4 Other_Predefined_Routines

The predefined functions EXPO and UNDEFINED will not be implemented. EXPO seems to have little use and UNDEFINED would have such limited usefulness in the NOS/VE environment as to make it virtually worthless. The extended form of TRUNC, i.e., TRUNC(A,N) will not be implemented.

3.2.4.5 FORTRAN_and_EXTERN_directives

The FORTRAN directive will not be implemented. The EXTERN directive is replaced by the EXTERNAL directive.

3.2.4.6 VALUE_Initialization_Part

The value initialization part differs in that structured variables are initialized using structured constants as defined in 3.2.1.8 . This form is part of the ANSI candidate extension library and is likely to become standard. It provides the same functionality as the predecessor extension, but the syntax is entirely different.

3.2.5 PROCESSOR LIMITATIONS

The following are the known limitations of Pascal, in addition to any limits imposed by NOS/VE.

3.2.5.1 Arrays

The maximum length allowed for an array variable is 268435456 bytes.

3.2.5.2 Sets

Type Set is limited to positive values of ordinal types, with ordinal range 0..255 .

3.2.5.3 Identifier_Length

This is not a true restriction, merely a caution. Use of identifiers which are greater than 31 characters will result in a warning diagnostic. There may be interface problems with such identifiers, especially should two such be spelled alike in the first 31 characters. Attribute and reference listings only use the first 31 characters, thus similar spellings may result in entries spelled the same (with different information). Debug will have trouble with identifiers spelled alike in the first 31 characters, i.e., only one of such identifiers will be accessable. The system interface limits names to 31 characters, thus program

parameter files and "external" procedures/functions with names longer than 31 characters may encounter problems.

3.2.5.4 Source_Input

The length of source input lines is limited to 255 characters.

3.2.5.5 Interactive_Input

The length of interactive input lines is limited to 150 characters.

3.2.5.6 String_Length

The length of a string expression is limited to 65535 characters.

3.2.5.7 Number_of_Files_Open

The number of files open by any task is limited to 100. This is a NOS/VE limitation. Should that limitation change, Pascal can support more open files.

3.3 INTERFACES

User interface to Pascal 180 will be SIS conforming. Below are listed the control statement parameters accepted by Pascal:

Parameter	Alias	Description
INPUT	I	Input file. I = <file> This parameter specifies the source input file name to Pascal. I=$NULL will result in a job log diagnostic and termination of compilation. Single specified value parameter. Default: I=$INPUT
BINARY	B	Binary Object Code output file. B = <file>

This parameter specifies the file to contain the object code produced by Pascal.

B=$NULL indicates no object file is to be output.

Single specified value parameter.
Default: B=$LOCAL.LGO

LIST L Listing file.

L = <file>

This parameter specifies the file to which Pascal will write the source listing, diagnostics, object listing, statistics and reference/attributes.

L=$NULL results in no listings output.

Single specified value parameter.
Default: L=$LIST

ERROR E Error file.

E = <file>

This parameter specifies the file to which Pascal will write the text of diagnostics, of EL level or higher. Diagnostics are also written to the L file, if present. If E and L name the same file, only one copy of the diagnostic will be output.

E=$NULL results in no error file.

Single specified value parameter.
Default: E=$ERRORS

ERROR_LEVEL EL Error level.

EL = <option>

This parameter indicates the severity level of diagnostic which will be

output by Pascal.

W Warning level and higher diagnostics are output.

F Fatal level diagnostics only are output.

Single specified value parameter.
Default: EL=W

LIST_OPTIONS LO Listing options.

LO = <option list>

The options of this parameter specify the information which is to appear on the L file. Multiple options may be specified.

A Attributes. A list of the attributes of each entity in the program.

O Object listing. A listing of the object program with assembler mnemonics.

R Cross reference listing. A listing which shows definition and uses of all entities of the program.

S Source listing. Source listing of the program.

LO=NONE causes no listing options to be selected.

Multiple option parameter.
Default: LO=S.

OPTIMIZATION_LEVEL OL Optimization level.

OL = <option>

This parameter controls the style of code generated by Pascal.

DEBUG Object code is stylized to facilitate debugging. Instructions are grouped to correspond to source statements.

LOW Production quality code, but not highly optimized.

Single specified value parameter.
Default: OL=LOW

DEBUG_AIDS DA Debugging aids option.

DA = <option list>

This parameter specifies the debug options to be selected. Multiple options may be selected.

DT Debug tables. Generate line number and symbol tables as part of the object code.

Multiple option parameter.
Default: DA=NONE

RUNTIME_CHECKS RC Runtime checks code generation.

RC = <option list>

This parameter controls which runtime checks will be compiled into the object program. Multiple options may be selected.

F Files checking. Selects checking of errors involving file variables and buffer variables.

