Clause 4

Declarations

The language defines several kinds of entities that are declared explicitly or implicitly by declarations.

```
declaration ::=
      type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | group_template_declaration
    | group_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration
    | primary_unit
    | architecture_body[1]
```

For each form of declaration, the language rules define a certain region of text called the *scope* of the declaration (see 10.2). Each form of declaration associates an identifier with a named entity. Only within its scope, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules (see 10.3). At such places the identifier is said to be a *name* of the entity; the name is said to *denote* the associated entity.

This ~~section~~ clause[2] describes type and subtype declarations, the various kinds of object declarations, alias declarations, attribute declarations, component declarations, and group and group template declarations. The other kinds of declarations are described in ~~Section~~ Clause[3] 1 and ~~Section~~ Clause[4] 2.

A declaration takes effect through the process of elaboration. Elaboration of declarations is discussed in ~~Section~~ Clause[5] 12.

4.1 Type declarations

A type declaration declares a type.

1. LCS 3.
2. To conform to IEEE rules.
3. To conform to IEEE rules.
4. To conform to IEEE rules.
5. To conform to IEEE rules.

```
type_declaration ::=
      full_type_declaration
    | incomplete_type_declaration

full_type_declaration ::=
    type identifier is type_definition ;

type_definition ::=
      scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition
    | protected_type_definition
```

The types created by the elaboration of distinct type definitions are distinct types. The elaboration of the type definition for a scalar type or a constrained array type creates both a base type and a subtype of the base type.

The simple name declared by a type declaration denotes the declared type, unless the type declaration declares both a base type and a subtype of the base type, in which case the simple name denotes the subtype and the base type is anonymous. A type is said to be *anonymous* if it has no simple name. For explanatory purposes, this standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier.

NOTES

1—Two type definitions always define two distinct types, even if they are lexically identical. Thus, the type definitions in the following two integer type declarations define distinct types:

type A **is range** 1 **to** 10;
type B **is range** 1 **to** 10;

This applies to type declarations for other classes of types as well.

2—The various forms of type definition are described in ~~Section~~ Clause[6] 3. Examples of type declarations are also given in that ~~section~~ clause[7].

4.2 Subtype declarations

A subtype declaration declares a subtype.

```
subtype_declaration ::=
    subtype identifier is subtype_indication ;

subtype_indication ::=
    [ resolution_function_name ] type_mark [ constraint ]

type_mark ::=
      type_name
    | subtype_name

constraint ::=
      range_constraint
    | index_constraint
```

6. To conform to IEEE rules.
7. To conform to IEEE rules.

A type mark denotes a type or a subtype. If a type mark is the name of a type, the type mark denotes this type and also the corresponding unconstrained subtype. The base type of a type mark is, by definition, the base type of the type or subtype denoted by the type mark.

A subtype indication defines a subtype of the base type of the type mark.

If a subtype indication includes a resolution function name, then any signal declared to be of that subtype will be resolved, if necessary, by the named function (see 2.4); for an overloaded function name, the meaning of the function name is determined by context (see 2.3 and 10.5). It is an error if the function does not meet the requirements of a resolution function (see 2.4). The presence of a resolution function name has no effect on the declarations of objects other than signals or on the declarations of files, aliases, attributes, or other subtypes.

If the subtype indication does not include a constraint, the subtype is the same as that denoted by the type mark. The condition imposed by a constraint is the condition obtained after evaluation of the expressions and ranges forming the constraint. The rules defining compatibility are given for each form of constraint in the appropriate ~~section~~ clause[8]. These rules are such that if a constraint is compatible with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. An error occurs if any check of compatibility fails.

The direction of a discrete subtype indication is the same as the direction of the range constraint that appears as the constraint of the subtype indication. If no constraint is present, and the type mark denotes a subtype, the direction of the subtype indication is the same as that of the denoted subtype. If no constraint is present, and the type mark denotes a type, the direction of the subtype indication is the same as that of the range used to define the denoted type. The direction of a discrete subtype is the same as the direction of its subtype indication.

A subtype indication denoting an access type, a file type, or a protected type ~~may~~ must[9] not contain a resolution function. Furthermore, the only allowable constraint on a subtype indication denoting an access type is an index constraint (and then only if the designated type is an array type).

A subtype indication denoting a subtype of a record type, a file type, or a protected type ~~may~~ must[10] not contain a constraint.

NOTE

—A subtype declaration does not define a new type.

4.3 Objects

An *object* is a named entity that contains (has) a value of a given type. An object is one of the following:

— An object declared by an object declaration (see 4.3.1)

— A loop or generate parameter (see 8.9 and 9.7)

— A formal parameter of a subprogram (see 2.1.1)

— A formal port (see 1.1.1.2 and 9.1)

— A formal generic (see 1.1.1.1 and 9.1)

— A local port (see 4.5)

— A local generic (see 4.5)

8. To conform to IEEE rules.
9. IR1000.4.7.
10. IR1000.4.7.

— An implicit signal GUARD defined by the guard expression of a block statement (see 9.1)In addition, the following are objects, but are not named entities:

— An implicit signal defined by any of the predefined attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION (see 14.1)

— An element or slice of another object (see 6.3, 6.4, and 6.5)

— An object designated by a value of an access type (see 3.3)

There are four classes of objects: constants, signals, variables, and files. The variable class of objects also has an additional subclass: shared variables. The class of an explicitly declared object is specified by the reserved word that must or may appear at the beginning of the declaration of that object. For a given object of a composite type, each subelement of that object is itself an object of the same class and subclass, if any, as the given object. The value of a composite object is the aggregation of the values of its subelements.

Objects declared by object declarations are available for use within blocks, processes, subprograms, or packages. Loop and generate parameters are implicitly declared by the corresponding statement and are available for use only within that statement. Other objects, declared by interface declarations, create channels for the communication of values between independent parts of a description.

