

3. Data types

1

Please review this clause thoroughly; I've removed many examples and some sections seem skimpy. Plus, I still feel some sections still rely on the examples listed to define the syntax and/or semantics.

5

The *e* language has a number of predefined data types, including the integer and Boolean scalar types common to most programming languages. In addition, new scalar data types (*enumerated types*) that are appropriate for programming, modeling hardware, and interfacing with hardware simulators can be created. The *e* language also provides a powerful mechanism for defining object-oriented hierarchical data structures (*structs*) and ordered collections of elements of the same type (*lists*).

10

The following sections provide a basic explanation of *e* data types:

- e data types
- Memory requirements for data types
- Untyped expressions
- Assignment rules
- Precision rules for numeric operations
- Automatic type casting
- Defining and extending scalar types
- Type conversion between scalars and strings

15

20

3.1 e data types

25

Most *e* expressions have an explicit data type. These data types are described in the following sections:

- Scalar types
- Scalar subtypes
- Enumerated scalar types
- Casting of enumerated types in comparisons
- Struct types
- Struct subtypes
- Referencing fields in when constructs
- List types
- The string type
- The external_pointer type

30

35

Certain expressions, such as HDL objects, have no explicit data type. See 3.3 for information on how these expressions are handled.

40

3.1.1 Scalar types

Scalar types in *e* are one of: numeric, Boolean, or enumerated. Table 1 shows the predefined numeric and Boolean types.

45

Both signed and unsigned integers can be of any size and, thus, of any range. See 3.1.2 for information on how to specify the size and range of a scalar field or variable explicitly. *See also*: predefined constants and constraint Boolean expressions (Clause 2).

50

55

Table 1—Predefined scalar types

Type name	Function	Default size for packing	Default value
int	Represents numeric data, both negative and non-negative integers.	32 bits	0
uint	Represents unsigned numeric data, non-negative integers only.	32 bits	0
bit	An unsigned integer in the range 0–1.	1 bit	0
byte	An unsigned integer in the range 0–255.	8 bits	0
time	An integer in the range 0–2 ⁶³ -1.	64 bits	0
bool	Represents truth (logical) values, TRUE(1), and FALSE (0).	1 bit	FALSE (0)

3.1.2 Scalar subtypes

A *scalar subtype* can be named and created by using a scalar modifier to specify the range or bit width of a scalar type. Unbounded integers are a predefined scalar subtype. The following sections describe scalar modifiers, named scalar subtypes, and unbounded integers in more detail.

3.1.2.1 Scalar modifiers

There are two types of scalar modifiers that can be used to modify predefined scalar types:

- range modifiers
- width modifiers

Range modifiers define the range of values that are valid. For example, the range modifier in the expression below restricts valid values to those between zero and 100, inclusive.

```
int [0..100]
```

Width modifiers define the width in bits or bytes. For example, the width modifiers in the expressions below restrict the bit width to 8.

```
int (bits: 8)
int (bytes: 1)
```

Width and range modifiers can also be used in combination, e.g.,

```
int [0..100] (bits: 7)
```

3.1.2.2 Named scalar subtypes

Named scalar subtypes are useful in a context where it is desirable to declare a counter variable, such as the variable “count”, in several places in the program, e.g.,

```
var count : int [0..100] (bits:7);
```

The type name can then be used to introduce new variables with this type, e.g.,

```
type int_count : int [0..99] (bits:7);
var count : int_count;
```

1

See 3.7.1 for more information on named scalar subtypes.

5

3.1.2.3 Unbounded integers

Unbounded integers represent arbitrarily large positive or negative numbers. Unbounded integers are specified as:

10

```
int (bits: *)
```

Use an unbounded integer variable when the exact size of the data is unknown. Unbounded integers can be used in expressions just as signed or unsigned integers are, with the following exceptions.

15

- Fields or variables declared as unbounded integers shall not be generated, packed, or unpacked.
- Unbounded unsigned integers are not allowed, so a declaration of a type such as `uint (bits:*)` shall generate a compile-time error.

20

3.1.3 Enumerated scalar types

The valid values for a variable or field can be defined as a list of symbolic constants, e.g., the following declaration defines the variable “kind” as having two legal values.

25

```
var kind: [immediate, register];
```

These symbolic constants have associated unsigned integer values. By default, the first name in the list is assigned the value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1. Explicit unsigned integer values can also be assigned to the symbolic constants.

