~~Section~~ Clause[1] 14

Predefined language environment

This ~~section~~ clause[2] describes the predefined attributes of VHDL and the packages that all VHDL implementations must provide.

14.1 Predefined attributes

Predefined attributes denote values, functions, types, and ranges associated with various kinds of named entities. These attributes are described below. For each attribute, the following information is provided:

— The kind of attribute: value, type, range, function, or signal.

— The prefixes for which the attribute is defined.

— A description of the parameter or argument, if one exists.

— The result of evaluating the attribute, and the result type (if applicable).

— Any further restrictions or comments that apply.

T'BASE

Kind:	Type.
Prefix:	Any type or subtype T.
Result:	The base type of T.
Restrictions:	This attribute is allowed only as the prefix of the name of another attribute; for example, T'BASE'LEFT.

T'LEFT

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The left bound of T.

T'RIGHT

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The right bound of T.

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

T'HIGH

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The upper bound of T.

T'LOW

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Same type as T.
Result:	The lower bound of T.

T'ASCENDING

Kind:	Value.
Prefix:	Any scalar type or subtype T.
Result Type:	Type Boolean
Result:	TRUE if T is defined with an ascending range; FALSE otherwise.

T'IMAGE(X)

Kind:	Function.
Prefix:	Any scalar type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	Type String.
Result:	The string representation of the parameter value, without leading or trailing whitespace. If T is an enumeration type or subtype and the parameter value is either an extended identifier or a character literal, the result is expressed with both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier) or apostrophe (in the case of a character literal); in the case of an extended identifier that has a backslash, the backslash is doubled in the string representation. If T is an enumeration type or subtype and the parameter value is a basic identifier, then the result is expressed in lowercase characters. If T is a numeric type or subtype, the result is expressed as the decimal representation of the parameter value without underlines or leading or trailing zeros (except as necessary to form the image of a legal literal with the proper value); moreover, an exponent may (but is not required to) be present and the language does not define under what conditions it is or is not present. If the exponent is present, the "e" is expressed as a lowercase character. If T is a physical type or subtype, the result is expressed in terms of the primary unit of T unless the base type of T is TIME, in which case the result is expressed in terms of the resolution limit (see 3.1.3.1); in either case, if the unit is a basic identifier, the image of the unit is expressed in lowercase characters. If T is a floating point type or subtype, the number of digits to the right of the decimal point corresponds to the standard form generated when the DIGITS parameter to TextIO. Write for type REAL is set to 0 (see 14.3). The result never contains the replacement characters described in 13.10.
Restrictions:	It is an error if the parameter value does not belong to the subtype implied by the prefix.

T'VALUE(X)

Kind:	Function.
Prefix:	Any scalar type or subtype T.
Parameter:	An expression of type String.
Result Type:	The base type of T.
Result:	The value of T whose string representation (as defined in ~~Section~~ Clause[3] 13) is given by the parameter. Leading and trailing whitespace is allowed and ignored. If T is a numeric type or subtype, the parameter ~~may~~ must[4] be expressed either as a decimal literal or as a based literal. If T is a physical type or subtype, the parameter ~~may~~ must[5] be expressed using a string representation of any of the unit names of T, with or without a leading abstract literal. The parameter must have whitespace between any abstract literal and the unit name. The replacement characters of 13.10 are allowed in the parameter.
Restrictions:	It is an error if the parameter is not a valid string representation of a literal of type T or if the result does not belong to the subtype implied by T.

T'POS(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	*universal_integer*.
Result:	The position number of the value of the parameter.

T'VAL(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression of any integer type.
Result Type:	The base type of T.
Result:	The value whose position number is the *universal_integer* value corresponding to X.
Restrictions:	It is an error if the result does not belong to the range T'LOW **to** T'HIGH.

T'SUCC(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value whose position number is one greater than that of the parameter.
Restrictions:	An error occurs if X equals T'HIGH or if X does not belong to the range T'LOW **to** T'HIGH.

3. To conform to IEEE rules.
4. IR1000.4.7.
5. IR1000.4.7.

T'PRED(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value whose position number is one less than that of the parameter.
Restrictions:	An error occurs if X equals T'LOW or if X does not belong to the range T'LOW **to** T'HIGH.

T'LEFTOF(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value that is to the left of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'LEFT or if X does not belong to the range T'LOW **to** T'HIGH.

T'RIGHTOF(X)

Kind:	Function.
Prefix:	Any discrete or physical type or subtype T.
Parameter:	An expression whose type is the base type of T.
Result Type:	The base type of T.
Result:	The value that is to the right of the parameter in the range of T.
Restrictions:	An error occurs if X equals T'RIGHT or if X does not belong to the range T'LOW **to** T'HIGH.

A'LEFT [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the left bound of the Nth index range of A.
Result:	Left bound of the Nth index range of A. (If A is an alias for an array object, then the result is the left bound of the Nth index range from the declaration of A, not that of the object.)

A'RIGHT [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Right bound of the Nth index range of A. (If A is an alias for an array object, then the result is the right bound of the Nth index range from the declaration of A, not that of the object.)

A'HIGH [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Upper bound of the Nth index range of A. (If A is an alias for an array object, then the result is the upper bound of the Nth index range from the declaration of A, not that of the object.)

A'LOW [(N)]

Kind:	Function.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type of the Nth index range of A.
Result:	Lower bound of the Nth index range of A. (If A is an alias for an array object, then the result is the lower bound of the Nth index range from the declaration of A, not that of the object.)

A'RANGE [(N)]

Kind:	Range.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	The type of the Nth index range of A.
Result:	The range A'LEFT(N) **to** A'RIGHT(N) if the Nth index range of A is ascending, or the range A'LEFT(N) **downto** A'RIGHT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)

A'REVERSE_RANGE [(N)]

Kind:	Range.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	The type of the Nth index range of A.
Result:	The range A'RIGHT(N) **downto** A'LEFT(N) if the Nth index range of A is ascending, or the range A'RIGHT(N) **to** A'LEFT(N) if the Nth index range of A is descending. (If A is an alias for an array object, then the result is determined by the Nth index range from the declaration of A, not that of the object.)