4.3.1 Object declarations

An object declaration declares an object of a specified type. Such an object is called an *explicitly declared object*.

```
object_declaration ::=
      constant_declaration
    | signal_declaration
    | variable_declaration
    | file_declaration
```

An object declaration is called a *single-object declaration* if its identifier list has a single identifier; it is called a *multiple-object declaration* if the identifier list has two or more identifiers. A multiple-object declaration is equivalent to a sequence of the corresponding number of single-object declarations. For each identifier of the list, the equivalent sequence has a single-object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple-object declaration; the equivalent sequence is in the same order as the identifier list.

A similar equivalence applies also for interface object declarations (see 4.3.2).

NOTE

—The subelements of a composite, declared object are not declared objects.

4.3.1.1 Constant declarations

A constant declaration declares a *constant* of the specified type. Such a constant is an *explicitly declared constant*.

```
constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;
```

If the assignment symbol ":=" followed by an expression is present in a constant declaration, the expression specifies the value of the constant; the type of the expression must be that of the constant. The value of a constant cannot be modified after the declaration is elaborated.

If the assignment symbol ":=" followed by an expression is not present in a constant declaration, then the declaration declares a *deferred constant*. Such a constant declaration ~~may only~~ must[11] appear in a package declaration. The corresponding full constant declaration, which defines the value of the constant, must appear in the body of the package (see 2.6).

Formal parameters of subprograms that are of mode **in** may be constants, and local and formal generics are always constants; the declarations of such objects are discussed in 4.3.2. A loop parameter is a constant within the corresponding loop (see 8.9); similarly, a generate parameter is a constant within the corresponding generate statement (see 9.7). A subelement or slice of a constant is a constant.

It is an error if a constant declaration declares a constant that is of a file type, an access type, a protected type, or a composite type that has a subelement that is a file type, an access type, or a protected type.

NOTE

—The subelements of a composite, declared constant are not declared constants.

Examples:

```
constant TOLERANCE : DISTANCE := 1.5 nm;
constant PI : REAL := 3.141592 ;
constant CYCLE_TIME : TIME := 100 ns;
constant Propagation_Delay : DELAY_LENGTH;       -- a deferred constant
```

4.3.1.2 Signal declarations

A signal declaration declares a *signal* of the specified type. Such a signal is an *explicitly declared signal*.

```
signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ] [ := expression ] ;

signal_kind ::=  register  |  bus
```

If the name of a resolution function appears in the declaration of a signal or in the declaration of the subtype used to declare the signal, then that resolution function is associated with the declared signal. Such a signal is called a *resolved signal*.

If a signal kind appears in a signal declaration, then the signals so declared are *guarded* signals of the kind indicated. For a guarded signal that is of a composite type, each subelement is likewise a guarded signal. For a guarded signal that is of an array type, each slice (see 6.5) is likewise a guarded signal. A guarded signal may be assigned values under the control of Boolean-valued *guard expressions* (or *guards*).When a given guard becomes False, the drivers of the corresponding guarded signals are implicitly assigned a null transaction (see 8.4.1) to cause those drivers to turn off. A disconnection specification (see 5.3) is used to specify the time required for those drivers to turn off.

If the signal declaration includes the assignment symbol followed by an expression, it must be of the same type as the signal. Such an expression is said to be a *default expression*. The default expression defines a *default value* associated with the signal or, for a composite signal, with each scalar subelement thereof. For a signal declared to be of a scalar subtype, the value of the default expression is the default value of the signal. For a signal declared to be of a composite subtype, each scalar subelement of the value of the default expression is the default value of the corresponding subelement of the signal.

In the absence of an explicit default expression, an implicit default value is assumed for a signal of a scalar subtype or for each scalar subelement of a composite signal, each of which is itself a signal of a scalar subtype. The implicit default value for a signal of a scalar subtype T is defined to be that given by T'LEFT.

11. IR1000.4.7.

It is an error if a signal declaration declares a signal that is of a file type, an access type, a protected type, or a composite type having a subelement that is a file type, an access type, or a protected type. It is also an error if a guarded signal of a scalar type is neither a resolved signal nor a subelement of a resolved signal.

A signal may have one or more *sources*. For a signal of a scalar type, each source is either a driver (see 12.6.1) or an **out**, **inout**, **buffer**, or **linkage** port of a component instance or of a block statement with which the signal is associated. For a signal of a composite type, each composite source is a collection of scalar sources, one for each scalar subelement of the signal. It is an error if, after the elaboration of a description, a signal has multiple sources and it is not a resolved signal. It is also an error if, after the elaboration of a description, a resolved signal has more sources than the number of elements in the index range of the type of the formal parameter of the resolution function associated with the resolved signal.

If a subelement or slice of a resolved signal of composite type is associated as an actual in a port map aspect (either in a component instantiation statement, a block statement,[12] or in a binding indication), and if the corresponding formal is of mode **out**, **inout**, **buffer**, or **linkage**, then every scalar subelement of that signal must be associated exactly once with such a formal in the same port map aspect, and the collection of the corresponding formal parts taken together constitute one source of the signal. If a resolved signal of composite type is associated as an actual in a port map aspect, that is equivalent to each of its subelements being associated in the same port map aspect.

If a subelement of a resolved signal of composite type has a driver in a given process, then every scalar subelement of that signal must have a driver in the same process, and the collection of all of those drivers taken together constitute one source of the signal.

The default value associated with a scalar signal defines the value component of a transaction that is the initial contents of each driver (if any) of that signal. The time component of the transaction is not defined, but the transaction is understood to have already occurred by the start of simulation.