30

```
var kind: [immediate = 1, register = 2];
```

The associated unsigned integer value of a symbolic constant in an enumerated type can be obtained by using the `.as_a()` type casting operator (see 3.8.1). Similarly, an unsigned integer value that is within the range of the values of the symbolic constants can be cast as the corresponding symbolic constant.

35

Value assignments can also be mixed, some can explicitly be assigned to symbolic constants and others might be automatically assigned. The following declaration assigns the value 3 to “immediate”; the value 4 is automatically assigned to “register”.

40

```
var kind: [immediate = 3, register];
```

An enumerated type can be named to facilitate its reuse throughout a program. For example, the first statement below defines a new enumerated type named “instr_kind”. The variable “i_kind” has the two legal values defined by the “instr_kind” type.

45

```
type instr_kind: [immediate, register];
var i_kind: instr_kind;
```

50

Enumerated types can be sized.

```
type instr_kind: [immediate, register] (bits: 2);
```

55

Variables or fields with an enumerated type can also be restricted to a range. This variable declaration excludes “foreign” from its legal values:

```
var p :packet_protocol [Ethernet..IEEE];
```

The default value for an enumerated type is zero (0), even if zero (0) is not a legal value for that type. For example, the variable “i_kind” has the value zero (0) until it is explicitly initialized or generated.

```
type instr_kind: [immediate = 1, register = 2];
var i_kind: instr_kind;
```

3.1.4 Casting of enumerated types in comparisons

Enumerated scalar types, like Boolean types, are not automatically converted to or from integers or unsigned integers in comparison operations (i.e., comparisons using the <, <=, >, >=, ==, or != operators). This is consistent with the strong typing in e and helps avoid the introduction of bugs if the order of symbolic names in an enumerated type declaration is changed. To perform such comparisons, explicit casting or tick notation (‘) needs to be used to specify the type.

3.1.5 Struct types

Structs are the basis for constructing compound data structures (*See also*: Clause 4). The default value for a struct is NULL. A struct type can also be used to define a variable. For more information on vars, see 16.2.

The following statement creates a struct type called “packet” with a field “protocol” of type “packet_protocol”.

```
struct packet {
    protocol: packet_protocol;
};
```

The struct type “packet” can then be used in any context where a type is required. For example, in this statement, “packet” defines the type of a field in another struct.

```
struct port {
    data_in : packet;
};
```

3.1.6 Struct subtypes

When a struct field has a Boolean type or an enumerated type, a struct subtype can be defined for one or more of the possible values for that field.

Example

The struct “packet” defined below has three possible subtypes based on its “protocol” field. The “gen_eth_packet” method below generates an instance of the “legal Ethernet packet” subtype, where legal == TRUE and protocol == Ethernet.

```
type packet_protocol: [Ethernet, IEEE, foreign];

struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
```

```

    legal: bool;
  };
  extend sys {
    gen_eth_packet () is {
      var packet: legal Ethernet packet;
      gen packet keeping {it.size < 10;};
      print packet;
    };
  };
};

```

To refer to a Boolean struct subtype, in this case, “legal packet”, use this syntax:

```
field_name struct_type
```

To refer to an enumerated struct subtype in a struct where no values are shared between the enumerated types, use this syntax:

```
value_name struct_type
```

In structs where more than one enumerated field can have the same value, use the following syntax instead to refer to the struct subtype:

```
value'field_name struct_type
```

The **extend**, **when**, or **like** constructs can also be used to add fields, methods, or method extensions that are required for a particular subtype. Use the **when** or **extend** construct (see Clause 4) to define struct subtypes with very similar results. These constructs are appropriate for most modeling purposes (*See also*: Annex A).

3.1.7 Referencing fields in when constructs

To refer to a field of a struct subtype outside of a **when**, **like**, or **extend** construct, assign a temporary name to the struct subtype [and then use that name](#). To reference a field in a **when** construct, first specify the appropriate value for the **when** determinant (see Annex A).

3.1.8 List types

List types hold ordered collections of data elements, where each data element conforms to the same type. Items in a list can be indexed with the subscript operator [], by placing a non-negative integer expression in the brackets. List indexes start at zero (0). To select an item from a list, specify its index, e.g., my_list[0] refers to the first item in the list named my_list.

Lists are defined by using the **list of** keyword in a variable or a field definition. The example below defines a list of bytes named “lob” and explicitly assigns five literal values to it. The print statement displays the first three elements of “lob”, 15, 31, and 63.

```

var lob: list of byte = {15;31;63;127;255};
print lob[0..2];

```

The following considerations also apply.

