UNISYS **Publication Change Notice (PCN)**

Date: February 28, 1989

Form—PCN number: 1169851-001

Title: A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation (Relative to the Mark 3.7.2 System Software Release)

Description

This letter announces the availability of PCN-001 to the A Series Pascal Programming Reference Manual, Volume 1: Basic Implementation, form number 1169851, dated July 1987. This PCN contains revisions and additions relative to the Mark 3.7.2 System Software Release.

Changes to the text are indicated by vertical black lines on the affected pages.

| Replace These Pages | | Add These Pages |
|---|---|---|
| vii | 431 | 134A |
| ix through xvii | 455 | 396A |
| 69 through 77 | 543 through 545 | 398A |
| 115 | 551 through 567 | 404A |
| 119 | | |
| 125 | | |
| 133 | | |
| 191 | | |
| 229 | | |
| 251 | | |
| 255 through 257 | | |
| 287 through 289 | | |
| 313 | | |
| 385 | | |
| 395 | | |
| 417 | | |

Retain the PCN cover sheet as a record of changes made to the basic publication.



Distribution Code SE
Printed in U S America

Page Status

| Page | Issue |
|---|---|
| iii through vi | Original |
| vii through xviii | PCN-001 |
| 1 through 68 | Original |
| 69 through 78 | PCN-001 |
| 79 through 114 | Original |
| 115 through 116 | PCN-001 |
| 117 through 118 | Original |
| 119 through 120 | PCN-001 |
| 121 through 124 | Original |
| 125 through 126 | PCN-001 |
| 127 through 132 | Original |
| 133 through 134B | PCN-001 |
| 135 through 190 | Original |
| 191 through 192 | PCN-001 |
| 193 through 228 | Original |
| 229 through 230 | PCN-001 |
| 231 through 250 | Original |
| 251 through 252 | PCN-001 |
| 253 through 254 | Original |
| 255 through 258 | PCN-001 |
| 259 through 286 | Original |
| 287 through 290 | PCN-001 |
| 291 through 312 | Original |
| 313 through 314 | PCN-001 |
| 315 through 384 | Original |
| 385 through 386 | PCN-001 |
| 387 through 394 | Original |
| 395 through 396B | PCN-001 |
| 397 through 398 | Original |
| 398A through 398B | PCN-001 |
| 399 through 404 | Original |
| 404A through 404B | PCN-001 |
| 405 through 416 | Original |
| 417 through 418 | PCN-001 |
| 419 through 430 | Original |
| 431 through 432 | PCN-001 |
| 433 through 454 | Original |
| 455 through 456 | PCN-001 |
| 457 through 542 | Original |
| 543 through 568 | PCN-001 |

E COMPARISON WITH ANSI PASCAL

Unisys Pascal and ANSI Pascal are compared in the areas of implementation-defined features, implementation-dependent features, and Unisys extensions to the ANSI standard.

F ERROR DETECTION AND REPORTING

Compile-time and run-time detection of errors is described. The errors defined in ANSI Pascal that are not detected in Unisys Pascal are also described.

G SYSTEM-SPECIFIC INFORMATION

All information applicable only to the implementation of Pascal on the A Series systems is provided.

H COMPILING PASCAL MODULES

A description for structuring a source file using modules t build a program and to build a library is given.

In addition, an explanation of railroad diagrams, a glossary, a bibliography, and an index appear at the end of this manual.

## RELATED PRODUCT INFORMATION

This list contains the Unisys documents you should read to best understand the material in this manual.

| Title | Form No. |
|---|---|
| BNA Architectural Description Reference Manual, Volume 1 | 1132172 |
| BNA Host Services Operations Guide | 1170339 |
| I/O Subsystem Programming Reference Manual | 1169984 |
| Operator Display Terminal (ODT) System Commands Operations Reference Manual | 1169612 |
| Pascal Programming Reference Manual, Volume 2: Product Interfaces | 1170107 |
| System Software Utilities Operations Reference Manual | 1170024 |

## COMPLIANCE STATEMENT

The Unisys Pascal compiler for A Series systems ("Unisys Pascal") complies with the requirements of ANSI/IEEE 770X3.97-1983 and with Level 0 ISO 7185 with the exception that the following features are not currently implemented:

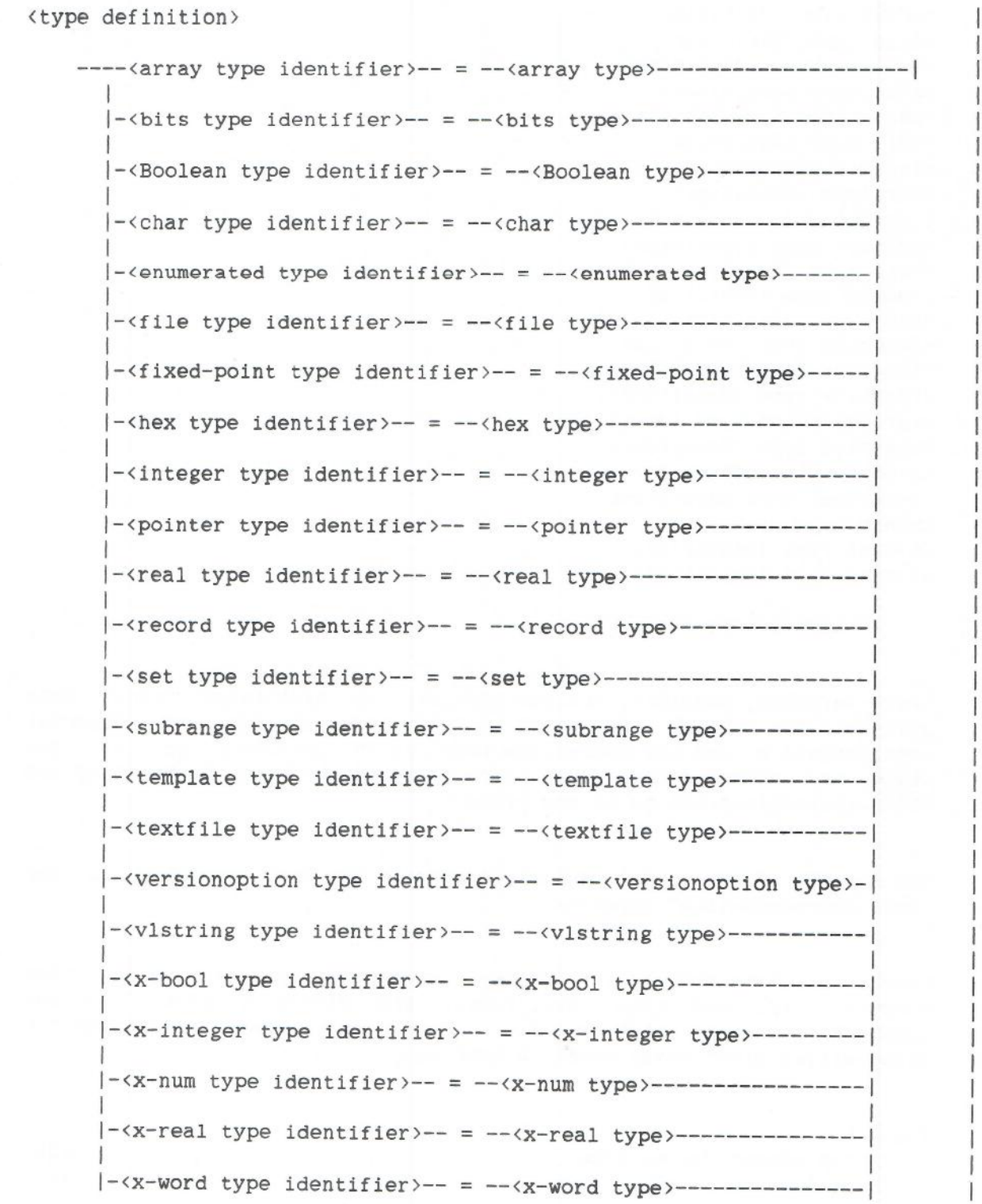
1. The declaration of files and textfiles as components of structured types

2. Dynamic variables of type file or type textfile

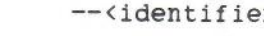
Further information on the relationship of Unisys Pascal to ANSI Pascal is contained in the "Comparison with ANSI Pascal" and the "Error Detection and Reporting" appendixes.

See also
Comparison with ANSI Pascal . . . . . 509
Error Detection and Reporting . . . . . 517

# CONTENTS


<type definition>
<array type identifier>-- = --<array type>
<bits type identifier>-- = --<bits type>
<Boolean type identifier>-- = --<Boolean type>
<char type identifier>-- = --<char type>
<enumerated type identifier>-- = --<enumerated type>
<file type identifier>-- = --<file type>
<fixed-point type identifier>-- = --<fixed-point type>
<hex type identifier>-- = --<hex type>
<integer type identifier>-- = --<integer type>
<pointer type identifier>-- = --<pointer type>
<real type identifier>-- = --<real type>
<record type identifier>-- = --<record type>
<set type identifier>-- = --<set type>
<subrange type identifier>-- = --<subrange type>
<template type identifier>-- = --<template type>
<textfile type identifier>-- = --<textfile type>
<versionoption type identifier>-- = --<versionoption type>
<vlstring type identifier>-- = --<vlstring type>
<x-bool type identifier>-- = --<x-bool type>
<x-integer type identifier>-- = --<x-integer type>
<x-num type identifier>-- = --<x-num type>
<x-real type identifier>-- = --<x-real type>
<x-word type identifier>-- = --<x-word type>

<array type identifier>
<bits type identifier>
<Boolean type identifier>
<char type identifier>
<enumerated type identifier>
<file type identifier>
<fixed-point type identifier>
<hex type identifier>
<integer type identifier>
<pointer type identifier>
<real type identifier>
<record type identifier>
<set type identifier>
<subrange type identifier>
<template type identifier>
<textfile type identifier>
<versionoption type identifier>
<vlstring type identifier>
<x-bool type identifier>
<x-integer type identifier>
<x-num type identifier>
<x-real type identifier>
<x-word type identifier>




Every variable, constant, and function has an associated "type" that defines its range of valid values, its internal and external representation, and the operations that can be performed on it. The <type and schema definitions> allow user-defined types to be named and their characteristics to be specified.

The details of each type's actual representation are described in the "Data Representation" appendix.

General concepts that apply to types, such as type categories, type compatibility, and type identifiers, are discussed under "Type and Schema Concepts" in this section. Types themselves are listed in alphabetical order under separate headings.