Examples:

signal S : STANDARD.BIT_VECTOR (1 **to** 10) ;

signal CLK1, CLK2 : TIME ;

signal OUTPUT : WIRED_OR MULTI_VALUED_LOGIC;

NOTES

1—Ports of any mode are also signals. The term *signal* is used in this standard to refer to objects declared either by signal declarations or by port declarations (or to subelements, slices, or aliases of such objects). It also refers to the implicit signal GUARD (see 9.1) and to implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, and 'TRANSACTION. The term *port* is used to refer to objects declared by port declarations only.

2—Signals are given initial values by initializing their drivers. The initial values of drivers are then propagated through the corresponding net to determine the initial values of the signals that make up the net (see 12.6.3).

3—The value of a signal ~~may be~~ is[13] indirectly modified by a signal assignment statement (see 8.4); such assignments affect the future values of the signal.

4—The subelements of a composite, declared signal are not declared signals.

Cross-References: Disconnection specifications, 5.3; Disconnection statements, 9.5; Guarded assignment, 9.5; Guarded blocks, 9.1; Guarded targets, 9.5; Signal guard, 9.1.

12. IR1000.3.2.
13. IR1000.4.7.

4.3.1.3 Variable declarations

A variable declaration declares a *variable* of the specified type. Such a variable is an *explicitly declared variable*.

variable_declaration ::=
[**shared**] **variable** identifier_list : subtype_indication [:= expression] ;

A variable declaration that includes the reserved word **shared** is a *shared variable declaration*. A shared variable declaration declares a *shared variable*. Shared variables are a subclass of the variable class of objects. The base type of the subtype indication of a shared variable declaration must be a protected type. Variables declared immediately within entity declarations, architecture bodies, packages, package bodies, and blocks must be shared variables. Variables declared immediately within subprograms and processes must not be shared variables. Variables may appear in protected type bodies; such variables, which must not be shared variables, represent shared data.

If a given variable declaration appears (directly or indirectly) within a protected type body, then the base type denoted by the subtype indication of the variable declaration ~~may~~ must[14] not be the protected type defined by the protected type body.

If the variable declaration includes the assignment symbol followed by an expression, the expression specifies an initial value for the declared variable; the type of the expression must be that of the variable. Such an expression is said to be an *initial value expression*. A variable declaration, whether it is a shared variable declaration or not, whose subtype indication denotes a protected type ~~may~~ must[15] not have an initial value expression (~~nor may it~~ moreover, it must not[16] include the immediately preceding assignment symbol).

If an initial value expression appears in the declaration of a variable, then the initial value of the variable is determined by that expression each time the variable declaration is elaborated. In the absence of an initial value expression, a default initial value applies. The default initial value for a variable of a scalar subtype T is defined to be the value given by T'LEFT. The default initial value of a variable of a composite type is defined to be the aggregate of the default initial values of all of its scalar subelements, each of which is itself a variable of a scalar subtype. The default initial value of a variable of an access type is defined to be the value **null** for that type.

NOTES

1—The value of a variable that is not a shared variable ~~may be~~ is[17] modified by a variable assignment statement (see 8.5); such assignments take effect immediately.

2—The variables declared within a given procedure persist until that procedure completes and returns to the caller. For procedures that contain wait statements, a variable ~~may therefore persist~~ therefore persists[18] from one point in simulation time to another, and the value in the variable is thus maintained over time. For processes, which never complete, all variables persist from the beginning of simulation until the end of simulation.

3—The subelements of a composite, declared variable are not declared variables.~~4~~— Since the language guarantees mutual exclusion of accesses to shared data, but not the order of access to such data by multiple processes in the same simulation cycle, the use of shared varaibles can be both non-portable and non-deterministic. For example, consider the following architecture:

14. IR1000.4.7.
15. IR1000.4.7.
16. IR1000.4.7.
17. IR1000.4.7.
18. IR1000.4.7.

```
architecture UseSharedVariables of SomeEntity is
    subtype ShortRange is INTEGER range -1 to 1;
    type ShortRangeProtected is protected
        procedure Set(V: ShortRange);
        procedure Get(V: out ShortRange);
    end protected;

    type ShortRangeProtectedis protected body
        variable Local: ShortRange := 0;
    begin
        procedure Set(V: ShortRange) is
        begin
            Local := V;
        end procedure Set;

        procedure Get(V: out ShortRange) is
        begin
            V := Local;
        end procedure Get;
    end protected body;

    shared variable Counter: ShortRangeProtected;

begin
PROC1: process
        variable V: ShortRange;
    begin
        Counter,Get( V );
        Counter.Set( V+1 );
        wait;
    end process PROC1;

PROC2: process
    variable V: ShortRange;
    begin
        Counter,Get( V );
        Counter.Set( V–1 );
        wait;
    end process PROC2;
end architecture UseSharedVariables;
```

In particular, the value of Counter after the execution of both processes is not guaranteed to be 0. The possible values of Counter could be –1, 0, or 1.

5—Variables that are not shared variables may have a subtype indication denoting a protected type.

Examples:

```
variable INDEX : INTEGER range 0 to 99 := 0 ;
    -- Initial value is determined by the initial value expression

variable COUNT : POSITIVE ;
    -- Initial value is POSITIVE'LEFT; that is,1.

variable MEMORY : BIT_MATRIX (0 to 7, 0 to 1023) ;
    -- Initial value is the aggregate of the initial values of each element
```

shared variable Counter: SharedCounter;
-- See 3.5.1 and 3.5.2 for the definitions of SharedCounter

shared variable addend, augend, result: ComplexNumber;
-- See 3.5.1 and 3.5.2 for the definition of ComplexNumber

variable bit_stack: VariableSizeBitArray;
-- See 3.5.1 and 3.5.2 for the definition of VariableSizeBitArray;

4.3.1.4 File declarations

A file declaration declares a *file* of the specified type. Such a file is an *explicitly declared file*.

file_declaration ::=
file identifier_list : subtype_indication [file_open_information] ;

file_open_information ::= [**open** *file_open_kind*_expression] **is** file_logical_name

file_logical_name ::= *string*_expression

The subtype indication of a file declaration must define a file subtype.