A'LENGTH [(N)]

Kind:	Value.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal_integer*, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	*universal_integer*.
Result:	Number of values in the Nth index range; i.e., if the Nth index range of A is a null range, then the result is 0. Otherwise, the result is the value of T'POS(A'HIGH(N)) – T'POS(A'LOW(N)) + 1, where T is the subtype of the Nth index of A.

A'ASCENDING [(N)]

Kind:	Value.
Prefix:	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype.
Parameter:	A locally static expression of type *universal integer*, the value of which must be greater than zero and must not exceed the dimensionality of A. If omitted, it defaults to 1.
Result Type:	Type Boolean.
Result:	TRUE if the Nth index range of A is defined with an ascending range; FALSE otherwise.

S'DELAYED [(T)]

Kind:	Signal.
Prefix:	Any signal denoted by the static signal name S.
Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
Result Type:	The base type of S.
Result:	A signal equivalent to signal S delayed T units of time. The value of S'DELAYED(t) at time T_n is always equal to the value of S at time T_n–t. Specifically:

Let R be of the same subtype as S, let T >= 0 ns, and let P be a process statement of the form

```
P: process (S)
   begin
      R <= transport S after T;
   end process ;
```

Assuming that the initial value of R is the same as the initial value of S, then the attribute 'DELAYED is defined such that S'DELAYED(T) = R for any T.

S'STABLE [(T)]

Kind:	Signal.
Prefix:	Any signal denoted by the static signal name S.
Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
Result Type:	Type Boolean.
Result:	A signal that has the value TRUE when an event has not occurred on signal S for T units of time, and the value FALSE otherwise. (See 12.6.2.)

S'QUIET [(T)]

Kind:	Signal.
Prefix:	Any signal denoted by the static signal name S.
Parameter:	A static expression of type TIME that evaluates to a nonnegative value. If omitted, it defaults to 0 ns.
Result Type:	Type Boolean.
Result:	A signal that has the value TRUE when the signal has been quiet for T units of time, and the value FALSE otherwise. (See 12.6.2.)

S'TRANSACTION

Kind:	Signal.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Bit.
Result:	A signal whose value toggles to the inverse of its previous value in each simulation cycle in which signal S becomes active.
Restriction:	A description is erroneous if it depends on the initial value of S'Transaction.

S'EVENT

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Boolean.
Result:	A value that indicates whether an event has just occurred on signal S. Specifically:

For a scalar signal S, S'EVENT returns the value TRUE if an event has occurred on S during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'EVENT returns TRUE if an event has occurred on any scalar subelement of S during the current simulation cycle; otherwise, it returns FALSE.

S'ACTIVE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Boolean.
Result:	A value that indicates whether signal S is active. Specifically:

For a scalar signal S, S'ACTIVE returns the value TRUE if signal S is active during the current simulation cycle; otherwise, it returns the value FALSE.

For a composite signal S, S'ACTIVE returns TRUE if any scalar subelement of S is active during the current simulation cycle; otherwise, it returns FALSE.

S'LAST_EVENT

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Time.
Result:	The amount of time that has elapsed since the last event occurred on signal S. Specifically:

For a signal S, S'LAST_EVENT returns the smallest value T of type TIME such that S'EVENT = True during any simulation cycle at time NOW – T, if such value exists; otherwise, it returns TIME'HIGH.

S'LAST_ACTIVE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Time.
Result:	The amount of time that has elapsed since the last time at which signal S was active. Specifically:

For a signal S, S'LAST_ACTIVE returns the smallest value T of type TIME such that S'ACTIVE = True during any simulation cycle at time NOW – T, if such value exists; otherwise, it returns TIME'HIGH.

S'LAST_VALUE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	The base type of S.
Result:	The previous value of S, immediately before the last change of S.

S'DRIVING

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	Type Boolean.
Result:	If the prefix denotes a scalar signal, the result is False if the current value of the driver for S in the current process is determined by the null transaction; True otherwise. If the prefix denotes a composite signal, the result is True if and only if R'DRIVING is True for every scalar subelement R of S; False otherwise. If the prefix denotes a null slice of a signal, the result is True.
Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute name appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**.

S'DRIVING_VALUE

Kind:	Function.
Prefix:	Any signal denoted by the static signal name S.
Result Type:	The base type of S.
Result:	If S is a scalar signal S, the result is the current value of the driver for S in the current process. If S is a composite signal, the result is the aggregate of the values of R'DRIVING_VALUE for each element R of S. If S is a null slice, the result is a null slice.
Restrictions:	This attribute is available only from within a process, a concurrent statement with an equivalent process, or a subprogram. If the prefix denotes a port, it is an error if the port does not have a mode of **inout**, **out**, or **buffer**. It is also an error if the attribute name appears in a subprogram body that is not a declarative item contained within a process statement and the prefix is not a formal parameter of the given subprogram or of a parent of that subprogram. Finally, it is an error if the prefix denotes a subprogram formal parameter whose mode is not **inout** or **out**, or if S'DRIVING is False at the time of the evaluation of S'DRIVING_VALUE.

E'SIMPLE_NAME

Kind:	Value.
Prefix:	Any named entity as defined in 5.1.
Result Type:	Type String.
Result:	The simple name, character literal, or operator symbol of the named entity, without leading or trailing whitespace or quotation marks but with apostrophes (in the case of a character literal) and both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier). In the case of a simple name or operator symbol, the characters are converted to their lowercase equivalents. In the case of an extended identifier, the case of the identifier is preserved, and any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'INSTANCE_NAME

Kind:	Value.
Prefix:	Any named entity other than the local ports and generics of a component declaration.
Result Type:	Type String.
Result:	A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, including the names of instantiated design entities. Specifically:

The result string has the following syntax:

```
instance_name ::=  package_based_path | full_instance_based_path

package_based_path ::=
    leader library_logical_name leader
        [[6] package_simple_name  leader ]
        { subprogram_ simple_name signature leader }[7]
        [ local_item_name ]
```

6. LCS 23.
7. LCS 23.

```
full_instance_based_path ::=  leader full_path_to_instance [ local_item_name ]

full_path_to_instance ::=  { full_path_instance_element leader }

local_item_name ::=
      simple_name
      character_literal
      operator_symbol

full_path_instance_element ::=
      [ component_instantiation_label @ ]
            entity_simple_name ( architecture_simple_name )
      | block_label
      | generate_label
      | process_label
      | loop_label
      | subprogram_simple_name signature[8]

generate_label ::=  generate_label [ ( literal ) ]

process_label ::=  [ process_label ]

leader ::=  :
```

Package-based paths identify items declared within packages. Full-instance-based paths identify items within an elaborated design hierarchy.