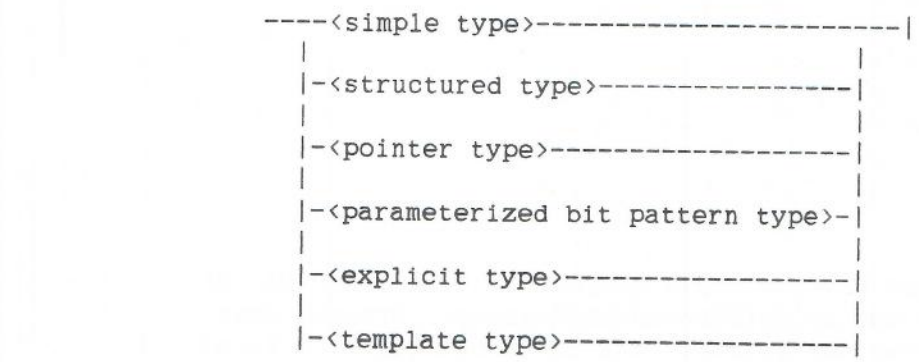
See also

Array Schema Definitions. . . . . 135
Data Representation . . . . . 481

## 2.5.1 TYPE AND SCHEMA CONCEPTS

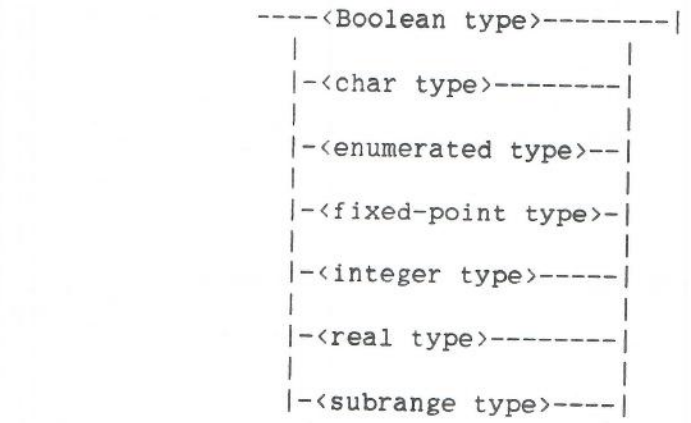
### TYPES

<type>



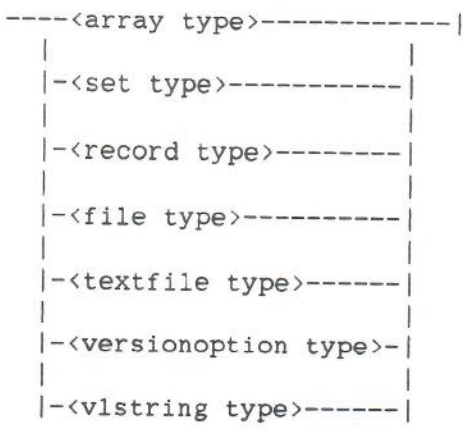

Types can be classified into categories that relate to their structure: simple types, structured types, pointer types, parameterized bit pattern types, explicit types, and template types.

<simple type>




Variables of simple types have only one component. The predefined types Boolean, char, fixed-point, integer, and real are simple types. User-defined derivatives of these predefined types, as well as enumerated types and subrange types, are also simple types.

<structured type>

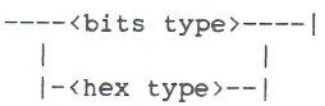



Variables of structured types have multiple components, which can be of one or more simple types or can be structured themselves. Arrays, sets, records, files, textfiles, version options, and vlstrings are structured types.

<pointer type>

Variables of pointer types contain values that are references to variables of simple or structured types.

<parameterized bit pattern type>




Variables of parameterized bit pattern types represent bit patterns of a specified length.

<explicit type>

Variables of explicit types represent numeric values in an explicitly specified format.

<template type>

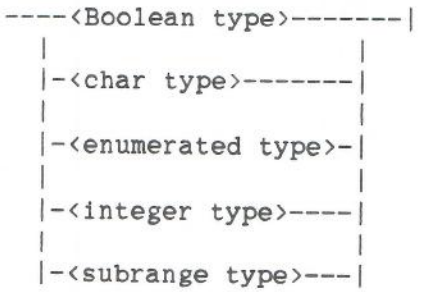
Template types change how variables of explicit array type or explicit record type are viewed or used.

A template can be attached to an <explicit array variable> or an <explicit record variable> at any byte offset (refer to "ATTACH_Template Procedure" in the "Predefined Procedures and Functions" section for additional information). After the template is attached, it acts as if it were a variable of the type for which it is a template. For example, a template attached to an <explicit array variable> acts as an <explicit array variable>. A <template type> cannot be used unless it is attached to an explicit array or record.

See also
ATTACH_Template Procedure . . . . . 368

## ORDINAL TYPES

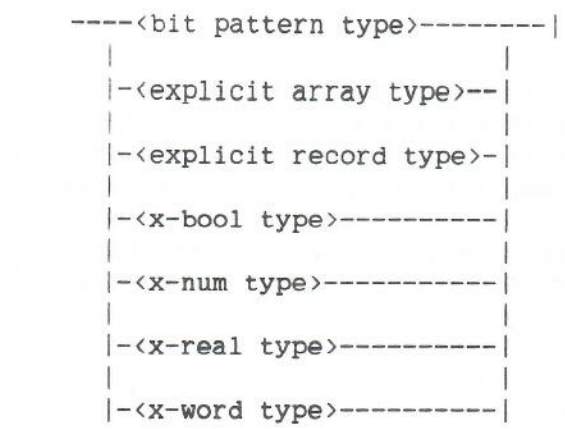
<ordinal type>




Most simple types are also "ordinal types." In an ordinal type, the values have a well-defined sequential relationship. Each value is assigned an "ordinal number" that uniquely identifies its position in the sequence. Thus, a value of an ordinal type can have a "successor" and a "predecessor" in the sequence. Values can also be compared (for example, greater than or less than) based on their ordinal numbers. The exceptions are <real type>s and <fixed-point type>s. They are simple types but not ordinal types.

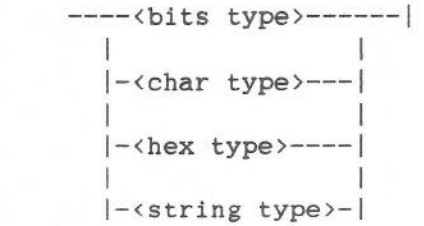
## EXPLICIT TYPES

<explicit type>




An <explicit type> is a type with the internal data format (and location, if within a record) that is explicitly specified. The <explicit type> is used primarily for importing data types from non-Pascal external interfaces such as the Advanced Data Dictionary System (ADDS).

<bit pattern type>




The <bit pattern type>s are a class of predefined types having associated values that are bit patterns.

<explicit array type>

This type is an <array type> designated as PACKED with a <component type> that is an <explicit type> represented by an integral number of bytes.

<explicit record type>

This type is a <record type> having <field type>s that are all <explicit type>s, and has specified field locations.

<x-bool type>

The <x-bool type> represents a Boolean value in an explicitly specified format.

<x-num type>

The <x-num type> represents numeric values in an explicitly specified format.

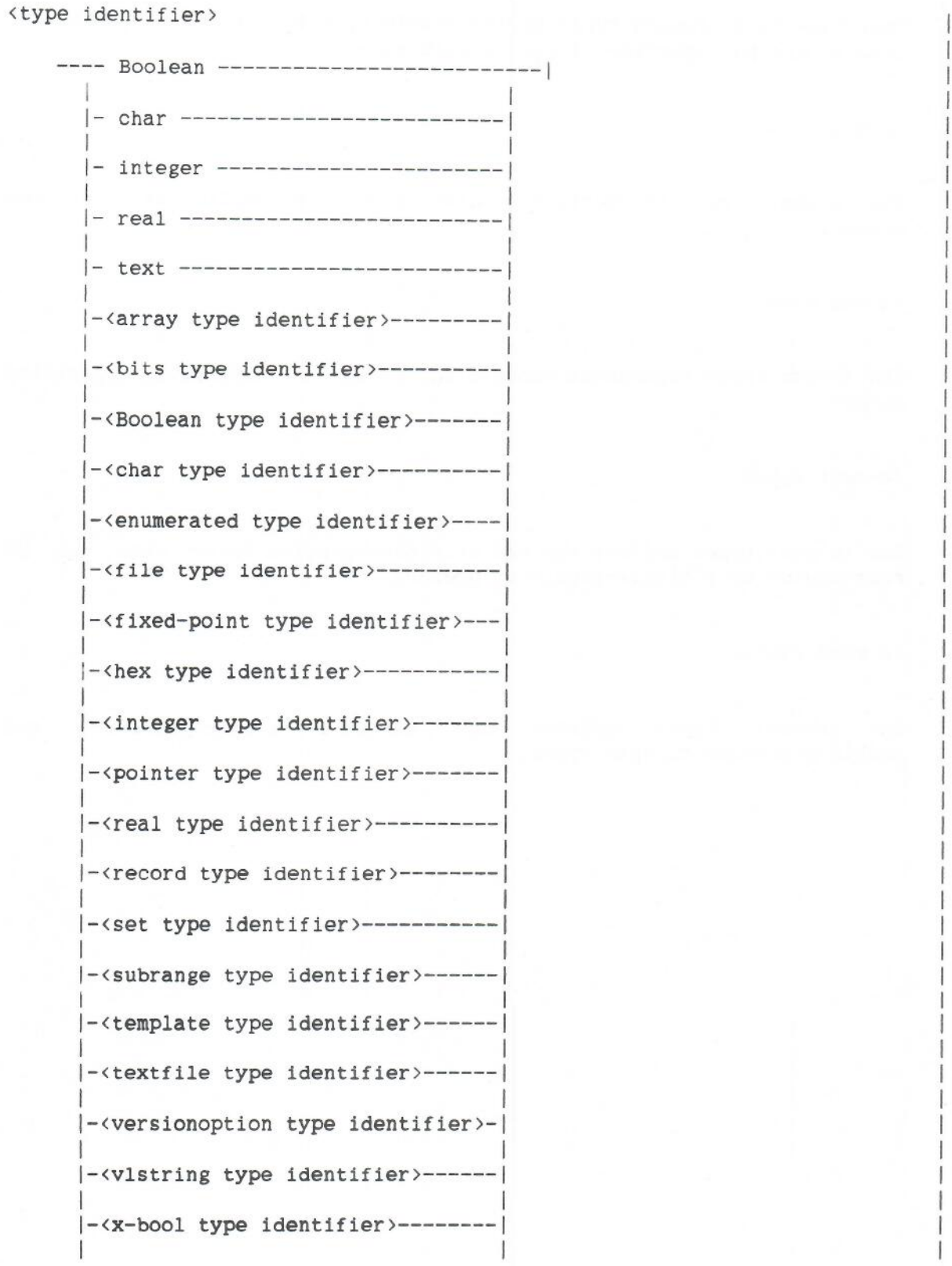
<x-real type>

The <x-real type> defines the set of floating-point values that can be represented by a single-precision operand.