If file open information is included in a given file declaration, then the file declared by the declaration is opened (see 3.4.1) with an implicit call to FILE_OPEN when the file declaration is elaborated (see 12.3.1.4). This implicit call is to the FILE_OPEN procedure of the first form, and it associates the identifier with the file parameter F, the file logical name with the External_Name parameter, and the file open kind expression with the Open_Kind parameter. If a file open kind expression is not included in the file open information of a given file declaration, then the default value of READ_MODE is used during elaboration of the file declaration.

If file open information is not included in a given file declaration, then the file declared by the declaration is not opened when the file declaration is elaborated.

The file logical name must be an expression of predefined type STRING. The value of this expression is interpreted as a logical name for a file in the host system environment. An implementation must provide some mechanism to associate a file logical name with a host-dependent file. Such a mechanism is not defined by the language.

The file logical name identifies an external file in the host file system that is associated with the file object. This association provides a mechanism for either importing data contained in an external file into the design during simulation or exporting data generated during simulation to an external file.

If multiple file objects are associated with the same external file, and each file object has an access mode that is read-only (see 3.4.1), then values read from each file object are read from the external file associated with the file object. The language does not define the order in which such values are read from the external file, nor does it define whether each value is read once or multiple times (once per file object).

The language does not define the order of and the relationship, if any, between values read from and written to multiple file objects that are associated with the same external file. An implementation may restrict the number of file objects that ~~may be~~ are[19] associated at one time with a given external file.

If a formal subprogram parameter is of the class **file**, it must be associated with an actual that is a file object.

Examples:

type IntegerFile **is file of** INTEGER;

19. IR1000.4.7.

```
file F1: IntegerFile;                              -- No implicit FILE_OPEN is performed
                                                   -- during elaboration.

file F2: IntegerFile is "test.dat";                -- At elaboration, an implicit call is performed:
                                                   -- FILE_OPEN (F2, "test.dat");
                                                   -- The OPEN_KIND parameter defaults to
                                                   -- READ_MODE.

file F3: IntegerFile open WRITE_MODE is "test.dat";
                                                   -- At elaboration, an implicit call is performed:
                                                   -- FILE_OPEN (F3, "test.dat", WRITE_MODE);
```

NOTE

—All file objects associated with the same external file should be of the same base type.

4.3.2 Interface declarations

An interface declaration declares an *interface object* of a specified type. Interface objects include *interface constants* that appear as generics of a design entity, a component, or a block, or as constant parameters of subprograms; *interface signals* that appear as ports of a design entity, component, or block, or as signal parameters of subprograms; *interface variables* that appear as variable parameters of subprograms; and *interface files* that appear as file parameters of subprograms.

interface_declaration ::=
    interface_constant_declaration
  | interface_signal_declaration
  | interface_variable_declaration
  | interface_file_declaration

interface_constant_declaration ::=
  [**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

interface_signal_declaration ::=
  [**signal**] identifier_list : [mode] subtype_indication [**bus**] [:= *static*_expression]

interface_variable_declaration ::=
  [**variable**] identifier_list : [mode] subtype_indication [:= *static*_expression]

interface_file_declaration ::=
  file identifier_list :[20] subtype_indication

mode ::= **in** | **out** | **inout** | **buffer** | **linkage**

If no mode is explicitly given in an interface declaration other than an interface file declaration, mode **in** is assumed.

For an interface constant declaration or an interface signal declaration, the subtype indication must define a subtype that is neither a file type, an access type, nor a protected type. ~~Morover~~ Moreover[21], the subtype indication ~~may~~ must[22] not denote a composite type with a subelement that is a file type, an access type, or a protected type.

For an interface file declaration, it is an error if the subtype indication does not denote a subtype of a file type.

20. IR1000.1.12.
21. Typo (noted by Ashenden).
22. IR1000.4.7.

If an interface signal declaration includes the reserved word **bus**, then the signal declared by that interface declaration is a guarded signal of signal kind **bus**.

If an interface declaration contains a ":=" symbol followed by an expression, the expression is said to be the *default expression* of the interface object. The type of a default expression must be that of the corresponding interface object. It is an error if a default expression appears in an interface declaration and any of the following conditions hold:

— The mode is **linkage**

— The interface object is a formal signal parameter

— The interface object is a formal variable parameter of mode other than **in**

— The subtype indication of the interface declaration denotes a protected type

In an interface signal declaration appearing in a port list, the default expression defines the default value(s) associated with the interface signal or its subelements. In the absence of a default expression, an implicit default value is assumed for the signal or for each scalar subelement, as defined for signal declarations (see 4.3.1.2). The value, whether implicitly or explicitly provided, is used to determine the initial contents of drivers, if any, of the interface signal as specified for signal declarations.

An interface object provides a channel of communication between the environment and a particular portion of a description. The value of an interface object may be determined by the value of an associated object or expression in the environment; similarly, the value of an object in the environment may be determined by the value of an associated interface object. The manner in which such associations are made is described in 4.3.2.2.

The value of an object is said to be *read* when one of the following conditions is satisfied:

— When the object is evaluated, and also (indirectly) when the object is associated with an interface object of the modes **in**, **inout**, or **linkage**.

— When the object is a signal and a name denoting the object appears in a sensitivity list in a wait statement or a process statement.

— When the object is a signal and the value of any of its predefined attributes 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, or 'LAST_VALUE is read.

— When one of its subelements is read.

— When the object is a file and a READ operation is performed on the file.