N Pointer checking. Selects checking of misuse of pointer variables and invalid usage of new and dispose procedures.

R Range checks. Selects range checking for subrange and set assignments and case variables.

S Subscript checks. Selects array subscript bound checking.

RC =ALL causes selection of all checks.

Multiple value specified parameter.
Default: RC=NONE

STANDARDS_DIAGNOSTICS SD Standards diagnostics.

SD = (level, standard)

This parameter specifies whether use of non-standard extensions in a program are to be diagnosed. The first option defines the error level to be assumed by such diagnostics and the second option determines which of the two standards is to apply.

Level:

W Standard errors result in warning errors.

F Standard errors result in fatal errors.

Standard:

ANSI The ANSI standard is the basis.

ISO The ISO standard is the basis.

SD = NONE causes standards errors not to be diagnosed.

SD = F (or W) causes the ISO standard to be the basis.

Multiple specified value parameter.
Default: SD=NONE

TERMINATION_ERROR_LEVEL TEL Termination error level.

TEL = <option>

This parameter indicates the severity

level of diagnostic which will cause Pascal to return an abnormal STATUS.

W Warning level and higher diagnostics cause abnormal STATUS.

F Fatal level diagnostics only cause abnormal STATUS.

Single specified value parameter.
Default: TEL=F

STATUS None Status variable.

STATUS = <status-variable>

This parameter specifies the name of the SCL status variable to be set by Pascal to indicate the occurrence of error conditions.

Single specified value parameter.
Default: No status information is returned.

3.4 ABORTS AND RECOVERY

3.4.1 COMPILE TIME

Detection of errors of EL or above (see section 3.3) will result in setting the STATUS variable. Action taken at that point is a user responsibility.

3.4.2 RUN TIME

Detection of an error at run time will result in the output of a diagnostic message, raising of an exception condition and termination of execution. If DEBUG or an abort-file is in force, the user will have that facility available.

3.5 PERFORMANCE

3.5.1 REQUIREMENTS

Performance specifications are provided in the Pascal 180 Design Requirements [reference 6].

3.5.2 PERFORMANCE TUNING

Users will get the best performance by selecting the options OL=LOW, DA=NONE and RC=NONE. These are currently the default settings for Pascal.

4.0 PRODUCT-LEVEL DESCRIPTION

Procedure and function calls in Pascal 180 are implemented in accordance with the SIS, section 5.2.8.

NOTE: Section 5.2.8 does not currently reflect the situatation as pertains to Pascal. This is the subject of DAP S4960.

5.0 ERRORS

5.0 ERRORS

5.1 CATASTROPHIC DIAGNOSTICS (COMPILE TIME)

Pascal 180 will provide the following catastrophic compile time diagnostics. Texts are listed below. These diagnostics will be output to the source listing (as well as the error file) following the source line which contained the error. Format of the diagnostics will be as follows:

CATASTR. *ERROR* {column number} (PA diag#) text

592021 Nature or severity of syntax error(s) prevents semantic analysis.

5.2 FATAL DIAGNOSTICS (COMPILE TIME)

Pascal 180 will provide the following fatal compile time diagnostics. Texts are listed below. These diagnostics will be produced in the same manner as the catastrophic diagnostics and will have the following format:

FATAL *ERROR* {column number} (PA diag#) text

590050 The CASE constant {sign} {constant} does not match the CASE constants in the variant record.
590051 The CASE constant {constant} does not match the CASE constants in the variant record.
590052 Long form of {procedure identifier} can only be used with pointers to variant records and dynamic strings.
590053 Long form of {procedure identifier} can only be used with pointers to variant records.
590054 Built-in procedure {procedure identifier} called in a functional context.
590055 Built-in function {function identifier} can only be called in expressions.
590056 Built-in {function identifier} expects {number} arguments but has been called with {number}.
590057 Built-in {procedure identifier} does not accept arguments of the form 'expr:expr' or 'expr:expr:expr'.
590058 ABS built-in function can be applied only to integers or reals.
590059 Math built-in function {function identifier} accepts only REAL and INTEGER arguments.
590060 The argument of CARD must be a set.
590061 CHR built-in function can be applied only to INTEGER

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

arguments.

590062 String argument to {function identifier} of length {number} is invalid. String must be of length 8 or greater.

590063 The argument of {function identifier} must be a string.

590064 Procedure {procedure identifier} must have at least one argument.

590065 First argument to procedure {procedure identifier} must be of type POINTER.

590066 Predefined file INPUT was not declared as a program parameter.

590067 Argument to EOF or EOLN must be a file.

590068 EOLN function can only be applied to files of type TEXT.

590069 The functions EOF and EOLN cannot have more than one argument.

590070 Argument of {procedure identifier} must be a file.

590071 Argument number {number} to INDEX function must be of type STRING or CHAR.

590072 LENGTH and MAXLENGTH functions can only be applied to STRING arguments.

590073 String length must be specified when allocating a dynamic string.

590074 String size of {number} is invalid when allocating a dynamic string. String size must be in the range of 1 to 65535.