<x-word type>

The <x-word type> defines the explicit single-precision and double-precision numeric types.

## TYPE IDENTIFIERS




In <type and schema definitions> and <variable declarations>, a type can usually be defined in one of two ways:

1. As a new type; that is, by using the <new array type>, <new bits type>, <new enumerated type>, <new file type>, <new hex type>, <new pointer type>, <new record type>, <new set type>, <new subrange type>, or <new vlstring type> syntax

2. As a derived type, where an <identifier> that has already been defined (or was predefined) as a type identifier is specified

In other contexts requiring a type specification, new types are not allowed and previously defined <type identifier>s must be used.

## ARRAY SCHEMATA

Schemata is the plural of the word schema, which means outline or plan. An array schema definition is a formal syntactical outline describing a collection of array types called a schema family. Each array type in this collection has the same primary characteristics as the others: the same component type, the same number of dimensions, and the same ordinal types for the subscripts of the corresponding dimensions.

Array schemata allow arrays with different bounds and different numbers of elements, but with types belonging to the same schema family, to be passed as actual parameters to the same formal parameter. The type of this formal parameter is designated by a schema identifier denoting the schema family, rather than a type identifier denoting the individual array type.

With array schemata, a formal parameter of a procedure or function can be denoted by an array schema identifier. The procedure or function can then handle any array type that is a member of the schema family.

See also

Array Schema Definitions. . . . . 135
Array Types . . . . . 85

## 2.5.2 TYPE COMPATIBILITY

### SAME TYPES

Because types can be defined in different ways, it is not always clear when two types are actually the same type. The concept of "same type" is used when describing how <variable parameter>s are matched in procedure and function invocations. More important, the definition of "same type" is used to define compatible types and assignment compatibility, which are described under later headings.

The <type identifier>s T1 and T2 are the same type if one of the following conditions is true:

1. One type is defined to be equal to the other.
2. Both types are the same type as a third type.

The simplest case of "same type" is when T1 is defined as equal to T2, as shown in the example below:

```
TYPE T1 = T2;   { same type, by condition 1}
```

Condition 2 describes the case in which T1 and T2 have a common ancestor, the simplest case of which is the following:

```
TYPE T3 = INTEGER;
     T1 = T3;   { same type, by condition 1 }
     T2 = T3;   { same type, by condition 1 }
{ T1 is the same type as T2, by condition 2 }
```

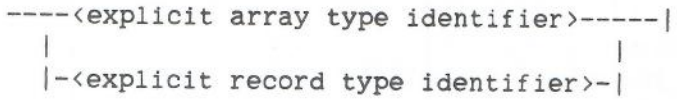
In the example above, T1 is the same type as T3, by condition 1. Similarly, T2 is also the same type as T3. Thus, T1 and T2 are the same type because they are both the same type as T3. By a somewhat longer chain, T1 and T2 can be recognized as the same type in the following example:

```
TYPE T5 = INTEGER;
     T4 = T5;
     T3 = INTEGER;
     T2 = T4;
     T1 = T3;
```

## TEMPLATE TYPES

<template type>


<template domain type>




A <template type> is a pattern that can be attached to an explicitly represented array or an explicitly represented record at any byte offset. Refer to "Template Procedures" for additional information.

The <template type>s are a Unisys extension to ANSI Pascal.

See also
Template Procedures . . . . . 366

**Example**

```
type binary_length_type = binary(16);
     text_area = packed array [1..65535] of char;

     msg = record
           length(1)  : binary_length_type;
           message(3) : text_area;
           end;
     msg_template = template(msg);
```

A variation of the "template" construct has been implemented that allows a template to be attached to a read-only parameter. The new construct is a read-only template, spelled "const_template".

**Example**

```
program demo_const_template;
   type
     rec =
        record
        first(1): packed array [1..1]  of char;
        rest    : packed array [1..10] of char;
        end;
     readonly_template = const_template( rec );
     large_area = packed array[1..65535] of char;
   var
     data_area: large_area;
   procedure takes_var_param( var r: rec );
     begin ... end;
   procedure p( const area: large_area );
     var t: readonly_template;
         c: char;
     begin
     attach_template( t, area, 1 );
     c := t.first;
     { The following statement results in a syntax error because }
     { a read-only template variable cannot be assigned to:      }
     t.first := c;
     { The following statement results in a syntax error because }
     { a read-only template variable cannot be passed as a       }
     { "var" parameter:                                          }
     takes_var_param( t );
     end;
   begin
   ...
   p( data_area );
   end.
```

A ‹template variable› declared with the "const_template" form of ‹template type› is a read-only ‹template variable›.

A read-only ‹template variable› is similar to a ‹template variable› declared with the "template" form of ‹template type›, except that it can be attached (using the "attach_template" procedure) to an ‹explicit array variable› or ‹explicit record variable› that is a read-only parameter. Also, the read-only ‹template variable› cannot be the target of an assignment statement or be used as the actual parameter corresponding to a formal "var" parameter. That is, the same restrictions that apply to a read-only parameter also apply to a read-only ‹template variable›.

**Example**

```
type str1 = string(10);
     str2 = string(80);
     str3 = str2;
```

Type "str1" is a <vlstring type identifier> that has a <maximum length> of 10 characters. "Str2" and "str3" are compatible <vlstring type identifier>s that have a <maximum length> of 80 characters.

## VERSIONOPTION TYPES

<versionoption type>

--<versionoption type identifier>--|

<release>
<cycle>
<patch>

-- <integer variable> --|

The <versionoption type> is a record type containing three integer fields. The <versionoption type> is used with the <versionoption function> to obtain the version assigned using the VERSION compiler control option. The <versionoption type> fields are in the following ranges:

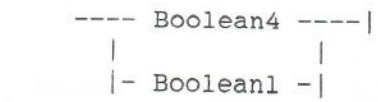
| Parameter | Range |
|---|---|
| <release> | 0..99 |
| <cycle> | 0..999 |
| <patch> | 0..9999 |

The fields of a <versionoption type> variable can be accessed the way fields of any record can be accessed, using dot qualification or the WITH statement. For an example of how to use the <versionoption type> and the <versionoption function>, refer to the VERSIONOPTION FUNCTION in the "Predefined Procedures and Functions" section.

See also
VERSIONOPTION FUNCTION. . . . . 404A

## X-BOOL TYPES

<x-bool type>




An <x-bool type> represents a Boolean value in an explicitly specified format, but cannot be treated directly as a Boolean value within a program. The explicit Boolean type must first be converted to a Boolean type through a type transfer process using the <x-bool type transfer function> before it can be referenced as a Boolean value. An explicit Boolean type can be transferred to either a <Boolean type> or a <bit pattern type>.

Attempting to transfer a value of a <bit pattern type> into an <x-bool type> that will not hold the value produces a range error, and vice versa.

Also, comparisons between explicit Boolean values cannot be performed. The <x-bool type> values must be converted to either <Boolean type> or <bit pattern type> values to be compared.

Like the <x-word type>, <x-bool type>s are used mainly to import COBOL-formatted records from the Advanced Data Dictionary System (ADDS).

The <x-bool type> is a Unisys extension to ANSI Pascal.

### Boolean4

A variable of type Boolean4 requires exactly 4 bits to represent a Boolean value in a packed 4-bit format. Fields of type Boolean4 are constrained to start on 4-bit boundaries within a record.

### Example

```
var  bool4_var1, bool4_var2   : Boolean4;
begin
     bool4_var1 := Boolean4(4'1');
     bool4_var2 := Boolean4(true);
end;
```

**Examples**

The example below shows how a variable of type s_digits(n) can be used. Given the definition for the s_digits type variable, the resulting relationships are true.

```
type s_digits8 = s_digits(8);


integer(s_digits8 (4'C1234567')) = 1234567
integer(s_digits8 (4'D1234567')) = -1234567
```

The example below shows how a variable of type s_digits(n,s) can be used. Given the definitions for the s_digits type and sfixed type variables, the resulting relationship is true.

```
type  s_digits5_8   = s_digits(5,8);
      sfixed4_8     = sfixed(4,8);


sfixed4_8 (s_digits5_8 (4'D1234')) = - 0.00001234
```

## Digits_s

A variable of type digits_s(n) represents an integer value in trailing sign, packed decimal format. In all other respects, digits_s(n) is the same as s_digits(n).

When the optional <scale factor> is used, a variable of type digits_s(n,s) requires exactly n*4 bits to represent a fixed-point value in the range -((10**(n-1))-1)/(10**s) to ((10**(n-1))-1)/(10**s).

Values for variables of type digits_s can be up to 24 digits long (1 digit is for the sign). Fields of s_digits type are constrained to start on 4-bit boundaries within records.

**Examples**

The example below shows how a variable of type digits_s(n) can be used. Given the definition for the digits_s type variable, the resulting relationships are true.

```
type digits_s8 = digits_s(8);
```

```
integer(digits_s8 (4'1234567C')) = 1234567
integer(digits_s8 (4'1234567D')) = -1234567
```

The example below shows how a variable of type digits_s(n,s) can be used. Given the definition for the digits_s type and sfixed type variables, the resulting relationships are true.

```
type  digits_s6_m3  = digits_s(6,-3);
      sfixed5_m3    = sfixed(5,-3);


sfixed5_m3 (digits_s6_m3 (4'12345C')) = 12345000
sfixed5_m3 (digits_s6_m3 (4'12345D')) = -12345000
```

## U display

A variable of type u_display(n) represents an integer value in unsigned, unpacked decimal format. It requires exactly "n" bytes and represents an integer in the range 0 (zero) to (10**n)-1. The value of "n" must be between 1 and 23, inclusive.

