# Preliminary
# *e* Language Reference Draft

4 December 2003

# Table of Contents

# 1 About This Book

The ***e*** language is an object-oriented programming language. Although ***e*** can be used to create any general-purpose software program, it has been designed to facilitate the verification of electronic designs. The verification-specific constructs that distinguish ***e*** from other object-oriented languages such as C++ include:

— Constructs to define legal values for data items (constraints)
— Constructs to describe sequences over time (temporal constructs)
— Constructs to support concurrency (multi-threaded execution)
— Constructs to support connectivity (bit-level access)

The ***e*** language also is designed to reduce the effort required to write tests and to make the high-level intent of the test readily apparent. In contrast to other object-oriented programming languages, ***e***'s unique extensibility lets you modify multiple data objects in a single, separate test file that is layered on top of the base verification environment. This extensibility feature allows you to address systemic, test-specific concerns that are not localized to a single data object's boundaries in a way that does not sacrifice modularity or readability.

This manual provides detailed information on the ***e*** programming language.

## 1.1 Conventions in This Book

This manual uses visual cues to help you locate and interpret information easily. These cues are explained in Table 1-1 on page 1.

**Table 1-1—Document Conventions**

| Visual Cue | Represents |
|---|---|
| `courier` | The Courier font indicates ***e*** or HDL code. For example, the following line indicates ***e*** code:<br><br>`keep opcode in [ADD, ADDI];` |
| bold | The bold font is used in descriptive text to indicate keywords. For example, the following sentence contains the keyword "keep":<br><br>Use the **keep** construct to define legal values for items.<br><br>The bold font is used in syntax descriptions to indicate text that must be typed exactly as it appears. For example, in the following sentence the keywords "keep" and "reset_soft", as well as the period and the parentheses must be typed as they appear:<br><br>keep *item*.reset_soft() |
| *italic* | The italic font represents user-defined variables that you must provide. For example, the following line instructs you to type "keep" as it appears, and then specify a boolean expression:<br><br>keep *constraint-bool-exp* |

**Table 1-1—Document Conventions *(continued)***

| Visual Cue | Represents |
|---|---|
| [ ] square brackets | Square brackets indicate optional parameters. For example, in the following construct the keywords "list of" are optional:<br><br>**var *name*: [list of] *type*** |
| **[ ]** bold brackets | Bold square brackets are required. For example, in the following construct you must type the bold square brackets as they appear:<br><br>**extend *enum-type-name*: [*name*,...]** |
| ***construct***, ... | An item, followed by a separator (usually a comma or a semicolon) and an ellipsis is an abbreviation for a list of elements of the specified type. For example, the following line means you can type a list of zero or more names separated by commas.<br><br>**extend *enum-type-name*: [*name*,...]** |
| \| | The \| character indicates alternative syntax or parameters. For example, the following line indicates that either the **bits** or **bytes** keyword should be used:<br><br>**type *scalar-type* (bits \| bytes: *num*)** |

## 1.2 Syntax Notation

Each construct section starts with the syntax for the construct. The syntax shows the construct, any arguments it accepts with their types, and the construct's return type if it has one.

When using the construct, terms in bold in the syntax are to be entered exactly as shown. Terms in italics are to be replaced by terms of your own. The argument types and the construct return type are for information only and are not entered.

For example, the syntax notation for the predefined pseudo-method named "first()" on page 576 is

***list*.first(*exp*:** bool**)**: list-type

This is what the notation means:

— The bold ".first" and the parentheses must be entered exactly.
— The parts in italics, "list" and "exp", must be replaced by a list name and an expression.
— ": bool" indicates that the expression must be a boolean expression.
— ": list-type" means that the pseudo-method returns an item of the list element type.

An example of a call to the ***list*.first()** pseudo-method is shown below, where "numbers" is a list of integer items and "my_number" is an integer. The pseudo-method returns the first integer in the list greater than 5:

```
my_number = numbers.first(it > 5)
```

# 2 *e* Basics

This chapter describes the structure of an ***e*** program, starting with the organization of ***e*** code into one or more files and the four categories of ***e*** constructs, and ending with a description of the struct hierarchy. This chapter also describes the ***e*** operators. It contains the following sections:

### See Also

— Chapter 9, “Temporal Expressions”
— Chapter 3, “Data Types”
— Chapter 24, “Predefined Routines Library”

## 2.1 Lexical Conventions

The following sections describe the lexical conventions of ***e***:

### 2.1.1 File Structure

***e*** code can be organized in multiple files. File names must be legal ***e*** names. The default file extension is “.e”. ***e*** code files are sometimes referred to as “modules” Each module contains at least one code segment and can also contain comments.

### See Also

— “Names, Keywords, and Macros” on page 9
— “Code Segments” on page 4
— “Comments and White Space” on page 4

### 2.1.2 Code Segments

A code segment is enclosed with a begin-code marker **<'** and an end-code marker **'>**. Both the begin-code and the end-code markers must be placed at the beginning of a line (left most), with no other text on that same line (no code and no comments). For example, the following three lines of code form a code segment:

```
<'
    import cpu_test_env;
'>
```

Several code segments can appear in one file. Each code segment consists of one or more statements.

#### See Also

— "Comments and White Space" on page 4
— "Statements" on page 12

### 2.1.3 Comments and White Space

***e*** files begin as a comment which ends when the first begin-code marker **<'** is encountered.

Comments within code segments can be marked with double dashes (**--**) or double slashes (**//**):

```
a = 5;        -- This is an inline comment
b = 7;        // This is also an inline comment
```

The end-code **'>** and the begin-code **<'** markers can be used in the middle of code sections, to write several consecutive lines of comment:

```
Import the basic test environment for the CPU...

<'
    import cpu_test_env;
'>

This particular test requires the code that bypasses bug#72 as
well as the constraints that focus on the immediate instructions.

<'
    import bypass_bug72;
    import cpu_test0012;
'>
```

#### See Also

— "Code Segments" on page 4

### 2.1.4 Literals and Constants

Literals are numeric, character and string values specified literally in ***e***. Operators can be applied to literals to create compound expressions. The following categories of literals and constants are supported in ***e***:

— "Unsized Numbers" on page 5
— "Sized Numbers" on page 5
— "MVL Literals" on page 6

— “Predefined Constants” on page 8
— “Literal String” on page 8
— “Literal Character” on page 9

### 2.1.4.1 Unsized Numbers

Unsized numbers are always positive and zero-extended unless preceded by a hyphen. Decimal constants are treated as signed integers and have a default size of 32 bits. Binary, hex, and octal constants are treated as unsigned integers, unless preceded by a hyphen to indicate a negative number, and have a default size of 32 bits. If the number cannot be represented in 32 bits then it is represented as an unbounded integer. See “Unbounded Integers” on page 77 for more information.

The notations shown in Table 2-1 can be used to represent unsized numbers.

**Table 2-1—Representing Unsized Numbers in Expressions**

| Notation | Legal Characters | Examples |
|---|---|---|
| Decimal integer | Any combination of 0-9 possibly preceded by a hyphen - for negative numbers. An underscore (_) can be added anywhere in the number for readability. | 12, 55_32, -764 |
| Binary integer | Any combination of 0-1 preceded by ***0b***. An underscore (_) can be added anywhere in the number for readability. | 0b100111, 0b1100_0101 |
| Hexadecimal integer | Any combination of 0-9 and a-f preceded by ***0x***. An underscore (_) can be added anywhere in the number for readability. | 0xff, 0x99_aa_bb_cc |
| Octal integer | Any combination of 0-7 preceded by 0o. An underscore (_) can be added anywhere in the number for readability. | 0o66_123 |
| K (kilo: multiply by 1024) | A decimal integer followed by a K or k. For example, 32K = 32768. | 32K, 32k, 128k |
| M (mega: multiply by 1024*1024) | A decimal integer followed by an M or m. For example, 2m = 2097152. | 1m, 4m, 4M |

#### See Also

— “Literals and Constants” on page 4

### 2.1.4.2 Sized Numbers

A sized number is a notation that defines a literal with a specific size in bits. The syntax is as follows:

***width-number*** ' (**b**|**o**|**d**|**h**|**x**) ***value-number***

The width number is a decimal integer specifying the width of the literal in bits. The value number is the value of the literal and can be specified in one of four radixes, as shown in Table 2-2.

NOTE— If the value number is more than the specified size in bits, its most significant bits are ignored. If the value number is less that the specified size, it is padded by zeros.

**Table 2-2—Radix Specification Characters**

| Radix | Represented By | Example |
|---|---|---|
| Binary | A leading 'b or 'B | 8'b11001010 |
| Octal | A leading 'o or 'O | 6'o45 |
| Decimal | A leading 'd or 'D | 16'd63453 |
| Hexadecimal | A leading 'h or 'H or 'x or 'X | 32'h12ffab04 |

### See Also

— “Literals and Constants” on page 4

### 2.1.4.3 MVL Literals

An MVL literal is based on the mvl type, which is a predefined enumerated scalar type in ***e***. The mvl type is defined as follows:

type mvl: [MVL_U,MVL_X,MVL_0,MVL_1,MVL_Z,MVL_W,MVL_L,MVL_H,MVL_N];

NOTE— MVL_N represents “don’t care”.

The mvl type is a superset of the capabilities provided by the @x and @z syntax allowed in HDL tick notation. For example, if a port is defined as type list of mvl, you can assign values with the $ access operator:

```
sig$ = {MVL_X;MVL_X;MVL_X} ; -- HDL tick notation is 'sig@x' = 0x3
```

If the port is a numeric type (uint, int, and so on), you can assign mvl values using the predefined MVL methods for ports. For example:

```
sig.put_mvl_list({MVL_X;MVL_X;MVL_X});
```

An MVL literal, which is a literal of type list of mvl, provides a more convenient syntax for assigning MVL values. The syntax of an MVL literal is as follows:

***width-number*** ' (**b**|**o**|**h**) ***value-number***

The width number is an unsigned decimal integer specifying the size of the list. The value number is any sequence of digits that are legal for the base, plus x, z, u, l, h, w, n.

Syntax rules:

— A single digit represents 4 bits in hexadecimal base, 3 bits in octal base and 1 bit in binary base. Similarly, the letters x, z, u, l, h, w, n represent 4 identical bits (for hexadecimal), 3 identical bits (for octal), or 1 bit (for binary). For example, 8’h1x is equivalent to 8’b0001xxxx.
— If the size of the value is smaller than the width, the value is padded to the left. A most significant bit (MSB) of 0 or 1 causes zero padding. If the MSB of the literal is x, z, u, l, h, w or n, then that mvl value is used for padding.

— If the size of the value is larger than the size specified for the list, the value is truncated leaving the LSB of the literal.
— An underscore can be used for breaking up long numbers to enhance readability. It is legal inside the size and inside the value. It is illegal at the beginning of the literal, between the base and the value, and between the single quote (') and the base.

### Examples

```
32'hffffxxxx
32'HFFFFXXXX
//16'_b1100uuuuu    --illegal because (_) is between (‘) and base
19'oL0001
14'D123
64'bz_1111_0000_1111_0000
```

### Notes

— Decimal literals are not supported.
— White space is not allowed as a separator between the width number and base or between the base and the value.
— The base and the value are not case sensitive.
— Size and base have to be specified.
— In the context of a Verilog comparison operator (!== or ===) or HDL tick access ('data' = 32'bx), only the 4-value subset is supported (0, 1, u, x).
— Verilog simulators support only the 4-value logic subset.
— An MVL literal of size 1 is of type list of mvl that has one element. It is not of type mvl. Thus, you cannot assign an MVL literal to a variable or field of type mvl:

```
var m: mvl = 1'bz; -- illegal; MVL_Z must be assigned
```

Syntactically, the same expression may be of a numeric type or MVL literal. For example, 1’b1 may represent either the number 1 or a list of MVL with the value {MVL_1}. A literal is considered to be an MVL literal when:

— The literal is assigned to a list of mvl, for example:

```
var v2: list of mvl = 16'b1;
```

— When the literal is passed to a method that receives a list of mvl
— When the literal is assigned to a port of type list of mvl using the $ operator
— When the literal is compared to list of mvl, for example:

```
check that v == 4'buuuu;
```

— When the literal is compared using the === and !== operators, for example:

```
check that 's' === 4'bz; -- limited to the 4-value subset
```

— When the literal is used in an HDL tick access assignment, for example:

```
's' = 8'bx1z; -- limited to the 4-value subset
```

— When the literal is an argument for a Verilog task, for example:

```
'task1'(8'h1x);)
```

— When the literal is used in a list operation, for example

```
l.add(32'b0)
```

If the type of the expression, according to the context, is numeric, or if the type cannot be extracted from the context, the default type remains uint, for example:

```
print 2'b11; -- prints unsigned integer value 3
print 2'bxx; -- syntax error
pack(NULL, 32'z) -- error
```

NOTE— The type-casting operations **as_a()** and **is a** do not propagate the context.

### See Also

— "Scalar Types" on page 75

### 2.1.4.4 Predefined Constants

A set of constants is predefined in ***e***, as shown in Table 2-3.

**Table 2-3—Predefined Constants**

| Constant | Description |
|---|---|
| TRUE | For boolean variables and expressions. |
| FALSE | For boolean variables and expressions. |
| NULL | For structs, specifies a NULL pointer. For character strings, specifies an empty string. |
| UNDEF | UNDEF indicates NONE where an index is expected. |
| MAX_INT | Represents the largest 32-bit **int** ($2^{31}$ -1) |
| MIN_INT | Represents the smallest 32-bit **int** ($-2^{31}$). |
| MAX_UINT | Represents the largest 32-bit **uint** ($2^{32}$-1). |

### See Also

— "Literals and Constants" on page 4

### 2.1.4.5 Literal String

A literal string is a sequence of zero or more ASCII characters enclosed by double quotes (" ").

The special escape sequences shown in Table 2-4 are allowed.

**Table 2-4—Escape Sequences in Strings**

| Escape Sequence | Meaning |
|---|---|
| \n | New-line |
| \t | Tab |
| \f | Form-feed |
| \" | Quote |
| \\ | Backslash |
| \r | Carriage-return |

#### Example

This example shows escape sequences used in strings.

```
extend sys {

    m() is {
        var header: string =
          "Name\tSize in Bytes\n----\t-------------\n";
        var p: packet = new;
        var pn: string = p.type().name;
        var ps: uint = p.type().size_in_bytes;
        outf("%s%s\t%d", header, pn, ps);
    };
};
```

#### Result

```
Name    Size in Bytes
----    -------------
packet  20
```

#### See Also

— “Literals and Constants” on page 4

### 2.1.4.6 Literal Character

A literal character is a single ASCII character, enclosed in quotation marks and preceded by ***0c***. This expression evaluates to the integer value that represents this character. For example, the literal character shown below is the single ASCII character “a” and evaluates to 0x0061.

```
var u: uint(bytes:2) = 0c"a"
```

NOTE— Literal characters can only be assigned to integers or unsigned integers without explicit casting.

#### See Also

— “Literals and Constants” on page 4

## 2.1.5 Names, Keywords, and Macros

The following sections describe the legal syntax for names and macros:

— “Legal e Names” on page 9
— “e Keywords” on page 10
— “Macros” on page 11

### 2.1.5.1 Legal *e* Names

User-defined names in ***e*** code consist of a case-sensitive combination of any length of the characters A-Z, a-z, 0-9, and underscore. They must begin with a letter. Names beginning with an underscore have a special meaning in ***e*** and are not recommended for general use. Names beginning with a number are not allowed.

The syntax of an ***e*** module name (a file name) is the same as the syntax of UNIX file names, with the following exceptions:

— '@' and '~' are not allowed as the first character of a file name.
— '[', ']', '{', '}' are not allowed in file names.
— Only one '.' is allowed in a file name.

NOTE— Many ASCII characters are not handled correctly by some UNIX commands when used in file names. These characters include control characters, spaces, and characters reserved for command line parsing, such as '-', '|', and '<'.

NOTE— Naming an ***e*** module "patch.e" or "test.e" can cause problems when you try to load the compiled file. If the module is to be compiled, do not name it patch.e or test.e.

### 2.1.5.2 *e* Keywords

The keywords listed below are the components of the ***e*** language. Some of the terms are keywords only when used together with other terms, such as "key" in "**list(key**:***key*****)**", "before" in "**keep gen *x* before *y***", or "computed" in "**define *def* as computed**".

| | | | | |
|---|---|---|---|---|
| all of | all_values | and | as a | as_a |
| assert | assume | async | attribute | before |
| bit | bits | bool | break | byte |
| bytes | c export | case | change | check that |
| compute | computed | consume | continue | cover |
| cross | cvl call | cvl callback | cvl method | cycle |
| default | define | delay | detach | do |
| down to | dut_error | each | edges | else |
| emit | event | exec | expect | extend |
| fail | fall | file | first of | for |
| force | from | gen | global | hdl pathname |
| if | #ifdef | #ifndef | in | index |
| int | is | is a | is also | is c routine |
| is empty | is first | is inline | is instance | is not a |
| is not empty | is only | is undefined | item | keep |
| keeping | key | like | line | list of |
| matching | me | nand | new | nor |
| not | not in | now | on | only |
| or | others | pass | prev | print |
| range | ranges | release | repeat | return |
| reverse | rise | routine | select | session |
| soft | start | state machine | step | struct |
| string | sync | sys | that | then |
| time | to | transition | true | try |
| type | uint | unit | until | using |

| | | | | |
|---|---|---|---|---|
| var | verilog code | verilog function | verilog import | verilog simulator |
| verilog task | verilog time | verilog timescale | verilog trace | verilog variable |
| vhdl code | vhdl driver | vhdl function | vhdl procedure | vhdl driver |
| vhdl simulator | vhdl time | when | while | with |
| within | | | | |

#### 2.1.5.3 Macros

***e*** macros (created with the **define** statement) can be defined with or without an initial ` character. There are two important characteristics of ***e*** macros defined with an initial ` character:

— They share the same name space as Verilog macros.
— You must always include the ` character when you reference the name.

