# Section 3

# Data types

Verilog-AMS HDL supports `integer`, `real`, and `parameter` data types as found in *IEEE 1364-1995 Verilog HDL*. It also modifies the `parameter` data types and introduces *array of real* as an extension of the `real` data type. Plus, it extends the `net` data types to support a new type called `wreal` to model real value nets.

Verilog-AMS HDL introduces a new data type, called *net_discipline*, for representing analog nets and declaring *disciplines* of all `nets` and `regs`. The *disciplines* define the `domain` and the natures of `potential` and `flow` and their associated attributes for *continuous* domains. A new data type called *genvar* is also introduced for use with behavioral loops.

## 3.1 Integer and real data types

The syntax for declaring `integer` and `real` is shown in Syntax 3-1.

integer_declaration ::=
**integer** list_of_identifiers **;**

real_declaration ::=
**real** list_of_identifiers **;**

list_of_identifiers ::=
var_name { **,** var_name }

var_name ::=
*variable*_identifier
| *array*_identifier array_range

array_range ::=
**[** *upper_limit*_constant_expression **:** *lower_limit*_constant_expression **]**

*Syntax 3-1—Syntax for integer and real declarations*

An *integer* declaration declares one or more variables of type integer. These variables can hold values ranging from $-2^{31}$ to $2^{31}-1$. Arrays of integers can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (`0`).

Arithmetic operations performed on integer variables produce 2’s complement results.

A *real* declaration declares one or more variables of type real. The real variables are stored as 64-bit quantities, as described by *IEEE STD-754-1985*, an IEEE standard for double precision floating point numbers.

Arrays of real can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (`0`).

Integers are initialized at the start of a simulation depending on how they are used. Integer variables whose values are assigned in an analog context default to an initial value of zero (`0`). Integer variables whose values are assigned in a digital context default to an initial value of `x`. Real variables are initialized to zero (`0`) at the start of a simulation.

*Examples:*

```
integer a[1:64]; // an array of 64 integer values
real float ; // a variable to store real value
real gain_factor[1:30] ;// array of 30 gain multipliers
             // with floating point values
```

## 3.2 Parameters

The syntax for parameter declarations is shown in Syntax 3-2.

The list of parameter assignments shall be a comma-separated list of assignments, where the right hand side of the assignment shall be a constant expression, that is, an expression containing only constant numbers and previously defined parameters.

For parameters defined as arrays, the initializer shall be a `constant_param_arrayinit` expression which is a list of constant expressions containing only constant numbers and previously defined parameters within `{` and `}` delimiters.

Parameters represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values which are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement or in the *module_instance* statement. It is not legal to use hierarchical name referencing (from within the analog block ) to access external analog variables or parameters.





parameter_declaration ::=
**parameter** [opt_type] list_of_param_assignments **;**

opt_type ::=
**real**
| **integer**

list_of_param_assignments ::=
declarator_init {**,** declarator_init}

declarator_init ::=
*parameter*_identifier **=** constant_expression **{** opt_value_range **}**
| *parameter_array*_identifier range **=** constant_param_arrayinit **{** opt_value_range **}**

opt_value_range ::=
**from** value_range_specifier
| **exclude** value_range_specifier
| **exclude** *value*_constant_expression

value_range_specifier ::=
start_paren expression1 **:** expression2 end_paren

start_paren ::=
**[** | **(**

end_paren ::=
**]** | **)**

expression1 ::=
constant_expression | **-inf**

expression2 ::=
constant_expression | **inf**

constant_param_arrayinit ::=
**{** param_arrayinit_element_list **}**

param_arrayinit_element_list ::=
param_arrayinit_element {**,** param_arrayinit_element }

param_arrayinit_element ::=
constant_expression
| **{** *replicator*_constant_expression **{**constant_expression**} }**

*Syntax 3-2—Syntax for parameter declaration*

By nature, analog behavioral specifications are characterized more extensively in terms of parameters than their digital counterparts. There are three fundamental extensions to the parameter declarations defined in *IEEE 1364-1995 Verilog HDL*:

- An optional type for the parameter can be specified in Verilog-AMS HDL. In *IEEE 1364-1995 Verilog HDL*, the type of a parameter defaults to the type of the default expression.

- A range of permissible values can be defined for each parameter. In *IEEE 1364-1995 Verilog HDL*, this check had to be done in user's model or was left as an implementation specific detail.
- Parameter arrays of basic integer and real data types can be specified.

### 3.2.1 Type specification

The parameter declaration can contain an optional *type* specification. In this sense, the **parameter** keyword acts more as a type qualifier than a type specifier. A default value for the parameter shall be specified.

*Examples:*

The following examples illustrate this concept:

```
parameter real slew_rate = 1e-3 ;
parameter integer size = 16 ;
```

If the *type* of a parameter is not specified, it is derived from the type of the value of the constant expression as in *IEEE 1364-1995 Verilog HDL*.

If the type of the parameter is specified, and the value assigned to the parameter conflicts with the type of the parameter, the value is coerced to the type of the parameter (see 4.1.1.1).

