Clause 3

Types

This ~~section~~ clause[1] describes the various categories of types that are provided by the language as well as those specific types that are predefined. The declarations of all predefined types are contained in package STANDARD, the declaration of which appears in ~~Section~~ Clause[2] 14.

A type is characterized by a set of values and a set of operations. The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type. The remaining operations of a type are the basic operations and the predefined operators (see 7.2). These operations are each implicitly declared for a given type declaration immediately after the type declaration and before the next explicit declaration, if any.

A *basic operation* is an operation that is inherent in one of the following:

— An assignment (in assignment statements and initializations)

— An allocator

— A selected name, an indexed name, or a slice name

— A qualification (in a qualified expression), an explicit type conversion, a formal or actual part in the form of a type conversion, or an implicit type conversion of a value of type *universal_integer* or *universal_real* to the corresponding value of another numeric type

— A numeric literal (for a universal type), the literal **null** (for an access type), a string literal, a bit string literal, an aggregate, or a predefined attribute

There are five classes of types. *Scalar* types are integer types, floating point types, physical types, and types defined by an enumeration of their values; values of these types have no elements. *Composite* types are array and record types; values of these types consist of element values. *Access* types provide access to objects of a given type. *File* types provide access to objects that contain a sequence of values of a given type. *Protected types* provide atomic and exclusive access to variables accessible to multiple processes.

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case where the constraint imposes no restriction is also included); a value is said to *satisfy* a constraint if it satisfies the corresponding condition. A *subtype* is a type together with a constraint. A value is said to *belong to a subtype* of a given type if it belongs to the type and satisfies the constraint; the given type is called the *base type* of the subtype. A type is a subtype of itself; such a subtype is said to be *unconstrained* (it corresponds to a condition that imposes no restriction). The base type of a type is the type itself.

---

1. To conform to IEEE rules.
2. To conform to IEEE rules.

The set of operations defined for a subtype of a given type includes the operations defined for the type; however, the assignment operation to an object having a given subtype only assigns values that belong to the subtype. Additional operations, such as qualification (in a qualified expression) are implicitly defined by a subtype declaration.

The term *subelement* is used in this manual in place of the term element to indicate either an element, or an element of another element or subelement. Where other subelements are excluded, the term *element* is used instead.

A given type must not have a subelement whose type is the given type itself.

A *member* of an object is either

—A slice ~~oif~~ of[3] the object,

—A subelement of the object, or

—A slice of a subelement of the object.

The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term *array object* is used for an object whose type is an array type; similarly, the term *access value* is used for a value of an access type.

NOTE

—The set of values of a subtype is a subset of the values of the base type. This subset need not be a proper subset.

3.1 Scalar Types

Scalar types consist of *enumeration types*, *integer types*, *physical types*, and *floating point types*. Enumeration types and integer types are called *discrete* types. Integer types, floating point types, and physical types are called *numeric* types. All scalar types are ordered; that is, all relational operators are predefined for their values. Each value of a discrete or physical type has a position number that is an integer value.

```
scalar_type_definition ::=
        enumeration_type_definition | integer_type_definition
      | floating_type_definition    | physical_type_definition

range_constraint ::= range range

range ::=
        range_attribute_name
      | simple_expression direction simple_expression

direction ::= to | downto
```

A range specifies a subset of values of a scalar type. A range is said to be a *null* range if the specified subset is empty.

The range L **to** R is called an *ascending* range; if L > R, then the range is a null range. The range L **downto** R is called a *descending* range; if L < R, then the range is a null range. The smaller of L and R is called the *lower bound*, and the larger, the *upper bound*, of the range. The value V is said to *belong to the range* if the relations (*lower bound* <= V) and (V <= *upper bound*) are both true and the range is not a null range. The operators >, <, and <= in the preceding definitions are the predefined operators of the applicable scalar type.

3. Typo.

For values of discrete or physical types, a value V1 is said to be *to the left of* a value V2 within a given range if both V1 and V2 belong to the range and either the range is an ascending range and V2 is the successor of V1 or the range is a descending range and V2 is the predecessor of V1. A list of values of a given range is in *left to right order* if each value in the list is to the left of the next value in the list within that range, except for the last value in the list.

If a range constraint is used in a subtype indication, the type of the expressions (likewise, of the bounds of a range attribute) must be the same as the base type of the type mark of the subtype indication. A range constraint is *compatible* with a subtype if each bound of the range belongs to the subtype or if the range constraint defines a null range. Otherwise, the range constraint is not compatible with the subtype.

The direction of a range constraint is the same as the direction of its range.

NOTE

—Indexing and iteration rules use values of discrete types.

3.1.1 Enumeration types

An enumeration type definition defines an enumeration type.

```
enumeration_type_definition ::=
    ( enumeration_literal { , enumeration_literal } )

enumeration_literal ::=  identifier | character_literal
```

The identifiers and character literals listed by an enumeration type definition must be distinct within the enumeration type definition. Each enumeration literal is the declaration of the corresponding enumeration literal; for the purpose of determining the parameter and result type profile of an enumeration literal, this declaration is equivalent to the declaration of a parameterless function whose designator is the same as the enumeration literal and whose result type is the same as the enumeration type.

An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character literal.

Each enumeration literal yields a different enumeration value. The predefined order relations between enumeration values follow the order of corresponding position numbers. The position number of the value of the first listed enumeration literal is zero; the position number for each additional enumeration literal is one more than that of its predecessor in the list.