A library logical name denotes a library; see 11.2. Since it is possible for[9] multiple logical names ~~may~~ to[10] denote the same library, the library logical name may not be unique.

The local item name in E'INSTANCE_NAME equals E'SIMPLE_NAME, unless E denotes a library, package, subprogram, or label. In this latter case, the package based path or full instance based path, as appropriate, will not contain a local item name.[11]

There is one full path instance element for each component instantiation, block statement, generate statement, process statement, or subprogram body in the design hierarchy between the top design entity and the named entity denoted by the prefix.

In a full path instance element, the architecture simple name must denote an architecture associated with the entity ~~interface~~ declaration[12] designated by the entity simple name; furthermore, the component instantiation label (and the commercial at following it) are required unless the entity simple name and the architecture simple name together denote the root design entity.

The literal in a generate label is required if the label denotes a generate statement with a for generation scheme; the literal must denote one of the values of the generate parameter.

A process statement with no label is denoted by an empty process label.

All characters in basic identifiers appearing in the result are converted to their lowercase equivalents. Both a leading and trailing reverse solidus surround an extended identifier appearing in the result;

8. LCS 23.
9. IR1000.4.7.
10. IR1000.4.7.
11. LCS 23.
12. Terminological correction.

any reverse solidus characters appearing as part of the identifier are represented with two consecutive reverse solidus characters.

E'PATH_NAME

Kind:	Value.
Prefix:	Any named entity other than the local ports and generics of a component declaration.
Result Type:	Type String.
Result:	A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, excluding the name of instantiated design entities. Specifically:

The result string has the following syntax:

```
path_name ::=  package_based_path | instance_based_path

instance_based_path ::=
    leader path_to_instance [ local_item_name ]

path_to_instance ::=  { path_instance_element leader }

path_instance_element  ::=
    component_instantiation_label
    | entity_simple_name
    | block_label
    | generate_label
    | process_label
    | subprogram_simple_name signature[13]
```

Package-based paths identify items declared within packages. Instance-based paths identify items within an elaborated design hierarchy.

The local item name in E'PATH_NAME equals E'SIMPLE_NAME, unless E denotes a library, package, subprogram, or label. In this latter case, the package based path or instance based path, as appropriate, will not contain a local item name.[14]

There is one path instance element for each component instantiation, block statement, generate statement, process statement, or subprogram body in the design hierarchy between the root design entity and the named entity denoted by the prefix.

Examples:

```
library Lib: ;[15]
package P is                                    -- All design units are in this library:
                                                -- P'PATH_NAME = ":lib:p:"
                                                -- P'INSTANCE_NAME = ":lib:p:"
    procedure Proc (F: inout INTEGER);          -- Proc'PATH_NAME = ":lib:p:proc[integer]:"[16]
                                                -- Proc'INSTANCE_NAME = ":lib:p:proc[integer]:"[17]
    constant C: INTEGER := 42;                  -- C'PATH_NAME = ":lib:p:c"
end package P;                                  -- C'INSTANCE_NAME = ":lib:p:c"
```

13. LCS 23.
14. LCS 23.
15. IR1000.1.8.
16. LCS 23.
17. LCS 23.

```
package body P is
    procedure Proc (F: inout INTEGER) is
        variable x: INTEGER;                -- x'PATH_NAME = ":lib:p:proc[integer]18:x"
    begin                                   -- x'INSTANCE_NAME = ":lib:p:proc[integer]19:x"
        . . .
    end;
end;

library Lib;
use Lib.P.all;                              -- Assume that E is in Lib and
entity E is                                 -- E is the top-level design entity:
                                            -- E'PATH_NAME = ":e:"
                                            -- E'INSTANCE_NAME =  ":e(a):"
    generic (G: INTEGER);                   -- G'PATH_NAME = ":e:g"
                                            -- G'INSTANCE_NAME =  ":e(a):g"
    port (P: in INTEGER);                   -- P'PATH_NAME = ":e:p"
end entity E;                               -- P'INSTANCE_NAME =  ":e(a):p"

architecture A of E is
    signal S: BIT_VECTOR (1 to G);          -- S'PATH_NAME = ":e:s"
                                            -- S'INSTANCE_NAME =  ":e(a):s"
    procedure Proc1 (signal sp1: NATURAL; C: out INTEGER) is
                                            -- Proc1'PATH_NAME =  ":e:proc1[natural,integer]20:"
                                            -- Proc1'INSTANCE_NAME =
                                            --                 "21:e(a):proc1[natural,integer]22:"
                                            -- C'PATH_NAME =  ":e:proc1[natural,integer]23:c"
                                            -- C'INSTANCE_NAME =
                                            --                 ":e(a):proc1[natural,integer]24:c"
        variable max: DELAY_LENGTH;         -- max'PATH_NAME = ":e:proc1[natural,integer]25:max"
                                            -- max'INSTANCE_NAME =
    begin                                   --                 ":e(a):proc1[natural,integer]26:max"
        max := sp1 * ns;
        wait on sp1 for max;
        c := sp1;
    end procedure Proc1;
begin
p1: process
        variable T: INTEGER := 12;              -- T'PATH_NAME =  ":e:p1:t"
    begin                                       -- T'INSTANCE_NAME =  ":e(a):p1:t"
        . . .
    end process p1;

    process
        variable T: INTEGER := 12;              -- T'PATH_NAME =  ":e::t"
    begin                                       -- T'INSTANCE_NAME =  ":e(a)::t"
        . . .
    end process;
end architecture;
```