When the optional ‹scale factor› is used, a variable of type u_display(n,s) requires exactly "n" bytes to represent a fixed-point value in the range 0 (zero) to ((10**n)-1)/(10**s) in an unsigned, display numeric format.

Values for variables of type u_display can be up to 23 bytes long.

### Examples

The example below shows how a variable of type u_display(n) can be used. Given the definition for the u_display type variable, the resulting relationship is true.

```
type u_display4 = u_display(4);


integer(u_display4 ('1234')) = 1234
```

## Word48(s)

A variable of type word48(s) requires exactly 48 bits to represent a fixed-point value in single-precision format. The <scale factor>, "s", must be between 47 and -68, inclusive. Fields of type word48 are constrained to start on byte boundaries within a record.

**Example**

```
type  word48_2       = word48(2);
      word48_m2      = word48(-2);
      fixed3_2       = fixed(3,2);
      fixed10_m2     = fixed(10,-2);
```

Given the preceding definitions, the following relationships are true:

```
fixed3_2 (word48_2 (4'0000000000FF'))     = 2.55
fixed10_m2 (word48_m2 (4'0000000000FF')) = 25500
```

## Word96(s)

A variable of type word96(s) requires exactly 96 bits to represent a fixed-point value in double-precision format. The <scale factor>, "s", must be between 47 and -68, inclusive. Fields of type word96 are constrained to start on byte boundaries within a record.

**Example**

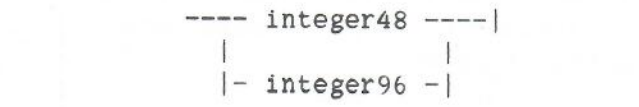
```
type  word96_2       = word96(2);
      word96_m2      = word96(-2);
      sfixed10_4     = fixed(10,4);
      fixed4_m2      = fixed(4,-2);
```

Given the above definitions, the following relationships are true:

```
sfixed10_4 (word96_2 (4'4000000000FF000000000000')) = -2.55
fixed4_m2 (word96_m2 (4'0000000000FF000000000000')) = 25500
```

## X-INTEGER TYPES

<x-integer type>




### Integer48

For convenience, integer48 is included as a predefined word48(0) type that represents a single-precision integer value. Thus, type transfer is possible between integer48 and integer types. All rules concerning type word48 remain the same. Fields of type integer48 are constrained to start on byte boundaries within a record.

**Example**

```
var  int48_var1, int48_var2   : integer48;
begin
     int48_var1 := integer48(4'0000000000FF');
     int48_var2 := integer48(4'4000000000FF');
end;
```

Given the preceding assignments of int48_var1 and int48_var2, the following relationships are true:

```
integer (int48_var1) = 255
integer (int48_var2) = -255
```

## Integer96

Similarly, integer96 is included as a predefined word96(0) type that represents a double-precision integer value. Thus, type transfer is possible between integer96 and integer types. All rules concerning type word96 remain the same. Fields of type integer96 are constrained to start on byte boundaries within a record.

**Example**

```
var  int96_var1, int96_var2  : integer96;
begin
     int96_var1 := integer96 (4'4000000000FF000000000000');
     int96_var2 := integer96 (255);
end;
```

Given the above assignments of int96_var1 and int96_var2, the following relationships are true:

```
integer (int96_var1) = -255
integer (int96_var2) = 255
```

The <bits constant>s and <hex constant>s are Unisys extensions to ANSI Pascal.

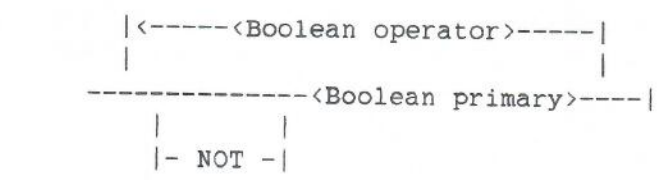
See also
Type Transfer Procedures and Functions. . . . . 316
Variables . . . . . 407

**Example**

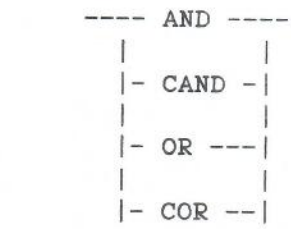
```
program bit_patterns;

const bitmask = 1'00010001';

var hex_byte : hex(2);
    bits_byte: bits(8);

begin
bits_byte := bits_mask;
hex_byte := bits_byte or 1'10000000';
end.
```

## BOOLEAN EXPRESSIONS

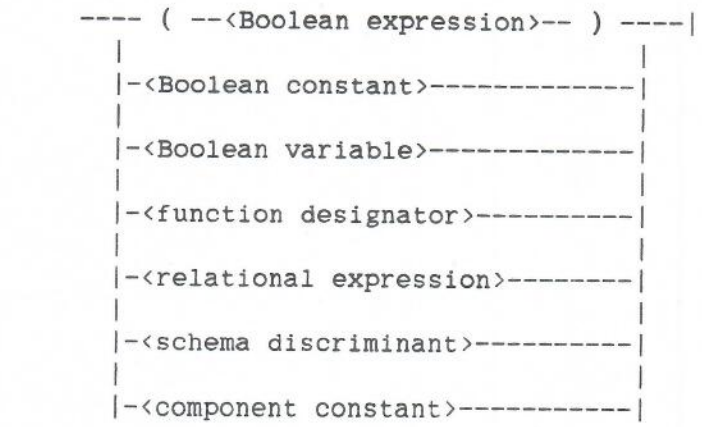
<Boolean expression>



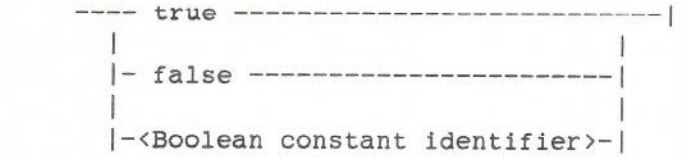

<Boolean operator>




<Boolean primary>



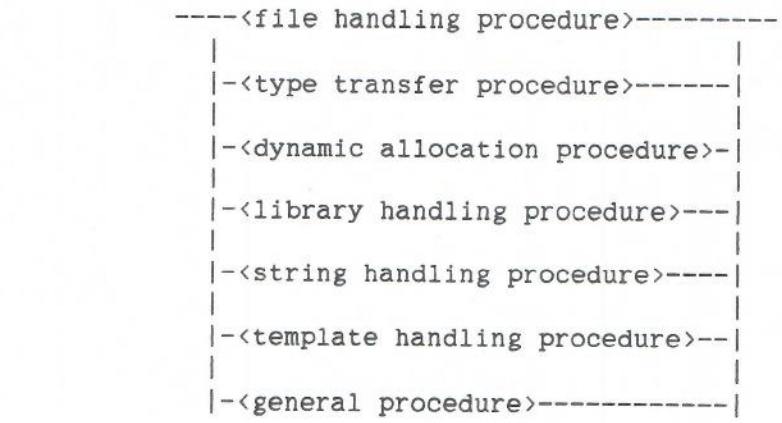

<Boolean constant>



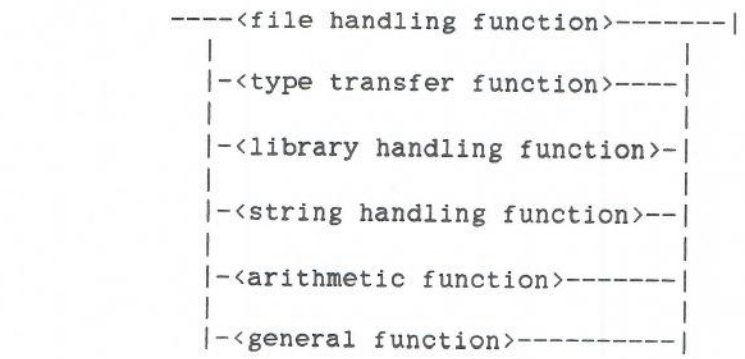

A <Boolean expression> generates a value of the <Boolean type>.

# 5 PREDEFINED PROCEDURES AND FUNCTIONS

<predefined procedure>




<predefined function>




Many Pascal features, including input/output (I/O) facilities and dynamic variables, are made available through predefined procedures and functions. Although procedures and functions are syntactically different constructs, this difference is not emphasized in this section. Instead, procedures and functions relating to a particular program application, such as file handling, are grouped together to provide some continuity in their descriptions. Within each group, the procedures and functions are listed alphabetically.

