Extending SystemVerilog Data Types to Nets

Revision 3

This document proposes a set of SystemVerilog extensions to allow data types to be used to declare nets. The *Overview* section provides some rationale for the extensions as well as a brief summary. The *Data Objects and Data Types* section presents some important concepts. These two sections are not intended to be included in the LRM. The formal proposal itself is covered in the *Proposed SystemVerilog LRM Changes* section. An additional proposal to extend the variable declaration syntax to allow the keyword **var** is discussed in *Proposed "var" Extension*.

Overview

SystemVerilog extended Verilog by adding powerful new data types and operators that can be used to declare and manipulate parameters and variables. Extensions like packed structs provide a very convenient abstraction for manipulating an object that is really just a bit vector.

SystemVerilog did not extend these new data types to nets. However, with the addition of continuous assignments to variables, hardware designers can use the extended data types with variables to model many common network behaviors. Users would like to have these convenient abstractions for nets too, because other common network behaviors — bidirectionality, multiple driver resolution, and delays — cannot be modeled with variables.

We propose to extend SystemVerilog by making a subset of the new data types available for nets too. In this first IEEE revision of SystemVerilog, we would like to allow a net or port to have any fixed-size data type that is based on four-state logic. You can use new SystemVerilog data types to declare parameters and variables, and by extension you can use new data types to declare nets too. For example:

```
wire w0; // Verilog-style: 1-bit wire, defaults to logic data type
wire logic w1; // 1-bit wire with explicit logic data type
wire logic [7:0] w2;
wire logic signed w3;
wire logic signed [7:0] w4;
wire logic signed [7:0] w5 [15:0];

trireg (large) logic #(0,0,0) cap1;

typedef struct packed { logic ecc; logic [7:0] data; } MemLoc;
wire MemLoc memsig;
```

```
typedef enum logic [2:0]
    { MOVop, SUBop, ANDop, ADDop = 4, ORop, LDop, XORop } opcodeT;
wire opcodeT opcode;

wire enum bit [1:0] { CLEAR, WARNING, ERROR } status;
```

The extensions to net data types are syntactically and semantically backward compatible with existing Verilog code. The extensions recommended at this time are limited to supporting four-state data types with nets. The recommended syntax and semantics have been carefully planned to allow future extensions, such as support for nets with two-state data types and real data types. These additional extensions are not included at this time because of schedule constraints.

Data Objects and Data Types

You can look at Verilog data objects as having two primary characteristics.

One is the "kind" of the object (i.e., variable vs. parameter vs. net) and the other is the "data type" of the object (integer vs. real vs. scalar bit, etc).

Roughly, the object kind indicates what you can do with the object. Only parameters can be modified with defparam statements, only variables can be assigned by procedural assignments,only nets have values that are resolved from their drivers, etc.