Thus, if you import a file of Verilog macros containing the following macro:

```
`define WORD_WIDTH 8
```

defining the following ***e*** macro results in a name conflict:

```
define `WORD_WIDTH 16;
```

With either macro defined, the correct way to reference it is as follows:

```
struct t {
    f: uint (bits: `WORD_WIDTH);
};
```

**See Also**

— Chapter 20, “Preprocessor Directives”

## 2.2 Syntactic Elements

Every ***e*** construct belongs to a construct category that determines how the construct can be used. There are four categories of ***e*** constructs:

| | |
|---|---|
| Statements | Statements are top-level constructs and are valid within the begin-code **<'** and end-code **'>** markers. See “Statements” on page 12 for a list and brief description of ***e*** statements. |
| Struct members | Struct members are second-level constructs and are valid only within a struct definition. See “Struct Members” on page 13 for a list and brief description of ***e*** struct members. |
| Actions | Actions are third-level constructs and are valid only when associated with a struct member, such as a method or an event. See “Actions” on page 14 for a list and brief description of ***e*** actions. |
| Expressions | Expressions are lower-level constructs that can be used only within another ***e*** construct. See “Expressions” on page 19 for a list and brief description of ***e*** expressions. |

The syntax hierarchy roughly corresponds to the level of indentation shown below:

```
statements
    struct members
        actions
            expressions
```

### See Also

— "Statements" on page 12
— "Struct Members" on page 13
— "Actions" on page 14
— "Expressions" on page 19

## 2.2.1 Statements

Statements are the top-level syntactic constructs of the ***e*** language and perform the functions related to extending the ***e*** language and interface with the simulator.

Statements are valid within the begin-code **<'** and end-code **'>** markers. They can extend over several lines and are separated by semicolons. For example, the following code segment has two statements:

```
<'
    import bypass_bug72;
    import cpu_test0012;
'>
```

In general, within a given ***e*** module, statements can appear in any order except that **import** statements must appear before any other statements. No statements other than **verilog import**, preprocessor directives or defines (**#ifdef**, **#ifndef**, **define**, **define as**, **define as computed**) can precede **import** statements. See "import" on page 635 for an example of a special case where this restriction also applies to **import** statements in different ***e*** modules.

Here is the complete list of ***e*** statements:

| | |
|---|---|
| struct | Defines a new data structure. See "Defining Structs: struct" on page 118. |
| type | Defines an enumerated data type or scalar subtype. See "type enumerated scalar" on page 98, "type scalar subtype" on page 100, or "type sized scalar" on page 101 |
| extend | Modifies a previously defined struct or type. See "Extending Structs: extend type" on page 121 or "extend type" on page 103 |
| define | Extends the ***e*** language by defining new commands, actions, expressions, or any other syntactic element. See Chapter 13, "Macros", "define as" on page 429, or "define as computed" on page 436. |
| #ifdef, #ifndef | Used together with define statements to place conditions on the ***e*** parser. See "#ifdef, #ifndef" on page 627. |
| import | Reads in an ***e*** file. See "import" on page 635. |

| | |
|---|---|
| verilog import | Reads in Verilog macro definitions from a file. See“verilog import” on page 799 . |
| verilog code | Writes Verilog code to the stubs file, which is used to interface ***e*** programs with a Verilog simulator. See “verilog code” on page 795. |
| verilog time | Specifies Verilog simulator time resolution. See“verilog time” on page 803 . |
| verilog variable reg \| wire | Specifies a Verilog register or wire that you want to drive from ***e***. See “verilog variable reg \| wire” on page 804. |
| verilog variable memory | Specifies a Verilog memory that you want to access from ***e***. See “verilog variable memory” on page 810. |
| verilog function | Specifies a Verilog function that you want to call from ***e***. See “verilog function” on page 797. |
| verilog task | Specifies a Verilog task that you want to call from ***e***. See “verilog task” on page 801. |
| vhdl code | Writes VHDL code to the stubs file, which is used to interface ***e*** programs with a VHDL simulator. See “vhdl code” on page 813. |
| vhdl driver | Used to drive a VHDL signal continuously via the resolution function. See “vhdl driver” on page 815. |
| vhdl function | Declares a VHDL function defined in a VHDL package. See “vhdl function” on page 819. |
| vhdl procedure | Declares a VHDL procedure defined in a VHDL package. See “vhdl procedure” on page 822. |
| vhdl time | Specifies VHDL simulator time resolution. See “vhdl time” on page 829. |

### See Also

— “Struct Members” on page 13
— “Actions” on page 14
— “Expressions” on page 19

## 2.2.2 Struct Members

Struct member declarations are second-level syntactic constructs of the ***e*** language that associate the entities of various kinds with the enclosing struct.

Struct members can only appear inside a struct type definition statement (see “Defining Structs: struct” on page 118). They can extend over several lines and are separated by semicolons. For example, the following struct “packet” has two struct members, len and data:

```
<'
struct packet{
    %len: int;
    %data[len]: list of byte;
};
'>
```

A struct can contain multiple struct members of any type in any order. Here is a brief description of ***e*** struct members:

| | |
|---|---|
| field declaration | Defines a data entity that is a member of the enclosing struct and has an explicit data type. |
| method declaration | Defines an operational procedure that can manipulate the fields of the enclosing struct and access runtime values in the DUT. |
| subtype declaration | Defines an instance of the parent struct in which specific struct members have particular values or behavior. |
| constraint declaration | Influences the distribution of values generated for data entities and the order in which values are generated. |
| coverage declaration | Defines functional test goals and collects data on how well the testing is meeting those goals. |
| temporal declaration | Defines ***e*** events and their associated actions. |

### See Also

— "Defining Fields: field" on page 125
— "Rules for Defining and Extending Methods" on page 459
— "Creating Subtypes with When" on page 133
— "Defining Constraints" on page 270
— "Defining Coverage Groups: cover" on page 373
— Chapter 10, "Temporal Struct Members"

## 2.2.3 Actions

***e*** actions are lower-level procedural constructs that can be used in combination to manipulate the fields of a struct or exchange data with the DUT.

Actions can extend over several lines and are separated by semicolons. An action block is a list of actions separated by semicolons and enclosed in curly brackets, **{ }**.

Actions must be associated with a struct member, specifically a method or an event, or issued interactively as commands at the command line. Here is an example of an action (an invocation of a method, "transmit()") associated with an event, xmit_ready. Another action, **out()** is associated with the transmit() method.

```
<'
struct packet{
    event xmit_ready is rise('top.ready');
    on xmit_ready {transmit();};
    transmit() is {
      out("transmitting  packet...");
    };
};
'>
```

The following sections describe the ***e*** actions:

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Executing Actions Iteratively" on page 16

- “Controlling Program Flow” on page 16
- “Invoking Methods and Routines” on page 17
- “Performing Time-Consuming Actions” on page 17
- “Generating Data Items” on page 18
- “Detecting and Handling Errors” on page 18
- “Printing” on page 18

### 2.2.3.1 Creating or Modifying Variables

| | |
|---|---|
| “var” on page 487 | Defines a local variable. |
| “=” on page 489 | Assigns or samples values of fields, local variables, or HDL objects. |
| “op=” on page 491 | Performs a complex assignment (such as add and assign, or shift and assign) of a field, local variable, or HDL object. |
| “force” on page 830 | Forces a Verilog net or wire to a specified value, over-riding the value from driven from the DUT. |
| “release” on page 834 | Releases the Verilog net or wire that was previously forced. |

#### See Also

- “Executing Actions Conditionally” on page 15
- “Executing Actions Iteratively” on page 16
- “Controlling Program Flow” on page 16
- “Invoking Methods and Routines” on page 17
- “Performing Time-Consuming Actions” on page 17
- “Generating Data Items” on page 18
- “Detecting and Handling Errors” on page 18
- “Printing” on page 18

### 2.2.3.2 Executing Actions Conditionally

| | |
|---|---|
| “if then else” on page 533 | Executes an action block if a condition is met and a different action block if it is not. |
| “case labeled-case-item” on page 534 | Executes one action block out of multiple action blocks depending on the value of a single expression. |
| “case bool-case-item” on page 536 | Evaluates a list of boolean expressions and executes the action block associated with the first expression that is true. |

#### See Also

- “Creating or Modifying Variables” on page 15
- “Executing Actions Iteratively” on page 16
- “Controlling Program Flow” on page 16
- “Invoking Methods and Routines” on page 17
- “Performing Time-Consuming Actions” on page 17
- “Generating Data Items” on page 18
- “Detecting and Handling Errors” on page 18
- “Printing” on page 18

### 2.2.3.3 Executing Actions Iteratively

| | |
|---|---|
| "while" on page 538 | Executes an action block repeatedly until a boolean expression becomes FALSE. |
| "repeat until" on page 539 | Executes an action block repeatedly until a boolean expression becomes TRUE. |
| "for each in" on page 540 | For each item in a list that is a specified type, executes an action block. |
| "for from to" on page 543 | Executes an action block for a specified number of times. |
| "for" on page 544 | Executes an action block for a specified number of times. |
| "for each line in file" on page 545 | Executes an action block for each line in a file. |
| "for each file matching" on page 546 | Executes an action block for each file in the search path. |

**See Also**

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Controlling Program Flow" on page 16
— "Invoking Methods and Routines" on page 17
— "Performing Time-Consuming Actions" on page 17
— "Generating Data Items" on page 18
— "Detecting and Handling Errors" on page 18
— "Printing" on page 18

### 2.2.3.4 Controlling Program Flow

| | |
|---|---|
| "break" on page 547 | Breaks the execution of the enclosing loop. |
| "continue" on page 548 | Stops execution of the enclosing loop and continues with the next iteration of the same loop. |

**See Also**

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Executing Actions Iteratively" on page 16
— "Invoking Methods and Routines" on page 17
— "Performing Time-Consuming Actions" on page 17
— "Generating Data Items" on page 18
— "Detecting and Handling Errors" on page 18
— "Printing" on page 18

### 2.2.3.5 Invoking Methods and Routines

| | |
|---|---|
| "method()" on page 478 | Calls a regular method. |
| "tcm()" on page 475 | Calls a TCM. |
| "start tcm()" on page 477 | Launches a TCM as a new thread (a parallel process). |
| "Calling Predefined Routines: routine()" on page 793 | Calls an ***e*** predefined routine. |
| "compute method()" on page 480 | Calls a value-returning method without using the value returned. |
| "return" on page 481 | Returns immediately from the current method to the method that called it. |

#### See Also

- "Creating or Modifying Variables" on page 15
- "Executing Actions Conditionally" on page 15
- "Executing Actions Iteratively" on page 16
- "Controlling Program Flow" on page 16
- "Performing Time-Consuming Actions" on page 17
- "Generating Data Items" on page 18
- "Detecting and Handling Errors" on page 18
- "Printing" on page 18

### 2.2.3.6 Performing Time-Consuming Actions

| | |
|---|---|
| "emit" on page 307 | Causes a named event to occur. |
| "sync" on page 365 | Suspends execution of the current TCM until the temporal expression succeeds. |
| "wait" on page 367 | Suspends execution of the current time-consuming method until a given temporal expression succeeds. |
| "all of" on page 369 | Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **all of** action is reached only when all branches of the **all of** have been fully executed. |
| "first of" on page 370 | Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **first of** action is reached when any of the branches in the **first of** has been fully executed. |
| "state machine" on page 883 | Defines a state machine. |

#### See Also

- "Creating or Modifying Variables" on page 15
- "Executing Actions Conditionally" on page 15
- "Executing Actions Iteratively" on page 16
- "Controlling Program Flow" on page 16
- "Invoking Methods and Routines" on page 17
- "Generating Data Items" on page 18
- "Detecting and Handling Errors" on page 18
- "Printing" on page 18

#### 2.2.3.7 Generating Data Items

| | |
|---|---|
| "gen" on page 296 | Generates a value for an item, while considering all relevant constraints. |

**See Also**

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Executing Actions Iteratively" on page 16
— "Controlling Program Flow" on page 16
— "Invoking Methods and Routines" on page 17
— "Performing Time-Consuming Actions" on page 17
— "Detecting and Handling Errors" on page 18
— "Printing" on page 18

#### 2.2.3.8 Detecting and Handling Errors

| | |
|---|---|
| "check that" on page 441 | Checks the DUT for correct data values. |
| "dut_error()" on page 443 | Defines a DUT error message string. |
| "assert" on page 456 | Issues an error message if a specified boolean expression is not true. |
| "warning()" on page 450 | Issues a warning message. |
| "error()" on page 451 | Issues an error message when a user error is detected. |
| "fatal()" on page 452 | Issues an error message, halts all activities, and exits immediately. |
| "try" on page 454 | Catches errors and exceptions. |

**See Also**

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Executing Actions Iteratively" on page 16
— "Controlling Program Flow" on page 16
— "Invoking Methods and Routines" on page 17
— "Performing Time-Consuming Actions" on page 17
— "Generating Data Items" on page 18
— "Printing" on page 18

#### 2.2.3.9 Printing

| | |
|---|---|
| "set_config()" on page 766 | Sets options for various categories, including printing. |

**See Also**

— "Creating or Modifying Variables" on page 15
— "Executing Actions Conditionally" on page 15
— "Executing Actions Iteratively" on page 16
— "Controlling Program Flow" on page 16
— "Invoking Methods and Routines" on page 17
— "Performing Time-Consuming Actions" on page 17
— "Generating Data Items" on page 18
— "Detecting and Handling Errors" on page 18

### 2.2.4 Expressions

Expressions are constructs that combine operands and operators to represent a value. The resulting value is a function of the values of the operands and the semantic meaning of the operators.

A few ***e*** expressions, such as expressions that restrict the range of valid values of a variable, must evaluate to constants at compile time. More typically, expressions are evaluated at run time, resolved to a value of some type, and assigned to a variable or field of that type. Strict type checking in ***e*** is enforced.

Each expression must contain at least one operand, which can be:

- A literal value
- A constant
- An ***e*** entity, such as a method, field, list, or struct
- An HDL entity, such as a signal

A compound expression applies one or more operators to one or more operands.

#### See Also

- Chapter 3, “Data Types”

## 2.3 Struct Hierarchy and Name Resolution

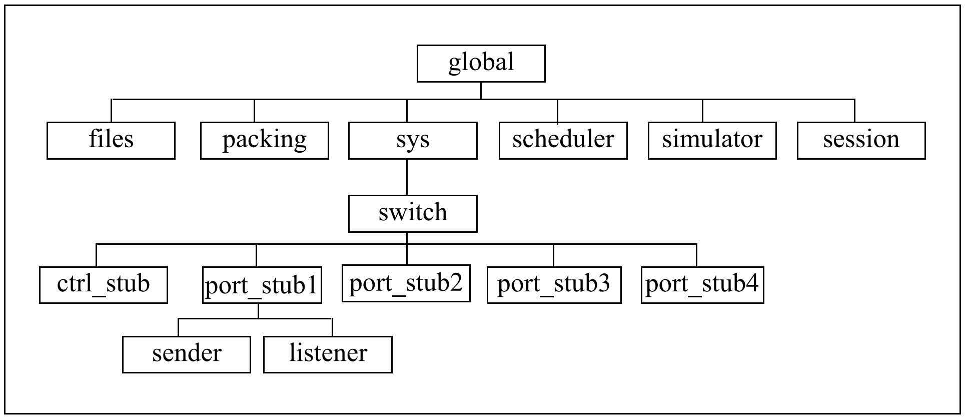
The following sections explain the struct hierarchy of an ***e*** program and how to reference entities within the program:

- “Struct Hierarchy” on page 19
- “Referencing e Entities” on page 21
- “Implicit Variables” on page 24
- “Name Resolution Rules” on page 26

### 2.3.1 Struct Hierarchy

Because structs can be instantiated as the fields of other structs, a typical ***e*** program has many levels of hierarchy. Every ***e*** program contains several predefined structs as well as user-defined structs. Figure 2-1 on page 20 shows the partial hierarchy of a typical ***e*** program. The predefined structs are shown in **bold**.

**Figure 2-1—Diagram of Struct Hierarchy**



### 2.3.1.1 Global Struct

The predefined struct **global** is the root of all ***e*** structs. All predefined structs and most predefined methods are part of the **global** struct.

It is highly recommended that you do not extend the **global** struct.