## 5.1 FILE HANDLING PROCEDURES AND FUNCTIONS

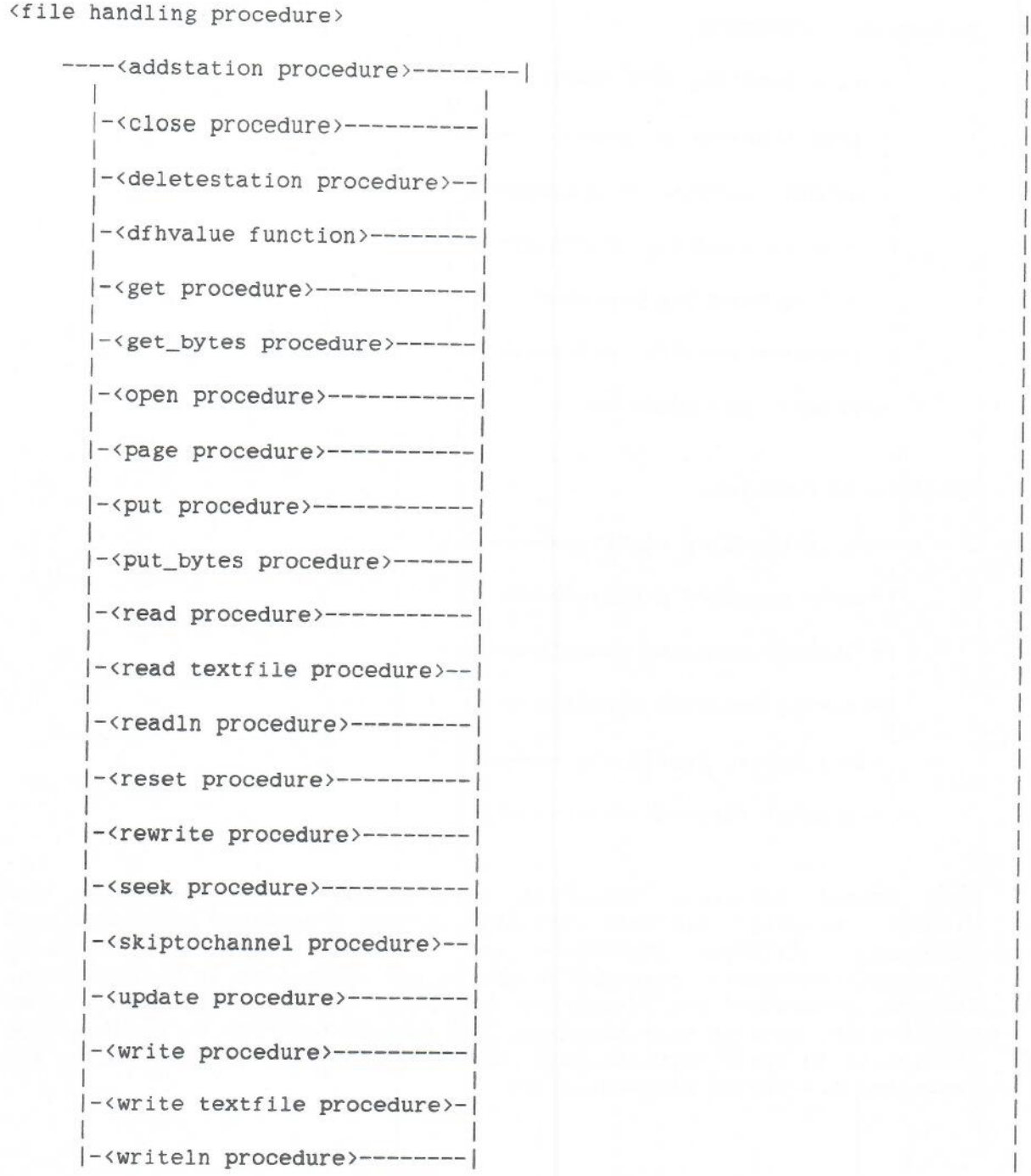


TABLE 5-2. MAPPING OF ATTRIBUTE TYPES WITH PASCAL DATA TYPES

| I/O Type | Pascal Type |
|---|---|
| integer (without mnemonics) | integer |
| integer (with mnemonics) | integer, for "setattribute" and "getattribute"; mnemonic, otherwise. |
| Boolean | Boolean |
| pointer | string, vlstring |
| real (numeric) | real |
| real (SERIALNO) | string or vlstring (6 significant characters) |
| real (bits -- STATE, ATTVALUE, USERINFO) | real |
| translatetable | [not available] |
| event | [available through the <wait procedure>] |
| STATIONLIST | [available through the <addstation procedure> and the <deletestation procedure>] |

Pointer-valued attributes are always assumed by the system to have an EBCDIC representation. Thus, if the STRINGS compiler control option has been set to ASCII, which causes the compiler to assume that all string variables are represented in ASCII, it will automatically translate from ASCII to EBCDIC or EBCDIC to ASCII when setting or accessing pointer-valued attributes.

When accessing a pointer-valued attribute, if the value is shorter than the string variable into which it is being stored, the value is blank filled on the right; if the value is being stored into a vlstring variable, it is not blank filled. If the value is longer than the length of the string variable or the maximum length of the vlstring variable into which it is being stored, an error occurs.

NOTE

A period (".") is not returned as part of the value.

If a null pointer-valued attribute is placed into a string variable, the variable will contain all blanks. If a null pointer-valued attribute is placed into a vlstring variable, the variable has a null value and a length of 0.

When setting the SERIALNO attribute, only the first six characters are used. A value less than six characters in length is blank filled on the right to six characters. A null SERIALNO is specified with a string or vlstring variable at least six characters long, in which the first six characters are NUL ("chr(0)").

The SERIALNO attribute, when accessed, returns a value six characters in length. If the SERIALNO is returned in a string variable longer than six characters, the value is blank filled on the right. If the SERIALNO is returned in a vlstring variable, the value is not blank filled. An error occurs if the value is longer than the length of the string variable or the maximum length of the vlstring variable.

MAXSUBFILES file attribute, or when a subfile index is not specified and the value of MAXSUBFILES is greater than 1.

Iores( Openall )
This value is returned only if the value of the KIND attribute is PORT. It is returned when an attempt is made to open all subfiles and an error occurs on any one of them.

The exceptions described here can occur when an OPEN is attempted on a PORT file:

Iores( unsupportedfunction )
This error occurs if a request is made for an unsupported function.

Iores( Unsupportedprotocol )
This error occurs when an incompatible protocol is detected.

Iores( Protocolerror )
An error in the protocol was detected.

Iores( Lackofresources )
The OPEN procedure cannot be completed due to lack of resources.

Iores( Hostnotinhostgroup )
The OPEN procedure failed because the host specified in YOURHOST is not a member of the specified YOURHOSTGROUP.

Iores( unauthorizedforag )
The current user is not an authorized user of the specified APPLICATIONGROUP.

Iores( Bnaipcunavailable )
This error occurs if the OPEN procedure failed because the PORT Level Manager was not available to the PORT element.

Iores( Badattributesforopen )
This error occurs if the OPEN procedure failed because the file attributes were set incorrectly.

Iores( Unavailablefunction )
This error occurs if a request is made for an unavailable function.

Iores( Unsupportedintmode )
This error occurs if the intmode attribute was set to a value not supported by BNA.

Iores( Networkworkingnotsupported )
This error occurs if networking is not supported.

Iores( Unsupportedparameterrslt )
This error is returned if the SERVICE attribute is set to TCPPUSHEDMSGSERVICE and if OFFER is used as an option for the OPEN procedure.

Iores( Localipcnotsupportedrslt )
This error is returned if the SERVICE attribute is set to TCPPUSHEDMSGSERVICE and if YOURHOST is not specified, implying a local OPEN procedure. Local ports are not allowed for TCPPUSHEDMSGSERVICE.

Iores( Unsupportedtranslationrslt )
This error is returned if the SERVICE attribute is set to TCPPUSHEDSERVICE, if the TRANSLATE attribute is set to the value FULLTRANS, and if the value of the INTMODE and EXTMODE attributes of the port file are not equal. This condition implies that the user requested a translation that is not supported by TCPPUSHEDMSGSERVICE.

**Get, Put, Read, Readln, Write, Writeln**

These exceptions can occur when an attempt is made to read or write a file.

Iores( Ok )
The operation was successful.

Iores( Badindex )
This value is returned only if the value of the KIND attribute is PORT. It is returned when a subfile index is specified that has a value less than 0 or greater than the value of the MAXSUBFILES attribute, or when a subfile index is not specified and MAXSUBFILES is greater than 1.

Iores( Dataerr )
This value is returned if a data error occurred, an invalid value was received by a textfile input operation, or a line overflow occurred on textfile output.

Iores( Deleted )
The value of the FILEORGANIZATION attribute is RELATIVE and a deleted or duplicated record was referenced.

Iores( Parity )
A parity error on the associated physical medium was encountered. Note that the I/O subsystem will report this error only after performing several unsuccessful retries.

Iores( Porterr )
This value is returned only if the value of the KIND attribute is PORT. It is returned when one of the following errors occurs:

1. A broadcast write failed for one or more subfiles.

2. A "put" with the "dontwait" option failed because no buffer was available.

3. A "get" with the "dontwait" option failed because no data was available.

Iores( Eof )
End-of-file or end-of-page was encountered. Note that this is "end-of-file" as defined by the I/O subsystem, which does not correspond exactly to the value of the Pascal <eof function>. Several conditions can cause this exception to occur without causing the <eof function> to return "true". Also, this exception does not always occur in generation mode, whereas the <eof function> will always return "true" in generation mode. An end-of-page exception occurs when the PAGESIZE attribute has been set and the end of a logical page has been encountered on output.

Iores( Brk )
A break on output from the associated physical device was encountered.

Iores( Timeout )
The TIMELIMIT file attribute has been set, and the specified time limit expired before the I/O operation completed.

Iores( Security )
A security violation was attempted.

**Close**

These exceptions may occur when the I/O subsystem close routines are invoked to close the file:

Iores( Ok )
The operation was successful.

Iores( Unknown )
An error has occurred, but its nature cannot be determined.

Iores( Notopen )
The file or subfile is already closed.

Iores( Datalost )
All data was not successfully delivered to the destination.

Iores( Recordcount )
A record count error occurred.

Iores( Blockcount )
A block count error occurred.

Iores( Badindex )
This value is returned only if the value of the KIND attribute is PORT. It is returned when a subfile index less than zero or greater than the value of the MAXSUBFILES file attribute is specified, and when a subfile index is not specified and the value of MAXSUBFILES is greater than 1.

Iores( Closeall )
This value is returned only if the value of the KIND attribute is PORT. It is returned when an attempt is made to close all open subfiles and an error occurs on at least one of them.

When handling exception conditions, special attention should be paid to the timing of the operations, particularly regarding textfile I/O. (Refer to the discussion of "Textfiles," especially "Lazy I/O" in this section.)

Under certain rare circumstances, it is possible that an "open" operation on a tape file may perform an implicit "close", in which case a "close" result may be returned. Similarly, a "close" operation on a tape file may perform an implicit open.

The ability to return I/O results is a Unisys extension to ANSI Pascal.

See also
Inspection Mode Operations on Textfiles . . . . . 243
Iores Function. . . . . 282

## PAGE PROCEDURE

<page procedure>


The <page procedure> is identical in action to a call on the <skiptochannel procedure> with the channel number specified as 1. That is, it causes a <writeln procedure> without carriage control, followed by a skip-to-top-of-page action. If the <textfile variable> is omitted, the action applies to the textfile "output".

If the <page procedure> is invoked for a file that is not associated with a printer, the effect is equivalent to invoking the <writeln procedure>.

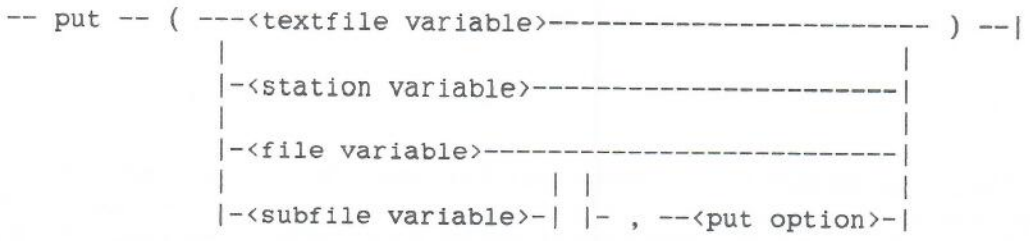
An error occurs if the file is in inspection mode or if the file is not open prior to the execution of the <page procedure>.