— When the object is a file of type STD.TEXTIO.TEXT and the procedure STD.TEXTIO.READLINE is called with the given object associated with the formal parameter F of the given procedure.[23]

The value of an object is said to be *updated* when one of the following conditions is satisfied:

— When it is the target of an assignment, and also (indirectly) when the object is associated with an interface object of the modes **out**, **buffer**, **inout**, or **linkage**.

— When one of its subelements is updated.

— When the object is a file and a WRITE operation is performed on the file.

23. Inspired by Steve Bailey.

— When the object is a file of type STD.TEXTIO.TEXT and the procedure STD.TEXTIO.WRITELINE is called with the given object associated with the formal parameter F of the given procedure.[24]

~~Only~~ It is an error if an object other than a[25] signal, variable, or file ~~objects may be~~ object is[26] updated.

An interface object has one of the following modes:

in. The value of the interface object ~~may only be read. In addition, any attributes~~ is allowed to be read, but it must not be updated. Reading an attribute[27] of the interface object ~~may be read, except that attributes~~ is allowed, unless the interface object is a subprogram signal parameter and the attribute is one of 'STABLE, 'QUIET, 'DELAYED, ~~and~~ 'TRANSACTION, 'DRIVING, or 'DRIVING_VALUE[28] ~~of a subprogram signal parameter may not be read within the corresponding subprogram~~[29]. ~~For a file object, operation ENDFILE is allowed.~~

out. The value of the interface object ~~may be updated~~ is allowed to be updated, but it must not be read[30]. Reading the attributes of the interface element, other than the predefined attributes 'STABLE, 'QUIET, 'DELAYED, 'TRANSACTION, 'EVENT, 'ACTIVE, 'LAST_EVENT, 'LAST_ACTIVE, and 'LAST_VALUE, is allowed. No other reading is allowed.

inout. ~~The~~ Reading and updating the[31] value of the interface object ~~may be both read and updated~~ is allowed[32]. Reading the attributes of the interface object, other than the attributes 'STABLE, 'QUIET, 'DELAYED, and 'TRANSACTION of a signal parameter, is also permitted. ~~For a file object, all file operations (see 3.4.1) are allowed.~~

buffer. ~~The~~ Reading and updating the[33] value of the interface object ~~may be both read and updated~~ is allowed[34]. Reading the attributes of the interface object is also permitted.

linkage. ~~The~~ Reading and updating the[35] value of the interface object ~~may be both read and updated~~ is allowed[36], but only by appearing as an actual corresponding to an interface object of mode **linkage**. No other reading or updating is permitted.

NOTES

~~1—Although signals of modes **inout** and **buffer** have the same characteristics with respect to whether they may be read or updated, a signal of mode **inout** may be updated by zero or more sources, whereas a signal of mode **buffer** must be updated by at most one source (see 1.1.1.2).~~

~~2—~~[37]A subprogram parameter that is of a file type must be declared as a file parameter.~~3~~ 2[38]—Since shared variables are a subclass of variables, a shared variable may be associated as an actual with a formal of class variable.

3—Ports of mode **linkage** may be removed from a future version of the language. See Annex F.[39]

24. Noted by Steve Bailey.
25. IR1000.4.7.
26. IR1000.4.7.
27. IR1000.4.7.
28. Missed in 1076-1993; noted by Bert Molenkamp.
29. IR1000.4.7.
30. IR1000.4.7.
31. IR1000.4.7.
32. IR1000.4.7.
33. IR1000.4.7.
34. IR1000.4.7.
35. IR1000.4.7.
36. IR1000.4.7.
37. LCS 24.
38. LCS 24.
39. LCS 25.

~~4~~—Interface file objects are modeless.[40]

4.3.2.1 Interface lists

An interface list contains the declarations of the interface objects required by a subprogram, a component, a design entity, or a block statement.

```
interface_list ::=
    interface_element { ; interface_element }

interface_element ::=  interface_declaration
```

A *generic* interface list consists entirely of interface constant declarations. A *port* interface list consists entirely of interface signal declarations. A *parameter* interface list may contain interface constant declarations, interface signal declarations, interface variable declarations, interface file declarations, or any combination thereof.

A name that denotes an interface object ~~may~~ must[41] not appear in any interface declaration within the interface list containing the denoted interface object except to declare this object.

NOTE

—The above restriction makes the following three interface lists illegal:

```
entity E is
    generic  (G1:INTEGER; G2:INTEGER := G1);                    -- illegal
    port     (P1:STRING;    P2:STRING(P1'RANGE));               -- illegal

    procedure X  (Y1, Y2: INTEGER;Y3: INTEGER range Y1 to Y2);  -- illegal
end E;
```

However, the following interface lists are legal:

```
entity E is
    generic(G1, G2, G3, G4:INTEGER);
    port(P1, P2:STRING (G1 to G2));

    procedure X(Y3:INTEGER range G3 to G4);
end E;
```

4.3.2.2 Association lists

An association list establishes correspondences between formal or local generic, port, or parameter names on the one hand and local or actual names or expressions on the other.

```
association_list ::=
    association_element { , association_element }

association_element ::=
    [ formal_part => ] actual_part

formal_part ::=
      formal_designator
    | function_name ( formal_designator )
    | type_mark ( formal_designator )
```

40. Suggested by Steve Bailey.
41. IR1000.4.7.

formal_designator ::=
 *generic*_name
 | *port*_name
 | *parameter*_name

actual_part ::=
 actual_designator
 | *function*_name (actual_designator)
 | type_mark (actual_designator)

actual_designator ::=
 expression
 | *signal*_name
 | *variable*_name
 | *file*_name
 | **open**

Each association element in an association list associates one actual designator with the corresponding interface element in the interface list of a subprogram declaration, component declaration, entity declaration, or block statement. The corresponding interface element is determined either by position or by name.

An association element is said to be *named* if the formal designator appears explicitly; otherwise, it is said to be *positional*. For a positional association, an actual designator at a given position in an association list corresponds to the interface element at the same position in the interface list.