### 2.3.1.2 Sys Struct

The system struct is instantiated under **global** as **sys**.

All fields and structs in **sys** not marked by an exclamation point (!) are generated automatically during the **generate_test** phase. Any structs or fields outside of **sys** that need generation must be generated explicitly.

Time is stored in a 64-bit integer field named **sys.time**. When ***e*** is linked with an event-driven simulator, **sys.time** shows the current simulator time. When ***e*** is linked with a cycle-based simulator, **sys.time** shows the current simulator cycle. **sys.time** is influenced by the current timescale. See "verilog time" on page 803 and "vhdl time" on page 829 for information on how the timescale is determined.

### 2.3.1.3 Packing Struct

Packing and unpacking are controlled by a predefined struct under **global** named **packing**. Packing and unpacking prepare ***e*** data sent to or received from the DUT. Under the packing struct are five predefined structs. You can create your own packing order by copying one of these structs and modifying one or more of its parameters.

### 2.3.1.4 Files Struct

The files struct provides predefined methods for manipulating files.

### 2.3.1.5 Scheduler Struct

The **scheduler** struct contains predefined methods that allow you to access active TCMs and terminate them.

### 2.3.1.6 Simulator Struct

The **simulator** struct controls the HDL simulator and has a predefined method that allows access to Verilog macros at run time.

### 2.3.1.7 Session Struct

The **session** struct holds the status of the current simulator session, related information, and events. Fields available in the **session** struct that are of general interest include:

— session.user_time
— session.system_time
— session.check_ok
— session.events

The first three fields listed above help you determine the time and memory used in a particular session. The following sections describe the **check_ok** field and the **events** field.

#### 2.3.1.7.1 session.check_ok

This field is of boolean type, and is set TRUE after every check, if the check succeeds. Otherwise, it is set to FALSE. This field allows you to extend checking of a behavior without the need to duplicate the **if** clause.

The following example show how this is accomplished.

```
post_generate() is also {
  check that mlist.size() > 0 else dut_error("Empty list");
  if session.check_ok then {
   check that mlist[0] == 0xa else dut_error("Error at index 0");
  };
};
```

#### 2.3.1.7.2 session.events

This field contains the names of all user-defined events that occurred during the test, and how many times each user-defined event occurred. The name of the event is preceded by the struct type and a double underscore:

*struct_type__event_name*

If an event is defined in a when subtype, the name of the event in the **session.events** field is prefixed by the subtype and a double underscore:

*subtype__struct_type__event_name*

### 2.3.2 Referencing *e* Entities

The following sections describe how to reference ***e*** entities:

— "Structs and Fields" on page 22
— "Method and Routine Names" on page 23
— "Enumerated Type Values" on page 23

### 2.3.2.1 Structs and Fields

Any user-defined struct can be instantiated as a field of the **sys** struct or of another struct. Thus every instantiated struct and its fields have a place in the struct hierarchy and their names include a path reflecting that place.

The **keep** constraints in the following example show the use of paths to identify the fields u and kind:

```
<'
struct switch {
    ctrl: ctrl_stub;
    port: port_stub;

    keep soft port.sender.cell.u == 0xff;
    keep ctrl.init_command.kind == RD;
};
struct ctrl_stub {
    init_command: ctrl_cmd;
};
struct simplex {
    kind: [TX, RX];
    cell: cell;
};
struct port_stub {
    sender: TX simplex;
    listener: RX simplex;
};
struct cell {
    u: uint;
};
struct ctrl_cmd {
    kind: [RD, WR];
    addr: int;
};
extend sys {
    switch : switch;
};
'>
```

#### Notes

— The name of the **global** struct can be omitted from the path to a field or a struct.

— The name of the enclosing struct is not included in the path if the current struct is the enclosing struct. For example, prefixing the name port.sender.cell.u in the example above with the name of the enclosing struct, switch, is an error.

— In certain contexts, you can use the implicit variables **me** or **it** in the path to refer to the enclosing struct. For example, prefixing the name port.sender.cell.u in the example above with **me** is legal. See "Implicit Variables" on page 24 for more information.

— A special syntax is required to reference struct subtypes and fields under struct subtypes. This syntax is described in "Struct Subtypes" on page 80.

#### See Also

— "Struct Subtypes" on page 80

— "Implicit Variables" on page 24

— "." on page 71

### 2.3.2.2 Method and Routine Names

The names of all methods and routines must be followed immediately by parentheses, both when you define the method and when you call it.

The predefined methods of any struct, such as **pre_generate()** or **init()**, and all user-defined methods, are associated with a particular struct. Thus, like structs and fields, every user-defined method has a place in the struct hierarchy and its name includes a path reflecting that place.

The example below illustrates the names used to call user-defined and predefined methods.

```
<'
struct meth {
    %size: int;
    %taken: int;

    get_free(size: int, taken: int): int is inline {
        result = size - taken;};
};
extend sys {
    !area: int;
    mi: meth;

    post_generate() is also {
        sys.area = sys.mi.get_free(sys.mi.size, sys.mi.taken);
        print sys.area;
    };
};
'>
```

Some predefined methods, such as the methods used to manipulate lists, are pseudo-methods. They are not associated with a particular struct. These methods are called by appending their name to the expression that you want to manipulate. Here is an example of how to call the list pseudo-method **.size()**:

```
<'
struct meth {
    %data: list of int;

    keep data.size() <= 10;
};
'>
```

User-defined routines, like predefined routines, are associated with the **global** struct. You can omit **global** from the path when the context is unambiguous. See "Name Resolution Rules" on page 26 for more information.

#### See Also

— "Invoking Methods and Routines" on page 17

### 2.3.2.3 Enumerated Type Values

Names for enumerated type values must be unique within each type. For example, defining a type as "my_type: [a, a, b]" results in an error because the name "a" is not unique.

However, the same name can be used in more than one enumerated type. For example, the following two enumerated types define the same value names:

```
type destination: [a, b, c, d];
type source: [a, b, c, d];
```

To refer to an enumerated type value in a struct where no values are shared between the enumerated types, you can use just the value name. In structs where more than one enumerated field can have the same value, you must use the following syntax to refer to the value when the type is not clear from the context:

```
type_name'value
```

In the following **keep** constraint, it is clear that the type of “dest” is “destination”, so you can use just the value name “b”:

```
type destination: [a, b, c, d];
type source: [a, b, c, d];
struct packet {
    dest: destination;
    keep me.dest == b;
```

However, because the type of the variable “tmp” below is not specified, it is necessary to use the full name for the enumerated type value “destination'b”:

```
m() is {
    var tmp := destination'b;
};
```

### See Also

— “Enumerated Scalar Types” on page 77

## 2.3.3 Implicit Variables

Many ***e*** constructs create implicit variables. The scope of these implicit variables is the construct that creates them. Two of these implicit variables, **me** and **it**, are used in pathnames when referencing ***e*** entities.

This section describes the implicit variables:

— “it” on page 24
— “me” on page 25
— “result” on page 26
— “index” on page 26

NOTE— With the exception of **result**, you cannot assign values to implicit variables. An assignment such as “**me** = packet” generates an error.

### 2.3.3.1 it

The constructs that create the implicit variable **it** are:

— list pseudo-methods
— for each
— gen...keeping

— keep for each
— keep .is_all_iterations()
— new with
— list with key declaration

The implicit variable **it** always refers to the current item.

Wherever **it.*field*** can be used, the shorthand notation ***.field*** can be used in its place. For example, **it**.len can be abbreviated to .len, with a leading dot. A typical use of **it** is to refer to each item in a list within a loop.

```
for each in sys.packets{
    it.len = 5;
    .good = TRUE;
};
```

In the code above, .good is shorthand for **it**.good. The scope of the **it** variable is restricted to the **for** loop.

In many places it is legal to designate and use a name other than the implicit **it**. In the following example, **it** is replaced with a variable name, “p”, that is declared in the iterating action.

```
for each (p) in sys.packets do {
    print p.len;
};
```

### See Also

— “Implicit Variables” on page 24

### 2.3.3.2 me

The implicit variable **me** refers to the current struct and can be used anywhere in the struct. In the following example, **me** refers to the current instance of the packet struct, and **it** refers to the current value of tmp.

```
struct packet {
    data: uint;
    stm() is {
        var tmp: uint;
        gen tmp keeping {it < me.data};
        print data, tmp using hex;
    };
};
```

When referring to a field from another member of the same struct, the **me**. can be omitted. In the **keep** constraint shown below, the name “**me**.header.dest” is equivalent to the name “header.dest”.

```
struct packet {
    %header : header;

    keep header.dest == 0x55;
};
struct header {
    %dest     : int (bits : 8);
};
```

### See Also

— "Implicit Variables" on page 24

#### 2.3.3.3 result

The **result** variable returns a value of the method's return type. If no **return** action is encountered, **result** is returned by default. The following method returns the sum of "a" and "b":

```
sum(a: int, b: int): int is {
    result = a + b;
};
```

### See Also

— "Implicit Variables" on page 24

#### 2.3.3.4 index

The constructs that create the implicit variable **index** are:

— list pseudo-methods
— for each
— keep for each

The **index** variable is a non-negative integer that holds the current index of the item referred to by **it**. The scope of the **index** variable is limited to the action block.

The following loop assigns 5 to the len field of every item in the packets list and also assigns the **index** value of each item to its id field.

```
for each in packets do {
    packets[index].len = 5;
    .id = index;
};
```

### See Also

— "Implicit Variables" on page 24

### 2.3.4 Name Resolution Rules

The following sections describe how names are resolved, depending on whether the names include a path or not.

— "Names that Include a Path" on page 26
— "Names that Do Not Include a Path" on page 27

#### 2.3.4.1 Names that Include a Path

To resolve names that include a path, an entity of that name is searched for at the specified scope and an error message is issued if it is not found. In the following example, the names "sys.b.u" and ".u" in the **keep** constraints cannot be resolved, and an error is issued an error if those names are not commented out.

```
<'
struct b {
    u:uint;

    m() is {
        print u;
    };
};
struct c {
    u:uint;

    keep u > sys.bi.u;
    keep me.u > 5;
    -- keep u < sys.b.u;          // 'sys' does not have a field 'b'
    -- keep .u < sys.bi.u;        // no such variable 'it'

    m() is {
        print u;
    };
};

extend sys {
    bi: b;
    ci: c;

    post_generate() is also {
        sys.bi.m();
        ci.m();
    };
};
'>
```

NOTE— If the path begins with a period (.), the path is assumed to begin with the implicit variable **it**.

### See Also

— “Names that Do Not Include a Path” on page 27

### 2.3.4.2 Names that Do Not Include a Path

To resolve names that do not include a path, the following checks are performed, in order. The program stops after the first check that identifies the named object.

1) Check whether the name is a macro. If there are two macro definitions, choose the most recent one.
2) Check whether the name is one of the predefined constants. There cannot be two identical predefined constants.
3) Check whether the name is an enumerated type. There cannot be two identical enumerated types.
4) Check whether the name identifies a variable used in the current action block. If not, and if the action is nested, check whether the name identifies a variable in the enclosing action block. If not, this search continues from the immediately enclosing action block outwards to the boundary of the method.
5) Check whether the name identifies a member of the current struct:
    If the expression is inside a struct definition, the current struct is the enclosing struct.

If the expression is inside a method, the current struct is the struct to which the method belongs.

6) Check whether the name identifies a member of the **global** struct.
7) If the name is still unresolved, an error message is issued.

### Example

The following example illustrates how variables in the inner scopes hide those in the outer scopes:

```
m() is {
    var x: int = 6;
    if x > 4 then {
        var x: bool = TRUE;
        print x;
    };
    print x;
};
```

### Result

```
x = TRUE
x = 6
```

NOTE— Macros, predefined constants, and enumerated types have “global scope”, Which means they can be seen from anywhere within an *e* program. For that reason, their names must be unique:

— No two name macros can have the same ***name***, and no two replacement macros can have the same ***macro-name'nonterminal-type*** (Chapter 13, “Macros”).
— No user-defined constant can have the same name as a predefined constant (“Predefined Constants” on page 8).
— No two enumerated types can have the same ***enum-type-name*** (“Defining and Extending Scalar Types” on page 98).

### See Also

— “Names that Include a Path” on page 26

## 2.4 Operator Precedence

The following table summarizes all *e* operators in order of precedence. The precedence is the same as in the C language, with the exception of operators that do not exist in C. To change the order of computation, place parentheses around the expression that should be computed first.

**Table 2-5—Operators in Order of Precedence**

| Operator | Operation Type |
|---|---|
| “[ ]” on page 54 | List indexing (subscripting) |
| “[ .. ]” on page 58 | List slicing |
| “[ : ]” on page 55 | Bit slicing (selection) |

**Table 2-5—Operators in Order of Precedence *(continued)***

| Operator | Operation Type |
|---|---|
| f(...) | Method and routine calls (see "Invoking Methods and Routines" on page 17) |
| "." on page 71 | Field selection |
| "~" on page 31, "! (not)" on page 35 | Bitwise not, boolean not |
| "{... ; ...}" on page 60 | List concatenation |
| "%{... , ...}" on page 62 | Bit concatenation |
| "Unary + -" on page 40 | Unary plus, minus |
| *, /, % | Binary multiply, divide, modulus (see "+ - * / %" on page 41) |
| +, - | Binary add and subtract (see "+ - * / %" on page 41) |
| ">> <<" on page 33 | Shift right, shift left |
| "< <= > >=" on page 42 | Comparison |
| "is [not] a" on page 67 | Subtype identification |
| "== !=" on page 43 | Equality, inequality |
| "=== !==" on page 45 | Verilog four-state comparison |
| "~ !~" on page 47 | String matching |
| "in" on page 49 | Range list operator |
| & | Bitwise and (see "& \| ^" on page 32) |
| \| | Bitwise or (see "& \| ^" on page 32) |
| ^ | Bitwise xor (see "& \| ^" on page 32) |
| "&& (and)" on page 36 | boolean and |
| "\|\| (or)" on page 37 | boolean or |
| "=>" on page 37 | boolean implication |
| "? :" on page 73 | Conditional operator ("a ? b : c" means "if a then b else c") |

NOTE— Every operation in ***e*** is performed within the context of types and is carried out either with 32-bit precision or unbounded precision.

### See Also

— Chapter 3, “Data Types” for information on the precision of operations and assignment rules
— “Evaluation Order of Expressions” on page 30

## 2.5 Evaluation Order of Expressions

In ***e*** it is defined that “and” (&&) and “or” (||) use left-to-right lazy evaluation. Consider the following statement:

```
bool_1 = foo(x) && bar(x)
```

If foo(x) returns TRUE, then bar(x) will be evaluated as well, to determine whether bool_1 gets TRUE. If, however, foo(x) returns FALSE, then bool_1 gets FALSE immediately, and bar(x) is not executed. The argument to bar(x) is not even evaluated.

Expressions containing || are likewise evaluated in a lazy fashion: If the subexpression on the left of the “or” operator is TRUE, then the subexpression on the right is ignored.

Although ***e*** was implemented to use left-to-right evaluation for both compiled ***e*** code and interpreted ***e*** code, that evaluation order is not required by the language definition for operators other than && or ||.

Take for example the following statement:

```
bool_2 = foo(x) + bar(x)
```

If foo(x) or bar(x) has side effects (that is, if foo(x) changes the value of x or bar(x) changes the value of x), then the results of foo(x) + bar(x) might depend on which of the two subexpressions, foo(x) or bar(x), is evaluated first, so the results are not predictable according to the ***e*** language definition. Practically, the left-to-right evaluation implemented in ***e*** assures predictable results, but that order is not guaranteed for other compilers. Writing code that depends on evaluation order should be avoided.

### See Also

— “Operator Precedence” on page 28

## 2.6 Bitwise Operators

The following sections describe the ***e*** bitwise operators:

| | |
|---|---|
| “~” on page 31 | The bitwise unary negation operator changes each 0 bit to 1 and each 1 bit to 0 in a single expression. |
| “& \| ^” on page 32 | The binary bitwise AND, OR, and XOR operators compare each bit in one expression with the corresponding bit in a second expression to calculate the result. |
| “>> <<” on page 33 | The shift-right and shift-left operators shift the bits in an expression to the left or right a specified number of bits. |

### See Also

— “bitwise_op()” on page 730

## 2.6.1 ~

### Purpose

Unary bitwise negation

### Category

Expression

### Syntax

~*exp*

Syntax example:

```
print ~x using hex;
```

### Parameter

| | |
|---|---|
| *exp* | A numeric expression or an HDL pathname. |

### Description

Sets each 1 bit of an expression to 0 and each 0 bits to 1. Each bit of the result expression is the opposite of the same bit in the original expression.

### Example 1

This example shows the effect of the ~ operator on a 32-bit integer.

```
m() is {
    var x : int = 0xff;
    print ~x using hex;
};
```

### Result

```
~x = 0xffffff00
```

### Example 2

This example shows the effect of the ~ operator on a 2-bit integer.

```
m() is {
    var x : uint (bits:2) = 2;
    print ~x using bin;
};
```

### Result

```
~x = 0b01
```

### Example 3

This example shows the effect of the ~ operator on an untyped expression.