If the same identifier or character literal is specified in more than one enumeration type definition, the corresponding literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal is determined according to the rules for overloaded subprograms (see 2.3).

Each enumeration type definition defines an ascending range.

Examples:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING, FALLING, AMBIGUOUS) ;

type BIT is ('0','1') ;

type SWITCH_LEVEL is ('0','1','X') ;                    -- Overloads '0' and '1'
```

3.1.1.1 Predefined enumeration types

The predefined enumeration types are CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS.

The predefined type CHARACTER is a character type whose values are the 256 characters of the ISO 8859-1 character set. Each of the 191 graphic characters of this character set is denoted by the corresponding character literal.

The declarations of the predefined types CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, and FILE_OPEN_STATUS appear in package STANDARD in ~~Section~~ Clause[4] 14.

NOTES

1—The first ~~17~~ 33[5] nongraphic elements of the predefined type CHARACTER (from NUL through DEL) are the ASCII abbreviations for the nonprinting characters in the ASCII set (except for those noted in ~~Section~~ Clause[6] 14). The ASCII names are chosen as ISO 8859-1 does not assign them abbreviations. The next ~~16~~ 32[7] (C128 through C159) are also not assigned abbreviations, so names unique to VHDL are assigned.

2—Type BOOLEAN can be used to model either active high or active low logic depending on the particular conversion functions chosen to and from type BIT.

3.1.2 Integer types

An integer type definition defines an integer type whose set of values includes those of the specified range.

integer_type_definition ::= range_constraint

An integer type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the integer type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in an integer type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Integer literals are the literals of an anonymous predefined type that is called *universal_integer* in this standard. Other integer types have no literals. However, for each integer type there exists an implicit conversion that converts a value of type *universal_integer* into the corresponding value (if any) of the integer type (see 7.3.5).

The position number of an integer value is the corresponding value of the type *universal_integer*.

The same arithmetic operators are predefined for all integer types (see 7.2). It is an error if the execution of such an operation (in particular, an implicit conversion) cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the integer type).

An implementation may restrict the bounds of the range constraint of integer types other than type *universal_integer*. However, an implementation must allow the declaration of any integer type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

4. To conform to IEEE rules.
5. IR1000.1.5.
6. To conform to IEEE rules.
7. IR1000.1.5.

Examples:

type TWOS_COMPLEMENT_INTEGER **is range** –32768 **to** 32767;

type BYTE_LENGTH_INTEGER **is range** 0 **to** 255;

type WORD_INDEX **is range** 31 **downto** 0;

subtype HIGH_BIT_LOW **is** BYTE_LENGTH_INTEGER **range** 0 **to** 127;

3.1.2.1 Predefined integer types

The only predefined integer type is the type INTEGER. The range of INTEGER is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range.

NOTE

—The range of INTEGER in a particular implementation ~~may be determined from the~~ is determinable from the values of its[8] 'LOW and 'HIGH attributes.

3.1.3 Physical types

Values of a physical type represent measurements of some quantity. Any value of a physical type is an integral multiple of the primary unit of measurement for that type.

physical_type_definition ::=
    range_constraint
        units
            primary_unit_declaration
            { secondary_unit_declaration }
        end units [*physical_type*_simple_name]

primary_unit_declaration ::= identifier ;[9]

secondary_unit_declaration ::= identifier = physical_literal ;

physical_literal ::= [abstract_literal] *unit*_name

A physical type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the physical type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a physical type definition must be a locally static expression of some integer type, but the two bounds need not have the same integer type. (Negative bounds are allowed.)

Each unit declaration (either the primary unit declaration or a secondary unit declaration) defines a *unit name*. Unit names declared in secondary unit declarations must be directly or indirectly defined in terms of integral multiples of the primary unit of the type declaration in which they appear. The position numbers of unit names need not lie within the range specified by the range constraint.

If a simple name appears at the end of a physical type declaration, it must repeat the identifier of the type declaration in which the physical type definition is included.

8. IR1000.4.7.
9. Noted by Bert Molenkamp.

The abstract literal portion (if present) of a physical literal appearing in a secondary unit declaration must be an integer literal.

A physical literal consisting solely of a unit name is equivalent to the integer 1 followed by the unit name.

There is a position number corresponding to each value of a physical type. The position number of the value corresponding to a unit name is the number of primary units represented by that unit name. The position number of the value corresponding to a physical literal with an abstract literal part is the largest integer that is not greater than the product of the value of the abstract literal and the position number of the accompanying unit name.

The same arithmetic operators are predefined for all physical types (see 7.2). It is an error if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the physical type).

An implementation may restrict the bounds of the range constraint of a physical type. However, an implementation must allow the declaration of any physical type whose range is wholly contained within the bounds –2147483647 and +2147483647 inclusive.

Examples:

```
type DURATION is range –1E18 to 1E18
    units
        fs;                         -- femtosecond
        ps    =  1000 fs;           -- picosecond
        ns    =  1000 ps;           -- nanosecond
        us    =  1000 ns;           -- microsecond
        ms    =  1000 us;           -- millisecond
        sec   =  1000 ms;           -- second
        min   =  60 sec;            -- minute
    end units;

type DISTANCE is range 0 to 1E16
    units
    -- primary unit:
        ~~A~~ Å[10];                -- angstrom

    -- metric lengths:
        nm    =  10 ~~A~~ Å[11];    -- nanometer
        um    =  1000 nm;           -- micrometer (or micron)
        mm    =  1000 um;           -- millimeter
        cm    =  10 mm;             -- centimeter
        m     =  1000 mm;           -- meter
        km    =  1000 m;            -- kilometer

    -- English lengths:
        mil   =  254000 ~~A~~ Å[12]; -- mil
        inch  =  1000 mil;          -- inch
        ft    =  12 inch;           -- foot
        yd    =  3 ft;              -- yard
        fm    =  6 ft;              -- fathom
        mi    =  5280 ft;           -- mile
        lg    =  3 mi;              -- league
    end units DISTANCE;
```

10. Change that should have been made in 1076-1993.
11. Change that should have been made in 1076-1993.
12. Change that should have been made in 1076-1993.