- Multi-dimensional lists (lists of lists) are not supported. To create a list with sublists in it, first create a struct to contain the sublists and then create a list of such structs as the main list.
- The default value of a list is an empty list.
- To size lists that are variables, use a **keep** constraint.

3.1.9 Keyed lists

A keyed list data type is similar to hash tables or association lists found in other programming languages. If the element type of the list is a scalar type or a string type, then the hash key shall be the predefined implicit variable `it`.

Keyed lists shall not be generated. Trying to generate a keyed list shall result in an error. Therefore, keyed lists shall be defined with the do-not-generate sign (!).

The only restriction on the type of the list elements is that they themselves shall not be lists. However, they can be struct types containing fields that are lists.

See also: 17.1.6 and Clause 19.

3.1.10 The string type

The predefined type **string** is the same as the C NULL terminated (zero terminated) string type. A series of ASCII characters enclosed by quotes (“”) can be assigned to a variable or field of type string, for example:

```
var message: string;
message = "Beginning initialization sequence...";
```

Bits or bit ranges of a string cannot be accessed, but the string can be converted to a list of bytes and that list can be used to access a portion of the string, e.g., the print statement below displays “/test1”.

```
var dir: string = "/tmp/test1";
var tmp := dir.as_a(list of byte);
tmp = tmp[4..9];
print tmp.as_a(string);
```

The default value of a variable of type **string** is NULL.

See also: 17.1.4 and Clause 24.

3.1.11 The external_pointer type

The `external_pointer` type is used to hold a pointer into an external (non-e) entity, such as a C struct. Unlike pointers to structs in e, external pointers are not changed during garbage collection.

3.2 Memory requirements for data types

The amount of memory needed to store data types is listed in [Table 2](#).

Table 2—Storage sizes of datatypes

Type	Size in memory
All scalars up to 32 bits	4 bytes.
Scalars larger than 32 bits	The same as a list of bits of the appropriate size.
String	4 bytes (the pointer) + the size of the string + 1 byte (the NULL byte). A NULL string is just the pointer.

Table 2—Storage sizes of datatypes (Continued)

Type	Size in memory
Struct pointer	4 bytes.
Struct	8 bytes + the sum of the field sizes. A NULL struct is just the pointer (4 bytes).
List	4 bytes (a pointer to the list) + approximately 16 bytes (header) + the sum of the sizes of the elements. Lists of scalars of size up to 16 bits are packed to the nearest power of 2 (in bits). This is often the most efficient representation.

3.3 Untyped expressions

All *e* expressions have an explicit type, except for the following types:

- HDL objects, such as 'top.w_en'
- **pack()** expressions, such as “pack(packing.low, 5)”
- bit concatenations, such as “%{slb1, slb2};”

The default type of HDL objects is 32-bit uint, while **pack()** expressions and bit concatenations have a default type of `list of bit`. However, due to implicit packing and unpacking, these expressions can be converted to the required data type and bit-size in certain contexts.

- a) When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked and converted to the same type and bit-size as the expression on the left-hand side. Implicit unpacking is not supported for strings, structs, or lists of non-scalar types.
- b) When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned. Implicit packing is not supported for strings, structs, or lists of non-scalar types.
- c) When the untyped expression is the operand of any binary operator (+, -, *, /, %), the expression is assumed to be a numeric type. The precision of the operation is determined by the expected type and the type of the operands. See 3.5 for more information.
- d) When a **pack()** expression includes the parameter or the return value of a method call, the expression takes the type and size as specified in the method declaration. The method parameter or return value in the pack expression must be a scalar type or a list of scalar type.
- e) When an untyped expression appears in one of the following contexts, it is treated as a Boolean expression:

```

if (untyped_exp) then {...}
while (untyped_exp) do {...}
check that (untyped_exp)
not untyped_exp
rise(untyped_exp), fall(untyped_exp), true(untyped_exp)

```

When the type and bit-size cannot be determined from the context, the expression is automatically cast according to the following rules.

- The default type of an HDL signal is an unsigned integer.
- The default type of a pack expression and a bit concatenation expression is a list of bit.
- If no bit width specification is detected, the default width is 32 bits.

When expressions are untyped, an implicit pack/unpack is performed according to the expected type. See also: 17.2.15.