590075 Size argument for dynamic string allocation must be of type INTEGER.

590076 Dynamic string allocation requires one size argument but {number} were found.

590077 ODD built-in function can be applied only to INTEGER arguments.

590078 ORD built-in function can be applied only to ordinal arguments.

590079 The first argument of PACK must be an unpacked array.

590080 The third argument of PACK must be a packed array.

590081 The second argument of PACK must be of an ordinal type that is compatible with the index type of the first argument.

590082 The array arguments of PACK must have the same component type.

590083 Predefined file OUTPUT was not declared as a program parameter.

590084 Argument to PAGE must be a file.

590085 The procedure PAGE can only be applied to files of type TEXT.

590086 The procedure PAGE cannot have more than one argument.

590087 PRED built-in function can be applied only to ordinal arguments.

590088 Argument of REWRITE must be a file.

590089 ROUND built-in function can be applied only to REAL

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

arguments.

590090 SQR built-in function can be applied only to REAL or INTEGER arguments.

590091 SUCC built-in function can be applied only to ordinal arguments.

590092 SUBSTR function requires either 2 or 3 arguments but {number} were found.

590093 Substring start position of {number} is invalid. The start position must be >= 1.

590094 String size argument to SUBSTR function must be of type INTEGER.

590095 Start position argument to SUBSTR function must be of type INTEGER.

590096 Substring length of {number} is invalid. Substring length must be >= 0.

590097 Substring length can only be 0 or 1 for an argument of type CHAR.

590098 Substring with start position of {number} and length of {number} exceeds the length of the string.

590099 First argument to SUBSTR must be of type STRING or CHAR.

590100 TRUNC built-in function can be applied only to REAL arguments.

590101 File was not specified and predefined file {file identifier} was not declared as a program parameter.

590102 The procedures READ and WRITE must have at least one non file parameter.

590103 The procedures READLN and WRITELN can only be applied to textfiles.

590104 The second argument of UNPACK must be an unpacked array.

590105 The first argument of UNPACK must be a packed array.

590106 The third argument of UNPACK must be of an ordinal type that is compatible with the index type of the second argument.

590107 The array arguments of UNPACK must have the same component type.

590108 An active FOR control variable cannot be read into.

590109 Bound identifier {identifier} cannot be read into.

590110 Argument number {number} to READ is not assignment compatible with the component type of the file variable.

590111 Argument number {number} to WRITE is not assignment compatible with the component type of the file variable.

590112 The maximum number of format specifications is 2 -- found {number} format specifications.

590113 Argument number {number} is not an acceptable type for textfile output. Valid types are INTEGER, REAL, BOOLEAN, CHAR and STRING.

590114 Argument number {number} is not an acceptable type for textfile input. Valid types are INTEGER, REAL and CHAR.

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

590115 TotalWidth format specifier must be of type INTEGER.
590116 FracDigits format specifier must be of type INTEGER.
590117 TotalWidth:FracDigits format can only be used with type REAL.
590170 All conformant array actual parameters in a section must be of the same type.
590171 All variable string actual parameters in a section must be of the same type.
590172 It is invalid to use PROGRAM name {identifier} as a procedure or function.
590173 The identifier {identifier} has been used in a procedure or function call, but is unknown in the current scope.
590174 The identifier {identifier} has been used in a procedure or function call, but is neither.
590175 Procedure {procedure identifier} was called in a functional context.
590176 {identifier} is a function and can only be called in expressions.
590177 For procedure or function {identifier}, the number of actual parameters, {number}, does not match the number of formal parameters, {number}.
590178 Parameter number {number}: the parameters of {procedure identifier} cannot be 'expr:expr' or 'expr:expr:expr'.
590179 Parameter number {number}: an actual parameter cannot be a conformant array for a value formal parameter.
590180 Parameter number {number}: actual parameter not of same type as VAR formal parameter.
590181 The function {function identifier} was never assigned a return value.
590182 A conformant array's index must be of some ordinal type.
590183 The component of a packed conformant array cannot be another conformant array.
590184 Function result type must be simple or pointer type.
590185 The indefinite string type can only be used as a variable parameter not a value parameter.
590186 TYPE {identifier} cannot be used as a value parameter. Value parameters cannot be files or structured types which contain files.
590187 Identifier {identifier} is not a legal directive. The options are FORWARD and EXTERNAL.
590188 Procedure or function {identifier} resolves a FORWARD directive, but contains a parameter list or a return type. The declaration closest-containing the FORWARD directive is at line {line number}.
590189 Procedure or function {identifier} resolves a FORWARD directive, but does not include the required block. The declaration closest-containing the FORWARD directive

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

is at line {line number}.