```
variable x: distance;  variable y: duration;  variable z: integer;

x := 5 A Å[13] + 13 ft – 27 inch;
y := 3 ns + 5 min;
z := ns / ps;
x := z * mi;
y := y/10;
z := 39.34 inch / m;
```

NOTES

1— The 'POS and 'VAL attributes may be used to convert between abstract values and physical values.

2— The value of a physical literal whose abstract literal is either the integer value zero or the floating-point value zero is the same value (specifically zero primary units) no matter what unit name follows the abstract literal.

3.1.3.1 Predefined physical types

The only predefined physical type is type TIME. The range of TIME is implementation dependent, but it is guaranteed to include the range –2147483647 to +2147483647. It is defined with an ascending range. All specifications of delays and pulse rejection limits must be of type TIME. The declaration of type TIME appears in package STANDARD in ~~Section~~ Clause[14] 14.

By default, the primary unit of type TIME (1 femtosecond) is the *resolution limit* for type TIME. Any TIME value whose absolute value is smaller than this limit is truncated to zero (0) time units. An implementation may allow a given execution of a model (see 12.6) to select a secondary unit of type TIME as the resolution limit. Furthermore, an implementation may restrict the precision of the representation of values of type TIME and the results of expressions of type TIME, provided that values as small as the resolution limit are representable within those restrictions. It is an error if a given unit of type TIME appears anywhere within the design hierarchy defining a model to be executed, and if the position number of that unit is less than that of the secondary unit selected as the resolution limit for type TIME during the execution of the model, unless that unit is part of a physical literal whose abstract literal is either the integer value zero or the floating-point value zero[15].

NOTE

—By selecting a secondary unit of type TIME as the resolution limit for type TIME, it may be possible to simulate for a longer period of simulated time, with reduced accuracy, or to simulate with greater accuracy for a shorter period of simulated time.

Cross-References: Delay and rejection limit in a signal assignment, 8.4; Disconnection, delay of a guarded signal, 5.3; Function NOW, 14.2; Predefined attributes, functions of TIME, 14.1; Simulation time, 12.6.2 and 12.6.3; Type TIME, 14.2; Updating a projected waveform, 8.4.1; Wait statements, timeout clause in, 8.1; Elaboration of a declarative part, 12.3[16].

3.1.4 Floating-point types

Floating-point types provide approximations to the real numbers. ~~Floating point types are useful for models in which the precise characterization of a floating point calculation is not important or not determined.~~[17]