When the type and bit size of an HDL signal cannot be determined from the context, the expression is automaticallly cast as an unsigned 32-bit integer.

```
m() is {
    print 'top.clk';
    print ~'top.clk';
    print (~'top.clk')[0:0];
};
```

### Result

```
'top.clk' = 0x0
~'top.clk' = 0xffffffff
(~'top.clk')[0:0] = 0x1
```

### See Also

— “'HDL-pathname'” on page 838
— “Scalar Types” on page 75
— “Untyped Expressions” on page 87

## 2.6.2 & | ^

### Purpose

Binary bitwise operations

### Category

Expression

### Syntax

*exp1 operator exp2*

Syntax example:

```
print (x & y);
```

### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A numeric expression or an HDL pathname. |

***operator*** is one of the following:

| | |
|---|---|
| & | Performs an AND operation. |
| \| | Performs an OR operation. |
| ^ | Performs an XOR operation. |

### Description

Performs an AND, OR, or XOR of both operands, bit by bit.

### Example 1

```
m() is {
    var x: uint = 0xff03;
    var y: uint = 0x70f6;
    print (x & y);
};
```

### Result

```
(x & y) = 0x7002
```

### Example 2

```
m() is {
    var x: uint = 0xff03;
    'top.a' = 0x70f6;
    print (x | 'top.a');
};
```

### Result

```
(x | 'top.a') = 0xfff7
```

### Example 3

```
extend sys {
    m() is {
        var x: uint = 0xff03;
        var y: uint = 0x70f6;
        print (x ^ y);
    };
};
```

### Result

```
(x ^ y) = 0x8ff5
```

### See Also

— “'HDL-pathname'” on page 838
— “Scalar Types” on page 75

## 2.6.3 >> <<

### Purpose

Shift bits left or right

### Category

Expression

### Syntax

***exp1 operator exp2***

Syntax example:

```
outf("%x\n", x >> 4);
```

## Parameters

| | |
|---|---|
| *exp1* | A numeric expression or an HDL pathname. |

***operator*** is one of the following:

| | |
|---|---|
| << | Performs a shift-left operation. |
| >> | Performs a shift-right operation. |
| *exp2* | A numeric expression. |

## Description

Shifts each bit of the first expression to the right or to the left the number of bits specified by the second expression.

In a shift-right operation, the shifted bits on the right are lost, while on the left they are filled with 1, if the first expression is a negative integer, or 0, in all other cases.

In a shift-left operation, the shifted bits on the left are lost, while on the right they are filled with 0.

If the bit size of the second expression is greater than 32 bits, it is first truncated to 32 bits, and then the shift is performed. Truncation removes the most significant bits.

NOTE— The result of a shift by more than 31 bits is undefined.

## Example 1

```
m() is {
    var x: int = 0x8fff0011;
    outf("%x\n", x >> 4);
    var y: uint = 0x8fff0011;
    outf("%b\n", y >> 4);
};
```

## Result

```
f8fff001
1000111111111111000000000001
```

## Example 2

```
m() is {
    'top.a' = 0x8fff0011;
    outf("%x\n", 'top.a' << 4);
};
```

## Result

```
fff00110
```

## See Also

— "'HDL-pathname'" on page 838
— "Scalar Types" on page 75

## 2.7 Boolean Operators

The following sections describe the ***e*** boolean operators:

| | |
|---|---|
| "! (not)" on page 35 | Returns TRUE when an expression evaluates to FALSE, and vice versa. |
| "&& (and)" on page 36 | Returns TRUE if two expressions are both TRUE. |
| "\|\| (or)" on page 37 | Returns TRUE if one of two expressions is TRUE. |
| "=>" on page 37 | Returns TRUE when the first expression of two expressions is FALSE, or when both expressions are TRUE. |
| "now" on page 38 | Returns TRUE if an event has occurred in the current cycle. |

### 2.7.1 ! (not)

#### Purpose

Boolean not operation

#### Category

Expression

#### Syntax

!*exp*

**not *exp***

Syntax example:

```
out(!(3 > 2));
```

#### Parameters

| | |
|---|---|
| *exp* | A boolean expression or an HDL pathname. |

#### Description

Returns FALSE when the expression evaluates to TRUE  and returns TRUE when the expression evaluates to FALSE.

#### Example

```
m() is {
    'top.a' = 3;
    out(!('top.a' > 2));
    out(not FALSE);
};
```

#### Result

```
FALSE
```

```
TRUE
```

### See Also

- — “'HDL-pathname'” on page 838
- — “Scalar Types” on page 75

## 2.7.2 && (and)

### Purpose

Boolean and

### Category

Expression

### Syntax

***exp1* && *exp2***

***exp1* and *exp2***

Syntax example:

```
if (2 > 1) and (3 > 2) then {
    out("3 > 2 > 1");
};
```

### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A boolean expression or an HDL pathname. |

### Description

Returns TRUE if both expressions evaluate to TRUE; otherwise, returns FALSE.

### Example

```
m() is {
    'top.a' = 3;
    'top.b' = 2;
    if ('top.b' > 1) and ('top.a' > 2) then {
        out("'top.a' > 'top.b' > 1");
    };
};
```

### Result

```
'top.a' > 'top.b' > 1
```

### See Also

- — “'HDL-pathname'” on page 838
- — “Scalar Types” on page 75

### 2.7.3 || (or)

#### Purpose

Boolean or

#### Category

Expression

#### Syntax

***exp1*** || ***exp2***

***exp1*** **or** ***exp2***

Syntax example:

```
if FALSE || ('top.a' > 1) then {
    out("'top.a' > 1");
};
```

#### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A boolean expression or an HDL pathname. |

#### Description

Returns TRUE if one or both expressions evaluate to TRUE; otherwise, returns FALSE.

#### Example

```
m() is {
    'top.a' = 3;
    if FALSE || ('top.a' > 1) then {
        out("'top.a' > 1");
    };
};
```

#### Result

```
'top.a' > 1
```

#### See Also

- “'HDL-pathname'” on page 838
- “Scalar Types” on page 75

### 2.7.4 =>

#### Purpose

Boolean implication

### Category

Expression

### Syntax

***exp1* => *exp2***

Syntax example:

```
    out((2 > 1) => (3 > 2));
```

### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A boolean expression. |

### Description

The expression returns TRUE when the first expression is FALSE, or when the second expression is TRUE. This construct is the same as:

```
        (not exp1) or (exp2)
```

### Example

```
    m() is {
        out((2 > 1) => (3 > 2));
        out((1 > 2) => (3 > 2));
        out((2 > 1) => (2 > 3));
    };
```

### Result

```
    TRUE
    TRUE
    FALSE
```

### See Also

- “constraint-bool-exp” on page 292
- “Scalar Types” on page 75

## 2.7.5 now

### Purpose

Boolean event check

### Category

Boolean expression

### Syntax

**now @*event-name***

Syntax example:

```
if now @sys.tx_set then {out("sys.tx_set occurred");};
```

### Parameter

*event-name* The event to be checked.

### Description

Evaluates to TRUE if the event occurs before the **now** expression is encountered, in the same cycle in which the **now** expression is encountered.

However, if the event is consumed later during the same cycle, the **now** expression changes to FALSE. This means that the event can be missed, if it succeeds after the expression is encountered.

### Example 1

In the following, the sys.tx_set event is checked when the **if** action is encountered. If the sys.tx_set event has already occurred, in the same sys.clk cycle, the **out()** routine is called.

```
struct pkt {
    event clk is @sys.any;
    tcm_exa()@clk is {
       if now @sys.tx_set then {out("sys.tx_set occurred");};
    };
    run() is also {
       start tcm_exa();
    };
};
```

### Example 2

In this example, the **now** expression is FALSE until the tx_set event is emitted, which changes the expression to TRUE. When the event is consumed by “sync consume (@tx_set)”, the **now** expression changes back to FALSE.

```
struct pkt {
    event tx_set;
    tcm_exa()@sys.any is {
        print now @tx_set;
        emit tx_set;
        print now @tx_set;
        sync consume (@tx_set);
        print now @tx_set;
    };
    run() is also {
        start tcm_exa();
    };
};
extend sys {
    p_i: pkt;
};
```

### See Also

— Chapter 8, “Events”
— “Scalar Types” on page 75

## 2.8 Arithmetic Operators

The following sections describe the ***e*** arithmetic operators:

| | |
|---|---|
| “Unary + -” on page 40 | Perform arithmetic operations on a single operand. |
| “+ - * / %” on page 41 | Perform arithmetic operations on two operands. |

### 2.8.1 Unary + -

#### Purpose

Unary plus and minus

#### Category

Expression

#### Syntax

-*exp*

+*exp*

Syntax example:

```
out(5," == ", +5);
```

#### Parameter

| | |
|---|---|
| *exp* | A numeric expression or an HDL pathname. |

#### Description

Performs a unary plus or minus of the expression. The minus operation changes a positive integer to a negative one, and a negative integer to a positive one. The plus operation leaves the expression unchanged.

#### Example 1

```
m() is {
    out(5," == ", +5);
};
```

#### Result

```
0x5 == 0x5
```

#### Example 2

```
m() is {
        var x: int = 3;
        print -x;
        print -(-x);
    };
```

### Result

```
    -x = -3
    -(-x) = 3
```

### See Also

- "'HDL-pathname'" on page 838
- "Scalar Types" on page 75

## 2.8.2 + - * / %

### Purpose

Binary arithmetic

### Category

Expression

### Syntax

***exp1 operator exp2***

Syntax example:

```
    out(10 + 5);
```

### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A numeric expression or an HDL pathname. |

***operator*** is one of the following:

| | |
|---|---|
| + | Performs addition. |
| - | Performs subtraction. |
| * | Performs multiplication. |
| / | Performs division and returns the quotient, rounded down. |
| % | Performs division and returns the remainder. |

### Description

Performs binary arithmetic operations.

### Example 1

```
    m() is {
        out(4 * -5);
    };
```

### Result

```
0xffffffec
```

### Example 2

```
m() is {
    out(21 / 7);
    out(27 / 7);
};
```

### Result

```
0x3
0x3
```

### Example 3

```
m() is {
    out(23 % 7);
};
```

### Result

```
0x2
```

### See Also

— “'HDL-pathname'” on page 838
— “Scalar Types” on page 75
— “Arithmetic Routines” on page 721

## 2.9 Comparison Operators

The following sections describe the *e* comparison operators:

| | |
|---|---|
| “< <= > >=” on page 42 | Compares two numeric expressions or HDL pathnames. |
| “== !=” on page 43 | Determines whether two expressions are equal or not. |
| “=== !==” on page 45 | Performs a 4-state, Verilog-style comparison of HDL objects. |
| “~ !~” on page 47 | Determines whether two string expressions are equal or not. |
| “in” on page 49 | Determines whether an expression is in a list or a range. |

### 2.9.1 < <= > >=

### Purpose

Comparison of values

### Category

Expression

### Syntax

***exp1 operator exp2***

Syntax example:

```
print 'top.a' >= 2;
```

### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A numeric expression, or an HDL pathname. |

***operator*** is one of the following:

| | |
|---|---|
| < | Returns TRUE if the first expression is smaller than the second expression. |
| <= | Returns TRUE if the first expression is not larger than the second expression. |
| > | Returns TRUE if the first expression is larger than the second expression. |
| >= | Returns TRUE if the first expression is not smaller than the second expression. |

### Description

Compares two expressions.

### Example

```
m() is {
    'top.a' = 3;
    print 'top.a' >= 2;
};
```

### Result

```
'top.a' >= 2 = TRUE
```

### See Also

- "'HDL-pathname'" on page 838
- "Scalar Types" on page 75

## 2.9.2 == !=

### Purpose

Equality of values

### Category

Expression

### Syntax

***exp1 operator exp2***

Syntax example:

```
print lob1 == lob2;
print p1 != p2;
```

## Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | A numeric, boolean, string, list, or struct expression. |

***operator*** is one of the following

| | |
|---|---|
| == | Returns TRUE if the first expression evaluates to the same value as the second expression. |
| != | Returns TRUE if the first expression does not evaluate to the same value as the second expression. |

## Description

The equality operators compare the items and return a boolean result. All types of items are compared by value, except for structs which are compared by address. Comparison methods for the various data types are listed in Table 2-6.

**Table 2-6—Equality Comparisons for Various Data Types**

| Type | Comparison Method |
|---|---|
| integers, unsigned integers, booleans, HDL pathnames | Values are compared. |
| strings | The strings are compared character by character. |
| lists | The lists are compared item by item. |
| structs | The structs addresses are compared |

## Notes

- Enumerated type values can be compared as long as they are of the same type.
- Do not use these operators to compare a string to a regular expression. Use the ~ or the !~ operator instead.
- See "=== !==" on page 45 for a description of using this operator with HDL pathnames.

## Example

```
extend sys {
    p1: packet;
    p2: packet;

    m() is {
        var s: string = "/rtests/tmp";
        var b: bool = TRUE;
        var lob1: list of byte = {0xaa; 0xbb; 0xcc; 0xdd};
        var lob2: list of byte = lob1;

        print s == "/rtests/tmp";
        print b != FALSE;
        print lob1 == lob2;
        print p1 != p2;
        };
    };
```

### Result

```
    s == "/rtests/tmp" = TRUE
    b != FALSE = TRUE
    lob1 == lob2 = TRUE
    p1 != p2 = TRUE
```

### See Also

- — "=== !==" on page 45
- — "~ !~" on page 47
- — "'HDL-pathname'" on page 838
- — "Scalar Types" on page 75

## 2.9.3 === !==

### Purpose

Verilog-style four-state comparison operators

### Category

Expression

### Syntax

'***HDL-pathname***' [!== | ===] ***exp***

***exp*** [!== | ===] '***HDL-pathname***'

Syntax example:

```
    print 'TOP.reg_a' === 4'b1100;
```

### Parameters

| | |
|---|---|
| *HDL-pathname* | The full path name of an HDL object, optionally including expressions and composite data. See "'HDL-pathname'" on page 838 for more information. |
| === | Determines identity, as in Verilog. Returns TRUE if the left and right operands have identical values, considering also the x and z values. |
| !== | Determines non-identity, as in Verilog. TRUE if the left and right operands differ in at least 1 bit, considering also the x and z values. |
| == | Returns TRUE if after translating all x values to 0 and all z values to 1, the left and right operands are equal. |
| != | Returns TRUE if after translating all x values to 0 and all z values to 1, the left and right operands are non-equal. |
| *exp* | Either a literal with four-state values, a numeric expression, or another HDL pathname. |

### Description

Compares four-state values (0, 1, x and z) with the identity and non-identity operators (Verilog style operators). Alternatively, you can use the regular equal and non-equal operators. (A description of the regular identity and non-identity operators is included in "Parameters" on page 46, for clarity.)

There are three ways to use the identity (===) and non-identity (**!==**) operators:

— '***HDL-pathname***' = = = ***literal-number-with-x-and-z values***
This expression compares a HDL object to a literal number (for example 'top.reg' === 4'b11z0). It checks that the bits of the HDL object match the literal number, bit by bit (considering all four values 0, 1, x, z).

— '***HDL-pathname***' = = = ***number-exp***
This expression evaluates to TRUE if the HDL object is identical in each bit value to the integer expression ***number-exp***. Integer expressions in ***e*** cannot hold x and z values; thus the whole expression can be true only if the HDL object has no x or z bits and is otherwise equal to the integer expression.

— '***HDL-pathname***' = = = '***second-HDL-pathname***'
This expression evaluates to TRUE if the two HDL objects are identical in all their bits (considering all four values 0, 1, x, z).

### Example 1

As in Verilog, if the radix is not binary, the z and x values in a literal number are interpreted as more than one bit wide and are left-extended when they are the left-most literal. The width they assume depends on the radix. For example, in hexadecimal radix, each literal z counts as four z bits.

Thus the value assigned in the following statement is 20'bxxxx_xxxx_zzzz_0000_0001 .

```
'x.signal[19:0]' = 20'hxz01;
```

Because z is evaluated as 1 and x as 0 in ordinary expressions, the value printed by the following statement is 0000_0000_1111_0000_0001 .

```
print 'x.signal';
```

Because x is evaluated as 1 and other values as 0 in expressions with @x, the value printed by the following statement is 1111_1111_0000_0000_0000.

```
print 'x.signal@x';
```

Because z is evaluated as 1 and other values as 0 in expressions with @z, the value printed by the following statement is 0000_0000_1111_0000_0000.

```
print 'x.signal@z';
```

### Example 2

In the following example, both comparisons evaluate to TRUE.

```
'TOP.reg_a' = 4'b1100;
wait cycle;
print 'TOP.reg_a' === 4'b1100;
print 'TOP.reg_a' === 0xC;
```

`Example 3`

This example shows how to test a single bit to determine its current state.

```
case {
    'TOP.write_en' === 1'b0: {out("write_en is 0");};
    'TOP.write_en' === 1'b1: {out("write_en is 1");};
    'TOP.write_en' === 1'bx: {out("write_en is x");};
    'TOP.write_en' === 1'bz: {out("write_en is z");};
};
```

### See Also

— "'HDL-pathname'" on page 838
— "Scalar Types" on page 75

## 2.9.4 ~ !~

### Purpose

String matching

### Category

Expression

### Syntax

***"string" operator "pattern-string"***

Syntax example:

```
print s ~ "blue*";
print s !~ "/^Bl.*d$/";
```

## Parameters

| | |
|---|---|
| *string* | A legal ***e*** string. |

***operator*** is one of the following:

| | |
|---|---|
| ~ | Returns TRUE if the pattern string can be matched to the whole string. |
| !~ | Returns TRUE if the pattern string cannot be matched to the whole string. |
| *pattern-string* | Either an AWK-style regular expression or a native ***e*** regular expression. If the pattern string starts and ends with slashes, then everything inside the slashes is treated as an AWK-style regular expression. See "String Matching" on page 51 for more information. |

## Description

Matches a string against a pattern. There are two styles of string matching: native ***e*** style, which is the default, and AWK-style.