18. LCS 23.
19. LCS 23.
20. LCS 23.
21. Typo correction.
22. LCS 23.
23. LCS 23.
24. LCS 23.
25. LCS 23.
26. LCS 23.

```
entity Bottom is
    generic(GBottom : INTEGER);
    port    (PBottom : INTEGER);
end entity Bottom;

architecture BottomArch of Bottom is
    signal SBottom : INTEGER;
begin
    ProcessBottom : process
        variable V : INTEGER;
    begin
        if GBottom = 4 then
            assert V'Simple_Name = "v"
                and V'Path_Name = ":top:b1:b2:g1(4):b3:l1:processbottom:v"
                and V'Instance_Name =
                  ":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):processbottom:v";
            assert GBottom'Simple_Name = "bottom gbottom[27]"
                and GBottom'Path_Name = ":top:b1:b2:g1(4):b3:l1:gbottom"
                and GBottom'Instance_Name =
                  ":top(top):b1:b2:g1(4):b3:l1@bottom(bottomarch):gbottom";
        elsif GBottom = -1 then
            assert V'Simple_Name = "v"
                and V'Path_Name = ":top:l2:processbottom:v"
                and V'Instance_Name =
                  ":top(top):l2@bottom(bottomarch):processbottom:v";
            assert GBottom'Simple_Name = "gbottom"
                and GBottom'Path_Name = "top:l2:gbottom"
                and GBottom'Instance_Name =
                  ":top(top):l2@bottom(bottomarch):gbottom";
        end if;
    wait;
    end process ProcessBottom;
end architecture BottomArch;

entity Top is end Top;

architecture Top of Top is
    component BComp is
        generic(GComp : INTEGER)
        port    (PComp : INTEGER);
    end component BComp;

    signal S : INTEGER;
begin
    B1 : block
        signal S : INTEGER;
    begin
        B2 : block
            signal S : INTEGER;
        begin
            G1 : for I in 1 to 10 generate
                B3 : block
                    signal S : INTEGER;
                    for L1 : BComp use entity Work.Bottom(BottomArch)
                        generic map(GBottom => GComp)
                        port map(PBottom => PComp);
```

27. IR1000.1.13.

```
                    begin
                        L1 : BComp generic map (I) port map (S);
                        P1 : process
                            variable V : INTEGER;
                        begin
                            if I = 7 then
                                assert V'Simple_Name = "v"
                                    and V'Path_Name = ":top:b1:b2:g1(7):b3:p1:v"
                                    and V'Instance_Name = ":top(top):b1:b2:g1(7):b3:p1:v";
                                assert P1'Simple_Name = "p1"
                                    and P1'Path_Name = ":top:b1:b2:g1(7):b3:p1:"
                                    and P1'Instance_Name = ":top(top):b1:b2:g1(7):b3:p1:";
                                assert S'Simple_Name = "s"
                                    and S'Path_Name = ":top:b1:b2:g1(7):b3:s"
                                    and S'Instance_Name = ":top(top):b1:b2:g1(7):b3:s";
                                assert B1.S'Simple_Name = "s"
                                    and B1.S'Path_Name = ":top:b1:s"
                                    and B1.S'Instance_Name = ":top(top):b1:s";
                            end if;

                            wait;
                        end process P1;
                    end block B3;
                end generate;
            end block B2;
        end block B1;
        L2 : BComp generic map (-1) port map (S);
    end architecture Top;

    configuration TopConf of Top is
        for Top
            for L2 : BComp use
                entity Work.Bottom(BottomArch)
                    generic map     (GBottom => GComp)
                    port map        (PBottom => PComp);
            end for;
        end for;
    end configuration TopConf;
```

NOTES

1—The relationship between the values of the LEFT, RIGHT, LOW, and HIGH attributes is expressed in the following table:

		Ascending range	**Descending range**
T'LEFT	=	T'LOW	T'HIGH
T'RIGHT	=	T'HIGH	T'LOW

2—Since the attributes S'EVENT, S'ACTIVE, S'LAST_EVENT, S'LAST_ACTIVE, and S'LAST_VALUE are functions, not signals, they cannot cause the execution of a process, even though the value returned by such a function may change dynamically. It is thus recommended that the equivalent signal-valued attributes S'STABLE and S'QUIET, or expressions involving those attributes, be used in concurrent contexts such as guard expressions or concurrent signal assignments. Similarly, function STANDARD.NOW should not be used in concurrent contexts.

3—S'DELAYED(0 ns) is not equal to S during any simulation cycle where S'EVENT is true.

4—S'STABLE(0 ns) = (S'DELAYED(0 ns) = S), and S'STABLE(0 ns) is FALSE only during a simulation cycle in which S has had a transaction.

5—For a given simulation cycle, S'QUIET(0 ns) is TRUE if and only if S is quiet for that simulation cycle.

6—If S'STABLE(T) is FALSE, then, by definition, for some t where 0 ns ≤ t ≤ T, S'DELAYED(t) /= S.

7—If T_S is the smallest value such that S'STABLE (T_S) is FALSE, then for all t where 0 ns ≤ t < T_S, S'DELAYED(t) = S.

8—S'EVENT should not be used within a postponed process (or a concurrent statement that has an equivalent postponed process) to determine if the prefix signal S caused the process to resume. However, S'LAST_EVENT = 0 ns can be used for this purpose.

9—The values of E'PATH_NAME and E'INSTANCE_NAME are not unique. Specifically, named entities in two different, unlabelled processes may have the same path names or instance names. Overloaded subprograms, and named entities within them, may also have the same path names or instance names.

10—If the prefix to the attributes 'SIMPLE_NAME, 'PATH_NAME, or 'INSTANCE_NAME denotes an alias, the result is respectively the simple name, path name or instance name of the alias. See 6.6.

11—For all values V of any scalar type T except a real type, the following relation holds:

V = T'Value(T'Image(V))

14.2 Package STANDARD

Package STANDARD predefines a number of types, subtypes, and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit. Package STANDARD ~~may not~~ cannot[28] be modified by the user.

The operators that are predefined for the types declared for package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_integer*), formal parameters, and undefined information (such as *implementation_defined*).

```
package STANDARD is

    -- Predefined enumeration types:

    type BOOLEAN is (FALSE, TRUE);

    -- The predefined operators for this type are as follows:

    -- function "and"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "or"   (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "nand" (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "nor"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "xor"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "xnor" (anonymous, anonymous: BOOLEAN) return BOOLEAN;

    -- function "not"(anonymous: BOOLEAN) return BOOLEAN;

    -- function "="  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "/=" (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "<"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function "<=" (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function ">"  (anonymous, anonymous: BOOLEAN) return BOOLEAN;
    -- function ">=" (anonymous, anonymous: BOOLEAN) return BOOLEAN;

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

28. IR1000.4.7.