```
floating_type_definition ::=  range_constraint
```

13. Change that should have been made in 1076-1993.
14. To conform to IEEE rules.
15. LCS 9.
16. LCS 9.
17. LCS 22.

A floating type definition defines both a type and a subtype of that type. The type is an anonymous type, the range of which is selected by the implementation; this range must be such that it wholly contains the range given in the floating type definition. The subtype is a named subtype of this anonymous base type, where the name of the subtype is that given by the corresponding type declaration and the range of the subtype is the given range.

Each bound of a range constraint that is used in a floating type definition must be a locally static expression of some floating-point type, but the two bounds need not have the same floating point type. (Negative bounds are allowed.)

Floating-point literals are the literals of an anonymous predefined type that is called *universal_real* in this standard. Other floating-point types have no literals. However, for each floating-point type there exists an implicit conversion that converts a value of type *universal_real* into the corresponding value (if any) of the floating-point type (see 7.3.5).

The same arithmetic operations are predefined for all floating-point types (see 7.2). A design is erroneous if the execution of such an operation cannot deliver the correct result (that is, if the value corresponding to the mathematical result is not a value of the floating-point type).

An implementation must choose a representation for all floating-point types except for *universal_real* that conforms either to IEEE Std 754 or to IEEE Std 854; in either case, a minimum representation size of 64 bits is required for this *chosen representation*.[18]

An implementation may restrict the bounds of the range constraint of floating-point types other than type *universal_real*. However, an implementation must allow the declaration of any floating-point type whose range is wholly contained within the bounds ~~−1.0E38 and +1.0E38 inclusive~~ allowed by the chosen representation[19]. ~~The representation of floating point types must include a minimum of six decimal digits of precision.~~[20]

NOTE

—An implementation is not required to detect errors in the execution of a predefined floating point arithmetic operation, since the detection of overflow conditions resulting from such operations ~~may~~ might[21] not be easily accomplished on many host systems.

3.1.4.1 Predefined floating point types

The only predefined floating point type is the type REAL. The range of REAL is host-dependent, but it is guaranteed to ~~include the range −1.0E38 to +1.0E38 inclusive~~ be the largest allowed by the chosen representation[22]. It is defined with an ascending range.

NOTE

—The range of REAL in a particular implementation ~~may be determined from the~~ is determinable from the values of its[23] 'LOW and 'HIGH attributes.

3.2 Composite types

Composite types are used to define collections of values. These include both arrays of values (collections of values of a homogeneous type) and records of values (collections of values of potentially heterogeneous types).

18. LCS 22.
19. LCS 22.
20. LCS 22.
21. IR1000.4.7.
22. LCS 22
23. IR1000.4.7.

composite_type_definition ::=
    array_type_definition
  | record_type_definition

An object of a composite type represents a collection of objects, one for each element of the composite object. ~~A composite type may only contain elements that are of scalar, composite, or access types;~~ It is an error if a composite type contains[24] elements of file types or protected types ~~are not allowed in a composite type~~[25]. Thus an object of a composite type ultimately represents a collection of objects of scalar or access types, one for each non-composite subelement of the composite object.

3.2.1 Array types

An array object is a composite object consisting of elements that have the same subtype. The name for an element of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of its elements.

array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::=
  array (index_subtype_definition { , index_subtype_definition })
    of *element*_subtype_indication

constrained_array_definition ::=
  array index_constraint **of** *element*_subtype_indication

index_subtype_definition ::= type_mark **range** <>

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete*_subtype_indication | range

An array object is characterized by the number of indices (the dimensionality of the array); the type, position, and range of each index; and the type and possible constraints of the elements. The order of the indices is significant.

A one-dimensional array has a distinct element for each possible index value. A multidimensional array has a distinct element for each possible sequence of index values that can be formed by selecting one value for each index (in the given order). The possible values for a given index are all the values that belong to the corresponding range; this range of values is called the *index range*.

An unconstrained array definition defines an array type and a name denoting that type. For each object that has the array type, the number of indices, the type and position of each index, and the subtype of the elements are as in the type definition. The *index subtype* for a given index position is, by definition, the subtype denoted by the type mark of the corresponding index subtype definition. The values of the left and right bounds of each index range are not defined but must belong to the corresponding index subtype; similarly, the direction of each index range is not defined. The symbol <> (called a *box*) in an index subtype definition stands for an undefined range (different objects of the type need not have the same bounds and direction).

A constrained array definition defines both an array type and a subtype of this type:

— The array type is an implicitly declared anonymous type; this type is defined by an (implicit) unconstrained array definition, in which the element subtype indication is that of the constrained array definition and in which the type mark of each index subtype definition denotes the subtype defined by the corresponding discrete range.

24. IR1000.4.7.
25. IR1000.4.7.

— The array subtype is the subtype obtained by imposition of the index constraint on the array type.

If a constrained array definition is given for a type declaration, the simple name declared by this declaration denotes the array subtype.

The direction of a discrete range is the same as the direction of the range or the discrete subtype indication that defines the discrete range. If a subtype indication appears as a discrete range, the subtype indication must not contain a resolution function.

Examples:

—Examples of constrained array declarations:

```
type MY_WORD is array (0 to 31) of BIT ;
    -- A memory word type with an ascending range.

type DATA_IN is array (7 downto 0) of FIVE_LEVEL_LOGIC ;
    -- An input port type with a descending range.
```

—Example of unconstrained array declarations:

```
type MEMORY is array (INTEGER range <>) of MY_WORD ;
    -- A memory array type.
```

—Examples of array object declarations:

```
signal DATA_LINE : DATA_IN ;
    -- Defines a data input line.

variable MY_MEMORY : MEMORY (0 to 2**n–1) ;
    -- Defines a memory of 2n 32-bit words.
```

NOTE

—The rules concerning constrained type declarations mean that a type declaration with a constrained array definition such as

```
type T is array (POSITIVE range MINIMUM to MAX) of ELEMENT;
```

is equivalent to the sequence of declarations