590190 Function {function identifier} is required to have a return type.

590191 {identifier} has an unacceptable type; a program parameter must be a BOOLEAN, an INTEGER, a STRING, or a FILE.

590192 Program parameter {identifier} has not been declared as a variable.

590193 Only an array or a string variable can be passed to a conformant array parameter.

590194 Formal and actual array parameters have incompatible index types.

590195 Packing of the actual and formal parameter do not match.

590196 Formal and actual array parameters must have the same component type.

590197 Only a variable string may be the actual argument to formal parameter of the indefinite string type.

590198 The parameter list for actual parameter {number} does not match the parameter list of the formal parameter.

590199 The actual argument for a procedure parameter cannot be a function.

590200 Return types don't match for actual and formal function parameters.

590201 Undeclared identifier {identifier} is used as actual parameter.

590202 Illegal actual argument for procedure or function formal parameter.

590212 The identifier {identifier} is unknown in the current scope.

590213 Non-constant identifier {identifier} used in constant definition.

590214 The constant declaration of {identifier} contains a signed STRING, which is illegal.

590215 The constant declaration of {identifier} contains a signed CHAR, which is illegal.

590216 The constant declaration of {identifier} contains a signed BOOLEAN, which is illegal.

590217 The constant declaration of {identifier} contains a signed enumerated constant, which is illegal.

590218 The INTEGER constant {number} is greater than MAXINT (9223372036854775807).

590219 The REAL constant {real string} is out of range.

590220 REAL constant used where an ordinal is required.

590221 String constant used where an ordinal is required.

590222 REAL constant identifier {identifier} used where an ordinal is required.

590223 String constant identifier {identifier} used where an ordinal is required.

590224 Non-constant identifier {identifier} used where an

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

ordinal constant is required.
590225 Ordinal constants other than those of type INTEGER cannot be signed.
590226 {identifier} must be type identifier.
590227 {identifier} must be a set type identifier.
590228 {identifier} must be a set or record type identifier.
590229 {identifier} must be an array type identifier.
590230 {identifier} must be a record type identifier.
590236 The CASE constant {constant} is not in the range of the selector type.
590237 The type of the CASE constant is not compatible with the type of the selector.
590238 Cannot duplicate CASE constant values; the offending CASE constant is {identifier}.
590239 CASE constants exceed cardinality of selector type.
590240 CASE constants do not exhaust selector type and there is no OTHERWISE.
590241 An array's index must be of some ordinal type.
590242 The array has too many elements.
590243 CASE selector must be of an ordinal type.
590244 The type {identifier} has been used in its own definition.
590245 The identifier {identifier} has been used as a type, but is unknown is the current scope.
590246 The identifier {identifier} has been used as a type, but is not a type.
590247 The component type of a file may not be another file or a structured type with a component of type file.
590248 The lower and upper bounds of a range must be of the same type.
590249 The bounds of a subrange must be of some ordinal type.
590250 A subrange's lower bound must be less than or equal to its upper bound.
590251 The base type of a set must be an ordinal type.
590252 The base type for a set of integers must be within 0..255.
590253 The scheme identifer in a valable string declaration must be the required identifier STRING.
590254 The maximum length of a variable string must be of type INTEGER.
590255 The maximum length of a variable string must be greater than or equal to 1.
590256 The maximum length of a variable string must be less than or equal to 65535.
590265 {identifier} which is an entire structured operand is incompatable with operator {operator symbol}.
590266 The left operand is incompatible with operator {operator symbol}.
590267 The right operand is incompatible with operator {operator symbol}.

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

590268 Operator {operator symbol} is not defined for the operand.
590269 The right operand which is of type {identifier} is incompatible with operator {operator symbol}.
590270 The left operand which is of type {identifier} is incompatible with operator {operator symbol}.
590271 The operands with types {identifier} and {identifier} are incompatible with {operator symbol}.
590272 Operator IN: the ordinal type of the left operand is not compatible with the base type of the right operand.
590273 The ordinal operands are incompatible with operator {operator symbol}.
590274 The pointer operands must be of the same type for operator {operator symbol}.
590275 The set operands must have compatible base types for operator {operator symbol}.
590276 The set operands must both be packed or unpacked for operator {operator symbol}.
590277 Length of concatenated string is {number} but the maximum string length allowed is 65535.
590278 Constant expression out of range.
590279 Constant subscript out of range.
590280 Substring index must be >= 1.
590281 Lower substring index, {number} must be less than or equal to upper substring index which is {number}.
590282 Upper substring index, {number} must be less than or equal to maximum length of the string which is {number}.
590283 The constant value NIL cannot be signed.
590284 String constants cannot be signed.
590285 Identifier {identifier} is unknown in the current scope.
590286 Ordinal constants cannot be signed.
590287 Identifier {identifier} is not a string constant.
590288 The identifier {identifier} is not a constant.
590289 Elements of a set of INTEGER must be in the range 0..255. {number} is out of this range.
590290 The elements of a set constructor must all be of the same ordinal type.
590291 The elements of a set constructor must of some ordinal type.
590292 Expression components must be constants, variables, or function calls. Identifier {identifier} is none of these.
590293 An active FOR control variable cannot be assigned to or passed as a VAR parameter.
590294 Bound identifier {identifier} cannot be assigned to or passed as a VAR parameter.
590295 Procedure {procedure identifier} cannot be the target of an assignment.
590296 Tried to assign to function {function identifier} outside of the function.