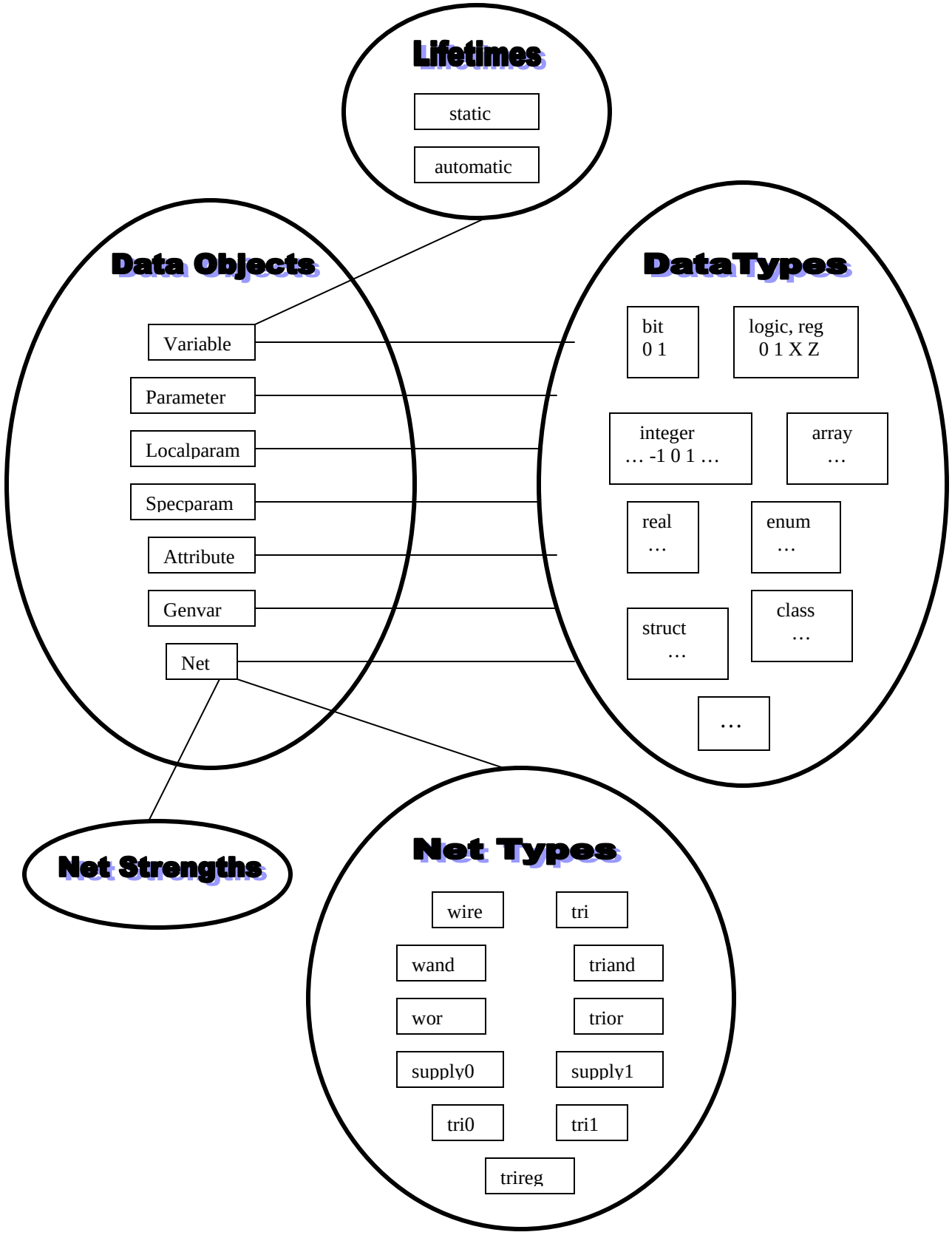
Roughly, the data type of an object indicates the values the object can take on. A data object of type 'real' can take on the value 3.14, an object of a bit vector type can take on the value 4'b0xz1, etc.

These two characteristics of a data object are largely orthogonal. As examples, a variable can be of any data type, and a bit vector can be the data type of any kind of data object.

Certain kinds of data objects have additional characteristics. For example, a net has a "net type", such as wire or trireg, that determines how its value is computed.

Diagram 1 below illustrates these concepts. This diagram shows the relationships between a data object and its significant properties. A data object is a construct that has a name and a value associated with it. Thus, an important characteristic of a data object is the set of values that it can have — that is, its "data type". Certain kinds of data objects have additional properties of interest. For example, a variable is also characterized by its lifetime, and a net is also characterized by its net type.

Diagram 1: Data Object Characteristics



Proposed SystemVerilog LRM Changes

This section proposes a set of specific LRM changes to extend SystemVerilog to allow all fixed-size four-state data types on nets. The basis for this extension is covered in sections 3 (Data types), 5 (Data declarations), and 18 (Hierarchy). These core changes are proposed first, followed by a set of changes throughout the LRM to provide a consistent view of data types and data objects.

Annex A.2.1.3

CHANGE:

```
net_declaration ::=
   net_type_or_trireg [drive_strength|charge_strength] [vectored|scalared]
     [signing] {packed_dimensions} [delay3] list_of_net_decl_assignments;
```

TO:

```
net_declaration ::=
   net_type_or_trireg [drive_strength|charge_strength] [vectored|scalared]
     data_type_or_implicit [delay3] list_of_net_decl_assignments;
```

In Syntax 18-4 and Annex A.2.2.1

CHANGE:

```
port_type ::= [net_type_or_trireg] [signing] {packed_dimension}
```

TO:

```
port_type* ::= [net_type_or_trireg] [signing] {packed_dimension}  |
               [net_type_or_trireg] data_type
```

*When a port_type contains a data_type, it shall only be legal to omit the explicit net_type_or_trireg when declaring an inout port.