After a match using either of the two styles, a local pseudo-variable $0 holds the whole matched string, and the pseudo-variables $1, $2,...$27 hold the sub strings matched. The pseudo-variables are set only by the ~ operator and are local to the function that does the string match. If the ~ operator produces fewer than 28 substrings, then the unneeded variables are left empty.

## Example 1

The first two patterns use ***e*** style; the next two use AWK.

```
m() is {
    var s: string = "BlueBird";

    print s ~ "Blue*";
    print s ~ "blue*";
    print s ~ "/^Bl.*d$/";
    print s ~ "/^bl.*d$/";
};
```

## Result

```
s ~ "Blue*" = TRUE
s ~ "blue*" = TRUE
s ~ "/^Bl.*d$/" = TRUE
s ~ "/^bl.*d$/" = FALSE
```

## Example 2

The first pattern uses ***e*** style; the next uses AWK.

```
m() is {
    var s: string = "BlueBird";

    print s !~ "blue*";
    print s !~ "/^Bl.*d$/";
};
```

## Result

```
s !~ "blue*" = FALSE
```

```
s !~ "/^Bl.*d$/" = FALSE
```

### See Also

— "The string Type" on page 86

## 2.9.5 in

### Purpose

Check for value in a list or specify a range for a constraint.

### Category

Expression

### Syntax

***exp1* in *exp2***

Syntax example:

```
keep x in [1..5];
check that x in {1;2;3;4;5};
```

### Parameters

| | |
|---|---|
| *exp1* | When the second expression is a range list, in a **keep** constraint, for example, then the type of the first expression has to be of a type comparable to the type of the range list. For a range list, square brackets are used.<br>When the second expression is a list, in a **check**, for example, then the type of the first expression can be one of the following:<br>— A type that is comparable to the element type of the second expression.<br>— A list of type that is comparable to the element type of the second expression.<br>For a list, curly braces are used. |
| *exp2* | Either a list or a range list. A range list is a list of constants or expressions that evaluate to constants. Expressions that use variables or struct fields cannot appear in range lists. |

### Description

For a check evaluates TRUE if the first expression is included or contained in the second expression. For a constraint, designates the range for the first expression.

### Example 1

This example checks to make sure that a variable is generated correctly by confirming that its value is in a list of values.

```
extend sys {
    x: int (bits: 64);
    keep x in [1..5];
    run() is also {
        check that x in {1;2;3;4;5};
    };
};
```

### Example 2

This example illustrates the use of **in** with square brackets, [], to designate a range of values for a constraint.

```
type pm_type: [PC_A, PC_B, PC_C, MM_A, MM_B];
extend sys {
    pm: pm_type;
    keep pm in [PC_A, PC_B, PC_C];
};
```

### Example 3

When two lists are compared and the first one has more than one repetition of the same value (for example, in {1;2;1}, 1 is repeated twice), then at least the same number of repetitions has to exist in the second list for the operator to succeed.

In this example, the list y is constrained to have 0 or 2 elements. The first check makes sure that y contains 0 to 2 instances of the numbers 0, 1, 2, and 3. An error is issued for the second check.

```
<'
extend sys {
    y: list of uint (bits: 2);
    keep y.size() in {0;2};

    run() is also {
        check that y in {0;0;1;1;2;2;3;3};
        check that {1;1;2} in {1;2;3;4};
    };
};
'>
```

### Result

```
    *** Error: Dut error at time 0
Checked at line 12 in basics66.e (sys.run):
check that {1;1;2} in {1;2;3;4}
```

### Example 4

This example illustrates that the order of the list items does not influence the result of the comparison. No error is issued.

```
<'
extend sys {
    run() is also {
        check that {1;2;3} in {3;2;1};
        check that {1;1;2} in {1;2;1;2};
    };
};
'>
```

### See Also

— “List Types” on page 84
— “[ range,...]” on page 64

## 2.10 String Matching

There are two styles of string matching: native ***e*** style, which is the default, and an AWK-like style. If the pattern starts and ends with slashes, then everything inside the slashes is treated as an AWK-style regular expression.

The following sections describe these two styles of string matching:

— “Native e Elite String Matching” on page 51
— “AWK-Style String Matching” on page 52

### See Also

— “List Types” on page 84
— “Arithmetic Routines” on page 721

### 2.10.1 Native *e* Elite String Matching

Native ***e*** string matching is attempted on all patterns that are not enclosed in slashes. ***e*** style is similar to UNIX filename matching.

Native string matching uses the meta-characters shown in the following table.

**Table 2-7—Meta-Characters in Native String Matching**

| Character String | Meaning |
|---|---|
| " " (blank) | Any sequence of white space (blanks and tabs) |
| * | Any sequence of non-white space characters, possibly empty (""). "a*" matches "a", "ab", and "abc", but not "ab c". |
| ... | Any sequence of characters |

Native style string matching always matches the full string to the pattern. For example: r does not match Bluebird, but *r* does.

A successful match results in assigning the local pseudo-variables $1 to $27 with the substrings corresponding to the non-blank meta-characters present in the pattern.

Native style string matching is case-insensitive.

### Example

```
m() is {
    var x := "pp kkk";
    print x ~ "* *";
    print $1; print $2;
    print x ~ "...";
    print $1;
};
```

### Result

```
x ~ "* *" = TRUE
$1 = "pp"
$2 = "kkk"
x ~ "..." = TRUE
$1 = "pp kkk"
```

### See Also

— “AWK-Style String Matching” on page 52
— “The string Type” on page 86
— “String Routines” on page 736

## 2.10.2 AWK-Style String Matching

In an AWK-style string matching you can use the standard AWK regular expression notation to write complex patterns. This notation uses the “/.../” format for the pattern to specify AWK-style regular expression syntax.

AWK style supports special characters such as . * [ \ ^ $ +? <>, when those characters are used in the same ways as in UNIX regular expressions (regexp).

The + and ? characters can be used in the same ways as in UNIX extended regular expression (egrep).

In AWK-style regular expressions, you can also use the following Perl shorthand notations, each representing a single character.

**Table 2-8—Perl-Style Regular Expressions Supported**

| Shorthand Notation | Meaning |
|---|---|
| ` | A shortest match operator: ` (back tick). |
| \d | Digit: [0-9] |
| \D | Non-digit |
| \s | Any white-space single char |
| \S | Any non-white-space single |
| \w | Word char: [a-zA-Z0-9_] |
| \W | Non-word char |

After doing a match, you can use the local pseudo-variables $1, $2...$27, which correspond to the parenthesized pieces of the match. $0 stores the whole matched piece of the string.

### Example 1

```
m() is {
    var x := "pp--kkk";
    print (x ~ "/--/");
    print (x ~ "/^pp--kkk$/");
};
```

### Result

```
x ~ "/--/" = TRUE
x ~ "/^pp--kkk$/" = TRUE
```

### Example 2

AWK-style matching is longest match. A shortest match operator is also supported: ` (back tick). The pattern “/x.`y/” matches the minimal such substring.

```
m() is {
    var s := "x x y y";
    print s ~ "/x(.`)y/";       // Prints TRUE
    print $1;                   // Prints " x " Matches x x y
    print s ~ "/x(.*)y/";       // Prints TRUE
    print $1;                   // Prints " x y "Matches x x y y
};
```

### Result

```
s ~ "/x(.`)y/" = TRUE
$1 = " x "
s ~ "/x(.*)y/" = TRUE
$1 = " x y "
```

### Example 3

After doing a match, you can use the local pseudo-variables $1, $2...$27, which correspond to the parenthesized pieces of the match. For instance:

```
m() is {
    var x := "pp--kkk";
    if x ~ "/^(p*)--(k*)$/" then {print $1, $2;};
};
```

### Result

```
$1 = "pp"
$2 = "kkk"
```

### See Also

— “Native e Elite String Matching” on page 51
— “The string Type” on page 86
— “String Routines” on page 736

## 2.11 Extraction and Concatenation Operators

The following sections describe the *e* extraction and concatenation operators:

| | |
|---|---|
| “[ ]” on page 54 | Extracts or sets a single item from a list. |
| “[ : ]” on page 55 | Extracts or sets consecutive bits or slices of a scalar, a list of bits, or a list of bytes. |

| | |
|---|---|
| "[ .. ]" on page 58 | List slicing operator |
| "[ range,...]" on page 64 | Range list operator |
| "{... ; ...}" on page 60 | List concatenation |
| "%{... , ...}" on page 62 | Bit concatenation |

### 2.11.1 [ ]

#### Purpose

List index operator

#### Category

Expression

#### Syntax

*list-exp*[*exp*]

Syntax example:

```
ints[size] = 8;
```

#### Parameters

| | |
|---|---|
| *list-exp* | An expression that returns a list. |
| *exp* | A numeric expression. |

#### Description

Extracts or sets a single item from a list.

#### Notes

— Indexing is only allowed for lists. To get a single bit from a scalar, use bit extraction. See "[ : ]" on page 55.

— Checking list boundaries to see if the specified element exists is done only in interpretive mode.

#### Example

```
<'
extend sys {
    packets[7]: list of packet;
    ints[15]: list of int;
    size: int [0..15];
    m() is {
        print packets[5];
        ints[size] = 8;
        print ints[size];
    };
};
'>
```

### Result

```
        packets[5] = packet-@0: packet
          ---------------------------------------------  @basics69
    0       protocol:                         atm
    1       len:                              1
    2       data:                             (1 items)
      ints[size] = 8
```

### See Also

- — “List Types” on page 84
- — Chapter 19, “List Pseudo-Methods Library”

## 2.11.2 [ : ]

### Purpose

Select bits or bit slices of an expression

### Category

Expression

### Syntax

***exp*****[[*****high-exp*****]:[*****low-exp*****][:*****slice*****]]**

Syntax example:

```
   print u[15:0] using hex;
```

### Parameters

| | |
|---|---|
| *exp* | A numeric expression, an HDL pathname, or an expression returning a list of bit or a list of byte. |
| *high-exp* | A non-negative numeric expression. The high expression has to be greater than or equal to the low expression. To extract a single slice, use the same expression for both the high expression and the low expression. |
| *low-exp* | A non-negative numeric expression, less than or equal to the high expression. |
| *slice* | A numeric expression. The default is **bit**. |

### Description

Extracts or sets consecutive bits or slices of a scalar, a list of bits, or a list of bytes.

When used on the left-hand-side of an assignment operator, the bit extract operator sets the specified bits of a scalar, a list of bits, or a list of bytes to the value on the right-hand-side (RHS) of the operator. The RHS value is chopped or zero/sign extended, if needed.

When used in any context except the left-hand-side of an assignment operator, the bit extract operator extracts the specified bits of a scalar, a list of bits, or a list of bytes.

#### 2.11.2.1 Slice and Size of the Result

The slice parameter affects the size of the slice that is set or extracted. With the default slice (**bit**), the bit extract operator always operates on a 1-bit slice of the expression. When extracting from a scalar expression, by default the bit extract operator returns an expression that is the same type and size as the scalar expression. When extracting from a list of bit or a list of byte, by default the result is a positive unbounded integer.

By specifying a different slice (**byte**, **int**, or **uint**), you can cause the bit operator to operate on a larger number of bits.

For example, the first print statement displays the lower two bytes of big_i, 4096. The second print statement displays the higher 32-bit slice of big_i, -61440.

```
var big_i: int (bits: 64) = 0xffff1000ffff1000;
print big_i[1:0:byte];
print big_i[1:1:int];
```

**Result**

```
big_i[1:0:byte] = 0x0000000000001000
big_i[1:1:int] = 0xfffffffffff1000
```

#### 2.11.2.2 Accessing Nonexistent Bits

If the expression is a numeric expression or an HDL pathname, any reference to a non-existent bit is an error. However, for unbounded integers, all bits logically exist and will be 0 for positive numbers, 1 for negative numbers. It is an error to extract nonexisting bits from list items. When setting non-existing bits in list items, new zero items are added.

**Notes**

— The [***high*** : ***low***] order of the bit extract operator is the opposite of [***low***.. ***high***] order of the list extract operator.

— The bit extract operator has a special behavior in packing. Packing the result of a bit extraction uses the exact size in bits (***high*** - ***low*** + 1). The size of this pack expression is (5 - 3 + 1) + (i - 3 + 1).
```
pack(packing.low, A[5:3], B[i:3]);
```

**Example 1**

This is a simple example showing how to extract and set the bits in an unsigned integer.

```
var x : uint = 0x8000_0a60;
print x[11:4];
print x[31:31];
x[3:0] = 0x7;
print x;
x[2:1:byte] = 0x1234;
print x;
```

**Result**

```
x[11:4] = 0xa6
x[31:31] = 0x1
x = 0x80000a67
x = 0x80123467
```

### Example 2

This example shows how to extract and set the bits in a list of bit.

```
var y : list of bit = {0;1;0;1;1;0;1;0;0;0;0};
print y using bin;
print y[6:1] using bin;
y[6:1] = 0xff;
print y using bin;
var x : uint = y[:];
print x using hex;
```

### Result

```
y =  (11 items, bin):
                                     0 0 0  0 1 0 1  1 0 1 0    .0

y[6:1] = 0b101101
y =  (11 items, bin):
                                     0 0 0  0 1 1 1  1 1 1 0    .0

x = 0x7e
```

### Example 3

This example shows how to extract and set the bits in a list of byte.

```
var z : list of byte = {0x12;0x34;0x56};
print z;
print z[1:0];
print z[1:0:byte];
z[2:2:byte] = 0x48;
print z;
```

### Result

```
z =  (3 items, hex):
                                                56 34 12        .0

z[1:0] = 0x2
z[1:0:byte] = 0x3412
z =  (3 items, hex):
                                                48 34 12        .0
```

### Example 4

This example shows how to use variables in the bit extract operator.

```
var x : uint = 0x80065000;
var i : uint = 16;
var j : uint = 4;
print x[i+j:i-j] using hex;
```

### Result

```
x[i+j:i-j] = 0x65
```

### Example 5

This example shows how to use variables in the bit extract operator. “r” will be an unbounded integer containing 32 bits, extracted starting from byte 1 of the list of bit.

```
var lob : list of bit;
gen lob keeping {.size() < 128};
print lob;
var i : uint = 1;
var r:= lob[i+3:i:byte];
print r using bin;
```

### Result

```
lob =  (40 items, hex):
        1 0 1 1  1 0 0 1  0 1 1 1  1 0 1 1  1 0 0 0  0 0 0 1    .0
                          1 0 1 0  0 1 0 1  1 0 0 1  1 1 0 0    .24

r = 0b0000000010100101100111001011100101111011
```

### See Also

— “Bit Slice Operator and Packing” on page 514
— “'HDL-pathname'” on page 838
— “List Types” on page 84
— “Scalar Types” on page 75

## 2.11.3 [ .. ]

### Purpose

List slicing operator

### Category

Expression

### Syntax

***exp*[[*low-exp*]..[*high-exp*]]**

Syntax example:

```
size: int [0..14];
```

### Parameters

| | |
|---|---|
| *exp* | An expression returning a list or a scalar. |
| *low-exp* | An expression evaluating to a positive integer. The default is 0. |
| *high-exp* | An expression evaluating to a positive integer. The default is the expression size on bits - 1. |

### Description

Accesses the specified list items and returns a list of the same type as the expression. If the expression is a list of bit it returns a list of bit. If the expression is a scalar, it is implicitly converted to a list of bit.

The rules for the list slicing operator are as follows:

— A list slice of the form a[m..n] requires that n>=m>=0 and n<a.size(). The size of the slice in this case is always n-m+1.
— A list slice of the form a[m..] requires that m>=0 and m<=a.size(). The size of the slice in this case is always a.size()-m.
— When assigning to a slice the size of the rhs must be the same as the size of the slice, specfically when the slice is of form a[m..] and m==a.size() then the rhs must be an empty list.

These rules are also true for the case of list slicing a numeric value, for example

```
var i:int;
print i[m..n];
print i[m..];
```

This operator interprets the numeric value as a list of bits and returns the slice of that list. In the above example, the first print is legal if n>=m>=0 and n<32 and the second is legal if m>=0 and m<=32.

### Notes

— This operator is not supported for unbounded integers.
— The only case where a list slice operation returns an empty list is in the case of a[m..] where m==a.size().