3.4 Assignment rules

Assignment rules define what is a legal assignment and how values are assigned to entities. The following sections describe various aspects of assignments

3.4.1 What is an assignment?

There are several legal ways to assign values:

- Assignment actions
- Return actions
- Parameter passing
- Variable declaration

Here is an example of an assignment action, where a value is explicitly assigned to a variable “x” and to a field “sys.x”.

```

extend sys{
  x: int;
  m() is {
    sys.x = '~/top/address';
    var x: int;
    x = sys.x + 1;
  };
};

```

Here’s an example of a **return** action, which implicitly assigns a value to the **result** variable:

```

extend sys {
  n(): int (bits: 64) is {
    return 1;
  };
};

```

Here’s an example of assigning a value (6) to a method parameter (“i”):

```

extend sys {
  k(i: int) @sys.any is {
    wait [i] * cycle;
  };

  run() is also {
    start k(6);
  };
};

```

Here’s an example of how variables are assigned during declaration:

```

extend sys {
  b() is {
    var x: int = 5;
    var y:= "ABC";
  };
};

```

Values shall not be assigned to fields during declaration, however.

3.4.2 Assignments create identical references

1

Assigning one struct, list, or value to another object of the same type results in two references pointing to the same memory location, so that changes to one of the objects also occur in the other object immediately.

5

Example

```
data1: list of byte;
data2: list of byte;
run() is also {
    data2 = data1;
    data1[0] = 0;
};
```

10

After generation, the two lists data1 and data2 are different lists. However, after the data2=data1 assignment, both lists refer to the same memory location; therefore, changing the data1[0] value also changes the data2[0] value immediately.

15

3.4.3 Assignment to different but compatible types

20

****Add a lead-in sentence****

3.4.3.1 Assignment of numeric types

25

Any numeric type (e.g., **uint**, **int**, or one of their subtypes) can be assigned with any other numeric type. Untyped expressions, such as HDL objects, can also appear in assignments of numeric types (see 3.3).

Automatic casting is performed when a numeric type is assigned to a different numeric type and automatic extension or truncation is performed if the types have a different bit-size (see 3.6). *See also:* 3.5 on how precision is determined for operations involving numeric types.

30

3.4.3.2 Assignment of Boolean types

A Boolean type can only be assigned with another Boolean type.

35

```
var x: bool;
x = 'top.a' >= 16;
```

3.4.3.3 Assignment of enumerated types

40

An enumerated type can be assigned with that same type or its scalar subtype. (The scalar subtype differs only in range or bit-size from the base type.) The following example shows:

- An assignment of the same type:

45

```
var x: color = blue;
```

- An assignment of a scalar subtype:

```
var y: color2 = x;
```

50

To assign any scalar type (numeric, enumerated, or Boolean type) to any different scalar type, use the **.as_a()** operator (see 3.8.1).

55

3.4.3.4 Assignment of structs

An entity of type struct can be assigned with a struct of that same type or with one of its subtypes. The following example shows:

- A same type assignment:

```
p2 = p1;
```

- An assignment of a subtype (Ether_8023 packet):

```
var p: Ether_8023 packet;
set_cell(p);
```

- An assignment of a derived struct (cell_8023):

```
set_cell(p: packet) is
p.cell = new cell_8023;
```

Although a subtype can be assigned to its parent struct without any explicit casting, to perform the reverse assignment (assign a parent struct to one of its subtypes), the `.as_a()` method needs to be used (see 3.8.1).

3.4.3.5 Assignment of strings

A string can be assigned only with strings, as shown below.

```
extend sys {
  m(): string is {
    return "aaa"; // assignment of a string
  };
};
```

3.4.3.6 Assignment of lists

An entity of a type list can be assigned only with a list of the same type. In the following example, the assignment of “list1” to “x” is legal because both lists are lists of integers.

```
extend sys {
  list1: list of int;
  m() is {
    var x: list of int = list1;
  };
};
```

However, an assignment such as “var y: list of int (bits: 16) = list1;” is not legal, because “list1” not the same list type as “y”. “y” has a size modifier, so it is a subtype of “list1”.

Use the `.as_a()` method to cast between lists and their subtypes (see 3.8.1).