3.1 Introduction

CHANGE:

Verilog-2001 has net data types, which can have 0, 1, X, or Z, plus 7 strengths, giving 120 values. It also has variable data types such as reg, which have 4 values 0, 1, X, Z. These are not just different data types, they are used differently. SystemVerilog adds another 4-value data type, called logic (see Sections 3.3.2 and 5.5).

TO:

Verilog-2001 has data objects that can take on values from a small number of predefined value systems: the set of four-state logic values, vectors and arrays of logic values, and the set of floating point values. SystemVerilog extends Verilog by introducing some of the data types that conventional programming languages provide, such as enumerations and structures.

In extending the type system, SystemVerilog makes a distinction between an object and its data type. A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects, or to define user-defined data types that are constructed from other data types.

The Verilog-2001 logic system is based on a set of four state values: 0, 1, X, and Z. Although this four-state logic is fundamental to the language, it does not have a name. SystemVerilog has given this primitive data type a name, logic. This new name can be used to declare objects and to construct other data types from the four-state data type.

The additional strength information associated with bits of a net is not considered part of the data type.

CHANGE:

Verilog-2001 provides arbitrary fixed length arithmetic using reg data types. The reg type can have bits at X or Z, however, and so are less efficient than an array of bits, because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a bit type which can only have bits with 0 or 1 values. See Section 3.3.2 on 2-state data types.

TO:

Verilog-2001 provides arbitrary fixed length arithmetic using 4-state logic. The 4-state type can have bits at X or Z, however, and so may be less efficient than an array of bits, because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a bit data type that can only have bits with 0 or 1 values. See Section 3.3.2 on 2-state data types.

5.1 Introduction

CHANGE:

There are several forms of data in SystemVerilog: literals (see Section 2), parameters (see Section 21), constants, variables, nets, and attributes (see Section 6)

TO:

There are several forms of data in SystemVerilog: literals (see Section 2), parameters (see Section 21), constants, variables, nets, and attributes (see Section 6). A data object is a named entity that has a data value associated with it, such as a parameter, a variable, or a net.

ADD TO END OF SECTION:

SystemVerilog extends the set of data types that are available for modeling Verilog storage and transmission elements. In addition to the Verilog-2001 data types, new predefined data types and user-defined data types can be used to declare constants, variables, and nets.

5.2 Data declaration syntax

ADD TO SYNTAX BOX:

```
net_declaration ::=
    net_type_or_trireg [drive_strength|charge_strength] [vectored|scalared]
      data_type_or_implicit [delay3] list_of_net_decl_assignments;
```

New section "5.5 Nets", right after section 5.4 Variables

A net declaration begins with a net type that determines how the values of the nets in the declaration are resolved. The declaration can include optional information such as delay values and drive or charge strength.

Verilog-2001 restricts the data type of a net to a scalar, a bit vector, or an array of scalars or bit vectors. In SystemVerilog, any four-state data type can be used to declare a net. For example:

```
trireg (large) logic #(0,0,0) cap1;

typedef logic [31:0] addressT;
wire addressT w1;
```

```
wire struct packed { logic ecc; logic [7:0] data; } memsig;
```

If a data type is not specified in the net declaration then the data type of the net is logic.

Certain restrictions apply to the data type of a net. A valid data type for a net shall be one of the following:

1. A four-state integral type , including a packed array or packed struct

2. An unpacked array or unpacked struct, where each element has a valid data type for a net

The effect of this recursive definition is that a net is composed entirely of four-state bits, and is treated accordingly. There is no change to the Verilog-2005 network semantics. In addition to a signal value, each bit of a net shall have additional strength information. When bits of signals combine, the strength and value of the resulting signal shall be determined as in Verilog-2005 section 7.10.

There is no change in the treatment of the signed property across hierarchical boundaries.

A lexical restriction applies to the use of the reg keyword in a net or port declaration. A Verilog net type keyword shall not be followed directly by the reg keyword. Thus, the following declarations are in error:

```
tri reg r;
inout wire reg p;
```

The reg keyword can be used in a net or port declaration if there are lexical elements between the net type keyword and the reg keyword.

18.1 Introduction

CHANGE:

An important enhancement in SystemVerilog is the ability to pass any data type through module ports, including nets, and all variable types including reals, arrays and structures.

TO:

An important enhancement in SystemVerilog is the ability to pass a value of any data type through module ports, using nets or variables. This includes reals, arrays and structures.