See also

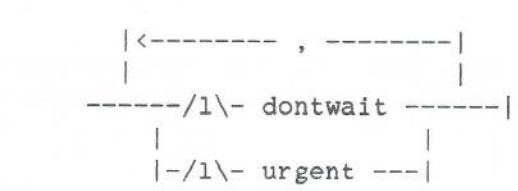
Skiptochannel Procedure . . . . . 306
Writeln Procedure . . . . . 315

## PUT PROCEDURE

<put procedure>




<put option>




The <put procedure> writes the value of the buffer variable to the file denoted by <textfile variable>, <file variable>, <subfile variable>, or <station variable>. The value of the buffer variable then becomes undefined.

An error occurs if the file is in inspection mode or if the file is not open prior to execution of the <put procedure>. An error also occurs if a <textfile variable> is specified and the <put procedure> causes the line to exceed the length determined by the value of the MAXRECSIZE file attribute.

A <subfile variable> is meaningful only if the KIND of the file is PORT. If a <subfile variable> is used and the associated <subfile index> is 0, a "broadcast write" (in BNA terminology) is performed; that is, the value of the buffer variable is written to each subfile. If a <subfile index> greater than 0 and less than or equal to the value of the MAXSUBFILES file attribute is specified, the data is written only to the specified subfile. The LASTSUBFILE file attribute is updated to the value of <subfile index> if the operation completes without error. An error occurs if an invalid <subfile index> is specified or if the value of MAXSUBFILES is greater than 1 and a <subfile variable> is not used.

A <station variable> is meaningful only if the KIND of the file is REMOTE. If a <station variable> is used and the associated <station index> is 0, a "broadcast write" is performed. If a <station index> greater than 0 is specified, the data is written only to the specified station. The LASTSUBFILE file attribute will be updated to the value of <station index> when the operation completes. If a <station variable> is not used, but the specified file is a remote file with multiple stations, the write is directed to the station denoted by the current value of the LASTSUBFILE attribute.

The "dontwait" <open option> is meaningful only if the file is a port file. If the file is a port file and the "dontwait" <open option> is not specified, the program is suspended until buffers are available to perform the write. If the "dontwait" <open option> is specified, the program is not suspended when buffers are not available. Instead, control returns to the program without performing the write. The success or failure of the operation can be determined by examining the associated I/O result. If an exception occurs and the I/O result is not handled programmatically, the program is terminated. Refer to "I/O Exception Handling" for information concerning I/O results.

The "urgent" clause is meaningful only when the Transmission Control Protocol/Internet Protocol (TCP/IP) is being used. This clause sets the "urgent indication" with the data. For more information on TCP/IP, refer to the "BNA Host Services Operations Guide."

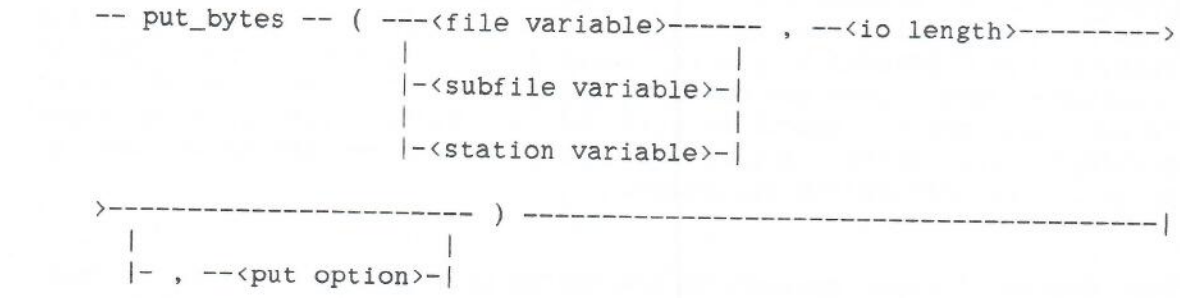
The use of <subfile variable>s, <station variable>s, and the "dontwait" and "urgent" <open option>s are Unisys extensions to ANSI Pascal.

See also

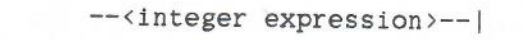
I/O Exception Handling. . . . . 253

## PUT BYTES PROCEDURE

<put_bytes procedure>




<io length>




The <put_bytes procedure> allows a program to do a variable-length write from a file's <buffer variable>. The <put_bytes procedure> writes <io length> bytes of the file's buffer variable to the specified file. If <io length> exceeds either the length of the file's buffer variable or the MAXRECSIZE of the file, the smaller number of bytes is written.

NOTE

Actually, an I/O which is <io length> FRAMESIZE UNITS will be requested. The compiler will set FRAMESIZE to 8 for all files to which this procedure can be applied.

As with the <get_bytes procedure>, whether a variable-length I/O is actually performed depends upon the values of the BLOCKSTRUCTURE, SIZEVISIBLE, and KIND file attributes. For further information, see the description of the <get_bytes procedure> and the example programs below.

The <component type> of the file must be an <explicit array type> or <explicit record type>.

The <put option> is the same as the <put option> parameter defined for the <put procedure>. It is not allowed if the first parameter is a <station variable>.

## Real Expression

The write operation treats variables of the <real type> and <fixed-point type> similarly. If the <frac digit> is provided, the number is written in fixed-point format; if it is not, the number is written in floating-point format. The default <field width> for a <real expression> is 15 characters.

In floating-point format, the number contains the following components:

1. A sign ("-" if the number is negative, blank if it is positive)
2. The first significant digit (or 0, if the number is 0)
3. A decimal point (.)
4. The fractional part (at least one digit)
5. The exponent symbol (E)
6. The sign of the exponent ("+" or "-")
7. Two digits of the exponent

If the <field width> specified is smaller than the minimum number of characters necessary to represent the number, the <field width> specification is ignored, and the number is written with one digit of fractional part. If the specified <field width> is larger, the number is expanded by adding trailing zeros to the fractional part.

In fixed-point format, the number will contain the following components:

1. A minus sign (-) if the number is negative
2. The integral part of the number (trunc(<real expression>))
3. A decimal point (.)
4. The <frac digit> of the fractional part of the number

If a specified <field width> is smaller than the minimum number of characters necessary to represent the number in fixed-point format, the <field width> specification is ignored and the entire number is written, including the <frac digit> of the fractional part. If the specified <field width> is larger, the number is written with leading blanks. If the number of significant digits requested is fewer than the number of significant digits in the system's representation of the number, the number will be rounded at the last digit written.

A space for a sign character is reserved for numbers in fixed-point format. If the number is negative, the minus sign is placed in that space. If the number is positive, the plus sign is not displayed; however, the space for the sign character is still reserved and a blank is displayed. Because of this characteristic, the extra space should be considered when output for fixed-point numbers is being formatted.

**Examples**

| Statement | Result |
|---|---|
| write(f,1.2345:20) | " 1.2345000000000E+00" |
| write(f,-27.1828e-3:14) | "-2.7182800E-02" |
| write(f,0.31:3) | " 3.1E-01" |
| write(f,-96E12:7) | "-9.6E+13" |
| write(f,0.317269:3) | " 3.2E-01" |
| write(f,-965E12:7) | "-9.7E+14" |
| write(f,0.31726e7:7:3) | "3172600.000" |
| write(f,-965E12:1:7) | "-965000000000000.0000000" |
| write(f,0.31726e7:13:3) | " 3172600.000" |
| write(f,-965E-2:12:7) | " -9.6500000" |
| write(f,3.1776e-1:13:3) | " 0.318" |
| write(f,-962.5E-2:12:2) | " -9.63" |

See also

## TRUNC FUNCTION

<trunc function>

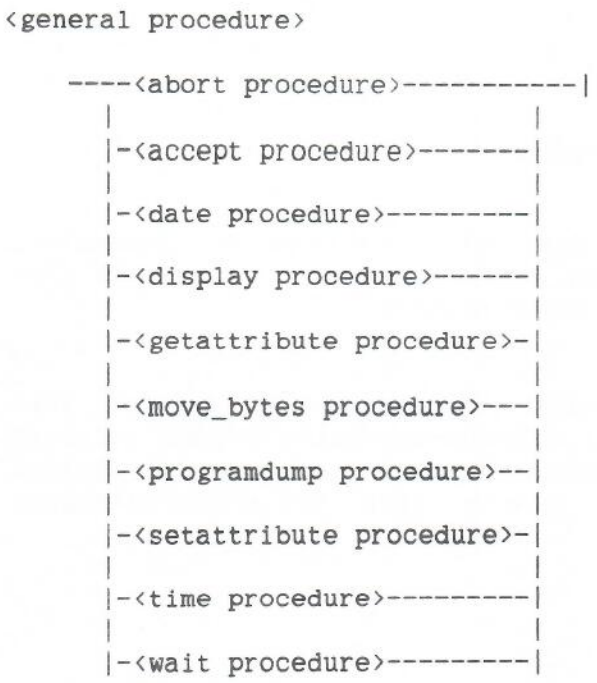

The <trunc function> returns the integer value, computed by truncation, of the specified <real expression>. If the result is greater than "maxint" or less than "-maxint", an error occurs.

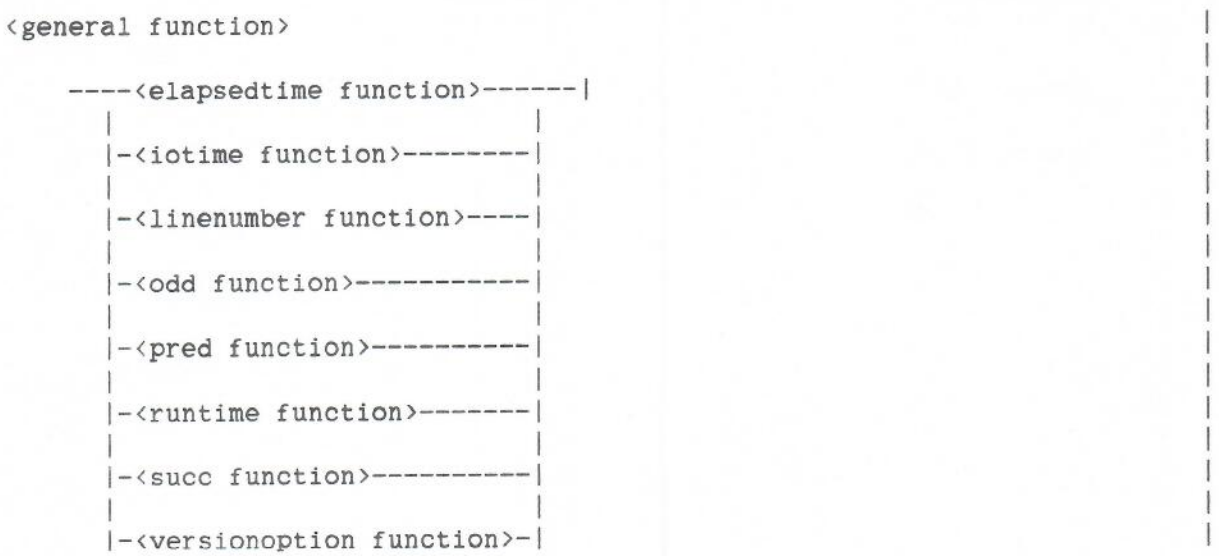
If the nearest integer to the <fixed-pt expression> is greater than "maxint" or less than "-maxint", a double-precision integer value is returned. It is an error if the absolute value of the integer nearest to the <fixed-pt expression> is greater than the double-precision integer value.