```
-- The predefined operators for this type are as follows:

-- function "and"  (anonymous, anonymous: BIT) return BIT;
-- function "or"   (anonymous, anonymous: BIT) return BIT;
-- function "nand" (anonymous, anonymous: BIT) return BIT;
-- function "nor"  (anonymous, anonymous: BIT) return BIT;
-- function "xor"  (anonymous, anonymous: BIT) return BIT;
-- function "xnor" (anonymous, anonymous: BIT) return BIT;

-- function "not" (anonymous: BIT) return BIT;

-- function "="  (anonymous, anonymous: BIT) return BOOLEAN;
-- function "/=" (anonymous, anonymous: BIT) return BOOLEAN;
-- function "<"  (anonymous, anonymous: BIT) return BOOLEAN;
-- function "<=" (anonymous, anonymous: BIT) return BOOLEAN;
-- function ">"  (anonymous, anonymous: BIT) return BOOLEAN;
-- function ">=" (anonymous, anonymous: BIT) return BOOLEAN;

type CHARACTER is (
    NUL,    SOH,    STX,    ETX,    EOT,    ENQ,    ACK,    BEL,
    BS,     HT,     LF,     VT,     FF,     CR,     SO,     SI,
    DLE,    DC1,    DC2,    DC3,    DC4,    NAK,    SYN,    ETB,
    CAN,    EM,     SUB,    ESC,    FSP,    GSP,    RSP,    USP,

    ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',
    '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',
    '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
    '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',

    '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
    'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
    'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
    'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',

    '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
    'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
    'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
    'x',    'y',    'z',    '{',    '|',    '}',    '~',    DEL,

    C128,   C129,   C130,   C131,   C132,   C133,   C134,   C135,
    C136,   C137,   C138,   C139,   C140,   C141,   C142,   C143,
    C144,   C145,   C146,   C147,   C148,   C149,   C150,   C151,
    C152,   C153,   C154,   C155,   C156,   C157,   C158,   C159,

    ' ',[29] '¡',   '¢',    '£',    '',     '¥',    ' ',[30] '§',
    '¨',    '©',    'ª',    '«',    '¬',    '-',    '®',    '¯',
    '°',    '±',    '²',    '³',    '´',    'µ',    '¶',    '·',
    '¸',    '¹',    'º',    '»',    '¼',    '½',    '¾',    '¿',

    'À',    'Á',    'Â',    'Ã',    'Ä',    'Å',    'Æ',    'Ç',
    'È',    'É',    'Ê',    'Ë',    'Ì',    'Í',    'Î',    'Ï',
    'Ð',    'Ñ',    'Ò',    'Ó',    'Ô',    'Õ',    'Ö',    '5',
    'Ø',    'Ù',    'Ú',    'Û',    'Ü',    'Ý',    'Þ',    'ß',
```

29. The nonbreaking space character.
30. The soft hyphen character.

```
    'à',    'á',    'â',    'ã',    'ä',    'å',    'æ',    'ç',
    'è',    'é',    'ê',    'ë',    'ì',    'í',    'î',    'ï',
    'ð',    'ñ',    'ò',    'ó',    'ô',    'õ',    'ö',    '÷',
    'ø',    'ù',    'ú',    'û',    'ü',    'ý',    'þ',    'ÿ');

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "/="  (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "<"   (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "<=" (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function ">"   (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function ">=" (anonymous, anonymous: CHARACTER) return BOOLEAN;

type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function "/="  (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function "<"   (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function "<=" (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function ">"   (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function ">=" (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;

-- type universal_integer  is range implementation_defined;

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "/="  (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "<"   (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "<=" (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function ">"   (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function ">=" (anonymous, anonymous: universal_integer) return BOOLEAN;

-- function "+"   (anonymous: universal_integer) return universal_integer;
-- function "-"   (anonymous: universal_integer) return universal_integer;
-- function "abs"(anonymous: universal_integer) return universal_integer;

-- function "+"      (anonymous, anonymous: universal_integer) return universal_integer;
-- function "–"      (anonymous, anonymous: universal_integer) return universal_integer;
-- function "*"      (anonymous, anonymous: universal_integer) return universal_integer;
-- function "/"      (anonymous, anonymous: universal_integer) return universal_integer;
-- function "mod"  (anonymous, anonymous: universal_integer) return universal_integer;
-- function "rem"  (anonymous, anonymous: universal_integer) return universal_integer;

-- type universal_real  is range implementation_defined;

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "/="  (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "<"   (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "<=" (anonymous, anonymous: universal_real) return BOOLEAN;
-- function ">"   (anonymous, anonymous: universal_real) return BOOLEAN;
-- function ">=" (anonymous, anonymous: universal_real) return BOOLEAN;
```

```
-- function "+"   (anonymous: universal_real) return universal_real;
-- function "–"   (anonymous: universal_real) return universal_real;
-- function "abs" (anonymous: universal_real) return universal_real;

-- function "+"   (anonymous, anonymous: universal_real) return universal_real;
-- function "–"   (anonymous, anonymous: universal_real) return universal_real;
-- function "*"   (anonymous, anonymous: universal_real) return universal_real;
-- function "/"   (anonymous, anonymous: universal_real) return universal_real;

-- function "*"   (anonymous: universal_real; anonymous: universal_integer) return universal_real;
-- function "*"   (anonymous: universal_integer; anonymous: universal_real) return universal_real;
-- function "/"   (anonymous: universal_real; anonymous: universal_integer) return universal_real;

-- Predefined numeric types:

type INTEGER is range implementation_defined;

-- The predefined operators for this type are as follows:

-- function "**"  (anonymous: universal_integer; anonymous: INTEGER return universal_integer;
-- function "**"  (anonymous: universal_real; anonymous: INTEGER)    return universal_real;

-- function "="   (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "/="  (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "<"   (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "<="  (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function ">"   (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function ">="  (anonymous, anonymous: INTEGER) return BOOLEAN;