```
subtype index_subtype is POSITIVE range MINIMUM to MAX;
type array_type is array (index_subtype range <>) of ELEMENT;
subtype T is array_type (index_subtype);
```

where *index_subtype* and *array_type* are both anonymous. Consequently, T is the name of a subtype and all objects declared with this type mark are arrays that have the same index range.

3.2.1.1 Index constraints and discrete ranges

An index constraint determines the index range for every index of an array type and, thereby, the corresponding array bounds.

For a discrete range used in a constrained array definition and defined by a range, an implicit conversion to the predefined type INTEGER is assumed if each bound is either a numeric literal or an attribute, and if the type of both bounds (prior to the implicit conversion) is the type *universal_integer*. Otherwise, both bounds must be of the same discrete type, other than *universal_integer*; this type must be determined independently of the context,

but using the fact that the type must be discrete and that both bounds must have the same type. These rules apply also to a discrete range used in an iteration scheme (see 8.9) or a generation scheme (see 9.7).

If an index constraint appears after a type mark in a subtype indication, then the type or subtype denoted by the type mark must not already impose an index constraint. The type mark must denote either an unconstrained array type or an access type whose designated type is such an array type. In either case, the index constraint must provide a discrete range for each index of the array type, and the type of each discrete range must be the same as that of the corresponding index.

An index constraint is *compatible* with the type denoted by the type mark if and only if the constraint defined by each discrete range is compatible with the corresponding index subtype. If any of the discrete ranges defines a null range, any array thus constrained is a *null array*, having no ~~components~~ elements[26]. An array value *satisfies* an index constraint if at each index position the array value and the index constraint have the same index range. (Note, however, that assignment and certain other operations on arrays involve an implicit subtype conversion.)

The index range for each index of an array object is determined as follows:

— For a variable or signal declared by an object declaration, the subtype indication of the corresponding object declaration must define a constrained array subtype (and thereby, the index range for each index of the object). The same requirement exists for the subtype indication of an element declaration, if the type of the record element is an array type, and for the element subtype indication of an array type definition, if the type of the array element is itself an array type.

— For a constant declared by an object declaration, the index ranges are defined by the initial value, if the subtype of the constant is unconstrained; otherwise, they are defined by this subtype (in which case the initial value is the result of an implicit subtype conversion).

— For an attribute whose value is specified by an attribute specification, the index ranges are defined by the expression given in the specification, if the subtype of the attribute is unconstrained; otherwise, they are defined by this subtype (in which case the value of the attribute is the result of an implicit subtype conversion).

— For an array object designated by an access value, the index ranges are defined by the allocator that creates the array object (see 7.3.6).

— For an interface object declared with a subtype indication that defines a constrained array subtype, the index ranges are defined by that subtype.

— For a formal parameter of a subprogram that is of an unconstrained array type and that is associated in whole (see 4.3.2.2), the index ranges are obtained from the corresponding association element in the applicable subprogram call.

— For a formal parameter of a subprogram that is of an unconstrained array type and whose subelements are associated individually (see 4.3.2.2), the index ranges are obtained as follows:

  The directions of the index ranges of the formal parameter are ~~that of the~~ those of the base[27] type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.

— For a formal generic or a formal port of a design entity or of a block statement that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a formal generic) or port map aspect (in the case of a formal port) of the applicable (implicit or explicit) binding indication.

26. IR1000.2.3.
27. IR1000.1.10.

— For a formal generic or a formal port of a design entity or of a block statement that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows:

The directions of the index ranges of the formal generic or formal port are ~~that of the~~ those of the base[28] type of the formal; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the formal.

— For a local generic or a local port of a component that is of an unconstrained array type and that is associated in whole, the index ranges are obtained from the corresponding association element in the generic map aspect (in the case of a local generic) or port map aspect (in the case of a local port) of the applicable component instantiation statement.

— For a local generic or a local port of a component that is of an unconstrained array type and whose subelements are associated individually, the index ranges are obtained as follows:

The directions of the index ranges of the local generic or local port are ~~that of the~~ those of the base[29] type of the local; the high and low bounds of the index ranges are respectively determined from the maximum and minimum values of the indices given in the association elements corresponding to the local.

If the index ranges for an interface object or member of an interface object are obtained from the corresponding association element (when associating in whole) or elements (when associating individually), then they are determined either by the actual part(s) or by the formal part(s) of the association element(s), depending upon the mode of the interface object, as follows:

— For an interface object or member of an interface object whose mode is **in**, **inout**, or **linkage**, if the actual part includes a conversion function or a type conversion, then the result type of that function or the type mark of the type conversion must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object or value denoted by the actual designator(s).

— For an interface object or member of an interface object whose mode is **out**, **buffer**, **inout**, or **linkage**, if the formal part includes a conversion function or a type conversion, then the parameter subytpe of that function or the type mark of the type conversion must be a constrained array subtype, and the index ranges are obtained from this constrained subtype; otherwise, the index ranges are obtained from the object denoted by the actual designator(s).

For an interface object of mode **inout** or **linkage**, the index ranges determined by the first rule must be identical to the index ranges determined by the second rule.

Examples:

```
type Word is array (NATURAL range <>) of BIT;
type Memory is array (NATURAL range <>) of Word (31 downto 0);

constant A_Word: Word := "10011";
    -- The index range of A_Word is 0 to 4
```

28. IR1000.1.10.
29. IR1000.1.10.

```
entity E is
    generic (ROM: Memory);
    port (Op1, Op2: in Word;  Result: out Word);
end entity E;
    -- The index ranges of the generic and the ports are defined by the actuals associated
    -- with an instance bound to E; these index ranges are accessible via the predefined
    -- array attributes (see 14.1).

signal A, B: Word (1 to 4);
signal C: Word (5 downto 0);

Instance: entity E
    generic map ((1 to 2) => (others => '0') (1 to 2 => (others => '0'))[30])
    port map (A, Op2(3 to 4) => B (1 to 2), Op2(2) => B (3), Result => C (3 downto 1));
        -- In this instance, the index range of ROM is 1 to 2 (matching that of the actual),
        -- The index range of Op1  is 1 to 4 (matching the index range of A), the index range
        -- of Op2 is 2 to 4, and the index range of Result is (3 downto 1)
        -- (again matching the index range of the actual).
```

3.2.1.2 Predefined array types

The predefined array types are STRING and BIT_VECTOR, defined in package STANDARD in ~~Section~~ Clause[31] 14.

The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype POSITIVE:

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH ;
type STRING is array (POSITIVE range <>) of CHARACTER ;
```

The values of the predefined type BIT_VECTOR are one-dimensional arrays of the predefined type BIT, indexed by values of the predefined subtype NATURAL:

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH ;
type BIT_VECTOR is array (NATURAL range <>) of BIT ;
```

Examples:

```
variable MESSAGE : STRING(1 to 17) := "THIS IS A MESSAGE" ;

signal LOW_BYTE : BIT_VECTOR (0 to 7) ;
```

3.2.2 Record types

A record type is a composite type, objects of which consist of named elements. The value of a record object is a composite value consisting of the values of its elements.