*Example:*

```
parameter real size = 10 ;
```

Here, `size` is coerced to `10.0`.

### 3.2.2 Value range specification

A parameter declaration can contain optional specifications of the permissible range of the values of a parameter. More than one range can be specified for inclusion or exclusion of values as legal values for the parameter.

The use of brackets, **[** and **]**, indicate inclusion of the end points in the valid range. The use of parenthesis, **(** and **)**, indicate exclusion of the end points from the valid range. It is possible to include one end point and not the other using **[ )** and **( ]**. The first expression in the range shall be numerically smaller than the second expression in the range.

*Examples:*

```
parameter real neg_rail = -15 from [-50:0) ;
parameter integer pos_rail = 15 from (0:50) ;
parameter real gain = 1 from [1:1000] ;
```

Here, the default value for `neg_rail` is `-15` and it is only allowed to acquire values within the range of `-50 <= neg_rail < 0`. Similarly, the default value for parameter `pos_rail` is `15` and it is only allowed to acquire values within the range of





`0 < pos_rail < 50`. And, the default value for `gain` is `1` and it is allowed to acquire values within the range of `1<= gain <= 1000`.

The keyword **inf** can be used to indicate infinity. If preceded by a negative sign, it indicates negative infinity.

*Example:*

```
parameter real val3=0 from [0:inf) exclude (10:20) exclude (30:40];
```

A single value can be excluded from the possible valid values for a parameter.

*Example:*

```
parameter real res = 1.0 exclude 0 ;
```

Here, the value of a parameter is checked against the specified range.

***Issue #38: For which value does the range check apply?***

Range checking applies to the value of the parameter for the instance and not against the default values specified in the device. It shall be an error only if the value of the parameter is out of range during simulation.

### 3.2.3 Parameter arrays

Verilog-AMS HDL includes behavioral extensions which utilize arrays. It requires these arrays be initialized in their definitions and allow overriding their values as with other parameter types. The declaration of arrays of parameters is in a similar manner to those of parameters and register arrays of reals and integers in *IEEE 1364-1995 Verilog HDL*.

Parameter arrays have the following restrictions. Failure to follow these restrictions shall result in an error.

- A type of a parameter array shall be given in the declaration.
- An array assigned to an instance of a module shall be of the exact size of the array bounds of that instance.
- If the array size is changed via a parameter assignment, the parameter array shall be assigned an array of the new size from the same module as the parameter assignment that changed the parameter array size.

*Example:*

```
parameter real poles[0:3] = { 1.0, 3.198, 4.554, 2.00 };
```

## 3.3 Genvars

*Genvars* are integer-valued variables which compose static expressions for instantiating structure behaviorally such as accessing analog signals within behavioral looping constructs. The syntax for declaring genvar variables is shown in Syntax 3-3.

```
genvar_declaration ::=
    genvar list_of_genvar_identifiers ;

list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }
```

*Syntax 3-3—Syntax for genvar declaration*

The static nature of genvar variables is derived from the limitations upon the contexts in which their values can be assigned.

*Examples:*

```
genvar i;
analog begin
    ...
    for (i = 0; i < 8; i = i + 1) begin
        V(out[i]) <+ transition(value[i], td, tr);
    end
    ...
end
```

The genvar variable `i` can only be assigned within the for-loop control. Assignments to the genvar variable `i` can consist only of expressions of static values, e.g., parameters, literals, and other genvar variables.

## 3.4 Net_discipline

In addition to the data types supported by *IEEE 1364-1995 Verilog HDL*, an additional data type, *net_discipline*, is introduced in Verilog-AMS HDL for continuous time and mixed signal simulation. *net_discipline* is used to declare analog nets ,as well as declaring the domains of digital nets and regs.

A signal can be digital, analog, or mixed, and is a hierarchical collection of nets which are contiguous (because of port connections). For analog and mixed-signals, a single node is associated with all continuous nets segments of the signal. The fundamental characteristic of analog and mixed signals is the values of the associated node are determined by the simultaneous solution of equations defined by the instances connected to the node using Kirchhoff's conservation laws. In general, a node represents a point of physical connections between nets of continuous-time description and it obeys conservation-law semantics.

A net is characterized by the discipline it follows. For example, all low-voltage nets have certain common characteristics, all mechanical nets have certain common characteristics, etc. Therefore, a *net* is always declared as a type of discipline. In this sense, a discipline is a user-defined type for declaring a net.





A *discipline* is characterized by the domain and the attributes defined in the *natures* for `potential` and `flow`.

### 3.4.1 Natures

A *nature* is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes can be declared and assigned constant values in a nature.

The nature declarations are at the same level as discipline and module declarations in the source text. That is, natures are declared at the top level and nature declarations do not nest inside other nature declarations, discipline declarations, or module declarations.