-- function "+"   (anonymous: INTEGER) return INTEGER;
-- function "–"   (anonymous: INTEGER) return INTEGER;
-- function "abs" (anonymous: INTEGER) return INTEGER;

-- function "+"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "–"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "*"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "/"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "mod"    (anonymous, anonymous: INTEGER) return INTEGER;
-- function "rem"    (anonymous, anonymous: INTEGER) return INTEGER;

-- function "**"  (anonymous: INTEGER; anonymous: INTEGER) return INTEGER;

type REAL is range implementation_defined;

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: REAL) return BOOLEAN;
-- function "/="  (anonymous, anonymous: REAL) return BOOLEAN;
-- function "<"   (anonymous, anonymous: REAL) return BOOLEAN;
-- function "<="  (anonymous, anonymous: REAL) return BOOLEAN;
-- function ">"   (anonymous, anonymous: REAL) return BOOLEAN;
-- function ">="  (anonymous, anonymous:  REAL) return BOOLEAN;

-- function "+"   (anonymous: REAL) return REAL;
-- function "–"   (anonymous: REAL) return REAL;
-- function "abs" (anonymous: REAL) return REAL;
```

```
-- function "+"   (anonymous, anonymous: REAL) return REAL;
-- function "–"   (anonymous, anonymous: REAL) return REAL;
-- function "*"   (anonymous, anonymous: REAL) return REAL;
-- function "/"   (anonymous, anonymous: REAL) return REAL;

-- function "**"  (anonymous: REAL; anonymous: INTEGER) return REAL;

-- Predefined type TIME:

type TIME is range implementation_defined
    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
        hr      =   60 min;         -- hour
    end units;

-- The predefined operators for this type are as follows:

-- function "="   (anonymous, anonymous: TIME) return BOOLEAN;
-- function "/="  (anonymous, anonymous: TIME) return BOOLEAN;
-- function "<"   (anonymous, anonymous: TIME) return BOOLEAN;
-- function "<="  (anonymous, anonymous: TIME) return BOOLEAN;
-- function ">"   (anonymous, anonymous: TIME) return BOOLEAN;
-- function ">="  (anonymous, anonymous: TIME) return BOOLEAN;

-- function "+"   (anonymous: TIME) return TIME;
-- function "–"   (anonymous: TIME) return TIME;
-- function "abs" (anonymous: TIME) return TIME;

-- function "+"   (anonymous, anonymous: TIME) return TIME;
-- function "–"   (anonymous, anonymous: TIME) return TIME;

-- function "*"   (anonymous: TIME;    anonymous: INTEGER)  return TIME;
-- function "*"   (anonymous: TIME;    anonymous: REAL)     return TIME;
-- function "*"   (anonymous: INTEGER; anonymous: TIME)     return TIME;
-- function "*"   (anonymous: REAL;    anonymous: TIME)     return TIME;
-- function "/"   (anonymous: TIME;    anonymous: INTEGER)  return TIME;
-- function "/"   (anonymous: TIME;    anonymous: REAL)     return TIME;

-- function "/"   (anonymous, anonymous: TIME) return universal_integer;

subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;

-- A function that returns the current simulation time, Tc (see 12.6.4):

impure pure[31] function NOW return DELAY_LENGTH;

-- Predefined numeric subtypes:

subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
```

31. LCS 9.

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

-- Predefined array types:

type STRING is array (POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:

-- function "="  (anonymous, anonymous: STRING) return BOOLEAN;
-- function "/=" (anonymous, anonymous: STRING) return BOOLEAN;
-- function "<"  (anonymous, anonymous: STRING) return BOOLEAN;
-- function "<=" (anonymous, anonymous: STRING) return BOOLEAN;
-- function ">"  (anonymous, anonymous: STRING) return BOOLEAN;
-- function ">=" (anonymous, anonymous: STRING) return BOOLEAN;

-- function "&" (anonymous: STRING;    anonymous: STRING)    return STRING;
-- function "&" (anonymous: STRING;    anonymous: CHARACTER) return STRING;
-- function "&" (anonymous: CHARACTER; anonymous: STRING)    return STRING;
-- function "&" (anonymous: CHARACTER; anonymous: CHARACTER) return STRING;

type BIT_VECTOR is array (NATURAL range <>) of BIT;

-- The predefined operators for this type are as follows:

-- function "and"  (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "or"   (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "nand" (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "nor"  (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "xor"  (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "xnor" (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;

-- function "not" (anonymous: BIT_VECTOR) return BIT_VECTOR;

-- function "sll" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;
-- function "srl" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;
-- function "sla" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;
-- function "sra" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;
-- function "rol" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;
-- function "ror" (anonymous: BIT_VECTOR; anonymous: INTEGER) return BIT_VECTOR;

-- function "="  (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "/=" (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "<"  (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "<=" (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function ">"  (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function ">=" (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;

-- function "&" (anonymous: BIT_VECTOR; anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "&" (anonymous: BIT_VECTOR; anonymous: BIT)        return BIT_VECTOR;
-- function "&" (anonymous: BIT;        anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "&" (anonymous: BIT;        anonymous: BIT)        return BIT_VECTOR;
```

```
    -- The predefined types for opening files:

    type FILE_OPEN_KIND is (
        READ_MODE,              -- Resulting access mode is read-only.
        WRITE_MODE,             -- Resulting access mode is write-only.
        APPEND_MODE);           -- Resulting access mode is write-only; information
                                -- is appended to the end of the existing file.

    -- The predefined operators for this type are as follows:

    -- function "="  (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
    -- function "/=" (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
    -- function "<"  (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
    -- function "<=" (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
    -- function ">"  (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
    -- function ">=" (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;

    type FILE_OPEN_STATUS is (
        OPEN_OK,            -- File open was successful.
        STATUS_ERROR,       -- File object was already open.
        NAME_ERROR,         -- External file not found or inaccessible.
        MODE_ERROR);        -- Could not open file with requested access mode.

    -- The predefined operators for this type are as follows:

    -- function "="  (anonymous, anonymous: FILE_OPEN_STATUS) return BOOLEAN;
    -- function "/=" (anonymous, anonymous: FILE_OPEN_STATUS) return BOOLEAN;
    -- function "<"  (anonymous, anonymous: FILE_OPEN_STATUS  return BOOLEAN;
    -- function "<=" (anonymous, anonymous: FILE_OPEN_STATUS) return BOOLEAN;
    -- function ">"  (anonymous, anonymous: FILE_OPEN_STATUS) return BOOLEAN;
    -- function ">=" (anonymous, anonymous: FILE_OPEN_STATUS) return BOOLEAN;

    -- The 'FOREIGN attribute:

    attribute FOREIGN: STRING;

end STANDARD;
```

The 'FOREIGN attribute ~~may~~ must[32] be associated only with architectures (see 1.2) or with subprograms. In the latter case, the attribute specification must appear in the declarative part in which the subprogram is declared (see 2.1).