3.5 Precision rules for numeric operations

For precision rules, there are two types of numeric expressions in *e*:

- *context-independent* expressions, where the precision of the operation (bit-width) and numeric type (signed or unsigned) depend only on the types of the operands

- *context-dependent* expressions, where the precision of the operation and the numeric type depend on the precision and numeric type of other expressions involved in the operation (the *context*), as well as the types of the operands

A numeric operation in *e* is performed in one of three possible combinations of precision and numeric type:

- a) Unsigned 32-bit integer (**uint**)
- b) Signed 32-bit integer (**int**)
- c) Infinite signed integer (**int (bits: *)**)

The *e* language has rules for determining the context of an expression or deciding the precision, and performing data conversion and sign extension.

3.5.1 Determining the context of an expression

The rules for defining the context of an expression are applied in the following order:

- a) In an assignment (*lhs = rhs*), the right-hand side (*rhs*) expression inherits the context of the left-hand side (*lhs*) expression.
- b) A sub-expression inherits the context of its enclosing expression.
- c) In a binary-operator expression (*lho OP rho*), the right-hand operand (*rho*) inherits context from the left-hand operand (*lho*), as well as from the enclosing expression.

Table 3 summarizes context inheritance for each type of operator that can be used in numeric expressions.

Table 3—Summary of context inheritance in numeric operations

Operator	Function	Context
* / % + - < <= > >= == != === !== & ^	Arithmetic, comparison, equality, and bitwise Boolean	The right-hand operand inherits context from the left-hand operand (<i>lho</i>), as well as from the enclosing expression. <i>lho</i> inherits only from the enclosing expression.
~! unary + -	Bitwise not, Boolean not, unary plus, minus	The operand inherits context from the enclosing expression.
[]	List indexing	The list index is context independent.
[..]	List slicing	The indices of the slice are context independent.
[:]	Bit slicing	The indices of the slice are context independent.
f(...)	Method or routine call	The context of a parameter to a method is the type and bit-width of the formal parameter.
{..., ...}	List concatenation	Context is passed from the lhs of the assignment, but not from left-to-right between the list members.
%{..., ...}	Bit concatenation	The elements of the concatenation are context independent.
>>, <<	Shift	Context is passed from the enclosing expression to the left operand. The context of the right operand is always a 32-bit uint .
<i>lho</i> in [<i>i</i> : <i>j</i>]	Range list operator	All three operands are context independent. (The range specifiers <i>i</i> and <i>j</i> shall be constant.)
&&,	Boolean	All operands are context independent.

Table 3—Summary of context inheritance in numeric operations (Continued)

Operator	Function	Context
$a ? b : c$	Conditional operator	a is context independent, b inherits the context from the enclosing expression, c inherits context from b as well as from the enclosing expression
.as_a()	Casting	The operand is context independent.
abs(), odd() even()	Arithmetic routine	The parameter is context independent.
min(), max()	Arithmetic routine	The right parameter inherits context from the left parameter (lp), as well as from the enclosing expression. lp inherits only from the enclosing expression.
ilog2(), ilog10(), isqrt()	Arithmetic routine	The context of the parameter is always a 32-bit uint .
ipow()	Arithmetic routine	Both parameters inherit the context of the enclosing expression, but the right parameter does not inherit context from the left.

3.5.2 Deciding precision and performing data conversion and sign extension

The rules for deciding precision, and performing data conversion and sign extension are:

- determine the context of the expression, which can be comprised of a maximum of two types
- if all types involved in an expression and its context are 32 bits in width or less:
 - 1) The operation is performed in 32 bits.
 - 2) If any of the types is unsigned, the operation is performed with unsigned integers. Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers, unless preceded by a hyphen.
 - 3) Each operand is automatically cast, if necessary, to the required type. Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.
- If any of the types is greater than 32 bits:
 - 1) The operation is performed in infinite precision (**int (bits:*)**).
 - 2) Each operand is zero-extended, if it is unsigned, or sign-extended, if it is signed, to infinite precision.

3.6 Automatic type casting

During assignment of a type to a different, but compatible type, automatic type casting is performed in the following contexts

- Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types. For example:

```
var x: uint;
var y: int;
x = y;
```

- Untyped expressions are automatically cast on assignment. See 3.3 for more information.
- Sized scalars are automatically type cast to differently sized scalars of the same type.
- Struct subtypes are automatically cast to their base struct type.

There are three important ramifications to automatic type casting: 1

- a) If the two types differ in bit-size, the assigned value is extended or truncated to the required bit-size.
- b) Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers. 5
- c) There is no automatic casting to a reference parameter (see 15.3).