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

590297 Non-variable {identifier} used as a variable access.
590298 The dereference operator, ^, can only be applied to pointers and files.
590299 Field qualification applied to a non-record.
590300 Cannot pass a field of a packed record as a VAR parameter.
590301 Field name {identifier} is not a field of the current record.
590302 Cannot pass tag field {identifier} as a VAR parameter.
590303 Substring index must be of type integer.
590304 Substring notation can only be applied to a fixed or variable length string.
590305 A substring cannot be passed as a VAR parameter.
590306 Only arrays or strings can be subscripted.
590307 Cannot pass an element of a packed array as a VAR parameter.
590308 STRING index must be of type INTEGER.
590309 An array subscript must be of some ordinal type.
590310 The subscript type is not compatible with the array index type.
590311 Variable access required. Expressions are not allowed in this context.
590336 Multiple definition of {identifier}; the current definition replaces the one at line {line number}.
590337 {identifier} has a use in this scope before its definition.
590348 Line length exceeded 255 characters, line ignored.
590358 Target array type is not compatible with value type.
590359 Target ordinal type is not compatible with value type.
590360 Target POINTER type is not compatible with value type.
590361 Target REAL type is not compatible with value type.
590362 Target RECORD type is not assignment compatible with the value's type.
590363 Target string length of {identifier} is less than source string length of {identifier}.
590364 Target string type not compatible with value type.
590365 Base of target set type is not compatible with base of value set type.
590366 Target set type is not compatible with value type.
590367 Target identifier is not valid for assignment.
590368 Files or structured types with file components cannot be used in assignment statements or as value parameters.
590369 The FORWARD directive for procedure or function {identifier} is not resolved in the current scope.
590370 The label {number} must appear on a statement in the current block.
590371 The elements of a set constructor must be of the same ordinal type as the base type of the set variable.
590372 The set element with ordinal value {number} is outside

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

	the set range.
590373	A set variable can only be initialized by a constant set constructor or a set constant.
590374	Variable can only be initialized with constants, constant set constructors and the word symbol NIL.
590375	The value {number} is outside the declared range of the variable {identifier}.
590376	The ordinal variable is not compatible with the value type.
590377	POINTER variables can only be initialized with the value NIL.
590378	A REAL variable can only be initialized by a constant REAL or constant INTEGER.
590379	A string variable cannot be initialized by a string constant of greater length.
590380	A string variable can only be initialized by a constant string value.
590381	Set assignment is invalid, both operands must be packed or unpacked.
590382	The FOR control variable, initial value, and final value must be of compatible ordinal types.
590383	The FOR control variable {identifier} must be declared in the current block.
590384	The FOR control variable {identifier} must not be the control variable of a containing FOR statement.
590385	The FOR control variable {identifier} must not be a formal parameter.
590386	The FOR control variable {identifier} is threatened by a statement in a procedure at the same level.
590387	The FOR control variable {identifier} is unknown in the current scope.
590388	Non-variable {identifier} used as a FOR control variable.
590389	Target label {number} of GOTO was not declared.
590390	The label {number} is not accessible from this GOTO statement.
590391	The expression in an IF statement must be of type BOOLEAN.
590392	{identifier} is a formal parameter and cannot be initialized.
590393	The variable {identifier} has already been initialized.
590394	Only varaibles of type BOOLEAN, INTEGER, REAL, POINTER, SET, ORDINAL and STRING can be initialized.
590396	Only variables can be initialized. {identifier} is not a variable.
590397	Only entire variables can be initialized.
590398	The label {number} is not in the range 0..9999.
590399	Label {number} was not declared in the current block.
590400	Label {number} has already been used on line {identifier}.
590401	Label {number} is not accessible to previous GOTO(s) that referenced it.

5.0 ERRORS
5.2 FATAL DIAGNOSTICS (COMPILE TIME)

590402 The expression in an UNTIL must be of type BOOLEAN.
590403 CASE constant is not compatible with the CASE index expression.
590405 The CASE index expression must be of some ordinal type.
590406 The expression in a WHILE statement must be of type BOOLEAN.
590407 Non-record used in a WITH list.
590408 The argument of CARD must be a set.
590500 Syntax error. {text describing the error.}
590501 Unexpected character.
590502 Unexpected end-of-line. String delimeter missing.
590503 Unexpected end-of-file. String delimeter missing.
590504 Syntax error. Unexpected end-of-file.