NOTES

1—The ASCII mnemonics for file separator (FS), group separator (GS), record separator (RS), and unit separator (US) are represented by FSP, GSP, RSP, and USP, respectively, in type CHARACTER in order to avoid conflict with the units of type TIME.

2—The declarative parts and statement parts of design entities whose corresponding architectures are decorated with the 'FOREIGN attribute and subprograms that are likewise decorated are subject to special elaboration rules. See 12.3 and 12.4.

3—The function STD.STANDARD.NOW is pure only within a single discrete time step; that is, within a set of simulation cycles whose T_cs are equal (see 12.6.4).[33]

32. IR1000.4.7.
33. LCS 9.

14.3 Package TEXTIO

Package TEXTIO contains declarations of types and subprograms that support formatted I/O operations on text files.

```
package TEXTIO is

    -- Type definitions for text I/O:

    type LINE is access STRING;                 -- A LINE is a pointer to a STRING value.

    -- The predefined operators for this type are as follows:

    -- function "="  (anonymous, anonymous: LINE) return BOOLEAN;
    -- function "/=" (anonymous, anonymous: LINE) return BOOLEAN;[34]

    type TEXT is file of STRING;                -- A file of variable-length ASCII records.

    -- The predefined operators for this type are as follows:

    -- procedure FILE_OPEN (file F: TEXT; External_Name; in STRING;
    --                      Open_Kind: in FILE_OPEN_KIND := READ_MODE);
    -- procedure FILE_OPEN (Status: out FILE_OPEN_STATUS; file F: TEXT;
    --                      External_Name: in STRING;
    --                      Open_Kind: in FILE_OPEN_KIND := READ_MODE);
    -- procedure FILE_CLOSE (file F: TEXT);
    -- procedure READ (file F: TEXT; VALUE: out STRING);
    -- procedure WRITE (file F: TEXT; VALUE: in STRING);
    -- function ENDFILE (file F: TEXT) return BOOLEAN;[35]

    type SIDE is (RIGHT, LEFT);                 -- For justifying output data within fields.

    -- The predefined operators for this type are as follows:

    -- function "="  (anonymous, anonymous: SIDE) return BOOLEAN;
    -- function "/=" (anonymous, anonymous: SIDE) return BOOLEAN;
    -- function "<"  (anonymous, anonymous: SIDE  return BOOLEAN;
    -- function "<=" (anonymous, anonymous: SIDE) return BOOLEAN;
    -- function ">"  (anonymous, anonymous: SIDE) return BOOLEAN;
    -- function ">=" (anonymous, anonymous: SIDE) return BOOLEAN;[36]

    subtype WIDTH is NATURAL;                   -- For specifying widths of output fields.

  -- Standard text files:

    file INPUT:  TEXT open READ_MODE  is "STD_INPUT";

    file OUTPUT: TEXT open WRITE_MODE is "STD_OUTPUT";

  -- Input routines for standard types:

    procedure READLINE (file F: TEXT; L: ~~out~~ inout[37] LINE);
```

34. Boyer.
35. Boyer.
36. Boyer.
37. Typo noted by Ashenden.

```
    procedure READ (L: inout LINE;VALUE: out BIT;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out BIT);

    procedure READ (L: inout LINE;VALUE: out BIT_VECTOR;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out BIT_VECTOR);

    procedure READ (L: inout LINE;VALUE: out BOOLEAN;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out BOOLEAN);

    procedure READ (L: inout LINE;VALUE: out CHARACTER;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out CHARACTER);

    procedure READ (L: inout LINE;VALUE: out INTEGER;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out INTEGER);

    procedure READ (L: inout LINE;VALUE: out REAL;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out REAL);

    procedure READ (L: inout LINE;VALUE: out STRING;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out STRING);

    procedure READ (L: inout LINE;VALUE: out TIME;GOOD: out BOOLEAN);
    procedure READ (L: inout LINE;VALUE: out TIME);

    -- Output routines for standard types:

    procedure WRITELINE (file F: TEXT; L: inout LINE);

    procedure WRITE (L: inout LINE;VALUE: in BIT;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in BIT_VECTOR;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in BOOLEAN;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in CHARACTER;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in INTEGER;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in REAL;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                     DIGITS: in NATURAL:= 0);

    procedure WRITE (L: inout LINE;VALUE: in STRING;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);

    procedure WRITE (L: inout LINE;VALUE: in TIME;
                     JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                     UNIT: in TIME:= ns);

    -- File position predicate:

--  function ENDFILE (file F: TEXT) return BOOLEAN;
```

end TEXTIO;

Procedures READLINE and WRITELINE declared in package TEXTIO read and write entire lines of a file of type TEXT. Procedure READLINE causes the next line to be read from the file and returns as the value of parameter L an access value that designates an object representing that line. If parameter L contains a nonnull access value at the start of the call, the object designated by that value is deallocated before the new object is created. The representation of the line does not contain the representation of the end of the line. It is an error if the file specified in a call to READLINE is not open or, if open, the file has an access mode other than read-only (see 3.4.1). Procedure WRITELINE causes the current line designated by parameter L to be written to the file and returns with the value of parameter L designating a null string. If parameter L contains a null access value at the start of the call, then a null string is written to the file. It is an error if the file specified in a call to WRITELINE is not open or, if open, the file has an access mode other than write-only.

The language does not define the representation of the end of a line. An implementation must allow all possible values of types CHARACTER and STRING to be written to a file. However, as an implementation is permitted to use certain values of types CHARACTER and STRING as line delimiters, it ~~may~~ might[38] not be possible to read these values from a TEXT file.