18.8 Port declarations

CHANGE:

With SystemVerilog, a port can be a declaration of a net, an interface, an event, or a variable of any type, including an array, a structure or a union.

TO:

With SystemVerilog, a port can be a declaration of an interface, an event, or a variable or net of any allowed data type, including an array, a structure or a union.

CHANGE:

If the first port direction but no type is specified, then the port type shall default to wire. This default type can be changed using the `default_nettype compiler directive, as in Verilog."

TO:

If the first port direction but no net type or data type is specified, then the port shall default to a net of net type wire. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

CHANGE:

For subsequent ports in the port list, if the type and direction are omitted, then both are inherited from the previous port. If only the direction is omitted, then it is inherited from the previous port. If only the type is omitted, it shall default to wire. This default type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// second port inherits its direction and type from previous port
module mh3 (input byte a, b);
   ...
endmodule
```

TO:

For subsequent ports in the port list, if the direction and the net type and data type are omitted, then the direction and any net type and data type are inherited from the previous port. If the direction is omitted, but a net type or data type is present, then the direction is inherited from the previous port. If the direction is present, but the net type and data types are omitted, then the port shall default to a net of net type wire. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// second port inherits its direction and data type
// from previous port
module mh3 (input byte a, b);
   ...
endmodule
```

For an inout port, if the net type is omitted, then the port shall default to a net of net type wire. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// the inout port defaults to a net of net type wire
module mh2 (inout integer a);
   ...
endmodule
```

18.11.3 Instantiation using implicit .name port connections

DELETE:

- A port connection between a net type and a variable type of the same bit length is a legitimate cast.

18.12 Port connection rules

CHANGE:

SystemVerilog extends Verilog port connections by making all variable data types available to pass through ports.

TO:

SystemVerilog extends Verilog port connections by making values of all data types on variables and nets available to pass through ports.

18.12.2 Port connection rules for nets

CHANGE:

If a port declaration has a wire type (which is the default), or any other net type,

TO:

If a port declaration has a net type, such as wire,

CHANGE:

- An output can be connected to a net type (or a concatenation of net types) or a compatible variable type (or a concatenation of variable types).

- An inout can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a variable type.

TO:

- An output can be connected to a net or variable (or a concatenation of nets or variables) of a compatible data type.

- An inout can be connected to a net (or a concatenation of nets) of a compatible data type, or left unconnected, but cannot be connected to a variable.

18.12.4 Compatible port types

CHANGE:

The same rules for assignment compatibility are used for compatible port types for ports declared as an input or an output variable, or for output ports connected to variables.

TO:

The same rules are used for compatible port types as for assignment compatibility.

3.6 chandle data type

CHANGE:

The chandle data type represents storage for pointers passed using the DPI Direct Programming Interface (see Section 27). The size of this type is platform dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

TO:

The chandle data type represents storage for pointers passed using the DPI Direct Programming Interface (see Section 27). The size of a value of this data type is platform

dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

3.7 String data type

CHANGE:

SystemVerilog includes a string data type, which is a variable size, dynamically allocated array of bytes. SystemVerilog also includes a number of special methods to work with strings.

TO:

SystemVerilog includes a string data type. The values of the string data type are dynamically allocated arrays of bytes of arbitrary size. SystemVerilog also includes a number of special methods to work with strings.

4.2 Packed and unpacked arrays

CHANGE:

Packed arrays can only be made of the single bit types (bit, logic, reg, wire, and the other net types) and recursively other packed arrays and packed structures.

TO:

Packed arrays can be made of only the single bit data types (bit, logic, reg) and recursively other packed arrays and packed structures.

New section 5.8.1, Matching types, approved for erratum 254

CHANGE:

3) An anonymous enum, struct, or union type matches itself among variables declared within the same declaration statement and no other types.

TO:

3) An anonymous enum, struct, or union type matches itself among data objects declared within the same declaration statement and no other data types.

CHANGE:

4) A typedef for an enum, struct, union, or class matches itself and the type of variables declared using that type within the scope of the type identifier.

TO:

4) A typedef for an enum, struct, union, or class matches itself and the type of data objects declared using that data type within the scope of the data type identifier.

5.8.1 Equivalent types

CHANGE:

3) An anonymous enum, struct, or union type is equivalent to itself among variables declared within the same declaration statement and no other types.

TO:

3) An anonymous enum, struct, or union type is equivalent to itself among data objects declared within the same declaration statement and no other data types.

CHANGE:

4) A typedef for an enum, unpacked struct, or unpacked union, or a class is equivalent to itself and variables declared using that type within the scope of the type identifier.

TO:

4) A typedef for an enum, unpacked struct, or unpacked union, or a class is equivalent to itself and to data objects that are declared using that data type within the scope of the data type identifier.

7.3 Assignment operators

CHANGE:

The semantics of such an assignment expression are those of a function which evaluates the right hand side, casts the right hand side to the left hand data type, stacks it, updates the left hand side and returns the stacked value. The type returned is the type of the left hand side data type. If the left hand side is a concatenation, the type returned shall be an unsigned integral value whose bit length is the sum of the length of its operands.

TO:

The semantics of such an assignment expression are those of a function that evaluates the right hand side, casts the right hand side to the left hand side data type, stacks it, updates the left hand side and returns the stacked value. The data type of the value that is returned is the data type of the left hand side. If the left hand side is a concatenation, then the data type of the value that is returned shall be an unsigned integral data type whose bit length is the sum of the length of its operands.

7.12 Concatenation

CHANGE:

SystemVerilog enhances the concatenation operation to allow concatenation of variables of type string. In general, if any of the operands is of type string, the concatenation is treated as a string, and all other arguments are implicitly converted to the string type (as described in Section 3.7). String concatenation is not allowed on the left hand side of an assignment, only as an expression.

TO:

SystemVerilog enhances the concatenation operation to allow concatenation of data objects of type string. In general, if any of the operands is of the data type string, the concatenation is treated as a string, and all other arguments are implicitly converted to the string data type (as described in Section 3.7). String concatenation is not allowed on the left hand side of an assignment, only as an expression.

CHANGE:

The replication operator (also called a multiple concatenation) form of braces can also be used with variables of type string. In the case of string replication, a non-constant multiplier is allowed.

TO:

The replication operator (also called a multiple concatenation) form of braces can also be used with data objects of type string. In the case of string replication, a non-constant multiplier is allowed.

7.16 Aggregate expressions

CHANGE:

Unpacked structure and array variables, literals, and expressions can all be used as aggregate expressions.

TO:

Unpacked structure and array data objects, as well as unpacked structure and array constructors, can all be used as aggregate expressions.

7.17 Operator overloading

CHANGE:

The overload declaration allows the arithmetic operators to be applied to data types that are normally illegal for them, such as unpacked structures. It does not change the meaning of the operators for those types where it is legal to apply them. This means that such code does not change behavior when operator overloading is used.

TO:

The overload declaration allows the arithmetic operators to be applied to data types that are normally illegal for them, such as unpacked structures. It does not change the meaning of the operators for those data types where it is legal to apply them. This means that such code does not change behavior when operator overloading is used.

CHANGE:

The overload declaration links an operator to a function prototype. The arguments are matched, and the type of the result is then checked. Multiple functions can have the same arguments and different return types. If no expected type exists because the operator is in a self-determined context, then a cast must be used to select the correct function. Similarly if more than one expected type is possible, due to nested operators, and could match more than one function, a cast must be used to select the correct function.

TO:

The overload declaration links an operator to a function prototype. The arguments are matched, and the data type of the result is then checked. Multiple functions can have the

same arguments and different return data types. If no expected data type exists because the operator is in a self-determined context, then a cast must be used to select the correct function. Similarly if more than one expected data type is possible, due to nested operators, and could match more than one function, a cast must be used to select the correct function.

CHANGE:

An expected result type exists in any of the following contexts:

TO:

An expected result data type exists in any of the following contexts:

CHANGE:

The overloading declaration links the + operator to each function prototype according to the equivalent argument types in the overloaded expression, which normally must match exactly. The exception is if the actual argument is an integral type and there is only one prototype with a corresponding integral argument, the actual is implicitly cast to the type in the prototype.

TO:

The overloading declaration links the + operator to each function prototype according to the equivalent argument data types in the overloaded expression, which normally must match exactly. The exception is if the actual argument is an integral type and there is only one prototype with a corresponding integral argument, the actual is implicitly cast to the data type in the prototype.

9.5 Continuous assignments