**Example**

| Statement | Result |
|---|---|
| trunc(3.5) | 3 |
| trunc(-3.5) | -3 |

## 5.8 GENERAL PROCEDURES AND FUNCTIONS

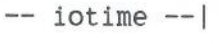



Many general procedures and functions are extensions to ANSI Pascal that allow the program access to system-specific features, such as file attributes, the program's accumulated run time, I/O time, and elapsed time, the interface to the Operator Display Terminal (ODT), and the system's time and date values. Other general procedures and functions listed here are part of ANSI Pascal and provide features not described elsewhere in this manual.

## IOTIME FUNCTION

<iotime function>




The <iotime function> returns, as a real value in units of seconds, the total I/O time that has been charged to the program.

The <iotime function> is a Unisys extension to ANSI Pascal.

## LINENUMBER FUNCTION

<linenumber function>

```
-- linenumber --|
```

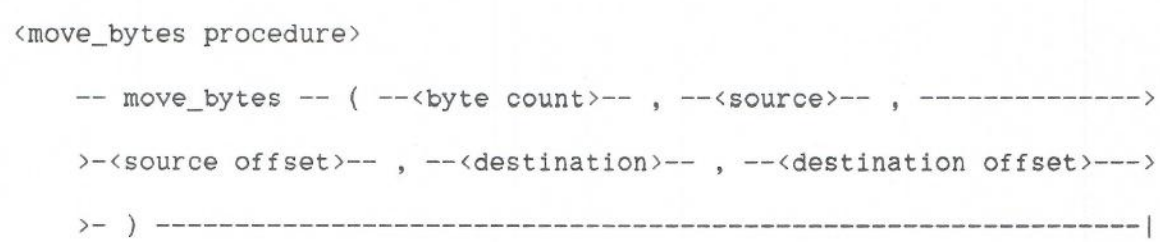
The <linenumber function> returns, as an integer value, the sequence number of the source file record where the function is used. The integer value returned by the <linenumber function> is in the range 0 through 99999999. This value is extracted from columns 73 through 80 of the record.

**Example**

```
PROCEDURE Assert( value1, value2, line_nbr : INTEGER );
BEGIN
IF value1 <> value2 THEN
   write( tracefile,
          'Assertion failure at line number ',
          line_nbr, '.', ' Value 1 = ', value1,
          ', Value 2  = ', value2, '.'
        );
END; {Assert}
...
Assert( linklibrary( GeneralSupportLib ),
        libres( SuccessfulLink ),
        linenumber
      );
...
```

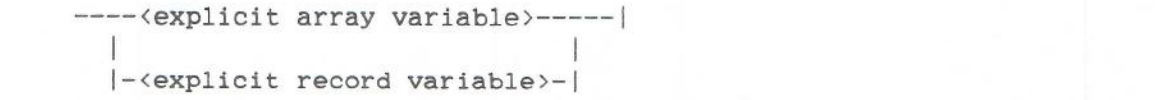
This program fragment checks to ensure that a link to the library "GeneralSupportLib" was successful. If not, the program writes out the message and sends the line number of the record where the link was unsuccessful.

## MOVE BYTES PROCEDURE

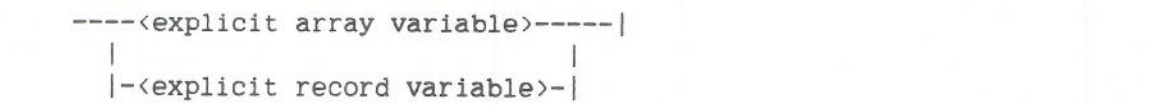



<byte count>
<source offset>
<destination offset>


<source>




<destination>



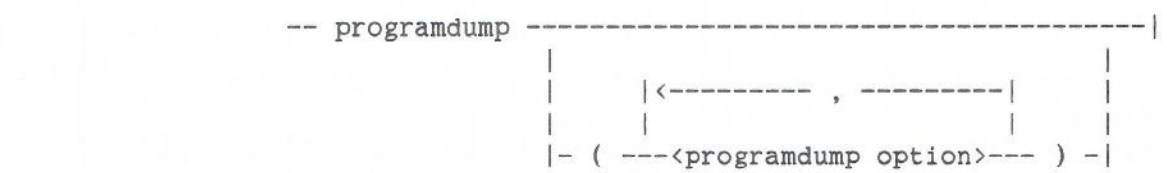

The <move-byte procedure> moves contiguous bytes from one place to another. It takes <byte count> contiguous bytes from <source>, starting at the <source offset> byte, and moves the contiguous bytes to <destination>, starting at the <destination offset> byte. Both <source offset> and <destination offset> begin at 1 (that is, they are 1 relative).

If the <move_bytes procedure> is used with an overlapping source and destination, the result is undefined. For example, if the first byte of the destination is the last byte of the source, the last byte moved may have the value of the source's first byte instead of its original value. If the source and destination must overlap, do the move in two steps.

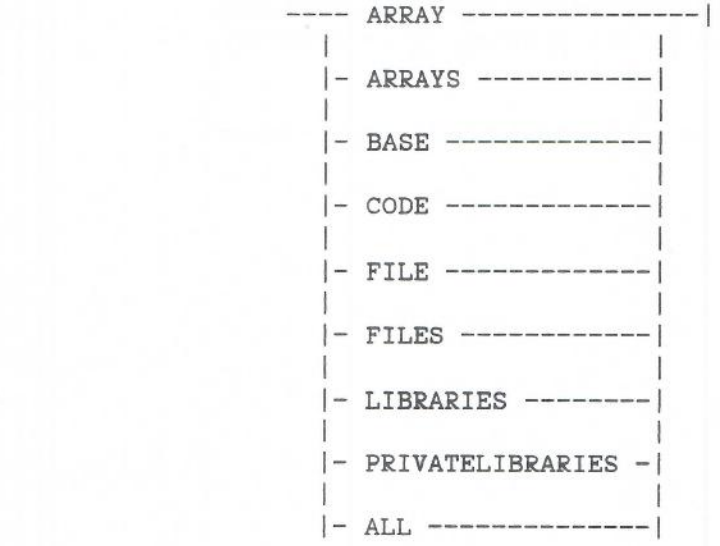
The <move_bytes procedure> is a Unisys extension to ANSI Pascal.

## PROGRAMDUMP PROCEDURE

<programdump procedure>




<programdump option>




The <programdump procedure> calls a procedure in the Master Control Program (MCP) that writes the contents of the program stack to the taskfile of the program. The <programdump option> specifies the items of the stack that are to be analyzed and written. The information produced by the PROGRAMDUMP call is written to the file specified by the TASKFILE task attribute of the program. Although task attribute access is not yet available in Pascal, the TASKFILE file attributes can be file equated at run time.

If no <programdump option> is specified, the stack is analyzed and printed according to the specifications in the OPTION task attribute of the program. If the contents of the program arrays are to be printed, the option ARRAY or ARRAYS must be specified.

The bottom (or "base") of the program stack is printed if the BASE option is specified. The MCP uses a portion of each stack to contain various words needed to control, identify, and log the program.

The segment dictionary of the program is printed if the CODE option is specified. The actual code is printed only for segments that are referenced by the program at the time of the PROGRAMDUMP call. Value arrays in the segment dictionary are printed when both the CODE option and either the ARRAY or ARRAYS option are specified.

If the program files are to be printed and analyzed, the FILE or FILES option must be specified. As each file is encountered, each word of the File Information Block (FIB) is separately named and, in some cases, analyzed.

The LIBRARIES option causes the stacks of all libraries being used by the program to be printed. The PRIVATELIBRARIES option causes the stacks of all private libraries being used by the program to be printed.

Specifying the ALL option is equivalent to specifying all the other options.

**Example**

```
ProgramDump;

ProgramDump( array, files ):

IF IdNote.NextLink <> NIL THEN
   BEGIN
   display('*** Program xyz taking program dump ***');
   ProgramDump( all );
   END;
```

The first use of the <programdump procedure> analyzes and writes the program stack information according to whatever specifications were in the OPTION task attribute of the program.

The second use of the <programdump procedure> analyzes the arrays and files associated with the program.

The third use of the <programdump procedure> analyzes all information about the program and writes the information to the task file.

## VERSIONOPTION FUNCTION

<versionoption function>

```
-- versionoptionfunction --|
```

The <versionoption function> returns to a <versionoption type> record the version, cycle, and patch numbers defined by the most recent VERSION compiler control option. The integer values returned are <integer> type and are in the following ranges:

| Parameter | Range |
|---|---|
| <release> | 0..99 |
| <cycle> | 0..999 |
| <patch> | 0..9999 |

The <versionoption type> is a predefined identifier.

**Example**

```
$VERSION 36.186.1986


PROGRAM version_example(  output  );
...
VAR
   version_nbr      : versionoptiontype;
   year             : 0..9999;
   month            : 1..12;
   day              : 1..31;
   hour             : 0..23;
   minute           : 0..59;
   second           : 0..59;
BEGIN
   ...
   version_nbr :=versionoption;
   date( year, month, day );
   time( hour, minute, second );
   ...
   writeln( output, 'Compiled with sample test version',
          version_nbr.release : 1, '.',
          version_nbr.cycle : 1,'.',
          version_nbr.patch : 1,
          { the ' : 1' writes the numbers in the smallest number of
          { spaces possible }
          ' on ', month : 1, '/', day : 1, '/', year : 1,
          ' at ', hour  : 1, ':', minute : 1, '.'
        );
   { Using the "with" statement...}
   with version_nbr do
      writeln( output, 'Compiled with sample test version',
             release : 1, '.', cycle : 1, '.', patch : 1, '.');
   ...
END.
```

This program fragment obtains the release number, the cycle number, and the patch number assigned at the beginning of the program and writes those numbers to the output file. In this example, the release number is 36, the cycle number is 186, and the patch number is 1986.