5.3 WARNING DIAGNOSTICS (COMPILE TIME)

Pascal 180 will provide the following warning compile time diagnostics. Texts are listed below. These will be produced in the same manner as the other compile time diagnostics and will have the following format:

WARNING *ERROR* {column number} (PA diag#) text

590338 Identifier {identifier} is longer than 31 characters. May cause interface problems.
590404 The CASE constant {constant} is not in the range of the case-index.

5.4 STANDARDS DIAGNOSTICS (COMPILE TIME)

Standards diagnostics are selectable by control statement option. These will be produced in the same manner as the other compile time diagnostics and will have the following format:

{level} {column number} (PA diag#) *{standard}* text

594979 {ANSI/ISO}: Array constant is non-standard.
594980 {ANSI/ISO}: Record constant is non-standard.
594981 {ANSI/ISO}: Set constant is non-standard.
594982 {ANSI/ISO}: Range in case constant list is non-standard.
594983 {ANSI/ISO}: Substring notation is non-standard.
594984 {ANSI/ISO}: Use of dynamic strings is non-standard.
594985 {ANSI/ISO}: Use of variable string type is non-standard.
594986 {ANSI/ISO}: Use of fixed length string with upper bound of 1 is non-standard.
594987 {ANSI/ISO}: Predefined identifier {identifier} is

non-standard.
594988 {ANSI/ISO}: Input of string type from a textfile is non-standard.
594989 {ANSI}: Conformant-array parameters are non-standard.
594990 {ANSI/ISO}: Use of the null string is non-standard.
594991 {ANSI/ISO}: OTHERWISE is non-standard.
594992 {ANSI/ISO}: Use of concatenation operator is non-standard.
594993 {ANSI/ISO}: Comparison of STRING and CHAR is non-standard.
594994 {ANSI/ISO}: Comparison of unequal size strings is non-standard.
594995 {ANSI/ISO}: Assignment of STRING and CHAR is non-standard.
594996 {ANSI/ISO}: Assignment of unequal size strings is non-standard.
594997 {ANSI/ISO}: Duplication of declaration section is non-standard.
594998 {ANSI/ISO}: Relaxed order of declarations is non-standard.
594999 {ANSI/ISO}: The VALUE initialization part is non-standard.

5.5 INTERNAL_ERROR_DIAGNOSTICS_(COMPILE_TIME)

Internal errors are indications of a compiler failure, and should be reported as problems, using the procedures for the site where noted. The diagnostics will be produced in the same manner as the other compile time diagnsotics and will have the following format:

CATASTR. *ERROR* {column number} (PA diag#) text

592000 Internal error. Xref_context_index < 1.
592001 Internal error. Xref_context_index > 100.
592002 Internal error. BIFS: unimplemented built_in function or procedure - {identifier}.
592003 Internal error. Evaluate_integer_constant: Bad integer constant {constant}.
592004 Internal error. Evaluate_real_constant: bad constant {constant}.
592005 Internal error. Map_type_definition: unknown node name for type definition.
592006 Internal error. SPAN: type not boolean, char, integer, or ordinal.
592007 Internal error. There are not enough inserts for message.
592008 Internal error. There are too many inserts for message.
592009 Internal error. Insert_ordinal_value: the type was not an ordinal type.
592010 Internal error. Insert_ordinal_value: the value is not in the range of the type.
592011 Internal error. Insert_string: too many inserts.
592012 Internal error. EXPR: unexpected type action.

5.0 ERRORS
5.5 INTERNAL ERROR DIAGNOSTICS (COMPILE TIME)

592013 Internal error. EXPR: unexpected code action.
592014 Internal error. Code_range_check: non_ordinal type used.
592015 Internal error. Code_subscript_check: non_ordinal type used.
592016 Internal error. Primary: not prepared to cope with node. Node name = {number}.
592017 Internal error. Stack_base_addr: asked for lexical level is greater than current lexical level.
592018 Internal error. Create_debug_st_entry: unexpected DT entry kind.
592019 Internal error. Hcp$include_file: TWS has tried to do an include file.
592020 Internal error. Hcp$process_a_tree: should not be called for Pascal.
592022 Internal error. Process_statement: unknown statement type.
592023 Internal error. Get_file_name called with identifier <> variable or field.
592024 Internal error. Get_file_name called with wrong node.
592025 Internal error. No space for table allocation. Table name is {identifier}.

5.6 JOB_LOG_MESSAGES_(COMPILE_TIME)

Pascal 180 will provide the following diagnostics to the job log. Texts are listed below. These diagnostics are unnumbered and will be output to the job log only.