3.7 Defining and extending scalar types 10

The following constructs can be used to define and extend scalar types:

- type enumerated scalar
- type scalar subtype
- type sized scalar 15
- extend type

3.7.1 type enumerated scalar 20

Purpose	Define an enumerated scalar type	
Category	Statement	
Syntax	type <i>enum-type-name</i> : [[<i>name</i> [= <i>exp</i>], ...]] [(bits bytes : <i>width-exp</i>)]	25
Parameters	<i>enum-type-name</i>	A legal <i>e</i> name. The name shall be different from any other predefined or enumerated type name because the name space for types is global.
	<i>name</i>	A legal <i>e</i> name. Each name shall be unique within the type. 30
	<i>exp</i>	A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1.
	<i>width-exp</i>	A positive constant expression. The valid range of values for sized enumerated scalar types is limited to the range 1 to $2^n - 1$, where <i>n</i> is the number of bits. 35

This defines an enumerated scalar type consisting of a set of names or name-value pairs. If no values are specified, the names get corresponding numerical values starting with 0 for the first name and casting can be done between the names and the numerical values. 40

Syntax example:

```
type PacketType :[ rx = 1, tx, ctrl ]; 45
```

50

55

3.7.2 type scalar subtype

Purpose	Define a scalar subtype	
Category	Statement	
Syntax	type <i>scalar-subtype-name</i> : <i>scalar-type</i> [<i>range</i> , ...]	
Parameters	<i>scalar-subtype-name</i>	A unique <i>e</i> name.
	<i>scalar-type</i>	Any previously defined enumerated scalar type, any of the predefined scalar types, including int , uint , bool , bit , byte , or time , or any previously defined scalar subtype.
	<i>range</i>	A constant expression or two constant expressions separated by two dots. All constant expressions shall resolve to legal values of the named type.

This defines a subtype of a scalar type by restricting the legal values that can be generated for this subtype to the specified range. The default value for variables or fields of this type “size” is zero (0), [which is](#) the default for all integers. The *range* only affects any generated values.

Syntax example:

```
type size: int [8, 16];
```

3.7.3 type sized scalar

Purpose	Define a sized scalar	
Category	Statement	
Syntax	type <i>sized-scalar-name</i> : type (bits bytes : <i>exp</i>)	
Parameters	<i>scalar-scalar-name</i>	A unique <i>e</i> name.
	<i>type</i>	Any previously defined enumerated type or any of the predefined scalar types, including int , uint , bool , or time .
	<i>exp</i>	A positive constant expression. The valid range of values for sized scalars is limited to the range 1 to $2n - 1$, where n is the number of bits.

This defines a scalar type with a specified bit width. The actual bit width is $exp * 1$ for bits and $exp * 8$ for bytes.

When assigning any expression into a sized scalar variable or field, the expression's value is truncated or extended automatically to fit into the variable. An expression with more bits than the variable is chopped down to the size of the variable. An expression with fewer bits is extended to the length of the variable. The added upper bits are filled with zero's (0) if the expression is unsigned or with the **appropriate sign bit** (0 or 1) if the expression is signed.

Syntax example:

```

type word      :uint(bits:16);
type address  :uint(bytes:2);

```

3.7.4 extend type

Purpose	Extend an enumerated scalar type	
Category	Statement	
Syntax	extend <i>enum-type</i>: [<i>name</i> [= <i>exp</i>], ...]	
Parameters	<i>enum-type</i>	Any previously defined enumerated type.
	<i>name</i>	A legal <i>e</i> name. Each name shall be unique within the type.
	<i>exp</i>	A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1.

This extends the specified enumerated scalar type to include the specified names or name-value pairs.

Syntax example:

```

type PacketType :[ rx, tx, ctrl ];
extend PacketType :[ status ];

```

3.8 Type conversion between scalars and strings

The **as_a()** expression is used to convert an expression from one data type to another. Information about how different types are converted, such as strings to scalars or lists of scalars, is contained in Table 4 and Table 5.

3.8.1 as_a()

Purpose	Casting operator	
Category	Expression	
Syntax	<i>exp.as_a</i> (<i>type</i> : type name): type	
Parameters	<i>exp</i>	Any <i>e</i> expression.
	<i>type</i>	Any legal <i>e</i> type.

This returns the expression converted into the specified type. Although some casting is done automatically (see 3.6), explicit casting is required to make assignments between different, but compatible types.

Syntax example:

```

print (b).as_a(uint);

```

3.8.1.1 Type conversion between scalars and lists of scalars

Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types.