```
record_type_definition ::=
    record
        element_declaration
      { element_declaration }
    end record [ record_type_simple_name ]
```

30. IR1000.1.3 (as corrected by Ashenden).
31. To conform to IEEE rules.

element_declaration ::=
identifier_list : element_subtype_definition ;

identifier_list ::= identifier { , identifier }

element_subtype_definition ::= subtype_indication

Each element declaration declares an element of the record type. The identifiers of all elements of a record type must be distinct. The use of a name that denotes a record element is not allowed within the record type definition that declares the element.

An element declaration with several identifiers is equivalent to a sequence of single element declarations. Each single element declaration declares a record element whose subtype is specified by the element subtype definition.

If a simple name appears at the end of a record type declaration, it must repeat the identifier of the type declaration in which the record type definition is included.

A record type definition creates a record type; it consists of the element declarations in the order in which they appear in the type definition.

Example:

```
type DATE is
    record
        DAY:    INTEGER range 1 to 31;
        MONTH:  MONTH_NAME;
        YEAR:   INTEGER range 0 to 4000;
    end record;
```

3.3 Access types

An object declared by an object declaration is created by the elaboration of the object declaration and is denoted by a simple name or by some other form of name. In contrast, objects that are created by the evaluation of allocators (see 7.3.6) have no simple name. Access to such an object is achieved by an *access value* returned by an allocator; the access value is said to *designate* the object.

access_type_definition ::= **access** subtype_indication

For each access type, there is a literal **null** that has a null access value designating no object at all. The null value of an access type is the default initial value of the type. Other values of an access type are obtained by evaluation of a special operation of the type, called an *allocator*. Each such access value designates an object of the subtype defined by the subtype indication of the access type definition. This subtype is called the *designated subtype* and the base type of this subtype is called the *designated type*. The designated type must not be a file type or a protected type; moreover, it ~~may~~ must[32] not have a subelement that is a file type or a protected type.

An object declared to be of an access type must be an object of class variable. An object designated by an access value is always an object of class variable.

The only form of constraint that is allowed after the name of an access type in a subtype indication is an index constraint. An access value belongs to a corresponding subtype of an access type either if the access value is the null value or if the value of the designated object satisfies the constraint.

32. IR1000.4.7.

Examples:

```
type ADDRESS is access MEMORY;
type BUFFER_PTR is access TEMP_BUFFER;
```

NOTES

1—An access value delivered by an allocator can be assigned to several variables of the corresponding access type. Hence, it is possible for an object created by an allocator to be designated by more than one variable of the access type. An access value can only designate an object created by an allocator; in particular, it cannot designate an object declared by an object declaration.

2—If the type of the object designated by the access value is an array type, this object is constrained with the array bounds supplied implicitly or explicitly for the corresponding allocator.

3.3.1 Incomplete type declarations

The designated type of an access type can be of any type except a file type or a protected type[33] (see 3.3). In particular, the type of an element of the designated type can be another access type or even the same access type. This permits mutually dependent and recursive access types. Declarations of such types require a prior incomplete type declaration for one or more types.

incomplete_type_declaration ::= **type** identifier ;

For each incomplete type declaration there must be a corresponding full type declaration with the same identifier. This full type declaration must occur later and immediately within the same declarative part as the incomplete type declaration to which it corresponds.

Prior to the end of the corresponding full type declaration, the only allowed use of a name that denotes a type declared by an incomplete type declaration is as the type mark in the subtype indication of an access type definition; no constraints are allowed in this subtype indication.

Example of a recursive type:

```
type CELL;                                  -- An incomplete type declaration.

type LINK is access CELL;

type CELL is
    record
        VALUE: INTEGER;
        SUCC: LINK;
        PRED: LINK;
    end record CELL;
variable HEAD : LINK := new CELL'(0, null, null);
variable \NEXT\ : LINK := HEAD.SUCC;
```

Examples of mutually dependent access types:

```
type PART;                                  -- Incomplete type declarations.
type WIRE;

type PART_PTR is access PART;
type WIRE_PTR is access WIRE;
```

33. Correction—missed during P1076a.

```
type PART_LIST is array (POSITIVE range <>) of PART_PTR;
type WIRE_LIST is array (POSITIVE range <>) of WIRE_PTR;

type PART_LIST_PTR is access PART_LIST;
type WIRE_LIST_PTR is access WIRE_LIST;

type PART is record
    PART_NAME: STRING (1 to MAX_STRING_LEN);
    CONNECTIONS: WIRE_LIST_PTR;
end record;

type WIRE is record
    WIRE_NAME: STRING (1 to MAX_STRING_LEN);
    CONNECTS: PART_LIST_PTR;
end record;
```

3.3.2 Allocation and deallocation of objects

An object designated by an access value is allocated by an allocator for that type. An allocator is a primary of an expression; allocators are described in 7.3.6. For each access type, a deallocation operation is implicitly declared immediately following the full type declaration for the type. This deallocation operation makes it possible to deallocate explicitly the storage occupied by a designated object.

Given the following access type declaration:

```
type AT is access T;
```

the following operation is implicitly declared immediately following the access type declaration:

```
procedure DEALLOCATE (P: inout AT) ;
```

Procedure DEALLOCATE takes as its single parameter a variable of the specified access type. If the value of that variable is the null value for the specified access type, then the operation has no effect. If the value of that variable is an access value that designates an object, the storage occupied by that object is returned to the system and may then be reused for subsequent object creation through the invocation of an allocator. The access parameter P is set to the null value for the specified type.

NOTE

—If ~~a pointer~~ an access value[34] is copied to a second variable and is then deallocated, the second variable is *not* set to **null** and thus references invalid storage.

3.4 File types

A file type definition defines a file type. File types are used to define objects representing files in the host system environment. The value of a file object is the sequence of values contained in the host system file.

file_type_definition ::= **file of** type_mark

The type mark in a file type definition defines the subtype of the values contained in the file. The type mark may denote either a constrained or an unconstrained subtype. The base type of this subtype must not be a file type, an access type, or a protected type. If the base type is a composite type, it must not contain a subelement of an access type, a file type, or a protected type. If the base type is an array type, it must be a one-dimensional array type.

34. Terminological correction.

Examples:

```
file of STRING          -- Defines a file type that can contain
                        -- an indefinite number of strings of arbitrary length.
file of NATURAL         -- Defines a file type that can contain
                        -- only nonnegative integer values.
```

3.4.1 File operations

The language implicitly defines the operations for objects of a file type. Given the following file type declaration:

```
type FT is file of TM;
```

where type mark TM denotes a scalar type, a record type, or a constrained array subtype, the following operations are implicitly declared immediately following the file type declaration:

```
procedure FILE_OPEN (file F: FT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_OPEN (Status: out FILE_OPEN_STATUS;
                     file F: FT;
                     External_Name: in STRING;
                     Open_Kind: in FILE_OPEN_KIND := READ_MODE);

procedure FILE_CLOSE (file F: FT);

procedure READ (file F: FT; VALUE: out TM);

procedure WRITE (file F: FT; VALUE: in TM);

function ENDFILE (file F: FT) return BOOLEAN;
```

The FILE_OPEN procedures open an external file specified by the External_Name parameter and associate it with the file object F. If the call to FILE_OPEN is successful (see below), the file object is said to be *open* and the file object has an *access mode* dependent on the value supplied to the Open_Kind parameter (see 14.2).

— If the value supplied to the Open_Kind parameter is READ_MODE, the access mode of the file object is *read-only*. In addition, the file object is initialized so that a subsequent READ will return the first value in the external file. Values are read from the file object in the order that they appear in the external file.

— If the value supplied to the Open_Kind parameter is WRITE_MODE, the access mode of the file object is *write-only*. In addition, the external file is made initially empty. Values written to the file object are placed in the external file in the order in which they are written.

— If the value supplied to the Open_Kind parameter is APPEND_MODE, the access mode of the file object is *write-only*. In addition, the file object is initialized so that values written to it will be added to the end of the external file in the order in which they are written.

In the second form of FILE_OPEN, the value returned through the Status parameter indicates the results of the procedure call:

— A value of OPEN_OK indicates that the call to FILE_OPEN was successful. If the call to FILE_OPEN specifies an external file that does not exist at the beginning of the call, and if the access mode of the file object passed to the call is write-only, then the external file is created.

— A value of STATUS_ERROR indicates that the file object already has an external file associated with it.

— A value of NAME_ERROR indicates that the external file does not exist (in the case of an attempt to read from the external file) or the external file cannot be created (in the case of an attempt to write or append to an external file that does not exist). This value is also returned if the external file cannot be associated with the file object for any reason.

— A value of MODE_ERROR indicates that the external file cannot be opened with the requested Open_Kind.

The first form of FILE_OPEN causes an error to occur if the second form of FILE_OPEN, when called under identical conditions, would return a Status value other than OPEN_OK.

A call to FILE_OPEN of the first form is *successful* if and only if the call does not cause an error to occur. Similarly, a call to FILE_OPEN of the second form is successful if and only if it returns a Status value of OPEN_OK.

If a file object F is associated with an external file, procedure FILE_CLOSE terminates access to the external file associated with F and closes the external file. If F is not associated with an external file, then FILE_CLOSE has no effect. In either case, the file object is no longer open after a call to FILE_CLOSE that associates the file object with the formal parameter F.

An implicit call to FILE_CLOSE exists in a subprogram body for every file object declared in the corresponding subprogram declarative part. Each such call associates a unique file object with the formal parameter F and is called whenever the corresponding subprogram completes its execution.

Procedure READ retrieves the next value from a file; it is an error if the access mode of the file object is write-only or if the file object is not open. Procedure WRITE appends a value to a file; it is similarly an error if the access mode of the file object is read-only or if the file is not open. Function ENDFILE returns FALSE if a subsequent READ operation on an open file object whose access mode is read-only can retrieve another value from the file; otherwise, it returns TRUE. Function ENDFILE always returns TRUE for an open file object whose access mode is write-only. It is an error if ENDFILE is called on a file object that is not open.

For a file type declaration in which the type mark denotes an unconstrained array type, the same operations are implicitly declared, except that the READ operation is declared as follows:

procedure READ (**file** F: FT; VALUE: **out** TM; LENGTH: **out** Natural);

The READ operation for such a type performs the same function as the READ operation for other types, but in addition it returns a value in parameter LENGTH that specifies the actual length of the array value read by the operation. If the object associated with formal parameter VALUE is shorter than this length, then only that portion of the array value read by the operation that can be contained in the object is returned by the READ operation, and the rest of the value is lost. If the object associated with formal parameter VALUE is longer than this length, then the entire value is returned and remaining elements of the object are unaffected by the READ operation.

An error will occur when a READ operation is performed on file F if ENDFILE(F) would return TRUE at that point.

At the beginning of the execution of any file operation, the execution of the file operation *blocks* (see 12.5) until exclusive access to the file object denoted by the formal parameter F can be granted. Exclusive access to the given file object is then granted and the execution of the file operation proceeds. Once the file operation completes, exclusive access to the given file object is rescinded.[35]

35. Noted as part of the P1076a cleanup initiated by Peter Ashenden.

NOTE

—Predefined package TEXTIO is provided to support formatted human-readable I/O. It defines type TEXT (a file type representing files of variable-length text strings) and type LINE (an access type that designates such strings). READ and WRITE operations are provided in package TEXTIO that append or extract data from a single line. Additional operations are provided to read or write entire lines and to determine the status of the current line or of the file itself. Package TEXTIO is defined in ~~Section~~ Clause[36] 14.

3.5 Protected types

A protected type definition defines a protected type. A protected type implements instantiatiable regions of sequential statements, each of which are guaranteed exclusive access to shared data. Shared data is a set of variable objects that may be potentially accessed as a unit by multiple processes.