CHANGE:

SystemVerilog removes this restriction, and permits continuous assignments to drive nets any type of variable.

TO:

SystemVerilog removes this restriction, and permits continuous assignments to drive nets and variables of any data type.

11.5 Object properties

CHANGE:

Any data type can be declared as a class property, except for net types since they are incompatible with dynamically allocated data.

TO:

There are no restrictions on the data type of a class property.

19.2: Interface syntax

CHANGE:

The aim of interfaces is to encapsulate communication. At the lower level, this means bundling variables and wires in interfaces, and can impose access restrictions with port directions in modports.

TO:

The aim of interfaces is to encapsulate communication. At the lower level, this means bundling variables and nets in interfaces, and can impose access restrictions with port directions in modports.

23.4 Expression size system function

CHANGE:

The $bits function can be used as an elaboration-time constant when used on fixed sized types; hence, it can be used in the declaration of other types or variables.

TO:

The $bits function can be used as an elaboration-time constant when used on fixed sized types; hence, it can be used in the declaration of other data types, variables or nets.

23.7: Array querying system functions

CHANGE:

SystemVerilog provides system functions to return information about a particular dimension of an array variable or type.

TO:

SystemVerilog provides system functions to return information about a particular dimension of an array data object or data type.

ANNEX J Glossary

ADD:

Data object – A data object is a named entity that has a data value associated with it. Examples of data objects are nets, variables and parameters. A data object has a data type that determines which values the data object can have.

ADD:

Data type - A data type is a set of values and a set of operations that can be performed on those values. Examples of data types are logic, real, and string. Data types can be used to declare data objects, or to define user-defined data types that are constructed from other data types.

CHANGE:

Aggregate - An aggregate expression, variable or type represents a set or collection of singular values. An aggregate type is any unpacked structure, unpacked union, or unpacked array data type.

TO:
Aggregate - An aggregate expression, data object or data type represents a set or collection of singular values. An aggregate data type is any unpacked structure, unpacked union, or unpacked array data type.

CHANGE:

Bit-stream - A bit-stream type or variables is any type that can be represented as a serial stream of bits. To qualify as a bit-stream type, each and every bit of the type must be individually addressable. This means that a bit-stream type can be any type that does not include a handle, chandle, real, shortreal, or event.

TO:

Bit-stream - A bit-stream data type is any data type whose values can be represented as a serial stream of bits. To qualify as a bit-stream data type, each and every bit of the values must be individually addressable. This means that a bit-stream data type can be any data type except for a handle, chandle, real, shortreal, or event.

CHANGE:

Dynamic - A dynamic type or variable is one that can be resized or re-allocated at runtime. Dynamic types include those that contain dynamic arrays, associative arrays, queues, or class handles.

TO:

Dynamic - A dynamic data type or variable has values that can be resized or re-allocated at runtime. Dynamic arrays, associative arrays, queues, class handles and data types that include such data types are dynamic data types.

CHANGE:

Enumerated type - Enumerated data types provide the capability to declare a variable which can have one of a set of named values. The numerical equivalents of these values can be specified. Enumerated types can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values. Section 3.10 discusses enumerated types.

TO:

Enumerated type - Enumerated data types provide the capability to declare a data object that can have one of a set of named values. The numerical equivalents of these values can be specified. Values of an enumerated data type can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values. Section 3.10 discusses enumerated types.

CHANGE:

Integral - An integral expression, variable or type is used to represent integral, or integer value They may also be called vectored values. Integrals may be signed or unsigned, sliced into smaller integral values, or concatenated into larger values.

TO:

Integral - An integral data type represents integral, or integer, values. Integral values may also be called vectored values. Integral values may be signed or unsigned, sliced into smaller integral values, or concatenated into larger values. An integral expression is an expression of an integral data type. An integral data object is an object of an integral data type.

CHANGE:

Singular - A singular expression, variable or type represents a single value, symbol, or handle. A singular type is any type except an unpacked structure, unpacked union, or unpacked array data type.

TO:

Singular - A singular expression, data object or data type represents a single value, symbol, or handle. A singular data type is any data type except an unpacked structure, unpacked union, or unpacked array data type.

Proposed “var” Extension

5.4. Variables

CHANGE:

A variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

TO:

One form of variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

Another form of variable declaration begins with the keyword "var". The data type is optional in this case. If a data type is not specified then the data type logic shall be inferred.