For other scalars and lists of scalars, there are a number of ways to perform type conversion, including the `as_a()` method, the `pack()` method, the `%{}` bit concatenation operator, and various string routines. Table 4 shows how to convert between scalars and lists of scalars.

In Table 4, `int` represents `int/uint` of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. `int(bits:x)` means `x` as any constant; variables shall not be used as the integer width.

The solutions `presume` variables are declared as

```
var int : int ;
var bool : bool ;
var enum : enum ;
var list_of_bit : list of bit ;
var list_of_byte : list of byte ;
var list_of_int : list of int ;
```

Any conversions not explicitly shown might have to be accomplished in two stages.

Table 4—Type conversion between scalars and lists of scalars

From	To	Solutions
int	list of bit	<code>list_of_bit = int[..]</code>
int	list of <code>int(bits:x)</code>	<code>list_of_int = %{}int}</code> <code>list_of_int = pack(pack.ing.low, int)</code> (LSB of int goes to list[0] for either choice)
list of bit list of byte	int	<code>int = list_of_bit[:]</code>
list of <code>int(bits:x)</code>	int	<code>int = pack(pack.ing.low, list_of_int)</code> (Use <code>pack.ing.high</code> for list in other order.)
<code>int(bits:x)</code>	<code>int(bits:y)</code>	<code>intx = inty</code> (Truncation or extension is automatic.) <code>intx.as_a(int(bits:y))</code>
bool	int	<code>int = bool.as_a(int)</code> (TRUE becomes 1, FALSE becomes 0.)
int	bool	<code>bool = int.as_a(bool)</code> (0 becomes FALSE, non-0 becomes TRUE.)
int	enum	<code>enum = int.as_a(enum)</code> (No checking is performed to make sure the int value is valid for the range of the enum.)
enum	int	<code>int = enum.as_a(int)</code> (Truncation is automatic.)

Table 4—Type conversion between scalars and lists of scalars (Continued)

From	To	Solutions
enum	bool	<i>enum.as_a</i> (bool) (Enumerated types with an associated unsigned integer value of 0 become FALSE; those with an associated non-0 values become TRUE. See 3.1.3 for more information on values associated with enumerated types.)
bool	enum	<i>bool.as_a</i> (enum) (Boolean types with a value of FALSE are converted to the enumerated type value that is associated with the unsigned integer value of 0; those with a value of TRUE are converted to the enumerated type value that is associated with the unsigned integer value of 1. No checking is performed to make sure the Boolean value is valid for the range of the enum.)
enum	enum	<i>enum1 = enum2.as_a</i> (enum1) (No checking is performed to make sure the int value is valid for the range of the enum.)
list of int(bits:x)	list of int(bits:y)	<i>listx.as_a</i> (list of int(bits:y)) (Same number of items, each padded or truncated.) <i>listy = pack</i> (<i>packing.low</i> , listx) (Concatenated data, different number of items.)

3.8.1.2 Type conversion between strings and scalars or lists of scalars

There are a number of ways to perform type conversion between strings and scalars or lists of scalars, including the *as_a*() method, the *pack*() method, the *%{}* bit concatenation operator, and various string routines. Table 5 shows how to convert between strings and scalars or lists of scalars.

In Table 5, **int** represents **int/uint** of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. **int(bits:x)** means **x** as any constant; variables shall not be used as the integer width.

The solutions **presume** variables are declared as

```
var int : int ;
var list_of_byte : list of byte ;
var list_of_int : list of int ;
var bool : bool ;
var enum : enum ;
var string : string ;
```

Any conversions not explicitly shown might have to be accomplished in two stages.

3.8.1.3 Type conversion between structs, struct subtypes, and lists of structs

Struct subtypes are automatically cast to their base struct type, so, for example, a variable of type “Ethernet packet” can be assigned to a variable of type “packet” without using *as_a*(). *as_a*() can be used to cast a base struct type to one of its subtypes; if a mismatch occurs, then NULL is assigned. For example, the “**print** pkt.*as_a*(foreign packet)” action results in “pkt.*as_a*(foreign packet) = NULL” if pkt is not a foreign packet.

When the expression to be converted is a list of structs, *as_a*() returns a new list of items whose type matches the specified type parameter. If no items match the type parameter, an empty list is returned. The list can contain items of various subtypes, but all items shall have a common parent type, i.e., the specified type parameter shall be a subtype of the type of the list.