### Example 1

This example shows the use of the list slicing operator on a list of integers and a list of structs.

```
<'
struct packet {
    protocol:[atm, eth];
    len : int [0..10];
    data[len]: list of byte;
};
extend sys {
    packets[7]: list of packet;
    ints[15]: list of int;
    size: int [0..15];
    m() is {
        print packets[5..];
        print ints[0..size];
    };
};
```

```
'>
```

### Result

```
packets[5..] =
item   type        protocol    len         data
---------------------------------------------------------------
0.     packet      atm         1           (1 items)
1.     packet      eth         5           (5 items)
  ints[0..size] =
0.      2030699911
1.      -419930323
2.      -1597130501
3.      -494877665
4.      -17370926
5.      -1450077749
6.      1428017017
7.      2036356410
8.      -1952412155
9.      -259249691
```

### Example 2

This example shows the use of the list slicing operator on a scalar expression and an HDL pathname.

```
<'
extend sys {
    m() is {
        var u : uint = 0xffffaaaa;
        print u[..15];
        'top.a' = 0xbbbbcccc;
        print 'top.a'[..15];
    };
};
'>
```

### Result

```
u[..15] =  (16 items, hex):
                                                   a a a a    .0
'top.a'[..15] =  (16 items, hex):
                                                   c c c c    .0
```

### See Also

— "'HDL-pathname'" on page 838
— "List Types" on page 84
— "Scalar Types" on page 75

## 2.11.4 {... ; ...}

### Purpose

List concatenation

### Syntax

**{*exp*; ...}**

Syntax example:

```
var x: list of uint = {1;2;3};
```

### Category

Expression

### Parameters

*exp* Any legal ***e*** expression, including a list. All expressions need to be compatible with the result type.

### Description

Returns a list built out of one or more elements or other lists. The result type is determined by the following rules:

— The type is derived from the context. In the following example, the result type is a list of **uint**:
```
var x: list of uint = {1;2;3};
```
— The type is derived from the first element type of the list. In the following example, the result type is a list of **int** 50 bits wide:
```
var y := {50'1; 2; 3};
```

### Example

```
<'
type color:[red, orange, yellow, green, blue, purple];
extend sys {
    m() is {
        var los: list of string = {"abc";"def"};
        var loc1: list of color = {red;green;blue};
        var loc2:={color'purple;loc1};
        print los;
        print loc1;
        print loc2;
    };
};
'>
```

### Result

```
los =
0.      "abc"
1.      "def"
    loc1 =
0.      red
1.      green
2.      blue
    loc2 =
0.      purple
1.      red
2.      green
```

```
    3.      blue
```

### See Also

— “List Types” on page 84

### 2.11.5 %{... , ...}

#### Purpose

Bit concatenation operator

#### Category

Expression

#### Syntax

%{***exp1***, ***exp2***, ...}

Syntax example:

```
    num1 = %{num2, num3};
    %{num2, num3} = num1;
```

#### Parameters

| | |
|---|---|
| ***exp1***, ***exp2*** | Expressions that receive lists of bits (when on the left-hand side of an assignment operator), or supply lists of bits (when on the right-hand side of an assignment operator). |

#### Description

Creates a list of bits from two or more expressions, or creates two or more smaller lists of bits from a given expression.

You can use the bit concatenation operator **%{}** for packing or unpacking operations that require the **packing.high** order.

- ***value-exp*** = %{***exp1***, ***exp2***,...} is equivalent to ***value-exp*** = **pack**(**packing.high**, ***exp1***, ***exp2***, ...).
- %{***exp1***, ***exp2***,...} = ***value-exp*** is equivalent to **unpack**(**packing.high**, ***value-exp***, ***exp1***, ***exp2***, ...).

Bit concatenations are untyped expressions. In many cases, the required type can be deduced from the context of the expression. See “Untyped Expressions” on page 87 for more information.

#### Example

This example shows several uses of the bit concatenation operator.

```
    extend sys {
        post_generate() is also {
            var num1 : uint (bits : 32);
            var num2 : uint (bits : 16);
            var num3 : uint (bits : 16);
            var bilist : list of bit;
            var bylist : list of byte;

            num2 = 0x1234;
            num3 = 0xabcd;
            num1 = %{num2, num3};
            print num1;

            num1 = 0x98765432;
            %{num2, num3} = num1;
            print num2, num3;
            print %{num2, num3};

            bilist = %{num2, num3};
            print bilist;
            bylist = %{num2, num3};
            print bylist;

        };
    };
```

### Result

```
        num1 = 0x1234abcd
        num2 = 0x9876
        num3 = 0x5432
        % {num2, num3} =  (32 items, hex):
                                            9 8 7 6  5 4 3 2    .0

        bilist =  (32 items, hex):
                                            9 8 7 6  5 4 3 2    .0

        bylist =  (4 items, hex):
                                                98 76 54 32     .0
```

### See Also

— "pack()" on page 516
— "unpack()" on page 521
— "swap()" on page 524
— "do_pack()" on page 526
— "do_unpack()" on page 529

## 2.12 Scalar Modifiers

You can create a scalar subtype by using a scalar modifier to specify the range or bit width of a scalar type. The following sections describe the scalar modifiers:

— "[ range,...]" on page 64
— "(bits | bytes : width-exp)" on page 65

### See Also

— "Scalar Types" on page 75
— "type sized scalar" on page 101
— "type scalar subtype" on page 100

### 2.12.1 [ range,...]

#### Purpose

Range modifier

#### Category

Expression

#### Syntax

**[*range*, ...]**

Syntax example:

```
u: uint[5..7, 15];
```

#### Parameter

| | |
|---|---|
| *range* | Either a constant expression, or a range of constant expressions in the form<br>`low-value..high-value`<br>If the scalar type is an enumerated type, it is ordered by the value associated with the integer value of each type item. |

#### Description

Creates a scalar subtype by restricting the range of valid values.

#### Example 1

The following example shows how to limit the values of an enumerated type and a numeric type.

```
<'
type color:[red, orange, yellow, green, blue, purple];
extend sys {
    bright: color[red..yellow];
    u: uint[5..7, 15];
};
'>
```

#### Example 2

The following example shows how to specify a list of possible values in a **keep** constraint.

```
<'
type color:[red, orange, yellow, green, blue, purple];
extend sys {
    bright: color;
    keep bright in [red, orange, yellow];
};
'>
```

### See Also

— "Scalar Subtypes" on page 76
— "type scalar subtype" on page 100

### 2.12.2 (bits | bytes : width-exp)

#### Purpose

Define a sized scalar

#### Category

Expression

#### Syntax

**(bits**|**bytes**: ***width-exp*****)**

Syntax example:

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

#### Parameter

| | |
|---|---|
| *width-exp* | 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 or bytes. |

#### Description

Defines a bit width for a scalar type. The actual bit width is ***exp * 1*** for bits and ***exp * 8*** for bytes. In the syntax example shown above, both types "word" and "address" have a bit width of 16.

#### Example

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

#### See Also

— "Scalar Types" on page 75
— "type sized scalar" on page 101

## 2.13 Parentheses

You can use parentheses freely to group terms in expressions, to improve the readability of the code. Parentheses are used in this way in some examples in this manual, although they are not syntactically required.

Parentheses are required in a few places in ***e*** code, such as at the end of the method or routine name in all method definitions, method calls, or routine calls. Required parentheses are shown in boldface in the syntax listings in this manual.

The following sections describe the contexts in which the parentheses are required to invoke a method, pseudo-method, or routine:

— "list.method()" on page 66
— "Calling Predefined Routines: routine()" on page 793
— "Invoking Methods" on page 474

## 2.14 list.method()

### Purpose

Execute list pseudo-method

### Category

Expression

### Syntax

***list-exp***. ***list-method*****(**[***param***,]...**)**[.***list-method*****(**[***param***,]...**)**. ...]

Syntax example:

```
print me.my_list.is_empty();
```

### Parameters

| | |
|---|---|
| *list-exp* | An expression that returns a list. |
| *list-method* | One of the list pseudo-methods described in Chapter 19, "List Pseudo-Methods Library" |

### Description

Executes a list pseudo-method on the specified list expression, item by item. When an item is evaluated, **it** stands for the item and **index** stands for its index in the list.

When a parameter is passed, that expression is evaluated for each item in the list.

### Example 1

This example shows how to call two simple list pseudo-methods. The **is_empty()** list method returns a boolean, while **size()** returns an int.

```
<'
extend sys {
    my_list: list of int;

    post_generate() is also {
        print me.my_list.is_empty();
        check that (me.my_list.size() > 5)
    };
};
'>
```

**Result**

```
        me.my_list.is_empty() = FALSE
```

**Example 2**

List method calls can be nested within any expression as long as the returned type matches the context. The following example filters the list my_packets to include only the ethernet kind, sorts the result in ascending order, and prints.

```
    <'
    struct packet {
            kind: [ethernet, atm, other];
            size: uint;
    };

    extend sys {
       packets[10]: list of packet;

        post_generate() is also {
           print packets.all(.kind==ethernet).sort(.size);
        };
    };
    '>
```

**Result**

```
        packets.all(.kind==ethernet).sort(.size) =
    item    type        kind        size
    ---------------------------------------------------------------
    0.      packet      ethernet    895996206
    1.      packet      ethernet    960947360
    2.      packet      ethernet    3889995846
```

**See Also**

— Chapter 19, “List Pseudo-Methods Library”
— “Implicit Variables” on page 24

## 2.15 Special-Purpose Operators

The following special purpose operators are supported:

| | |
|---|---|
| “is [not] a” on page 67 | Identify the subtype of a struct instance |
| “new” on page 69 | Allocate a new struct |
| “.” on page 71 | Refer to fields in structs |
| “‘” on page 73 | Used in names of ***e*** entities |
| “? :” on page 73 | Conditional operator |

### 2.15.1 is [not] a

**Purpose**

Identify the subtype of a struct instance

### Category

Boolean expression

### Syntax

***struct-exp* is a *subtype*** [(***name***)]

***struct-exp* is not a *subtype***

Syntax example:

```
if me is a long packet (l) {
    print l;
};
if me is not a long packet {
    print kind;
};
```

### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. |
| *subtype* | A subtype of the specified struct type. |
| *name* | The name of the local variable you want to create. This parameter cannot be used with **is not a** expressions. |

### Description

Identifies whether a struct instance is a particular subtype or not at run time.

If a name is specified, then a local temporary variable of that name is created in the scope of the action containing the **is a** expression. This local variable contains the result of ***struct-exp*.as_a(*type*)** when the **is a** expression returns TRUE.

### Notes

- — A compile time error results if there is no chance that the struct instance is of the specified type.
- — Unlike other constructs with optional ***name*** variables, the implicit **it** variable is not created when the optional name is not used in the **is a** expression.
- — The ***name*** parameter cannot be used with **is not a** expressions.

### Example

```
<'
type pack_kind :[long, short];
struct packet {
    kind: pack_kind;
    when long packet {
        a: int;
    };
    check_my_type() is {
        if me is a long packet (l) {
            print l;
        };
        if me is not a long packet {
            print kind;
        };
    };
};

extend sys {
    p:packet;
};
'>
```

### Result

```
        l = packet-@0: packet
    ----------------------------------------------   @expressions67
    0       kind:                           long
    1       long'a:                         -1786485835
```

### See Also

— "as_a()" on page 104

## 2.15.2 new

### Purpose

Allocate a new initialized struct

### Category

Expression

### Syntax

**new** [***struct-type*** [[(***name***)] **with** {***action;...***}]]

Syntax example:

```
var q : packet = new good large packet;
```

### Parameters

| | |
|---|---|
| *struct-type* | Either a struct type or a struct subtype. |
| *name* | An optional name, valid within the action block, for the new struct. If no name is specified, you can use the implicit variable **it** to refer to the new struct. |
| *action* | A list of one or more actions. |

### Description

Creates a new struct:

1) Allocates space for the struct.
2) Assigns default values to struct fields.
3) Invokes the **init()** method for the struct, which by default initializes all fields of scalar type, including enumerated scalar type, to zero. The initial value of a struct or list is NULL, unless the list is a sized list of scalars, in which case it is initialized to the proper size with each item set to the default value.

4) Invokes the **run()** method for the struct, unless the **new** expression is in a construct that is executed before the run phase. For example, if you use **new** in an extension to **sys.init()**, then the **run()** method is not invoked.
5) Executes the action-block, if one is specified.

If no subtype is specified, the type is derived from the context. For example, if the new struct is assigned to a variable of type packet, the new struct will be of type packet.

If the optional **with** clause is used, you can refer to the newly created **struct** either with the implicit variable **it**, or with an optional name.

NOTE— The new struct is a shallow struct. The fields of the struct that are of type struct are not allocated.

### Example

```
<'
struct packet {
    good : bool;
    size : [small, medium, large];
        length : int;
};
extend sys {
    post_generate() is also {
        var p : packet = new;
        print p;
        var q : packet = new good large packet;
        print q;
        var x := new packet (p) with {
            p.length = 5;
                print p;
            };
    };
};
'>
```

### Result

```
    p = packet-@0: packet
----------------------------------------------  @expressions69
0       good:                           FALSE
1       size:                           small
2       length:                         0
    q = good large packet-@1: good large packet
----------------------------------------------  @expressions69
0       good:                           TRUE
1       size:                           large
2       length:                         0
    p = packet-@2: packet
----------------------------------------------  @expressions69
0       good:                           FALSE
1       size:                           small
2       length:                         5
```

### See Also

— “The init() Method of sys” on page 645
— “The run() Method of sys” on page 646

— “Struct Subtypes” on page 80

### 2.15.3 .

#### Purpose

Refer to fields in structs

#### Category

Expression

#### Syntax

[[***struct-exp***].] ***field-name***

Syntax example:

```
keep soft port.sender.cell.u == 0xff;
```

#### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. |
| *field-name* | The name of the scalar field or list field to reference. |

#### Description

Refers to a field in the specified struct. If the struct expression is missing, but the period exists, the implicit variable **it** is assumed. If both the struct expression and the period (**.**) are missing, the field name is resolved according to the name resolution rules.

#### Notes

— When the struct expression is a list of structs, the expression cannot appear on the left-hand side of an assignment operator.
— When the field name is a list item, the expression returns a concatenation of the lists in the field.

#### Example 1

The following example shows the use of the “.” to identify the fields u and kind in the **keep** constraints:

```
<'
struct switch {
    ctrl: ctrl_stub;
    port: port_stub;

    keep soft port.sender.cell.u == 0xff;
    keep ctrl.init_command.kind == RD;
};
struct ctrl_stub {
    init_command: ctrl_cmd;
};
struct simplex {
    kind: [TX, RX];
    cell: cell;
};
struct port_stub {
    sender: TX simplex;
    listener: RX simplex;
};
struct cell {
    u: uint;
};
struct ctrl_cmd {
    kind: [RD, WR];
    addr: int;
};
extend sys {
    switch : switch;
};
'>
```

### Example 2

This example shows the effect of using the "." to access the fields in a list (switch.port) and to access a field that is a list (switch.port.data):

```
<'
struct switch {
    port: list of port_stub;

    keep soft port.size() == 4;
};

struct port_stub {
    data[5]: list of byte;
};

extend sys {
    switch : switch;

    post_generate() is also {
        print switch.port;
        print switch.port.data;
    };
};
'>
```

### Result

```
    switch.port =
item   type        data
----------------------------------------------------------------
0.     port_stub   (5 items)
1.     port_stub   (5 items)
2.     port_stub   (5 items)
3.     port_stub   (5 items)
  switch.port.data =  (20 items, dec):
        185  24 137 186  202   3 186 107  108 119  84 212       .0
                         129 224  56 145    3 252  61  58       .12
```

### See Also

— "Struct Hierarchy and Name Resolution" on page 19

### 2.15.4 '

#### Apostrophes

The apostrophe (**'**) is an important syntax element used in multiple ways in ***e*** source code. The actual context of where it is used in the syntax defines its purpose. A single apostrophe is used in the following places:

— When accessing HDL objects (for example: 'top.a')
— When defining the name of a syntactic construct in a macro definition (for example: show_time'command)
— When referring to struct subtypes (for example: b'dest Ethernet packet)
— When referring to an enumerated value not in context of an enumerated variable (for example: color'green)
— In the begin-code marker **<'** and in the end-code marker **'>**

#### See Also

— Chapter 13, "Macros"
— "Struct Subtypes" on page 80
— "Enumerated Type Values" on page 23
— "Code Segments" on page 4

### 2.15.5 ? :

#### Purpose

Conditional operator

#### Category

Expression

#### Syntax

*bool-exp* **?** *exp1* **:** *exp2*

Syntax example:

```
z = (flag ? 7 : 15);
```

#### Parameters

| | |
|---|---|
| *bool-exp* | A legal ***e*** expression that evaluates to TRUE or FALSE. |
| *exp1*, *exp2* | A legal ***e*** expression. |

#### Description

Evaluates one of two possible expressions, depending on whether a boolean expression evaluates to TRUE or FALSE. If the boolean expression is TRUE, then the first expression is evaluated. If it is FALSE, then the second expression is evaluated.