Named associations can be given in any order, but if both positional and named associations appear in the same association list, then all positional associations must occur first at their normal position. Hence once a named association is used, the rest of the association list must use only named associations.

In the following, the term *actual* refers to an actual designator, and the term *formal* refers to a formal designator.

The formal part of a named ~~element association~~ association element[42] may be in the form of a function call, where the single argument of the function is the formal designator itself, if and only if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**. In this case, the function name must denote a function whose single parameter is of the type of the formal and whose result is the type of the corresponding actual. Such a *conversion function* provides for type conversion in the event that data flows from the formal to the actual.

Alternatively, the formal part of a named ~~element association~~ association element[43] may be in the form of a type conversion, where the expression to be converted is the formal designator itself, if and only if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**. In this case, the base type denoted by the type mark must be the same as the base type of the corresponding actual. Such a type conversion provides for type conversion in the event that data flows from the formal to the actual. It is an error if the type of the formal is not closely related to the type of the actual. (See 7.3.5.)

Similarly, the actual part of a (named or positional) ~~element association~~ association element[44] may be in the form of a function call, where the single argument of the function is the actual designator itself, if and only if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. In this case, the function name must denote a function whose single parameter is of the type of the actual, and whose result is the type of the corresponding formal. In addition, the formal must not be of class **constant** for this interpretation to hold (the actual is interpreted as an expression that is a function call if the class of the formal is **constant**). Such a conversion function provides for type conversion in the event that data flows from the actual to the formal.

42. IR1000.2.1.
43. IR1000.2.1.
44. IR1000.2.1.

Alternatively, the actual part of a (named or positional) ~~element association~~ association element[45] may be in the form of a type conversion, where the expression to be type converted is the actual designator itself, if and only if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. In this case, the base type denoted by the type mark must be the same as the base type of the corresponding formal. Such a type conversion provides for type conversion in the event that data flows from the actual to the formal. It is an error if the type of the actual is not closely related to the type of the formal.

The type of the actual (after applying the conversion function or type conversion, if present in the actual part) must be the same as the type of the corresponding formal, if the mode of the formal is **in**, **inout**, or **linkage**, and if the actual is not **open**. Similarly, if the mode of the formal is **out**, **inout**, **buffer**, or **linkage**, and if the actual is not **open**, then the type of the formal (after applying the conversion function or type conversion, if present in the formal part) must be the same as the corresponding actual.

For the association of signals with corresponding formal ports, association of a formal of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal with the matching subelement of the actual, provided that no conversion function or type conversion is present in either the actual part or the formal part of the association element. If a conversion function or type conversion is present, then the entire formal is considered to be associated with the entire actual.

Similarly, for the association of actuals with corresponding formal subprogram parameters, association of a formal parameter of a given composite type with an actual of the same type is equivalent to the association of each scalar subelement of the formal parameter with the matching subelement of the actual. Different parameter passing mechanisms may be required in each case, but in both cases the associations will have an equivalent effect. This equivalence applies provided that no actual is accessible by more than one path (see 2.1.1.1).

A formal ~~may~~ must[46] be either an explicitly declared interface object or member (see ~~Section~~ Clause[47] 3) of such an interface object. In the former case, such a formal is said to be *associated in whole*. In the latter cases, named association must be used to associate the formal and actual; the subelements of such a formal are said to be *associated individually*. Furthermore, every scalar subelement of the explicitly declared interface object must be associated exactly once with an actual (or subelement thereof) in the same association list, and all such associations must appear in a contiguous sequence within that association list. Each association element that associates a slice or subelement (or slice thereof) of an interface object must identify the formal with a locally static name.

If an interface element in an interface list includes a default expression for a formal generic, for a formal port of any mode other than **linkage**, or for a formal variable or constant parameter of mode **in**, then any corresponding association list need not include an association element for that interface element. If the association element is not included in the association list, or if the actual is **open**, then the value of the default expression is used as the actual expression or signal value in an implicit association element for that interface element.

It is an error if an actual of **open** is associated with a formal that is associated individually. An actual of **open** counts as the single association allowed for the corresponding formal but does not supply a constant, signal, or variable (as is appropriate to the object class of the formal) to the formal.

NOTES

1—It is a consequence of these rules that, if an association element is omitted from an association list in order to make use of the default expression on the corresponding interface element, all subsequent association elements in that association list must be named associations.

2—Although a default expression can appear in an interface element that declares a (local or formal) port, such a default expression is not interpreted as the value of an implicit association element for that port. Instead, the value of the expression is used to determine the effective value of that port during simulation if the port is left unconnected (see 12.6.2).

45. IR1000.2.1.
46. IR1000.4.7.
47. To conform to IEEE rules.

3—Named association may not be used when invoking implicitly defined ~~operations~~ operators[48], since the formal parameters of these operators are not named (see 7.2).

4—Since information flows only from the actual to the formal when the mode of the formal is **in**, and since a function call is itself an expression, the actual associated with a formal of object class **constant** is never interpreted as a conversion function or a type conversion converting an actual designator that is an expression. Thus, the following association element is legal:

Param => F (**open**)

under the conditions that Param is a constant formal and F is a function returning the same base type as that of Param and having one or more parameters, all of which may be defaulted.5~~—No~~ It is an error if a[49] conversion function or type conversion ~~may appear~~ appears[50] in the actual part when the actual designator is **open**.

4.3.3 Alias declarations

An alias declaration declares an alternate name for an existing named entity.

alias_declaration ::=
    alias alias_designator [: subtype_indication] **is** name [signature] ;

alias_designator ::= identifier | character_literal | operator_symbol

An *object alias* is an alias whose alias designator denotes an object (that is, a constant, a variable, a signal, or a file). A *nonobject alias* is an alias whose alias designator denotes some named entity other than an object. An alias can be declared for all named entities except for labels, loop parameters, and generate parameters.