```
var byte my_byte;   // equivalent to "byte my_byte;"
var v;              // equivalent to "var logic v;"
var [15:0] vw;      // equivalent to "var logic [15:0] vw;"
var enum bit { clear, error } status;
input var logic data_in;
var reg r;
```

18.1 Introduction

CHANGE:

An important enhancement in SystemVerilog is the ability to pass any data type through module ports, including nets, and all variable types including reals, arrays and structures.

TO:

An important enhancement in SystemVerilog is the ability to pass a value of any data type through module ports, using nets or variables. This includes reals, arrays and structures.

18.8 Port declarations

CHANGE:

With SystemVerilog, a port can be a declaration of a net, an interface, an event, or a variable of any type, including an array, a structure or a union.

TO:

With SystemVerilog, a port can be a declaration of an interface, or a variable or net of any allowed data type, including an array, a structure or a union. Within section 18.8, the term *port kind* will be used to mean any of the net type keywords, or the keyword **var**, which are used to explicitly declare a port of one of these kinds. If these keywords are omitted in a port declaration, there are default rules for determining the port kind.

CHANGE:

If the first port direction but no type is specified, then the port type shall default to **wire**. This default type can be changed using the `default_nettype compiler directive, as in Verilog.

TO:

If the first port direction but no port kind or data type is specified, then the port shall default to a net of net type wire. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

CHANGE:

For subsequent ports in the port list, if the type and direction are omitted, then both are inherited from the previous port. If only the direction is omitted, then it is inherited from the previous port. If only the type is omitted, it shall default to wire. This default type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// second port inherits its direction and type from previous port
module mh3 (input byte a, b);
  ...
endmodule
```