#### Example

```
<'
```

```
extend sys {
    m() is {
        var z: int;
        var flag: bool;

        z = (flag ? 7 : 15);
        print flag, z;
    };
};
'>
```

### Result

```
flag = FALSE
z = 15
```

### See Also

— “Conditional Actions” on page 533

# 3 Data Types

The ***e*** language has a number of predefined data types including the integer and boolean scalar types common to most programming languages. In addition, you can create new scalar data types (enumerated types) that are appropriate for programming, modeling hardware, and interfacing with hardware simulators. 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).

This chapter contains the following topics:

— "Overview of e Data Types" on page 75
— "Defining and Extending Scalar Types" on page 98
— "Type Conversion Between Scalars and Strings" on page 104

**See Also**

— Chapter 4, "Structs, Fields, and Subtypes"
— Chapter 2, "e Basics"

## 3.1 Overview of *e* Data Types

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

— "e Data Types" on page 75
— "Memory Requirements for Data Types" on page 87
— "Untyped Expressions" on page 87
— "Assignment Rules" on page 89
— "Precision Rules for Numeric Operations" on page 93
— "Automatic Type Casting" on page 96

### 3.1.1 *e* Data Types

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

— "Scalar Types" on page 75
— "Scalar Subtypes" on page 76
— "Enumerated Scalar Types" on page 77
— "Struct Types" on page 80
— "Struct Subtypes" on page 80
— "List Types" on page 84
— "The string Type" on page 86
— "The external_pointer Type" on page 86

Certain expressions, such as HDL objects, have no explicit data type. See "Untyped Expressions" on page 87 for information on how these expressions are handled.

#### 3.1.1.1 Scalar Types

Scalar types in ***e*** are one of the following:

— Numeric
— Boolean

— Enumerated

Table 3-1, “Predefined Scalar Types”, on page 76 shows the predefined numeric and boolean types. See the notes below the table for important information about these predefined types.

**Table 3-1—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) |

NOTE— Both signed and unsigned integers can be of any size and, thus, of any range. See “Scalar Subtypes” on page 76 for information on how to specify the size and range of a scalar field or variable explicitly.

### Result

— Predefined constants, described in Chapter 2, “e Basics”
— Constraint boolean expressions, described in Chapter 2, “e Basics”

### 3.1.1.2 Scalar Subtypes

You can create a scalar subtype by using a scalar modifier to specify the range or bit width of a scalar type. You can also specify a name for the scalar subtype if you plan to use it repeatedly in your program. Unbounded integers are a predefined scalar subtype.

The following sections describe scalar modifiers, named scalar subtypes, and unbounded integers in more detail.

#### 3.1.1.2.1 Scalar Modifiers

There are two types of scalar modifiers that you can use together or separately to modify predefined scalar types:

1) Range modifiers
2) 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. The width modifiers in the expressions below restrict the bit width to 8.

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

You can use width and range modifiers in combination.

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

### 3.1.1.2.2 Named Scalar Subtypes

When you use a scalar modifier to limit the range or bit width of a scalar type, you can also specify a name.

Named scalar subtypes are useful in a context where, for example, you need to declare a counter variable like the variable “count” several places in the program.

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

By creating a named scalar type, you can use the type name when introducing new variables with this type.

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

See “type enumerated scalar” on page 98 for more information on named scalar subtypes.

### 3.1.1.2.3 Unbounded Integers

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

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

You can use an unbounded integer variable when you do not know the exact size of the data. You can use unbounded integers in expressions just as you use signed or unsigned integers.

#### Notes

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

#### See Also

— “type scalar subtype” on page 100
— “type sized scalar” on page 101
— “extend type” on page 103

### 3.1.1.3 Enumerated Scalar Types

You can define the valid values for a variable or field as a list of symbolic constants. For example, the following declaration defines the variable “kind” as having two legal values.

```
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. Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1. You can also assign explicit unsigned integer values to the symbolic constants.

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

The associated unsigned integer value of a symbolic constant in an enumerated type can be obtained using the **.as_a()** type casting operator. 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.

Casting an unsigned integer to a symbolic constant:

```
type signal_number: [signal_0, signal_1, signal_2, signal_3];
struct signal {
    cast_1() is {
        var temp_val: uint = 2;
        var signal_name: signal_number = temp_val.as_a(signal_number);
        print signal_name;
    };
};
```

Casting a symbolic constant to an unsigned integer:

```
type signal_number: [signal_0, signal_1, signal_2, signal_3];
struct signal {
    cast_2() is {
        var temp_enum: signal_number = signal_3;
        var signal_value: uint = temp_enum.as_a(uint);
        print signal_value;
    };
};
```

You can explicitly assign values to some symbolic constants and allow others to be automatically assigned. The following declaration assigns the value 3 to “immediate”; the value 4 is assigned to “register” automatically.

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

You can name an enumerated type to facilitate its reuse throughout your 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.

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

It is sometimes convenient to introduce a named enumerated type as an empty type.

```
type packet_protocol: [];
```

Once the protocols that are meaningful in the program are identified you can extend the definition of the type with a statement like:

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

Enumerated types can be sized.

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

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, even if zero is not a legal value for that type. For example, the variable “i_kind” has the value zero until it is explicitly initialized or generated.

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

### 3.1.1.4 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 (that is, comparisons using <, <=, >, >=, ==, or **!**= operators). This is consistent with the strong typing in ***e***, and helps avoid introduction of bugs if the order of symbolic names in an enumerated type declaration is changed, for example, while operations which are affected by the order of those names in the declaration remain unchanged (because they are in a different part of the code and therefore go unnoticed, perhaps).

Assume that I is an int, B is a bool, and E is an enumerated type. Since enumerated and boolean types are not automatically converted to or from integers or unsigned integers, you cannot use syntax such as “if (I) {...}”, or “if (B==1) {...}”, or “if (E<6) {...}”. In order to perform such comparisons, you must use explicit casting, or tick notation to specify the type. Examples of correct and incorrect syntax are shown in the sample code below.

```
    type my_enum: [A, B, C];
    struct etypes {
        x: my_enum;
        my_method() is {

            if (A.as_a(int) < B.as_a(int)) then {      // Load-time error:
                out("A is less than B");               // No such variable 'A'
            };

            if (A.as_a(int) == B.as_a(int)) then {    // Load-time error:
                out("A equals B");                    // No such variable 'B'
            };

            if (my_enum'A.as_a(int) < my_enum'B.as_a(int)) then {   // No error
                out ("A less than B");
            };

            if (my_enum'A < my_enum'B) then {     // Load-time error:
                out ("A less than B");     // The type of 'x' is 'my_enum'
            };                             // while expecting a numeric type

            if (x < A) then {               // Load-time error:
                out("x less than A");       // The type of 'x' is 'my_enum'
            };                              // while expecting a numeric type

            if (x == A) then {                                      // No error
                out ("x equals A");
            };
        };
    };
```

The first two **if** statements above cause load errors because it is possible for A or B or both to be used in more than one enumerated type declaration, and it is not possible to tell from the context which type they are, or their values. In the third **if** statement, the enumerated type is specified using the tick notation, so that statement is legal. Note that it is still necessary to cast A and B as ints in order to do the comparison, A < B, otherwise the error in the fourth case, my_enum'A < my_enum'B, occurs.

In the fifth case, x < A, the context of A is not clear at load time, so a loading error occurs. The context of A is clear in the last case, x == A, however, so this code loads with no problem.

#### See Also

- "type enumerated scalar" on page 98
- "extend type" on page 103

### 3.1.1.5 Struct Types

Structs are the basis for constructing compound data structures.

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

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

You can then use the struct type "packet" 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;
    };
```

You can also define a variable using a struct type.

```
    var data_in : packet;
```

The default value for a struct is NULL.

#### See Also

- Chapter 4, "Structs, Fields, and Subtypes"
- "var" on page 487

### 3.1.1.6 Struct Subtypes

When a struct field has a boolean type or an enumerated type, you can define a struct subtype for one or more of the possible values for that field. For 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, for example "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, you can use this syntax:

```
value_name struct_type
```

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

```
value'field_name struct_type
```

For example, if we define two enumerated types:

```
type destination: [a, b, c, d];
type source: [a, b, c, d];
```

And add two fields to the "packet" struct:

```
dest: destination;
src: source;
```

The syntax for referring to the type of an Ethernet packet with the destination "b" is:

```
b'dest Ethernet packet
```

because the name "b Ethernet packet" is ambiguous.

```
type packet_protocol: [Ethernet, IEEE, foreign];
type destination: [a, b, c, d];
type source: [a, b, c, d];

struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    legal: bool;
    dest: destination;
    src: source;
};
```

```
extend sys {
    gen_eth_packet () is {
        var packet: b'dest Ethernet packet;
        gen packet keeping {it.size > 511 and it.size < 1k};
        print packet;
    };
};
```

The example below shows another context where a struct subtype can be used.

```
type packet_protocol: [Ethernet, IEEE, foreign];
type destination: [a, b, c, d];
type source: [a, b, c, d];

struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    legal: bool;
    dest: destination;
    src: source;
};
extend sys {
plist: list of packet;
    print_Epackets() is {
        for each Ethernet packet (ep) in plist {
            print ep;
        };
    };
};
```

You can also use the **extend**, **when**, or **like** constructs to add fields, methods, or method extensions that are required for a particular subtype.

For example, the **extend** construct shown below adds a field and a method to the “Ethernet packet” subtype. The “Ethernet packet” subtype also inherits all the characteristics of the struct “packet”.

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

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

};

extend Ethernet packet {
    e_field: int;

    show() is {out("I am an Ethernet packet")};
};
```

The “Ethernet packet” subtype could also be defined with the **when** construct. The following “Ethernet packet” subtype is exactly equivalent to the Ethernet packet subtype defined by **extend**.

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

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

    when Ethernet packet {
        e_field: int;

        show() is {out("I am an Ethernet packet")};
    };
};
```

You can use either the **when** or the **extend** construct to define struct subtypes with very similar results. These constructs are appropriate for most modeling purposes. Under certain circumstances, you may prefer to use the **like** construct to create struct subtypes. See Chapter 4, “Structs, Fields, and Subtypes” for a detailed discussion of the use of these constructs to create struct subtypes.

### 3.1.1.7 Referencing Fields in When Constructs

The example below shows how to refer to a field of a struct subtype outside of a **when**, **like**, or **extend** construct by assigning a temporary name to the struct subtype.

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

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

    keep me is a Ethernet packet (ep) => ep.e_field == 1;

    when Ethernet packet {
        e_field: int;

        show() is {out("I am an Ethernet packet")};
    };
};
```

In order to reference a field in a **when** construct, you must specify the appropriate value for the **when** determinant. For example, consider the following struct and subtype:

```
type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    when IEEE packet {
        i_val: int;
    };
};
```

For any instance “pk_inst” of the packet struct, references to the “i_val” field are only valid if the **when** determinant is “IEEE”. The following are three ways to ensure that “pk_inst” is in fact an “IEEE packet” before referencing “i_val”.

— Test “pk_inst” to see if it is “IEEE packet”:

```
if pk_inst is a IEEE packet (ip) {ip.i_val = 1; };
```

or

```
pk_list.first(it is a IEEE packet (ip) and ip.i_val == 1);
```

Use this method if “pk_inst” is a packet that may or may not be “IEEE”. For example, “pk_inst” may be an element of a list of packets or may be a generated packet with no constraint on “protocol”.

— Define “pk_inst” as an “IEEE packet”:

```
var pk_inst: IEEE packet;
pk_inst.i_val = 1;
```

or

```
var pk_inst: IEEE packet;
gen pk_inst keeping {
    it is a IEEE packet (ip) and ip.i_val == 1
};
```

Use this method if you want “pk_inst” to always be “IEEE”. Note that you must either declare the variable or field to be type “IEEE packet” or use the **is a** syntax. It is not sufficient to say **gen** pk_inst **keeping** (.kind == IEEE; .i_val == 1}.

— Cast “pk_inst” as “IEEE packet”:

```
pk_inst.as_a(IEEE packet).i_val = 1;
```

This is shorthand for method 1 above. You can do it this way if you know that “pk_inst” is an “IEEE packet” but for some reason it is defined just as a packet. For example:

```
var pk_inst: packet;
gen pk_inst keeping {it is a IEEE packet};
pk_inst.as_a(IEEE packet).i_val = 1;
```

Note that if “pk_inst” is not an “IEEE packet” you will get an error stating that “struct is NULL”.

### See Also

— Chapter 4, “Structs, Fields, and Subtypes”
— “var” on page 487
— “Struct Hierarchy and Name Resolution” on page 19
— “Comparison of When and Like Inheritance” on page 142
— “when” on page 133
— “Extending Structs: extend type” on page 121
— “is [not] a” on page 67

### 3.1.1.8 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. You can select an item from a list by specifying its index. For example, 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];
```

NOTE— Multi-dimensional lists (lists of lists) are not supported. To create a list with sublists in it, you can 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.

### 3.1.1.8.1 Regular Lists

The following example shows two lists, “packets” and “all_lengths”.

```
type packet_protocol : [Ethernet, IEEE, foreign];
type length: int [0..10];
struct packet {
    protocol: packet_protocol;
    len: length;
};
extend sys {
    packets[10] : list of packet;
    do_print() is {
        var all_lengths: list of length;
        all_lengths = packets.len;
        print packets;
        print all_lengths;
    };
};
```

Each element of “packets” is a struct of type “packet”. Each element of “all_lengths” is a scalar value of type “length”.

Both “packets” and “all_lengths” have 10 elements because of the explicit size “[10]” specified in the “packets” declaration. You can only specify a list size in this manner for fields. To size lists that are variables, you have to use a **keep** constraint.

### 3.1.1.8.2 Keyed Lists

A keyed list data type is similar to hash tables or association lists found in other programming languages. The declaration below specifies that “packets” is a list of packets, and that the “protocol” field in the packet type is used as the hash key.

```
type packet_protocol : [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
};
var packets : list (key: protocol) of packet;
```

If the element type of the list is a scalar type or a string type, then the hash key must be the predefined implicit variable **it**.

```
struct person {
    name: string;
```

```
        id: int;
    };
    struct city {
        !persons: list(key: name) of person;
        !street_names: list(key: it) of string;
    };
```

### Notes

— Keyed lists cannot be generated. Trying to generate a keyed list results in an error. Therefore, keyed lists must be defined with the do-not-generate sign (an exclamation mark), as in the above example.

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

### See Also

— Chapter 4, “Structs, Fields, and Subtypes”

— “var” on page 487

— “Packing and Unpacking Lists” on page 503

— Chapter 19, “List Pseudo-Methods Library”

## 3.1.1.9 The string Type

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

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

You cannot access bits or bit ranges of a string, but you can convert a string to a list of bytes and then access a portion of the string. The print statement shown 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

— Chapter 24, “Predefined Routines Library”

— “Packing and Unpacking Strings” on page 501

## 3.1.1.10 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.

### 3.1.2 Memory Requirements for Data Types

The amount of memory needed to store data types is listed in Table 3-3.

**Table 3-2—Storage Sizes of DataTypes**

| Type | Size in Memory |
|---|---|
| All scalars up to 32 bits | 4 bytes |
| Scalars larger than 32 bits | Same as a list of bit of the appropriate size |
| String | 4 bytes (the pointer) + the size of the string + 1 byte (the NULL byte)<br><br>A NULL string is just the pointer. |
| Struct pointer | 4 bytes |
| Struct | 8 bytes + the sum of the field sizes<br><br>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<br><br>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. |

### 3.1.3 Untyped Expressions

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

- 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, because of implicit packing and unpacking, these expressions can be converted to the required data type and bit size in certain contexts.

- 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.
  The pack expression shown below, for example, is evaluated as 0x04, taking the type and bit size of “j”.

```
var j:int(bits:8);
j = pack(packing.low, 4);
```

NOTE— Implicit unpacking is not supported for strings, structs, or lists of non-scalar types. As a result, the following causes a load-time error if “i” is a string, a struct, or a list of a non-scalar type:

```
i = pack(packing.low, 5);
```

— When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned.
In the following example, the value of “j”, 0x4, is implicitly packed and converted to the size of 'top.a' before the value is driven:

```
'top.a' = j;
```

NOTE— Implicit packing is not supported for strings, structs, or lists of non-scalar types. As a result, the assignment above would cause a load-time error if “j” were a string, a struct, or a list of a non-scalar type.

— 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 “Precision Rules for Numeric Operations” on page 93 for more information.
Both 'top.a' and “pack(packing.low, -4)” are handled as numeric types.

```
print ('top.a' + pack(packing.low, 4) == 0);
```

— 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 **pack()** expression “pack(packing.low, data)” generates a list of bit that is implicitly unpacked into the required type **list of byte** as defined in the declaration of the send_data() method.

```
extend sys {
    data[10]:list of byte;
    send_data(d: list of byte) is {
        ...
    };
    run() is also {
        send_data(pack(packing.low, data));
    };
};
```

NOTE— The method parameter or return value in the pack expression must be a scalar type or a list of scalar type. For example, the following results in a load-time error:

```
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);
};
extend sys {
    instr: instruction;
    send_instr(i: instruction) is {
        ...
    };
    run() is also {;
```

```
            send_instr(pack(packing.low, 5)); --load-time error
        };
    };
```

— 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

— "Implicit Packing and Unpacking" on page 515

## 3.1.4 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:

— "What Is an Assignment?" on page 89
— "Assignments Create Identical References" on page 90
— "Assignment to Different but Compatible Types" on page 91

### 3.1.4.1 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";
        };
    };
```

NOTE— You cannot assign values to fields during declaration in this same manner.

#### 3.1.4.2 Assignments Create Identical References

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.

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

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.

### 3.1.4.3 Assignment to Different but Compatible Types

#### 3.1.4.3.1 Assignment of Numeric Types

Any numeric type (for example, **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 "Untyped Expressions" on page 87 for more information.