```
protected_type_definition ::=
      protected_type_declaration
    | protected_type_body
```

Each protected type declaration appearing immediately within a given declarative region (see 10.1) must have exactly one corresponding protected type body appearing immediately within the same declarative region and textually subsequent to the protected type declaration. Similarly, each protected type body appearing immediately within a given declarative region must have exactly one corresponding protected type declaration appearing immediately within the same declarative region and textually prior to the protected type body.

3.5.1 Protected type declarations

A protected type declaration declares the external interface to a protected type.

```
protected_type_declaration ::=
    protected
        protected_type_declarative_part
    end protected [ protected_type_simple_name ]

protected_type_declarative_part ::=
    { protected_type_declarative_item }

protected_type_declarative_item ::=
  subprogram_declaration
| attribute_specification
| use_clause
```

If a simple name appears at the end of a protected type declaration, it must repeat the identifier of the type declaration in which the protected type definition is included.

Each subprogram specified within a given protected type declaration defines an abstract operation, called a *method*, that operates atomically and exclusively on a single object of the protected type. In addition to the (implied) object of the protected type operated on by the subprogram, additional parameters may be explicitly specified in the formal parameter list of the subprogram declaration of the subprogram. Such formal parameters must not be of an access type or a file type; moreover, they must not have a subelement that is an access type or a file type. Additionally, in the case of a function subprogram, the return type of the function must not be of an access type or file type; moreover, it must not have a subelement that is an access type or a file type.

36. To conform to IEEE rules.

Examples:

```
type SharedCounter is protected
    procedure increment  (N: Integer := 1);
    procedure decrement (N: Integer := 1);
    impure function value return Integer;
end protected SharedCounter;

type ComplexNumber is protected
    procedure extract (variable r, i: out Real);
    procedure add (variable a, b: inout ComplexNumber);
end protected ComplexNumber;

type VariableSizedBitArray is protected
    procedure add_bit (index: Positive; value: Bit);
    impure function size return Natural;
end protected VariableSizedBitArray;
```

3.5.2 Protected type bodies

A protected type body provides the implementation for a protected type.

```
protected_type_body  ::=
    protected body
        protected_type_body_declarative_part
    end protected body [ protected_type_simple name ]

protected_type_body_declarative_part ::=
    { protected_type_body_declarative_item }

protected_type_body_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration
```

Each subprogram declaration appearing in a given protected type declaration shall have a corresponding subprogram body appearing in the corresponding protected type body.

NOTE

—Subprogram bodies appearing in a protected type body not conformant to any of the subprogram declarations in the corresponding protected type declaration are visible only within the protected type body. Such subprograms may have parameters and (in the case of functions) return types that are or contain access and file types.

Examples:

```
type SharedCounter is protected body

    variable counter: Integer := 0;

    procedure increment (N: Integer := 1) is
    begin
        counter := counter + N;
    end procedure increment;

    procedure decrement (N: Integer := 1) is
    begin
        counter := counter – N;
    end procedure decrement;

    impure function value return Integer is
    begin
        return counter;
    end function value;
end protected body SharedCounter;

type ComplexNumber is protected body

    variable re, im: Real;

    procedure extract (r, i: out Real) is
    begin
        r := re;
        i := im;
    end procedure extract;

    procedure add (variable a, b: inout ComplexNumber) is
        variable a_real, b_real: Real;
        variable a_imag, b_imag: Real;
    begin
        a.extract (a_real, a_imag);
        b.extract (b_real, b_imag);
        re  := a_real + b_real;
        im := a_imag + b_imag;
    end procedure add;
end protected body ComplexNumber;

type VariableSizeBitArray is protected body
    type bit_vector_access is access Bit_Vector;

    variable bit_array: bit_vector_access := null;
    variable bit_array_length: Natural := 0;
```

```
        procedure add_bit (index: Positive; value: Bit) is
            variable tmp: bit_vector_access;
        begin
            if index > bit_array_length then
                tmp := bit_array;
                bit_array := new bit_vector (1 to index);
                if tmp /= null then
                    bit_array (1 to bit_array_length) := tmp.all;
                    deallocate (tmp);
                end if;
                bit_array_length := index;
            end if;
            bit_array (index) := value;
        end procedure add_bit;

        impure function size return Natural is
        begin
            return bit_array_length;
        end function size;
    end protected body VariableSizeBitArray;
```