The alias designator in an alias declaration denotes the named entity specified by the name and, if present, the signature in the alias declaration. An alias of a signal denotes a signal; an alias of a variable denotes a variable; an alias of a constant denotes a constant; and an alias of a file denotes a file. Similarly, an alias of a subprogram (including an operator) denotes a subprogram, an alias of an enumeration literal denotes an enumeration literal, and so forth.

If the alias designator is a character literal, the name must denote an enumeration literal. If the alias designator is an operator symbol, the name must denote a function, and that function then overloads the operator symbol. In this latter case, the operator symbol and the function both must meet the requirements of 2.3.1.[51]

NOTES

1—Since, for example, the alias of a variable is a variable, every reference within this document to a designator (a name, character literal, or operator symbol) that requires the designator to denote a named entity with certain characteristics (for example, to be a variable) allows the designator to denote an alias, so long as the aliased name denotes a named entity having the required characteristics. This situation holds except where aliases are specifically prohibited.

2—The alias of an overloadable ~~object~~ named entity[52] is itself overloadable.

4.3.3.1 Object aliases

The following rules apply to object aliases:

a) A signature ~~may~~ must[53] not appear in a declaration of an object alias.

48. IR1000.2.4.
49. IR1000.4.7.
50. IR1000.4.7.
51. LCS 7.
52. IR1000.2.2.
53. IR1000.4.7.

b) The name must be a static name (see 6.1) that denotes an object. The base type of the name specified in an alias declaration must be the same as the base type of the type mark in the subtype indication (if the subtype indication is present); this type must not be a multi-dimensional array type. When the object denoted by the name is referenced via the alias defined by the alias declaration, the following rules apply:

— If the subtype indication is absent or if it is present and denotes an unconstrained array type:

— If the alias designator denotes a slice of an object, then the subtype of the object is viewed as if it were of the subtype specified by the slice

— Otherwise, the object is viewed as if it were of the subtype specified in the declaration of the object denoted by the name

— If the subtype indication is present and denotes a constrained array subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, the subtype denoted by the subtype indication must include a matching element (see 7.2.2) for each element of the object denoted by the name;

— If the subtype indication denotes a scalar subtype, then the object is viewed as if it were of the subtype specified by the subtype indication; moreover, it is an error if this subtype does not have the same bounds and direction as the subtype denoted by the object name.

c) The same applies to attribute references where the prefix of the attribute name denotes the alias.

d) A reference to an element of an object alias is implicitly a reference to the matching element of the object denoted by the alias. A reference to a slice of an object alias consisting of the elements e_1, e_2, ..., e_n is implicitly a reference to a slice of the object denoted by the alias consisting of the matching elements corresponding to each of e_1 through e_n.

4.3.3.2 Nonobject aliases

The following rules apply to nonobject aliases:

a) A subtype indication ~~may~~ must[54] not appear in a nonobject alias.

b) A signature is required if the name denotes a subprogram (including an operator) or enumeration literal. In this case, the signature is required to match (see 2.3) the parameter and result type profile of exactly one of the subprograms or enumeration literals denoted by the name.

c) If the name denotes an enumeration type, then one implicit alias declaration for each of the literals of the type immediately follows the alias declaration for the enumeration type; each such implicit declaration has, as its alias designator, the simple name or character literal of the literal and has, as its name, a name constructed by taking the name of the alias for the enumeration type and substituting the simple name or character literal being aliased for the simple name of the type. Each implicit alias has a signature that matches the parameter and result type profile of the literal being aliased.

d) Alternatively, if the name denotes a physical type, then one implicit alias declaration for each of the units of the type immediately follows the alias declaration for the physical type; each such implicit declaration has, as its alias designator, the simple name of the unit and has, as its[55] name, a name constructed by taking the name of the alias for the physical type and substituting the simple name of the unit being aliased for the simple name of the type.

54. IR1000.4.7.
55. Omission noted by Boyer.

e) Finally, if the name denotes a type, then implicit alias declarations for each predefined operator for the type immediately follow the explicit alias declaration for the type and, if present, any implicit alias declarations for literals or units of the type. Each implicit alias has a signature that matches the parameter and result type profile of the implicit operator being aliased.

Examples:

```
variable REAL_NUMBER : BIT_VECTOR (0 to 31);

alias SIGN : BIT is REAL_NUMBER (0);
      -- SIGN is now a scalar (BIT) value

alias MANTISSA : BIT_VECTOR (23 downto 0) is REAL_NUMBER (8 to 31);
      -- MANTISSA is a 24b value whose range is 23 downto 0.
      -- Note that the ranges of MANTISSA and REAL_NUMBER (8 to 31)
      -- have opposite directions.  A reference to MANTISSA (23 downto 18)
      -- is equivalent to a reference to REAL_NUMBER (8 to 13).

alias EXPONENT : BIT_VECTOR (1 to 7) is REAL_NUMBER (1 to 7);
      -- EXPONENT is a 7-bit value whose range is 1 to 7.

-- explicit alias:
alias STD_BIT is STD.STANDARD.BIT;

--  implicit aliases…
--  alias '0'      is STD.STANDARD.'0'      [return STD.STANDARD.BIT];

-- alias '1'       is STD.STANDARD.'1'      [return STD.STANDARD.BIT];

-- alias "and"     is STD.STANDARD."and"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "or"      is STD.STANDARD."or"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "nand"    is STD.STANDARD."nand"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "nor"     is STD.STANDARD."nor"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "xor"     is STD.STANDARD."xor"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "xnor"    is STD.STANDARD."xnor"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BIT];

-- alias "not"is STD.STANDARD."not"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT
--                 return STD.STANDARD.BIT];

-- alias "="       is STD.STANDARD."="
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];

-- alias "/="      is STD.STANDARD."/="
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];
```

```
-- alias "<"     is STD.STANDARD."<"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];

-- alias "<="    is STD.STANDARD."<="
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];

-- alias ">"     is STD.STANDARD.">"
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];

-- alias ">="    is STD.STANDARD.">="
--         [STD.STANDARD.BIT, STD.STANDARD.BIT return STD.STANDARD.BOOLEAN];
```

NOTE

—An alias of an explicitly declared object is not an explicitly declared object, nor is the alias of a subelement or slice of an explicitly declared object an explicitly declared object.