Each READ procedure declared in package TEXTIO extracts data from the beginning of the string value designated by parameter L and modifies the value so that it designates the remaining portion of the line on exit.

The READ procedures defined for a given type other than CHARACTER and STRING begin by skipping leading *whitespace characters*. A whitespace character is defined as a space, a nonbreaking space, or a horizontal tabulation character (SP, NBSP, or HT). For all READ procedures, characters are then removed from L and composed into a string representation of the value of the specified type. Character removal and string composition stops when a character is encountered that cannot be part of the value according to the lexical rules of 13.2; this character is not removed from L and is not added to the string representation of the value. The READ procedures for types INTEGER and REAL also accept a leading sign; additionally, there can be no space between the sign and the remainder of the literal. The READ procedures for types STRING and BIT_VECTOR also terminate acceptance when VALUE'LENGTH characters have been accepted. Again using the rules of 13.2, the accepted characters are then interpreted as a string representation of the specified type. The READ does not succeed if the sequence of characters removed from L is not a valid string representation of a value of the specified type or, in the case of types STRING and BIT_VECTOR, if the sequence does not contain VALUE'LENGTH characters.

The definitions of the string representation of the value for each data type are as follows:

— The representation of a BIT value is formed by a single character, either 1 or 0. No leading or trailing quotation characters are present.

— The representation of a BIT_VECTOR value is formed by a sequence of characters, either 1 or 0. No leading or trailing quotation characters are present.

— The representation of a BOOLEAN value is formed by an identifier, either FALSE or TRUE.

— The representation of a CHARACTER value is formed by a single character.

— The representation of both INTEGER and REAL values is that of a decimal literal (see 13.4.1), with the addition of an optional leading sign. The sign is never written if the value is nonnegative, but it is accepted during a read even if the value is nonnegative. No spaces can occur between the sign and the remainder of the value. The decimal point is absent in the case of an INTEGER literal and present in the case of a REAL literal. An exponent may optionally be present; moreover, the language does not define under what conditions it is or is not present. However, if the exponent is present, the “e” is written as a lowercase character. Leading and trailing zeroes are written as necessary to meet the requirements of the FIELD and DIGITS parameters, and they are accepted during a read.

38. IR1000.4.7.

— The representation of a STRING value is formed by a sequence of characters, one for each element of the string. No leading or trailing quotation characters are present.

— The representation of a TIME value is formed by an optional decimal literal composed following the rules for INTEGER and REAL literals described above, one or more blanks, and an identifier that is a unit of type TIME, as defined in package STANDARD (see 14.2). When read, the identifier can be expressed with characters of either case; when written, the identifier is expressed in lowercase characters.

Each WRITE procedure similarly appends data to the end of the string value designated by parameter L; in this case, however, L continues to designate the entire line after the value is appended. The format of the appended data is defined by the string representations defined above for the READ procedures.

The READ and WRITE procedures for the types BIT_VECTOR and STRING respectively read and write the element values in left-to-right order.

For each predefined data type there are two READ procedures declared in package TEXTIO. The first has three parameters: L, the line to read from; VALUE, the value read from the line; and GOOD, a Boolean flag that indicates whether the read operation succeeded or not. For example, the operation READ (L, IntVal, OK) would return with OK set to FALSE, L unchanged, and IntVal undefined if IntVal is a variable of type INTEGER and L designates the line "ABC". The success indication returned via parameter GOOD allows a process to recover gracefully from unexpected discrepancies in input format. The second form of read operation has only the parameters L and VALUE. If the requested type cannot be read into VALUE from line L, then an error occurs. Thus, the operation READ (L, IntVal) would cause an error to occur if IntVal is of type INTEGER and L designates the line "ABC".

For each predefined data type there is one WRITE procedure declared in package TEXTIO. Each of these has at least two parameters: L, the line to which to write; and VALUE, the value to be written. The additional parameters JUSTIFIED, FIELD, DIGITS, and UNIT control the formatting of output data. Each write operation appends data to a line formatted within a *field* that is at least as long as required to represent the data value. Parameter FIELD specifies the desired field width. Since the actual field width will always be at least large enough to hold the string representation of the data value, the default value 0 for the FIELD parameter has the effect of causing the data value to be written out in a field of exactly the right width (i.e., no leading or trailing spaces). Parameter JUSTIFIED specifies whether values are to be right- or left-justified within the field; the default is right-justified. If the FIELD parameter describes a field width larger than the number of characters necessary for a given value, blanks are used to fill the remaining characters in the field.

Parameter DIGITS specifies how many digits to the right of the decimal point are to be output when writing a real number; the default value 0 indicates that the number should be output in standard form, consisting of a normalized mantissa plus exponent (e.g., 1.079236E–23). If DIGITS is nonzero, then the real number is output as an integer part followed by '.' followed by the fractional part, using the specified number of digits (e.g., 3.14159).

Parameter UNIT specifies how values of type TIME are to be formatted. The value of this parameter must be equal to one of the units declared as part of the declaration of type TIME; the result is that the TIME value is formatted as an integer or real literal representing the number of multiples of this unit, followed by the name of the unit itself. The name of the unit is formatted using only lowercase characters. Thus the procedure call WRITE(Line, 5 ns, UNIT=>us) would result in the string value "0.005 us" being appended to the string value designated by Line, whereas WRITE(Line, 5 ns) would result in the string value "5 ns" being appended (since the default UNIT value is ns).

Function ENDFILE is defined for files of type TEXT by the implicit declaration of that function as part of the declaration of the file type.

NOTES

1—For a variable L of type Line, attribute L'Length gives the current length of the line, whether that line is being read or written. For a line L that is being written, the value of L'Length gives the number of characters that have already been written to the line; this is equivalent to the column number of the last character of the line. For a line L that is being read, the value

of L'Length gives the number of characters on that line remaining to be read. In particular, the expression L'Length = 0 is true precisely when the end of the current line has been reached.

2—The execution of a read or write operation may modify or even deallocate the string object designated by input parameter L of type Line for that operation; thus, a dangling reference may result if the value of a variable L of type Line is assigned to another access variable and then a read or write operation is performed on L.