The syntax for defining a nature is shown in Syntax 3-4.

```
nature_declaration ::=
    nature nature_name
        [ nature_descriptions ]
    endnature

nature_name ::=
      nature_identifier
    | nature_identifier : parent_identifier

parent_identifier ::=
      nature_identifier
    | discipline_identifier.flow
    | discipline_identifier.potential

nature_descriptions ::=
    nature_description { nature_description }

nature_description ::=
    attribute = constant_expression ;

attribute ::=
      abstol
    | access
    | ddt_nature
    | idt_nature
    | units
    | attribute_identifier
```

*Syntax 3-4—Syntax for nature declaration*

A nature shall be defined between the keywords **nature** and **endnature**. Each nature definition shall have a unique identifier as the name of the nature and shall include all the required attributes specified in 3.4.1.2.

*Examples:*

```
nature current
    units = "A" ;
    access = I ;
    idt_nature = charge ;
    abstol = 1u ;
endnature

nature voltage
    units = "V" ;
    access = V;
    abstol = 1u ;
endnature
```

#### 3.4.1.1 Derived natures

A nature can be derived from an already declared nature. This allows the new nature to have the same attributes as the attributes of the existing nature. The new nature is called a *derived nature* and the existing nature is called a *parent nature.* If a nature is not derived from any other nature, it is called a *base nature*.

In order to derive a new nature from an existing nature, the new nature name shall be followed by a colon (:) and the name of the parent nature in the nature definition.

A derived nature can declare additional attributes or override attribute values of the parent nature, with certain restrictions (as outlined in 3.4.1.2) for the predefined attributes.

The attributes of the derived nature are accessed in the same manner as accessing attributes of any other nature.

*Examples:*

```
nature Ttl_curr
    units = "A" ;
    access = I ;
    abstol = 1u ;
endnature

// An alias

nature Ttl_net_curr : Ttl_curr
endnature

nature New_curr : Ttl_curr// derived, but different
    abstol = 1m ;// modified for this nature
```

***Issue #84a: use maxval instead of max since max is a keyword***

```
    max = 12.3 ;// new attribute for this nature
    maxval = 12.3 ;// new attribute for this nature
endnature
```





#### 3.4.1.2 Attributes

Attributes define the value of certain quantities which characterize the nature. There are five predefined attributes — **abstol**, **access**, **idt_nature**, **ddt_nature**, and **units**. In addition, user-defined attributes can be defined in a nature (see 3.4.1.3). Attribute declaration assigns a constant expression to the attribute name, as shown in the example in 3.4.1.1.

**abstol**

The **abstol** attribute provides a tolerance measure (metric) for convergence of potential or flow calculations. It specifies the maximum negligible for signals associated with the nature.

This attribute is required for all base natures. It is legal for a derived nature to change **abstol**, but if left unspecified it shall inherit the `abstol` from its parent nature. The constant expression assigned to it shall evaluate to a real value.

**access**

The **access** attribute identifies the name for the access function. When the nature is used to bind a potential, the name is used as an access function for the potential; when the nature is used to bind a flow, the name is used as an access function for the flow. The usage of access functions is described further in 4.3.

This attribute is required for all base natures. It is illegal for a derived nature to change the access attribute; the derived nature always inherits the access attribute of its parent nature. If specified, the constant expression assigned to it shall be an identifier (by name, not as a string).

**idt_nature**

The **idt_nature** attribute provides a relationship between a nature and the nature representing its time integral.

**idt_nature** can be used to reduce the need to specified tolerances on the **idt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, which eliminates the need to give a tolerance with the operator.

If specified, the constant expression assigned to `idt_nature` shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a `nature` to be self-referencing with respect to its `idt_nature` attribute. In other words, the value of `idt_nature` can be the `nature` that the attribute itself is associated with.

The **idt_nature** attribute is optional; the default value is the `nature` itself. While it is possible to override the parent's value of `idt_nature` using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its `idt_nature`.

**ddt_nature**

The **ddt_nature** attribute provides a relationship between a nature and the nature representing its time derivative.

**ddt_nature** can be used to reduce the need to specified tolerances on the **ddt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, eliminating the need to give a tolerance with the operator.

If specified, the constant expression assigned to `ddt_nature` shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a `nature` to be self-referencing with respect to its `ddt_nature` attribute. In other words, the value of `ddt_nature` can be the `nature` that the attribute itself is associated with.

The **ddt_nature** attribute is optional; the default value is the `nature` itself. While it is possible to override the parent's value of `ddt_nature` using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its `ddt_nature`.

**units**

The **units** attribute provides a binding between the value of the access function and the units for that value. The **units** field is provided so simulators can annotate the continuous signals with their units and is also used in the *net compatibility rule check*.

This attribute is required for all base natures. It is illegal for a derived nature to define or change the `units`; the derived nature always inherits its parent nature `units`. If specified, the constant expression assigned to it shall be a string.

#### 3.4.1.3 User-defined attributes

In addition to the predefined attributes listed above, a `nature` can specify other attributes which can be useful for analog modeling. Typical examples include certain maximum and minimum values to define a valid range.

A user-defined attribute can be declared in the same manner as any predefined attribute. The name of the attribute shall be unique in the nature being defined and the value being assigned to the attribute shall be constant.