Message Texts

--WARNING-- Conflict use of parameter DEBUG_AIDS, options before NONE ignored.
--WARNING-- Conflict use of parameter LIST_OPTIONS, options before NONE ignored.
--WARNING-- Conflict use of parameter RUNTIME_CHECKS, options before NONE ignored.

5.7 JOB_STATUS_MESSAGES_(COMPILE_TIME)

Diagnostic Numbers and Texts
{information in brackets is variable}

590002 Pascal detected {error-level} error(s), TEL = {error-level} causes abnormal termination.
590004 INPUT file is $NULL, no compilation performed.
590005 The FILE_CONTENTS attribute for the INPUT file must be either UNKNOWN or LEGIBLE.
590006 The FILE_ORGANIZATION attribute for the INPUT file must

be SEQUENTIAL.
590007 The FILE_STRUCTURE attribute for the INPUT file be either UNKNOWN or DATA.
590008 The GLOBAL_ACCESS_MODE attribute for the INPUT file must contain READ.
590009 The FILE_CONTENTS attribute for the BINARY file must be either UNKNOWN or OBJECT.
590010 The FILE_ORGANIZATION attribute for the BINARY file must be SEQUENTIAL.
590011 The FILE_STRUCTURE attribute for the BINARY file be either UNKNOWN or DATA.
590012 The GLOBAL_ACCESS_MODE attribute for the BINARY file must contain MODIFY.
590013 The FILE_CONTENTS attribute for the LIST file must be either UNKNOWN or LEGIBLE or LIST.
590014 The FILE_ORGANIZATION attribute for the LIST file must be SEQUENTIAL.
590015 The FILE_STRUCTURE attribute for the LIST file be either UNKNOWN or DATA.
590016 The GLOBAL_ACCESS_MODE attribute for the LIST file must contain MODIFY.
590017 The FILE_CONTENTS attribute for the ERROR file must be either UNKNOWN or LEGIBLE or LIST.
590018 The FILE_ORGANIZATION attribute for the ERROR file must be SEQUENTIAL.
590019 The FILE_STRUCTURE attribute for the ERROR file be either UNKNOWN or DATA.
590020 The GLOBAL_ACCESS_MODE attribute for the ERROR file must contain MODIFY.
590021 STANDARDS_DIAGNOSTICS first option must be ERROR_LEVEL or NONE.
590022 STANDARDS_DIAGNOSTICS second option must be ANSI or ISO.
590023 STANDARDS_DIAGNOSTICS allowed only two options.
590024 STANDARDS_DIAGNOSTICS option conflict, options other then NONE ignored.

5.8 FATAL_DIAGNOSTICS (RUN_TIME)

Diagnostic Numbers and Texts
{information in brackets is variable}

595006 The value 0 is not allowed as the right operand to divide. !
595007 Attempted DISPOSE of a pointer variable with an invalid value: {address}.
595008 Attempted DISPOSE of a NIL pointer.
595011 Internal Error: Cannot allocate PDT on stack.
595012 Internal Error: Cannot allocate file table.
595013 Internal Error: Cannot allocate wsa in

5.0 ERRORS
5.8 FATAL DIAGNOSTICS (RUN TIME)

REWRITE for file {file}.