Table 5—Type conversion between strings and scalars or lists of scalars

From	To	ASCII convert?	Solutions
list of int list of byte	string	yes	<code>list_of_int.as_a(string)</code> (Each list item is converted to its ASCII character and the characters are concatenated into a single string. <code>int[0]</code> represents left-most character. If a list item is not a printable ASCII character, the string returned is empty.)
string	list of int list of byte	yes	<code>string.as_a(list of int)</code> (Each character in the string is converted to its numeric value and assigned to a separate element in the list. The left-most character becomes <code>int[0]</code>)
string	list of int	yes	<code>list_of_int = pack(packing.low, string)</code> <code>list_of_int = %{string}</code> (The numeric values of the characters are concatenated before assigning them to the list. Any pack option gives same result; null byte, 00, is the last item in the list.)
string	int	yes	<code>int = %{string}</code> <code>int = pack(packing.low, string)</code> (Any pack option gives the same result.)
int	string	yes	<code>unpack(packing.low, %{8'b0, int}, string)</code> (Any pack option with <code>scalar_reorder={}</code> gives the same result.)
string	int	no	<code>string.as_a(int)</code> (Converts to decimal.) <code>append("0b", string).as_a(int)</code> (Converts to binary.) <code>append("0x", string).as_a(int)</code> (Converts to hexadecimal.)
int	string	no	<code>int.as_a(string)</code> (Uses the current print radix.) <code>append(int)</code> (Converts int according to the current print radix.) <code>dec(int)</code> , <code>hex(int)</code> , <code>bin(int)</code> (Converts int according to a specific radix.)
string	bool	no	<code>bool = string.as_a(bool)</code> (Only "TRUE" and "FALSE" can be converted to Boolean; all other strings return an error.)
bool	string	no	<code>string = bool.as_a(string)</code>
string	enum	no	<code>enum = string.as_a(enum)</code>
enum	string	no	<code>string = enum.as_a(string)</code>

Assigning a struct subtype to a base struct type does not change the declared type. Thus, `as_a()` needs to be used to cast the base struct type as the subtype and access any of the subtype-specific struct members.

Subtypes created through **like** inheritance exhibit the same behavior as subtypes created through **when** inheritance.

3.8.1.4 Type conversion between simple lists and keyed lists

Simple lists can be converted to keyed lists and vice-versa. The hash key is dropped in converting a keyed list to a simple list. However, a key needs to be specified first to convert a simple list to a keyed list.

To convert a simple list of packets (“sys.packets”) to a keyed list, where the “len” field of the packet struct is the key:

Example

```
var pkts: list (key: len) of packet
pkts = sys.packets.as_a(list (key: len) of packet)
```

Using the **as_a()** method returns a copy of sys.packets, so the original sys.packets is still a simple list, not a keyed list. Thus, “print pkts.key_index(130)” returns the index of the item that has a “len” field of 130, while “print sys.packets.key_index(130)” shall return an error.

If a conversion between a simple list and a keyed list also involves a conversion of the type of each item, that conversion of each item follows the standard rules, e.g., when **as_a()** is used to convert an integer to a string, no ASCII conversion is performed. Similarly, if **as_a()** is used to convert a simple list of integers to a keyed list of strings, no ASCII conversion is performed.

No checking is performed to make sure the value is valid when casting from a numeric or Boolean type to an enumerated type, or when casting between enumerated types.

- The **as_a()** pseudo-method, when applied to a scalar list, creates a new list whose size is the same as the original size and then casts each element separately.
- When the **as_a()** operator is applied to a list of structs, the list items for which the casting failed are omitted from the list.
- **as_a()** can be used to convert a string to an enumerated type. The string has to match one of the possible values of that type letter-by-letter or a runtime error shall be issued.

See also: 2.15.1.

3.8.2 all_values()

Purpose	Access all values of a scalar type
Category	Pseudo routine
Syntax	all_values (<i>scalar-type</i> : type name): list of scalar type
Parameters	<i>scalar-type</i> Any legal <i>e</i> scalar type.

This returns a list that contains all the legal values of the specified scalar type. When that type is an enumerated type, the order of the items is the same as the order in which they were defined. When the type is a numeric type, the order of the items is from the smallest to the largest.

When the specified type has more than one million legal values, a compile time error shall be issued (to point out possible memory abuse).

Syntax example:

```
1      print all_values(reg_address);
```

5

10

15

20

25

30

35

40

45

50

55