### 3.4.2 Disciplines

A *discipline* description consists of specifying a `domain` type and binding any *natures* to `potential` or `flow`.

The syntax for declaring a discipline is shown in Syntax 3-5.





```
discipline_declaration ::=
    discipline discipline_identifier
        [ discipline_descriptions ]
    enddiscipline

discipline_descriptions ::=
    discipline_description { discipline_description }

discipline_description ::=
      nature_binding
    | domain_binding
    | attr_override

nature_binding ::=
    pot_or_flow nature_identifier ;

pot_or_flow ::=
      potential
    | flow

domain_binding ::=
      domain continuous ;
    | domain discrete ;

attr_override ::=
    pot_or_flow . attribute_identifier = constant_expression ;
```

*Syntax 3-5—Syntax for discipline declaration*

A *discipline* shall be defined between the keywords **discipline** and **enddiscipline**. Each discipline shall have a unique identifier as the name of the discipline.

The discipline declarations are at the same level as *nature* and *module* declarations in the source text. That is, disciplines are declared at the top level and discipline declarations do not nest inside other discipline declarations, nature declarations, or module declarations.

#### 3.4.2.1 Nature binding

Each discipline can bind a *nature* to its `potential` and `flow`.

Only the name of the nature is specified in the discipline. The nature binding for potential is specified using the keyword **potential**. The nature binding for flow is specified using the keyword **flow**.

The access function defined in the nature bound to potential is used in the model to describe the signal-flow which obeys Kirchhoff's Potential Law (KPL). This access function is called the *potential access function*.

The access function defined in the nature bound to flow is used in the model to describe a quantity which obeys Kirchhoff's Flow Law (KFL). This access function is called the *flow access function*.

Disciplines with two natures are called *conservative disciplines* and the nets associated with conservative disciplines are called *conservative nets*. Conservative disciplines shall not have the same *nature* specified for both the `potential` and the `flow`. Disciplines with a single potential nature are called *signal-flow disciplines* and the nets with signal-flow disciplines are called *signal-flow nets*. Only a `potential` nature is allowed to be specified for a signal-flow discipline, as shown in the following examples.

*Examples:*

Conservative discipline

```
discipline electrical
   potential Voltage ;
   flow Current ;
enddiscipline
```

Signal-flow disciplines

```
discipline voltage
   potential Voltage ;
enddiscipline

discipline current
   potential Current;
enddiscipline
```

### 3.4.3 Multi-disciplinary example

Disciplines in Verilog-AMS HDL allow designs of multi-disciplinaries to be easily defined and simulated. Disciplines can be used to allow unique tolerances based on the size of the signals and outputs displayed in the actual units of the discipline. This example shows how an application spanning multiple disciplines can be modeled in Verilog-AMS HDL. It models a DC-motor driven by a voltage source.

```
module motorckt;
   parameter real freq=100;
   ground gnd;

   electrical drive;
   rotational shaft;

   motor m1 (drive, gnd, shaft);
   vsource #(.freq(freq), .ampl(1.0)) v1 (drive, gnd);

endmodule

// vp:   positive terminal [V,A] vn:negative terminal [V,A]
// shaft:motor shaft [rad,Nm]
// INSTANCE parameters
// Km = motor constant [Vs/rad]Kf = flux constant  [Nm/A]
// j  = inertia factor [Nms^2/rad] D= drag (friction)  [Nms/rad]
// Rm = motor resistance [Ohms] Lm = motor inductance [H]
// A model of a DC motor driving a shaft
```





```
module motor(vp, vn, shaft);
   inout vp, vn, shaft;
   electrical vp, vn ;
   rotational shaft ;

   parameter real Km = 4.5, Kf = 6.2;
   parameter real j = .004, D = 0.1;
   parameter real Rm = 5.0, Lm = .02;

   analog begin
      V(vp, vn) <+ Km*Theta(shaft) + Rm*I(vp, vn) +
         ddt(Lm*I(vp, vn));
      Tau(shaft) <+ Kf*I(vp, vn) - D*Theta(shaft) -
         ddt(j*Theta(shaft));
      end
endmodule
```

#### 3.4.3.1 Domain binding

Analog signal values are represented in continuous time, whereas digital signal values are represented in discrete time. The **domain** attribute of the discipline stores this property of the signal. It takes two possible values, **discrete** or **continuous.** Signals with continuous-time domains are real valued. Signals with discrete-time domains can either be binary (`0`, `1`, `X`, or `Z`), integer or real values.

*Examples:*

```
discipline electrical
   domain continuous;
   potential Voltage;
   flow Current;
enddiscipline

discipline logic
   domain discrete ;
enddiscipline
```

The **domain** attribute is optional. The default value for domain is **continuous** for non-empty disciplines, e.g., those which specify nature bindings.

#### 3.4.3.2 Empty disciplines

It is possible to define a discipline with no nature bindings. These are known as e*mpty disciplines* and they can be used in structural descriptions to let the components connected to a net determine which natures are to be used for the net.