595014 Internal Error: Cannot allocate wsa in RESET for file {file}.

595018 EOF must be TRUE before PUT on file {file}.

595020 EOF must be true prior to use of PAGE on file {file}.

595022 EOF must be true prior to use of WRITE or WRITELN on file {file}.

595024 EOLN activated when EOF is TRUE for file {file} at line {line number}.

595026 Input of REAL number {number} from file {file} yields {math library} error.

595027 File {file} is undefined prior to use of GET.

595028 File {file} is undefined prior to use of PAGE.

595029 File {file} is undefined prior to use of PUT.

595030 File {file} is undefined prior to use of READ.

595031 File {file} is undefined prior to use of WRITE or WRITELN.

595033 File mode must be GENERATION prior to use of PAGE on file {file}.

595034 File mode must be GENERATION prior to use of PUT on file {file}.

595035 File mode must be GENERATION prior to use of WRITE or WRITELN on file {file}.

595037 File mode must be INSPECTION prior to use of GET on file {file}.

595038 File mode must be INSPECTION prior to use of READ on file {file}.

595039 The Fractional Digits value must be greater than or equal to 1. The value {number} is not allowed.

595040 Free of unallocated block in DISPOSE.

595041 GET attempted on file {file} when EOF is TRUE.

595044 Input attempted after end of file {file} using GET.

595046 Internal error: insufficient space to perform textfile output on file {file}.

595047 The integer value {number} read from file {file} exceeds MAXINT.

595049 Interactive file {file} must be a textfile.

595054 The value '{text}' read from file {file} is not a valid REAL number.

595055 Line length must be <= 150 for interactive input on file {file}.

595056 Internal error: Lower merge error in DISPOSE.

595057 More than 100 files used.

5.0 ERRORS
5.8 FATAL DIAGNOSTICS (RUN TIME)

595059 Pascal file {file} must have file organization of SEQUENTIAL.
595060 Pascal file {file} must have record type of VARIABLE.
595061 READ attempted when EOF is TRUE on file {file}.
595062 RESET attempted on undefined file {file}.
595063 The value {integer} is not allowed as the right operand of MOD at line {line number}. The value must be greater than zero.
595067 The value '{text}' read from file {file} is not a valid signed number.
595068 The TotalWidth value must be greater than or equal to 1. The value {number} is not allowed.
595069 Internal Error: Unable to allocate space for NEW variable in Heap.
595070 Internal Error: Upper merge error in DISPOSE.
595071 System condition detected: Arithmetic overflow at P register = {address}.
595073 System condition detected: Exponent overflow at P register = {address}.
595074 System condition detected: Exponent underflow at P register = {address}.
595075 System condition detected: Floating point indefinite at P register = {address}.
595076 System condition detected: Invalid BDP data at P register = {address}.
595077 Attempted output of REAL number to file {file} yields {math library} error.
595078 Attempted output of INTEGER number to file {file} yields {math library} error.
595079 Size of binary input item must be <= 2**31 - 1 bytes, current size is {number}.
595080 Internal Error: Error in math library routine MLP$MOVE_BYTES: {math library error}.
595081 RESET on file {file} failed because EOF occurred before buffer variable filled.
595082 Size of binary output item must be <= 2**31 - 1 bytes, current size is {number}.
595085 Program terminated by calling HALT.
595086 Internal Error: Cannot find stack frame for target label of GOTO statement.
595087 File buffer variable is undefined for file {file}.
595088 File buffer variable for file {file} is undefined when EOF is TRUE.
595089 Input of integer value {number} from file {file} yields {math library} error.

595090 Input line length {number} cannot be greater than page width of {number} for interactive file {file}.

595091 Character value '{text}' read from file {file} is outside the declared subrange of {text} .. {text}.

595092 Integer value {number} read from file {file} is outside the declared range of {number} .. {number}.

595093 The value {number} is out of range at line {number}.

595094 The ordinal value {number} is out of range at line {number}.

595095 The character value '{text}' is out of range at line {number}.

595096 The subscript value {number} is out of range at line {number}.

595097 The ordinal subscript value {number} is out of range at line {number}.

595098 The character subscript value '{text}' is out of range at line {number}.

595099 Attempted dereference of a pointer with an invalid value: {address} at line {number}.

595100 Attempted dereference of a NIL pointer at line {number}.

595101 The ordinal value of CHAR types must be within the range of 0..255. The ordinal value {number} is outside this range at line {number}.

595102 The ordinal value of CHAR types must be within the range of 0..255. The character subscript with ordinal value {number} is outside this range at line {number}.

595103 The CASE selector value {number} does not match any of the CASE constants ending at line {number}.

595104 The character case selector value '{text}' does not match any of the case constants at line {number}. The range of case constants is '{text}' .. '{text}'.

595105 The source string size of {identifier} is greater than the the target string size at line {line number}.

595106 The value {integer} is not acceptable as a substring index at line {line number}. Substring index must be greater than or equal to 1.

595107 Lower substring index {integer} must be less than or equal to upper substring index at line {line number}.

595108 Upper substring index {integer} must be less than or equal to length of the string at line {line number}.

595109 Length of concatenated string is {integer} at line {line

number}. Maximum string length allowed is 65535.

5.9 ERRORS NOT DETECTED

In conformance with the ISO and ANSI standards, a list of errors (in the sense of the standards) which are not detected follows:

Reference and access to component of inactive variant.
Removal of identifying-value from pointer-type of identified variable while reference exists.
Alteration of file-variable f while reference to f^ exists.
Set expression as value parameter which has elements not included in the base type of the formal parameter.
Buffer-variable undefined immediately prior to use of PUT.
Different variant specified for active variant created by NEW(p,c1,..,cn) .
DISPOSE(p) used when identifying-value created by NEW(p,c1,..,cn) .
DISPOSE(p,k1,..,km) applied to variable created by NEW different number of variants.
DISPOSE(p,k1,..,km) applied to variable created by NEW different variant list.
Undefined parameter of a pointer-type to DISPOSE.
Use of variable created by NEW(p,c1,..,cn) in assignment statement or actual parameter.
Components of unpacked array are both undefined and accessed for PACK.
Components of packed array are undefined for UNPACK.
f undefined when EOF(f) activated.
f undefined when EOLN(f) activated.
Use of undefined variable, or portion thereof.
Undefined function result upon completion of function.
Set expression in assignment statement which has elements not included in the base type of the variable-access.