## SUBFILE VARIABLE

<subfile variable>

```
--<file variable>-- ( -- subfile --<subfile index>-- ) --|
```

<subfile index>

```
--<integer expression>--|
```

A port file (a file having PORT as its KIND attribute) can have one or more associated "subfiles," each of which can be connected to a different complementary subfile. A <subfile variable> can be used instead of a <file variable> in certain predefined procedures and functions to refer to a particular subfile. In all cases, the type of the <subfile variable> is the same as the type of the <file variable> that it qualifies. Note also that all subfiles share the buffer variable of their associated <file variable>.

For more information on the use of subfiles in Pascal, refer to "Open Procedure," "Close Procedure," "Get Procedure," and "Put Procedure" under "File Handling Procedures and Functions." Also refer to "Wait Procedure," "Getattribute Procedure" and "Setattribute Procedure" under "General Procedures and Functions."

The <subfile variable> is a Unisys extension to ANSI Pascal.

See also
File Handling Procedures and Functions. . . . . 230
General Procedures and Functions. . . . . 386

## TEMPLATE VARIABLE

A <template variable> is a <variable> declared as a <template type>.

## TEXTFILE VARIABLE

A <textfile variable> is an <entire variable> declared as a <textfile type>.

**VERSIONOPTION VARIABLE**

A <versionoption variable> is a <variable> declared as a <versionoption type>.

**VLSTRING VARIABLE**

A <vlstring variable> is a <variable> declared as a <vlstring type>.

**X-BOOL VARIABLE**

An <x-bool variable> is a <variable> declared as an <x-bool type>.

**X-INTEGER VARIABLE**

An <x-integer variable> is a <variable> declared as an <integer type>.

**X-NUM VARIABLE**

An <x-num variable> is a <variable> declared as an <x-num type>.

**X-REAL VARIABLE**

An <x-real variable> is a <variable> declared as an <x-real type>.

**X-WORD VARIABLE**

An <x-word variable> is a <variable> declared as an <x-num type>.

## Reserved Word

<reserved word>

| | | | | | | |
|---|---|---|---|---|---|---|
| AND | DIV | FOR | INTERFACE | NOT | PROGRAM | TYPE |
| ARRAY | DO | FROM | LABEL | OF | RECORD | UNTIL |
| BEGIN | DOWNTO | FUNCTION | LIBRARY | OR | REPEAT | USAGE |
| CAND | ELSE | GOTO | MOD | OTHERWISE | SET | VAR |
| CASE | END | IF | MODULE | PACKED | THEN | WHILE |
| CONST | EXPORT | IMPORT | NIL | PROCEDURE | TO | WITH |
| COR | FILE | IN | | | | |

The <reserved word>s are language keywords that cannot be redefined by the programmer. In general, these are words the compiler uses to recognize declarations, statements, and operators.

## Predefined Identifier

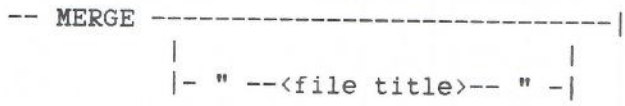
<predefined identifier>

| | | | |
|---|---|---|---|
| abort | dispose | log | scan_neq |
| abs | elapsedtime | mark | seek |
| accept | eof | max | setattribute |
| addstation | eoln | maxint | sfixed |
| adjust_template | erf | min | sin |
| arccos | exp | move_bytes | sinh |
| arcsin | false | new | skiptochannel |
| arctan | filevalue | odd | sqr |
| arctanh | fillchar | open | sqrt |
| attach_template | fixed | ord | string |
| Boolean | fixeddiv | output | succ |
| Boolean1 | freeze | pack | tan |
| Boolean4 | gamma | page | tanh |
| cancel | get | pos | template |
| char | get_bytes | pred | text |
| chr | getattribute | programdump | time |
| close | hex | put | true |
| concat | input | put_bytes | trunc |
| cos | insert | random | u_display |
| cosh | integer | read | unpack |
| cotan | integer48 | readln | update |
| date | integer96 | real | versionoption |
| delete | iores | real48 | versionoptiontype |
| deletestation | iotime | release | wait |
| delinklibrary | length | reset | word48 |
| dfhvalue | libraryvalue | rewrite | word96 |
| digits | libres | round | write |
| digits_s | linenumber | runtime | writeln |
| display | linklibrary | s_digits | z_display |
| display_s | ln | s_display | |
| display_z | lngamma | scan_eql | |

<Predefined identifier>s are <identifier>s that have a predefined meaning in Pascal; as with user-defined <identifier>s, <predefined identifier>s can be redefined, but the former definition becomes unavailable within the scope of the redefinition.

## MERGE OPTION

<merge option>




(Type: Boolean, Default: FALSE)

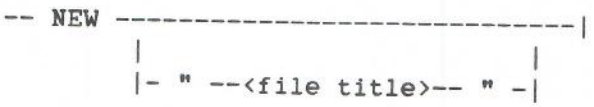
The MERGE option, when enabled, merges source language records from the primary input file (CARD) with source language records from the secondary input file (SOURCE). The <file title>, if present, causes the specified file to be read as the SOURCE file (the <file title> is used to assign the TITLE attribute of the SOURCE file). If no <file title> is specified, SOURCE is assumed. The single quotation mark (') can be substituted for the double quotation mark (") when delimiting the <file title>, but it must be used in pairs.

This option remains enabled throughout a compilation and cannot be disabled. Subsequent attempts to SET or RESET this option are treated as errors and are ignored.

This option can appear only on a compiler control record that is placed before any Pascal source record in the CARD file.

## NEW OPTION

<new option>




(Type: Boolean, Default: FALSE)

The NEW option, when enabled, creates a new source language symbolic (NEWSOURCE) of all source language records accepted for compilation. This new symbolic includes any input records that were omitted by the OMIT option and any permanent compiler control records, but it does not include any records that were discarded by enabling the DELETE option or the VOID option.

The <file title>, if present, causes the specified file to be written as the NEWSOURCE file (the <file title> is used to assign the TITLE attribute of the NEWSOURCE file). If no <file title> is specified, NEWSOURCE is assumed. The single quotation mark (') can be substituted for the double quotation mark (") when delimiting the <file title>, but it must be used in pairs.

This option remains enabled throughout a compilation and cannot be disabled. Subsequent attempts to SET or RESET this option are treated as errors and are ignored.

See also
DELETE Option . . . . . 446
OMIT Option . . . . . 457
VOID Option . . . . . 469

## NOBOUNDS OPTION

<nobounds option>


(Type: Boolean, Default: FALSE)

The NOBOUNDS option, when enabled, eliminates all range-checking code that the compiler would normally generate to verify the assignment compatibility of set variables, subrange variables, and vlstring variables. Range checking performed by the system, such as the range check on an array index when the array is an <entire variable> (not part of a record), cannot be disabled.

See also
Variables . . . . . 407

**repetitive statement**

A structured statement that specifies that its subcomponent statements are to be repeated.

**reserved word**

A word that has special meaning within a programming language and that generally cannot be redefined or redeclared by the programmer.

**run time**

The time during which an object code file is executed. Run time is also referred to as "execution time."

**same type**

A term applied to two types when they have been defined as equivalent to each other or both equivalent to a third type.

**schema**

An outline or plan.

In Pascal, an array declaration that includes one or more dynamic discriminants. In other words, a schema is a type of incomplete array declaration. Using an array schema as a formal parameter makes it possible to pass arrays with different bounds and different numbers of elements to the same formal parameter.

**schemata**

The plural of schema.

**sequential statement**

A structured statement in which the subcomponent statements are executed in the order in which they appear, without conditions or repetitions.

**simple statement**

A statement that performs a single action and contains no other statements.

**source program**

A program coded in a language one or more steps removed from machine language. It must be translated into machine language before use.

**standard character set**

The character set used by the compiler to represent the "char" type and string types. The standard character set is EBCDIC unless changed to ASCII using the STRINGS compiler control option.

**structured statement**

A statement that can have statements as subcomponents.

**structured type**

A data type that defines variables that can have multiple components.

**substring**

A portion of a string or vlstring variable.

**successor**

In an ordinal type, the value having the next highest ordinal number.

**TCP/IP**

See "Transmission Control Protocol/Internet Protocol"

**Transmission Control Protocol/Internet Protocol (TCP/IP)**

A host-to-host data communications protocol. In conjunction with BNA, Version 2, the Unisys TCP/IP software product provides a link to incompatible systems across a TCP/IP network, using the familiar BNA interface.

# BIBLIOGRAPHY

Advanced Data Dictionary System (ADDS) User's Guide (form 1180551). Unisys Corporation.

IEEE Standard Pascal, American National Standard. Computer Programming Language ANSI/IEEE, 770X3.97-1983.

Binder Programming Reference Manual (form 5014582). Unisys Corporation. Formerly Binder Reference Manual.

BNA Architectural Description Reference Manual, Volume 1 (form 1132172). Unisys Corporation.

BNA Host Services Operations Guide (form 1170339). Unisys Corporation.

BNA Version 1 Program Agent Programming Guide (form 1169653). Unisys Corporation.

CANDE Operations Reference Manual (form 1169869). Unisys Corporation.

Editor User's Guide (form 1169976). Unisys Corporation.

I/O Subsystem Programming Reference Manual (form 1169984). Unisys Corporation.

Message Translation Utility User's Guide (form 1169554). Unisys Corporation.

Operator Display Terminal (ODT) System Commands Operations Reference Manual (form 1169612). Unisys Corporation.

Pascal Programming Reference Manual, Volume 2: Product Interfaces (form 1170107). Unisys Corporation.

Pascal User Manual and Report. K. Jensen and N. Wirth. New York: Springer-Verlag, 1978.

System Software Utilities Operations Reference Manual (form 1170024).
Unisys Corporation.

Work Flow Language (WFL) Programming Reference Manual (form 1169802).
Unisys Corporation.

# NOTES

# NOTES

# NOTES