Such disciplines may have a domain binding or they may be domain-less, thus allowing the domain to be determined by the connectivity of the net (see 8.4).

*Examples:*

***Issue #41: We do not need nuetral disciplines if wire is already neutral.***

```
discipline neutral
enddiscipline

discipline interconnect
    domain continuous;
enddiscipline
```

#### 3.4.3.3 Discipline of wires and undeclared nets

It is possible for a module to have nets where there are no discipline declarations. If such a net appears bound only to ports in module instantiations, it may have no declaration at all or may be declared to have a wire type such as `wire`, `tri`, `wand`, `wor`, etc. If it is referenced in behavioral code, then it must have a wire type.

In these cases, the net shall be treated as having an empty discipline. If the net is referenced in behavioral code or if its net type is other than `wire`, then it shall be treated as having empty discipline with a domain binding of `discrete`, otherwise it shall be treated as having empty discipline with no domain binding.

This allows *netlists* (modules which describe connectivity only, with no behavior) which use wire as an interconnect to be valid in both *IEEE 1364-1995 Verilog HDL* and Verilog-AMS HDL. The domain shall be determined by the connectivity of the net (see 8.4).

#### 3.4.3.4 Overriding nature attributes from discipline

A discipline can override the value of the bound nature for the pre-defined attributes (except as restricted by 3.4.1.2), as shown for the flow `ttl_curr` in the example below. To do so from a discipline declaration, the bound nature and attribute needs to be defined, as shown for the `abstol` value within the discipline `ttl` in the example below. The general form is shown as the *attr_override* terminal in Syntax 3-5: the keyword **flow** or **potential**, then the hierarchical separator **.** and the attribute name, and, finally, set all of this equal to (=) the new value (e.g., `flow.abstol = 10u`).

*Examples:*

```
nature ttl_curr
    units = "A" ;
    access = I ;
    abstol = 1u ;
endnature

nature ttl_volt
    units = "V" ;
    access = V;
    abstol = 100u ;
endnature
```





```
discipline ttl
    potential ttl_volt ;
    flow ttl_curr ;
    flow.abstol = 10u ;
enddiscipline
```

#### 3.4.3.5 Deriving natures from disciplines

A nature can be derived from the *nature* bound to the `potential` or `flow` in a discipline. This allows the new nature to have the same attributes as the attributes for the nature bound to the `potential` or the `flow` of the discipline.

If the nature binding to the `potential` or the `flow` of a discipline changes, the new nature shall automatically inherit the attributes for the changed nature.

In order to derive a new nature from flow or potential of a discipline, the nature declaration shall also include the discipline name followed by the hierarchical separator (.) and the keyword **flow** or **potential**, as shown for `Ttl_net_curr` in the example below.

A nature derived from the `flow` or `potential` of a discipline can declare additional attributes or override values of the attributes already declared.

*Examples:*

```
nature Ttl_net_curr : Ttl.flow // from the example in 3.4.3.4
endnature     // abstol = 10u as modified in ttl

nature Ttl_net_volt : Ttl.potential // from the example in 3.4.3.4
    abstol = 1m ;// modified for this nature
    max = 12.3 ;// new attribute for this nature

    maxval = 12.3 ;// new attribute for this nature
endnature
```

### 3.4.4 Net discipline declaration

Each *net_discipline* declaration associates nets and regs with an already declared discipline. Syntax 3-6 shows how to declare disciplines of nets and regs.

```
net_discipline_declaration ::=
    discipline_identifier [ range ] list_of_nets ;
range ::=
    [ constant_expression : constant_expression ]

list_of_nets ::=
      net_identifier [ range ]
    | net_identifier [ range ] , list_of_nets
```

*Syntax 3-6—Syntax for net discipline declaration*

If a range is specified for a net, the net is called a *vector net*; otherwise it is called a *scalar net*. A vector net is also called an *bus*.

*Examples:*

```
electrical [MSB:LSB] n1 ; // MSB and LSB are parameters
voltage [5:0] n2, n3 ;
magnetic inductor ;
logic [10:1] connector1 ;
```

Nets represent the abstraction of information about signals. As with ports, nets represent component interconnections. Nets declared in the module interface define the ports to the module (see 7.3.4).

A net used for modeling a conservative system shall have a discipline with both access functions (`potential` and `flow`) defined. When modeling a signal-flow system, the discipline of a net can have only `potential` access functions. When modeling a discrete system, the discipline of a net can only have a `domain` of `discrete` defined.

Nets declared with an empty discipline do not have declared natures, so such nets can not be used in analog behavioral descriptions (because the access functions are not known). However, such nets can be used in structural descriptions, where they inherit the natures from the ports of the instances of modules that connect to them.

### 3.4.5 Ground declaration

Each ground declaration is associated with an already declared net of continuous discipline. The node associated with the net will be the global reference node in the circuit. If used in behavioral code, the net shall be used in only the differential source and probe forms, e.g., `V(gnd)` is not allowed. The net must be assigned a continuous discipline to be declared ground.