TO:

For subsequent ports in the port list, if the direction and the port kind and data type are omitted, then the direction and any port kind and data type are inherited from the previous port. If the direction is omitted, but a port kind or data type is present, then the direction is inherited from the previous port. If the direction is present, but the port kind and data types are omitted, then the port shall default to a net of net type **wire**. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// second port inherits its direction and data type from previous port
module mh3 (input byte a, b);
  ...
endmodule
```

For an **inout** port, if the port kind is omitted, then the port shall default to a net of net type **wire**. This default net type can be changed using the `default_nettype compiler directive, as in Verilog.

```
// the inout port defaults to a net of net type wire
module mh2 (inout integer a);
  ...
endmodule
```

18.11.3 Instantiation using implicit .name port connections

DELETE:

- A port connection between a net type and a variable type of the same bit length is a legitimate cast.

18.12 Port connection rules

CHANGE:

SystemVerilog extends Verilog port connections by making all variable data types available to pass through ports

TO:

SystemVerilog extends Verilog port connections by making values of all data types on variables and nets available to pass through ports.

18.12.2 Port connection rules for nets

CHANGE:

If a port declaration has a wire type (which is the default), or any other net type,

TO:

If a port declaration has a net type, such as wire,
CHANGE:

- An output can be connected to a net type (or a concatenation of net types) or a compatible variable type (or a concatenation of variable types).

- An **inout** can be connected to a net type (or a concatenation of net types) or left unconnected, but not to a variable type.

TO:

- An output can be connected to a net or variable (or a concatenation of nets or variables) of a compatible data type.

- An **inout** can be connected to a net (or a concatenation of nets) of a compatible data type, or left unconnected, but cannot be connected to a variable.

18.12.4 Compatible port types

CHANGE:

The same rules for assignment compatibility are used for compatible
port types for ports declared as an input or an output variable, or for output ports
connected to variables.

TO:

The same rules are used for compatible port types as for assignment compatibility.

A.1.4

REPLACE

net_port_header ::= [port_direction] port_type
variable_port_header ::= [port_direction] data_type

WITH

net_port_header ::= [port_direction] net_port_type

variable_port_header ::= [port_direction] variable_port~~data~~_type

A.2.1.2 and Syntax 18-4

REPLACE

inout_declaration ::=
    inout port_type list_of_port_identifiers

input_declaration ::=
        input port_type list_of_port_identifiers
    | **input** data_type list_of_variable_identifiers

output_declaration ::=
        output port_type list_of_port_identifiers
    | **output** data_type list_of_variable_port_identifiers

interface_port_declaration ::=
        interface_identifier list_of_interface_identifiers
    | interface_identifier **.** modport_identifier list_of_interface_identifiers

ref_declaration ::= **ref** data_type list_of_port_identifiers

WITH

inout_declaration ::=
    inout net_port_type list_of_port_identifiers

input_declaration ::=
        input net_port_type list_of_port_identifiers
    | **input** variable_port~~data~~_type list_of_variable_identifiers

output_declaration ::=
        output net_port_type list_of_port_identifiers
    | **output** variable_port~~data~~_type list_of_variable_port_identifiers

interface_port_declaration ::=
        interface_identifier list_of_interface_identifiers
    | interface_identifier **.** modport_identifier list_of_interface_identifiers

ref_declaration ::= **ref** variable_port~~data~~_type list_of_port_identifiers

A.2.1.3

REPLACE

net_declaration[14] ::=
net_type_or_trireg [drive_strength | charge_strength] [**vectored** | **scalared**]
[signing] { packed_dimension } [delay3] list_of_net_decl_assignments **;**

WITH

net_declaration[14] ::=
net_type ~~or_trireg~~ [drive_strength | charge_strength] [**vectored** | **scalared**]
~~[signing] { packed_dimension }~~
data_type_or_implicit [delay3] list_of_net_decl_assignments **;**

A.2.1.3 and Syntax 5-1

DELETE

~~variable_declaration ::=~~
~~data_type list_of_variable_decl_assignments ;~~

A.2.1.3 and Syntax 5-1

In data_declaration, REPLACE

[**const**] [lifetime] variable_declaration

WITH

[**const**] [**var**] [lifetime] ~~variable_declaration~~ data_type_or_implicit list_of_variable_decl_assignments **;**

[Note to editor: the **var** keyword and the semicolon **;** should both be **red**.]
and ADD the following to footnote 15
In a data_declaration it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.
[Note to editor: the **var** keyword should be **red**.]

A.2.2.1

REPLACE

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **tri0** | **tri1** | **wire** | **wand** | **wor**

port_type ::= [net_type_or_trireg] [signing] { packed_dimension }

net_type_or_trireg ::= net_type | **trireg**

WITH

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **trireg** | **tri0** | **tri1** | **wire** | **wand** | **wor**

net_port_type* ::= [net_type~~_or_trireg~~] data_type_or_implicit ~~[signing] { packed_dimension }~~

~~net_type_or_trireg ::= net_type | **trireg**~~

*When a net_port_type contains a data_type, it shall only be legal to omit the explict net_type when declaring an inout port.

variable_port_type ::= var_data_type
var_data_type ::= data_type | **var** data_type_or_implicit

[Note to editor: the **var** and **trireg** keywords should be **red**.]

Syntax 3-1

REPLACE

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **tri0** | **tri1** | **wire** | **wand** | **wor**

WITH
net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **trireg** | **tri0** | **tri1** | **wire** | **wand** | **wor**

[Note to editor: the **trireg** keyword should be **red**.]

Syntax 18-4

REPLACE

port_type ::= [net_type_or_trireg] [signing] { packed_dimension }

WITH

net_port_type* ::= [net_type~~_or_trireg~~] data_type_or_implicit ~~[signing] { packed_dimension }~~

*When a net_port_type contains a data_type, it shall only be legal to omit the explict net_type when declaring an inout port.

variable_port_type ::= var_data_type
var_data_type ::= data_type | **var** data_type_or_implicit

[Note to editor: the **var** keyword should be **red**.]

A.2.7 and Syntax 10-1

REPLACE

tf_port_item ::=
{ attribute_instance }
[tf_port_direction] data_type_or_implicit
port_identifier variable_dimension [= expression]

WITH

tf_port_item ::=
{ attribute_instance }
[tf_port_direction] [**var**] data_type_or_implicit
port_identifier variable_dimension [= expression]

[Note to editor: the **var** keyword should be **red**.]

A.2.7 and Syntax 10-1

REPLACE

tf_port_declaration ::=
{ attribute_instance } tf_port_direction data_type_or_implicit
list_of_tf_variable_identifiers **;**
WITH

tf_port_declaration ::=
{ attribute_instance } tf_port_direction [**var**] data_type_or_implicit
list_of_tf_variable_identifiers **;**

[Note to editor: the **var** keyword should be **red**.]

A.2.10, Syntax 17-4, and Syntax 17-14

REPLACE

assertion_variable_declaration ::=

data_type list_of_variable_identifiers ;

WITH

assertion_variable_declaration ::=
var_data_type list_of_variable_identifiers ~~;~~ **;**

[Note to editor: the semicolon **;** should be **red**.]

Table B-1

DELETE

~~Note: The keyword `var` is reserved for future extensions.~~