4.4 Attribute declarations

An attribute is a value, function, type, range, signal, or constant that may be associated with one or more named entities in a description. There are two categories of attributes: predefined attributes and user-defined attributes. Predefined attributes provide information about named entities in a description. ~~Section~~ Clause[56] 14 contains the definition of all predefined attributes. Predefined attributes that are signals ~~may~~ must[57] not be updated.

User-defined attributes are constants of arbitrary type. Such attributes are defined by an attribute declaration.

```
attribute_declaration ::=
    attribute identifier: type_mark ;
```

The identifier is said to be the *designator* of the attribute. An attribute may be associated with an entity declaration, an architecture, a configuration, a procedure, a function, a package, a type, a subtype, a constant, a signal, a variable, a component, a label, a literal, a unit, a group, or a file.

It is an error if the type mark denotes an access type, a file type, a protected type, or a composite type with a subelement that is an access type, a file type, or a protected type. The denoted type or subtype need not be constrained.

Examples:

```
type COORDINATE is record X,Y: INTEGER; end record;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
attribute LOCATION: COORDINATE;
attribute PIN_NO: POSITIVE;
```

NOTES

1—A given named entity E will be decorated with the user-defined attribute A if and only if an attribute specification for the value of attribute A exists in the same declarative part as the declaration of E. In the absence of such a specification, an attribute name of the form E'A is illegal.

2—A user-defined attribute is associated with the named entity denoted by the name specified in a declaration, not with the name itself. Hence, an attribute of an object can be referenced by using an alias for that object rather than the declared name of the object as the prefix of the attribute name, and the attribute referenced in such a way is the same attribute (and therefore has the same value) as the attribute referenced by using the declared name of the object as the prefix.

56. To conform to IEEE rules.
57. IR1000.4.7.

3—A user-defined attribute of a port, signal, variable, or constant of some composite type is an attribute of the entire port, signal, variable, or constant, not of its elements. If it is necessary to associate an attribute with each element of some composite object, then the attribute itself can be declared to be of a composite type such that for each element of the object, there is a corresponding element of the attribute.

4.5 Component declarations

A component declaration declares a ~~virtual design entity interface~~ an interface to a virtual design entity[58] that may be used in a component instantiation statement. A component configuration or a configuration specification can be used to associate a component instance with a design entity that resides in a library.

component_declaration ::=
  component identifier [**is**]
    [*local*_generic_clause]
    [*local*_port_clause]
  end component [*component*_simple_name] ;

Each interface object in the local generic clause declares a local generic. Each interface object in the local port clause declares a local port.

If a simple name appears at the end of a component declaration, it must repeat the identifier of the component declaration.

4.6 Group template declarations

A group template declaration declares a *group template*, which defines the allowable classes of named entities that can appear in a group.

group_template_declaration ::=
  group identifier **is** (entity_class_entry_list) ;

entity_class_entry_list ::=
  entity_class_entry { , entity_class_entry }

entity_class_entry ::= entity_class [<>]

A group template is characterized by the number of entity class entries and the entity class at each position. Entity classes are described in 5.1.

An entity class entry that is an entity class defines the entity class that may appear at that position in the group type. An entity class entry that includes a box (<>) allows zero or more group constituents to appear in this position in the corresponding group declaration; such an entity class entry must be the last one within the entity class entry list.

Examples:

group PIN2PIN **is** (**signal**, **signal**); -- Groups of this type consist of two signals.

group RESOURCE **is** (**label** <>); -- Groups of this type consist of any number of labels.

group DIFF_CYCLES **is** (**group** <>); -- A group of groups.

4.7 Group declarations

A group declaration declares a *group*, a named collection of named entities. Named entities are described in 5.1.

58. Terminological correction.

group_declaration ::=
    group identifier : *group_template*_name (group_constituent_list) ;

group_constituent_list ::= group_constituent { , group_constituent }

group_constituent ::= name | character_literal

It is an error if the class of any group constituent in the group constituent list is not the same as the class specified by the corresponding entity class entry in the entity class entry list of the group template.

A name that is a group constituent ~~may~~ must[59] not be an attribute name (see 6.6)~~, nor, if it contains a prefix, may that prefix be~~ . Moreover, if such a name contains a prefix, it is an error if the prefix is[60] a function call.

If a group declaration appears within a package body, and a group constituent within that group declaration is the same as the simple name of the package body, then the group constituent denotes the package declaration and not the package body. The same rule holds for group declarations appearing within subprogram bodies containing group constituents with the same designator as that of the enclosing subprogram body.

If a group declaration contains a group constituent that denotes a variable of an access type, the group declaration declares a group incorporating the variable itself, and not the designated object, if any.

Examples:

```
group G1: RESOURCE (L1, L2);                     -- A group of two labels.

group G2: RESOURCE (L3, L4, L5);                 -- A group of three labels.

group C2Q: PIN2PIN (PROJECT.GLOBALS.CK, Q);      -- Groups may associate named
                                                 -- entities in different declarative
                                                 -- parts (and regions).

group CONSTRAINT1: DIFF_CYCLES (G1, G3);         -- A group of groups.
```

59. IR1000.4.7.
60. IR1000.4.7.