Syntax 3-7 shows the syntax used for declaring the global reference node (*ground*).

```
ground_declaration ::=
    ground [ range ] list_of_nets;
```

*Syntax 3-7—Syntax for declaring ground*

*Examples:*

```
module loadedsrc(in, out);
   input in;
   output out;
   electrical in, out;
   electrical gnd;
   ground gnd;
   parameter real srcval = 5.0;
   resistor #(.r(10K)) r1(out,gnd);
   analog begin
      V(out) <+ V(in,gnd)*2;
   end
endmodule
```





### 3.4.6 Implicit nets

Nets can be used in a structural descriptions without being declared. In this case, the net is implicitly declared to be a scalar net with the empty discipline and undefined domain.

*Examples:*

```
module top(i1, i2, o1, o2, o3);
   input i1, i2;
   output o1, o2, o3;
   electrical i1, i2, o1, o2, o3;

   // ab1, ab2, cb1, cb2 are implicit nets, not declared
   blk_a a1( i1, ab1 );
   blk_a a2( i2, ab2 );
   blk_b b1( ab1, cb1 );
   blk_b b2( ab2, cb2 );
   blk_c c1( o1, o2, o3, cb1, cb2);
endmodule
```

## 3.5 Real net declarations

The **wreal**, or real net data type, represents a real-valued physical connection between structural entities. A wreal net shall not store its value. A wreal net can be used for real-valued nets which are driven by a single driver, such as a continuous assignment. If no driver is connected to a wreal net, its value shall be zero (`0.0`).

wreal nets can only be connected to other wreals or real expressions. They cannot be connected to any other wires, even explicitly declared 64-bit wires.

Syntax 3-8 shows the syntax for declaring digital nets.

```
digital_net_declaration ::=
        digital_net_declaration
      | wreal [ list_of_identifiers ] ;
```

*Syntax 3-8—Syntax for declaring digital nets*

*Examples:*

```
module foo(in, out);
  input in;
  output out;
  wreal in;
  electrical out;
  analog begin
    V(out) <+ in;
  end
endmodule

module top();
  real stim;
  electrical load;
  foo f1(stim, load);
  dut d1(load, out);
  always begin
    #1 stim = stim + 0.1;
  end
endmodule
```

## 3.6 Default discipline

Verilog-AMS HDL supports the `**default_discipline** compiler directive. This directive specifies a default discipline to be applied to any net which does not have an explicit discipline declaration. Its syntax is shown in Syntax 3-9.

default_discipline_directive ::=
`**default_discipline** [*discipline*_identifier [qualifier] [scope] ]

qualifier ::=
**integer** | **real** | **reg** | **wire** | **tri** | **wand** | **triand** | **wor** | **trior**
| **trireg** | **tri0** | **tri1** | **supply0** | **supply1**

scope ::=
*instance*_identifier

*Syntax 3-9—Syntax for setting default discipline compiler directive*

The scope of this directive is similar to the scope of the `**define** compiler directive. The default discipline is applied to all signals without a discipline declaration which appear in the text stream following the use of the `‘default_discipline` directive until either the end of the text stream or another `‘default_discipline` directive with the same combination of qualifier and scope (if applicable) is found in the subsequent text. Therefore, more than one `‘default_discipline` directive can be in force simultaneously, provided each differs in scope, qualifier, or both.





If this directive is used without a discipline name, it turns off all currently active default disciplines without setting a new default discipline. Subsequent signals without a discipline shall be associated with the empty discipline.

*Examples:*

```
`default_discipline logic

module behavnand(in1, in2, out);
   input in1, in2;
   output out;
   reg out;

   always begin
      out = ~(in1 && in2);
   end

endmodule
```

This example illustrates the usage of the `‘default_discipline` directive. The nets `in1`, `in2`, and `out` all have discipline `logic` by default.

There is a precedence of compiler directives; the more specific directives have higher precedence over general directives.

## 3.7 Discipline precedence

While a net itself can be declared only in the module to which it belongs, the discipline of the net can be specified in a number of ways.

- The discipline name can appear in the declaration of the net.
- The discipline name can be used in a declaration which makes an out of context reference to the net from another module.
- The discipline name can be used in a `‘default_discipline` compiler directive.

Discipline conflicts can arise if more than one of these methods is applied to the same net. Discipline conflicts shall be resolved using the following order of precedence:

1. A declaration from a module other than the module to which the net belongs using an out of module reference, e.g.,

   ```
   module example1;
      electrical example2.net;
   endmodule
   ```

2. The local declaration of the net in the module to which it belongs, e.g.,

   ```
   module example2;
      electrical net;
   endmodule
   ```

3. `‘default_discipline` with qualifier and scope, e.g.,

   ```
   `default_discipline electrical trireg example1.instance5;
   ```

4. `‘default_discipline` with scope only, e.g.,

   ```
   `default_discipline electrical example1.instance5;
   ```

5. `‘default_discipline` with qualifier only, e.g.,

   ```
   `default_discipline electrical trireg;
   ```

6. `‘default_discipline` without qualifier or scope, e.g.,

   ```
   `default_discipline electrical;
   ```

It is not legal to have two different disciplines at the same level of precedence for the same net.

## 3.8 Net compatibility

***Issue #86: The compatibility rules are not properly defined in this section. So, a proposal is included to reword this section.***

~~Certain operations can be done on nets only if the two (or more) nets are compatible. For example, if an access function has two nets as arguments, they shall be compatible. The nets are considered compatible if their respective disciplines are compatible. The following rules apply in deciding whether two disciplines are compatible:~~

~~*Self Rule*: A discipline is compatible with itself.~~

~~*Nature Compatibility Rule*: Two natures are compatible if they both exist and are derived from the same base nature.~~

~~*Nature Incompatibility Rule*: Two natures are not incompatible if they are compatible or if one or both do not exist.~~

~~*Units Value Rule*: All compatible natures shall have the same value for the attribute `units`. Since a child nature cannot override a base nature’s unit, this rule is always maintained.~~

~~*Potential Compatibility Rule*: If the natures of the two potentials are compatible and the natures of the two flows are not incompatible, then the two disciplines are considered compatible.~~

~~*Flow Compatibility Rule*: If the natures of the two flows are compatible and the natures of the two potential are not incompatible, then the two disciplines are considered compatible.~~

~~*Empty Discipline Rule*: An empty discipline is compatible with all disciplines of the same domain.~~





~~*Discrete Domain Rule*: Disciplines with the `discrete` domain attribute with the same signal value type (i.e., `bit`, `real`, and `integer`) are compatible with each other.~~

~~*Domain Incompatibility Rule*: Disciplines with different domain attributes are incompatible with each other.~~

~~*Signal Connection Rule*: It shall be an error to connect two ports or nets of different domains unless there is a `connect` statement (see 8.4) defined between these disciplines. It shall be an error to connect two ports or nets of the same domain with incompatible disciplines.~~

Certain operations can be done on nets only if the two (or more) nets are compatible. For example, if an access function has two nets as arguments, they must be compatible. The following rules shall apply to determine the compatibility of two (or more) nets:

*Discrete Domain Rule:* Digital nets with the same signal value type (ie. bit, real, integer) are compatible with each other if their disciplines are compatible, ie. the discipline has a discrete domain or is empty.

*Signal Domain Rule:* It shall be an error to connect two ports or nets of different domains unless there is a connect statement (see 8.4) defined between the disciplines of the nets or ports.

*Signal Connection Rule:* It shall be an error to connect two ports or nets of the same domain with incompatible disciplines.

### 3.8.1 Discipline and Nature Compatibility

The following rules shall apply to determine discipline compatibility:

*Self Rule (Discipline):* A discipline is compatible with itself.

*Empty Discipline Rule:* An empty discipline is compatible with all other disciplines, regardless of domain.

*Domain Incompatibility Rule:* Disciplines with different domain attributes are incompatible.

*Potential Incompatibility Rule:* Disciplines with incompatible potential natures are incompatible.

*Flow Incompatibility Rule:* Disciplines with incompatible flow natures are incompatible.

The following rules shall apply to determine nature compatibility:

*Self Rule (Nature):* A nature is compatible with itself.

*Non-Existent Binding Rule:* A nature is compatible with a non-existent discipline binding.

*Base Nature Rule:* A derived nature is compatible with its base nature.

*Derived Nature Rule:* Two natures are compatible if they are derived from the same base nature.

*Units Value Rule:* Two natures are compatible if they have the same value for the units attribute.

The following examples illustrates these rules.

***Issue #88: Correction of inconsistencies in the examples, and the explanation following the example. Also, references to derived disciplines will be dropped as the current BNF does not support this syntax.***

*Examples:*





```
nature Voltage
    access = V;
    units = "V";
    abstol = 1u;
endnature

nature Current
    access = I;
    units = "A";
    abstol = 1p;
endnature

~~nature lowvoltage: Voltage~~
~~    abstol = 1;~~
~~endnature~~

nature highvoltage: Voltage
    abstol = 1;
endnature

discipline electrical
    potential Voltage;
    nature Current;
endnature

~~discipline highvolt :electrical~~
~~    potential.abstol=1;~~
~~enddiscipline~~

discipline highvolt
    potential highvoltage;
    nature Current;
endnature

discipline sig_flow_v
    potential Voltage;
enddiscipline

discipline sig_flow_i
    potential Current;
enddiscipline
```

```
nature Position
    access = X;
    units = "m";
    abstol = 1u;
endnature

nature Force
    access = F;
    units = "N";
    abstol = 1n;
endnature

discipline rotational
    potential Position;
    flow Force;
enddiscipline

discipline sig_flow_x
    potential Position;
enddiscipline

discipline sig_flow_f
    potential Force;
enddiscipline

discipline empty
enddiscipline

discipline logic
    domain discrete;
enddiscipline

discipline continous_elec
    domain continous;
    potential Voltage;
    nature Current;
endnature
```

The following compatibility observations can be made from the above example:

- `Voltage` and ~~`lowvoltage`~~ `highvoltage` are compatible natures because they both exists and are derived from the same base natures.
- `electrical` and ~~`highv`~~ `highvolt` are compatible disciplines because the natures for both potential and flow exist and are derived from the same base natures.

- `electrical` and `sig_flow_v` are compatible disciplines because the nature for potential is same for both disciplines and the nature for flow does not exist in `sig_flow_v`.
- `electrical` and ~~`mechanical`~~ `rotational` are incompatible disciplines because the natures for both potential and flow are not derived from the same base natures.
- `electrical` and `sig_flow_x` are incompatible disciplines because the nature for both potentials are not derived from the same base nature.
- An *empty discipline* is compatible with all other disciplines of the same domain (determined independently) because it does not have a potential or a flow nature. Without natures, there can be no conflicting natures.
- `electrical` and `logic` are incompatible disciplines because the domains are different. ~~A connect statement must be used to connect these nets and or ports together.~~ A connect statement must be used to connect nets or ports of these disciplines together.
- `electrical` and `continous_elec` are compatible disciplines because the default domain for discipline `electrical` is `continous` and the specified natures for potential and flow are the same.

## 3.9 Branches

A *branch* is a path between two nets. If both nets are conservative, then the branch is a *conservative branch* and it defines a branch potential and a branch flow. If one net is a signal-flow net, then the branch is a *signal-flow branch* and it defines either a branch potential or a branch flow, but not both.

Each branch declaration is associated with two nets from which it derives a discipline. These nets are referred to as the *branch terminals*. Only one net need be specified, in which case the second net defaults to `ground` and the discipline for the branch is derived from the specified net. The disciplines for the specified nets shall be compatible (see 3.8).

The syntax for declaring branches is shown in Syntax 3-10.





```
branch_declaration ::=
    branch list_of_branches ;

list_of_branches ::=
    terminals list_of_branch_identifiers

terminals ::=
     ( net_or_port_scalar_expression )
   | ( net_or_port_scalar_expression , net_or_port_scalar_expression )

list_of_branch_identifiers ::=
     branch_identifier [ range ]
   | branch_identifier [ range ] , list_of_branch_identifiers
```

*Syntax 3-10—Syntax for branch declaration*

If one of the terminals of a branch is a vector net, then the other terminal shall either be a scalar net or a vector net of the same size. In the latter case, the branch is referred to as a *vector branch*. When both terminals are vectors, the scalar branches that make up the vector branch connect to the corresponding scalar nets of the vector terminals, as shown in Figure 3-1.

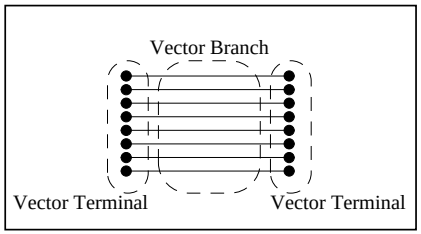


**Figure 3-1 Two vector terminals**

When one terminal is a vector and the other is a scalar, a singular scalar branch connects to each scalar net in the vector terminal and each terminal of the vector branch connects to the scalar terminal, as shown in Figure 3-2.

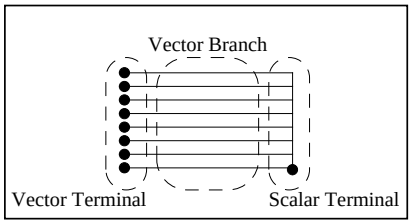


**Figure 3-2 One vector and one scalar terminal**

## 3.10 Namespace

The following subsections define the namespace.

### 3.10.1 Nature and discipline

Natures and disciplines are defined at the same level of scope as modules. Thus, identifiers defined as natures or disciplines have a global scope, which allows nets to be declared inside any module in the same manner as an instance of a module.

### 3.10.2 Access functions

Each access function name, defined before a module is parsed, is automatically added to that module's name space unless there is another identifier defined with the same name as the access function in that module's name space. Furthermore, the access function of each base nature shall be unique.

### 3.10.3 Net

The scope rules for net identifiers are the same as the scope rules for any other identifier declarations, except nets can not be declared anywhere other than in the port of a module or in the module itself. A net can only be declared inside a module block; a net can not be declared local to a block.

Access functions are uniquely defined for each net based on the discipline of the net. Each access function is used with the name of the net as its argument and a net can only be accessed through its access functions.

The hierarchical reference character ( . ) can be used to reference a net across the module boundary according to the rules specified in *IEEE 1364-1995 Verilog HDL*.





### 3.10.4 Branch

The scope rules for branch identifiers are the same as the scope rules for net identifiers. A branch can only be declared inside a module block; there is no local declaration for a branch.

Access functions are uniquely defined for each branch based on the discipline of the branch. The access function is used with the name of the branch as its argument and a branch can only be accessed through its access functions.

The hierarchical reference character ( . ) can be used to reference a net across the module boundary according to the rules specified in *IEEE 1364-1995 Verilog HDL*.