```
extend sys {
    !x1: int;
    x2: uint (bits: 3);
    !x3: int [10..100];

    post_generate() is also{
        x1 = x2;
        x3 = x1;
        var x: int (bits: 48) = x3;
    };
};
```

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 different bit size. See "Automatic Type Casting" on page 96 for more information. See "Precision Rules for Numeric Operations" on page 93 for information on how precision is determined for operations involving numeric types.

#### 3.1.4.3.2 Assignment of Boolean Types

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

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

#### 3.1.4.3.3 Assignment of Enumerated Types

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

The example below shows:

— An assignment of the same type:

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

— An assignment of a scalar subtype:

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

**Example**

```
type color: [red,green,blue];
type color2: color (bits: 2);

extend sys {
    m() is {
        var x: color = blue;
        var y: color2 = x;
    };
};
```

To assign any scalar type (numeric, enumerated, or boolean type) to any different scalar type, you must use the **.as_a()** operator.

#### 3.1.4.3.4 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):

```
set_cell(p);
```

— An assignment of a derived struct (cell_8023):

```
p.cell = new cell_8023;
```

### Example

```
type packet_kind: [Ether, Ether_8023];
struct cell {};

struct cell_8023 like cell {};
struct packet {
    packet_kind;
    !cell: cell;
};

extend sys {
    p1: packet;
    !p2: packet;
    !p3: packet;
    post_generate() is also {
        p2 = p1;
        var p: Ether_8023 packet;
        gen p;
        set_cell(p);
    };

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

Although you can assign a subtype to its parent struct without any explicit casting as shown above, to perform the reverse assignment (assign a parent struct to one of its subtypes), you must use the **.as_a()** method. See “as_a()” on page 104 for an example of how to do this.

#### 3.1.4.3.5 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
    };
};
```

#### 3.1.4.3.6 Assignment of Lists

An entity of 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;” would be an error, because “list1” not the same list type as “y”. “y” has a size modifier, so it is a subtype of “list1”. You can use the **.as_a()** method to cast between lists and their subtypes.

### See Also

— “Untyped Expressions” on page 87
— “Precision Rules for Numeric Operations” on page 93
— “Automatic Type Casting” on page 96

## 3.1.5 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:

— Unsigned 32-bit integer (**uint**)
— Signed 32-bit integer (**int**)
— Infinite signed integer (**int (bits: *)**)

The ***e*** language has rules for:

— Determining the context of an expression
— Deciding precision, and performing data conversion and sign extension

The following sections describe these rules and give an example of how these rules are applied:

— “Determining the Context of an Expression” on page 94
— “Deciding Precision and Performing Data Conversion and Sign Extension” on page 95
— “Example Application of Precision Rules” on page 95

### See Also

— “Operator Precedence” on page 28

### 3.1.5.1 Determining the Context of an Expression

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

1) In an assignment (***lhs*** = ***rhs***), the right-hand side (***rhs***) expression inherits the context of the left-hand side (***lhs***) expression.
2) A sub-expression inherits the context of its enclosing expression.
3) 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 3-3 summarizes context inheritance for each type of operator that can be used in numeric expressions.

**Table 3-3—Summary of Context Inheritance in Numeric Operations**

| Operator | Function | Context |
|---|---|---|
| * / % + -<br>< <= > >=<br>== !=<br>=== !==<br>& \| ^ | Arithmetic, comparison, equality, and bit-wise boolean | The right-hand operand inherits context from the left-hand operand (***lho***), as well as from the enclosing expression. ***lho*** inherits only from the enclosing expression. |
| ~ !<br>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 32-bit **uint**. |
| ***lho*** **in** [***i***..***j***] | Range list operator | All three operands are context independent. (The range specifiers ***i*** and ***j*** must be constant.) |
| &&, \|\| | Boolean | All operands are context independent. |
| *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. |

**Table 3-3—Summary of Context Inheritance in Numeric Operations *(continued)***

| Operator | Function | Context |
|---|---|---|
| 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 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. |

### 3.1.5.2 Deciding Precision and Performing Data Conversion and Sign Extension

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

— Determine the context of the expression. The context may be comprised of up to two types.
— If all types involved in an expression and its context are 32 bits in width or less:

- The operation is performed in 32 bits.
- If any of the types is unsigned, the operation is performed with unsigned integers.

  NOTE— 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.

- Each operand is automatically cast, if necessary, to the required type.

  NOTE— Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.

— If any of the types is greater than 32 bits:

- The operation is performed in infinite precision (**int (bits:*)**)
- Each operand is zero-extended, if it is unsigned, or sign-extended, if it is signed, to infinite precision.

### 3.1.5.3 Example Application of Precision Rules

Given the following assignment:

```
sum: int;
exp1: int (bytes:2);
exp2: uint (bits:4);
exp3: int (bits:4);
```

```
sum = exp1 + exp2 * exp3;
```

1) The precision of the multiplication operation (exp2 * exp3) is based on the four types involved here:
   The inherited context of the lhs expression (**int**)
   The inherited context of the lho (**int (bytes:2)**)
   The type of exp2 (4-bit **uint**)
   The type of exp3 (4-bit **int**)

   Because one of these four types is unsigned, the multiplication is done in 32-bit unsigned integer. Both exp2 and exp3 are converted to 32-bit uint and the multiplication operation is performed.
2) The precision of the addition operation is based on the three types involved here:
   The inherited context of the lhs expression (**int**)
   The type of exp1 (**int (bytes:2)**)
   The type of (exp2 * exp3) (**uint**)

   Because one of these types is unsigned, the addition is done in 32-bit unsigned integer. exp1 is converted to 32-bit **uint** and the addition operation is performed.
3) For the assignment operation, the result of the addition operation is converted to 32-bit **int** and assigned to sum.

### See Also

— "Untyped Expressions" on page 87
— "Assignment Rules" on page 89
— "Automatic Type Casting" on page 96

## 3.1.6 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 "Untyped Expressions" on page 87 for more information.

```
var j: uint = 0xff;
'top.a' = j;
```

— Sized scalars are automatically type cast to differently sized scalars of the same type.

```
type color: [red,green,blue];
type color2: color (bits: 2);
var x: color = blue;
var y: color2 = x;
```

— Struct subtypes are automatically cast to their base struct type.

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

```
struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined; // To be defined by children
};
extend Ethernet packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};
extend sys {
    m() is {
        var epkt: Ethernet packet = new;
        var pkt: packet = epkt;
    };
};
```

There are three important ramifications to automatic type casting:

1) If the two types differ in bit size, then the assigned value is extended or truncated to the required bit size. See Example 1 on page 97.
2) Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers. See Example 2 on page 98.
3) There is no automatic casting to a reference parameter. See "Parameter Passing" on page 484 for more information.

### Example 1

In the following example, "x" is a 32-bit signed integer, "y" is a 48-bit unsigned integer, and "z" is a 3-bit signed integer. Assigning "x" to "y" extends "x" to 48 bits. Assigning "x" to "z" chops "x" to 3 bits.

```
extend sys {
    m() is {
        var x: int = -1;
        var y: int (bits: 48) = x;
        var z: int (bits: 3) = x;
        print y,z;
    };
};
```

### Result

Calling "sys.m()" results in:

```
y = 0xffffffffffff
z = 0x7
```

### Example 2

```
m() is {
    var x: int = -1;
    var y: uint = MAX_UINT;
    var z: uint = 1;
    print x == y;
    print x > z;
};
```

### Result

The **int** value “x” (0xffffffff) is automatically cast to **uint** and becomes MAX_UINT. As a result, the print statements display the following:

```
x == y = TRUE
x > z = TRUE
```

### See Also

— “as_a()” on page 104
— “Untyped Expressions” on page 87
— “Assignment Rules” on page 89
— “Precision Rules for Numeric Operations” on page 93

## 3.2 Defining and Extending Scalar Types

You can use the following constructs to define and extend scalar types:

— “type enumerated scalar” on page 98
— “type scalar subtype” on page 100
— “type sized scalar” on page 101
— “extend type” on page 103

### 3.2.1 type enumerated scalar

#### Purpose

Define an enumerated scalar type

#### Category

Statement

#### Syntax

**type *enum-type-name***: **[**[***name***[=***exp***], ...]**]** [(**bits** | **bytes:** ***width-exp***)]

Syntax example:

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

## Parameters

| | |
|---|---|
| *enum-type-name* | A legal ***e*** name. The name must be different from any other predefined or enumerated type name because the name space for types is global. |
| *name* | A legal ***e*** name. Each name must be unique within the type. |
| *exp* | 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. Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1. |
| *width-exp* | 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 ***n*** is the number of bits. |

## Description

Defines an enumerated scalar type having the name you specify and 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.

## Example 1

This is a simple example of the basic syntax.

```
type PacketType :[ rx, tx, ctrl ];

struct packet {
    kind :PacketType;

    do_print() is {
        if kind == ctrl {
            out("This is a control packet.");
        };
    };
};
```

## Example 2

This example shows how HDL variables are automatically cast to the required scalar type.

```
type PacketType :[ rx, tx, ctrl ];

struct packet {
    kind :PacketType;

    set() is {
        kind = 'top.pkt_type';
    };
};
```

## Example 3

This example shows an enumerated type with a bit width:

```
type NetworkType :[ IP=0x0800, ARP=0x8060 ](bits:16);
```

```
struct header {
    dest_address :uint(bits:48);
    src_address  :uint(bits:48);
    type         :NetworkType;

    do_print() is {
        if type == IP {
            out("This is an IP packet.");
        };
    };
};
```

### Example 4

This example shows how to type cast between an enumerated type and an unsigned integer.

```
type signal_number: [signal_0, signal_1, signal_2, signal_3];
struct signal {
    cast_1() is {
        var temp_val: uint = 3;
        var signal_name: signal_number = temp_val.as_a(signal_number);
        print signal_name;
    };

    cast_2() is {
        var temp_enum: signal_number = signal_0;
    var signal_value: uint = temp_enum.as_a(uint);
        print signal_value;
    };
};
```

### See Also

— "type scalar subtype" on page 100
— "type sized scalar" on page 101
— "extend type" on page 103
— "as_a()" on page 104
— "Enumerated Scalar Types" on page 77

## 3.2.2 type scalar subtype

### Purpose

Define a scalar subtype

### Category

Statement

### Syntax

**type** ***scalar-subtype-name*****:** ***scalar-type*** **[*range, …*]**

Syntax example:

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

### Parameters

| | |
|---|---|
| *scalar-subtype-name* | A unique ***e*** name. |
| *scalar-type* | 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. |
| *range* | A constant expression or two constant expressions separated by two dots. All constant expressions must resolve to legal values of the named type. |

### Description

Defines a subtype of a scalar type by restricting the legal values that can be generated for this subtype to the specified range.

NOTE— The default value for variables or fields of this type “size” is zero, the default for all integers; the range affects only the generated values.

### Example 1

The integer subtype defined below includes all non-negative integers except 4,5, and 7.

```
type medium: uint [0..3,6,8..MAX_INT];
```

### Example 2

The following example defines the “inst” type, which has five legal instruction values, and the subtype “mem_inst”, which has only the values related to memory.

```
type inst: [add, sub, mul, div, load, store];
type mem_inst: inst [load..store];
```

Example 3

You can omit the range list, thus renaming the full range. The first example below gives the name “my_int” to the full range of integers. The second example gives the name “true_or_false” to the full range of the boolean type.

```
type my_int: int;
type true_or_false: bool;
```

### See Also

— “type enumerated scalar” on page 98
— “type sized scalar” on page 101
— “extend type” on page 103
— “Scalar Subtypes” on page 76

## 3.2.3 type sized scalar

### Purpose

Define a sized scalar

### Category

Statement

### Syntax

**type *sized-scalar-name*: *type* (bits | bytes: *exp*)**

Syntax example:

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

### Parameters

| | |
|---|---|
| *sized-scalar-name* | A unique ***e*** name. |
| *type* | Any previously defined enumerated type or any of the predefined scalar types, including **int**, **uint**, **bool**, or **time**. |
| *exp* | A positive constant expression. The valid range of values for sized scalars is limited to the range 1 to 2***n*** - 1, where ***n*** is the number of bits. |

### Description

Defines a scalar type with a specified bit width. The actual bit width is ***exp*** * 1 for bits and ***exp*** * 8 for bytes. In the example shown below, both types “word” and “address” have a bit width of 16.

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

### Example

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 if the expression is unsigned, or with the sign bit (zero or one) if it is a signed expression.

Here is an example of assigning an expression where the expression's value is truncated:

```
type SmallAddressType :uint(bits:2);

extend sys {
    chop_expression() is {
        var small_address :SmallAddressType;

        small_address = 0x2 * 8;
        out("small_address: ", small_address);
    };

    run() is also {
        chop_expression();
    };
};
```

### See Also

- — "type enumerated scalar" on page 98
- — "type scalar subtype" on page 100
- — "extend type" on page 103
- — "Assignment Rules" on page 89
- — "Precision Rules for Numeric Operations" on page 93

## 3.2.4 extend type

### Purpose

Extend an enumerated scalar type

### Category

Statement

### Syntax

**extend *enum-type*: [*name*[= *exp*], ...]**

Syntax example:

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

### Parameters

| | |
|---|---|
| *enum-type* | Any previously defined enumerated type. |
| *name* | A legal ***e*** name. Each name must be unique within the type. |
| *exp* | 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. Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1. |

### Description

Extends the specified enumerated scalar type to include the names or name-value pairs you specify.

### Example 1

This is an example of the basic syntax.

```
    type command   :[ ADD=0x00, SUB=0x02, AND=0x04,
                      XOR=0x06, UDEF=0xFF ] (bits: 8);

    extend command :[ ADDI=0x01, SUBI=0x03,
                      ANDI=0x05, XORI=0x07 ];
```

### Example 2

A common use of type extension is defining a protocol type and extending it as new protocols are added to the test environment. For example, you can define a packet header without having to know what specific network protocols are supported by the packet:

```
type NetworkType :[ ](bits:16);

struct header {
    dest_address :uint(bits:48);
    src_address  :uint(bits:48);
    type         :NetworkType;
};
```

As protocols are gradually added to the test environment, the new protocol type can be added without changes to the original code:

```
extend NetworkType :[ ARP=0x8060 ];
```

Then again for more protocols:

```
extend NetworkType :[ IP=0x0800 ];
```

### See Also

- "type enumerated scalar" on page 98
- "type scalar subtype" on page 100
- "type sized scalar" on page 101
- "e Data Types" on page 75

## 3.3 Type Conversion Between Scalars and Strings

This section contains:

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 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105 and Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107.

This section contains:

- "as_a()" on page 104
- "all_values()" on page 115

### 3.3.1 as_a()

#### Purpose

Casting operator

#### Category

Expression

### Syntax

***exp*.as_a(*type***: type name**)**: type

Syntax example:

```
print (b).as_a(uint);
```

### Parameters

| | |
|---|---|
| *exp* | Any ***e*** expression. |
| *type* | Any legal ***e*** type. |

### Description

Returns the expression, converted into the specified type. Although some casting is done automatically (see "Automatic Type Casting" on page 96), explicit casting is required in some cases when making assignments between different but compatible types.

### 3.3.1.1 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 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105 shows the recommended methods for converting between scalars and lists of scalars.

In Table 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105, **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 cannot be used as the integer width.

The solutions assume that there is a variables 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 may have to be accomplished in two stages.

**Table 3-4—Type Conversion Between Scalars and Lists of Scalars**

| From | To | Solutions |
|---|---|---|
| int | list of bit | list_of_bit = int**[..]** |

**Table 3-4—Type Conversion Between Scalars and Lists of Scalars *(continued)***

| From | To | Solutions |
|---|---|---|
| int | list of int(bits:x) | list_of_int = **%**{int}<br><br>list_of_int = **pack(packing.low**, int**)**<br><br>(LSB of int goes to list[0] for either choice) |
| list of bit<br><br>list of byte | int | int = list_of_bit**[:]** |
| list of int(bits:x) | int | int = **pack(packing.low**, list_of_int**)**<br><br>(Use **packing.high** for list in other order.) |
| int(bits:x) | int(bits:y) | intx = inty<br><br>(Truncation or extension is automatic.)<br><br>intx.**as_a(**int(bits:y)**)** |
| bool | int | int = bool.**as_a(**int**)**<br><br>(TRUE becomes 1, FALSE becomes 0.) |
| int | bool | bool = int.**as_a(**bool**)**<br><br>(0 becomes FALSE, non-0 becomes TRUE.) |
| int | *enum* | ***enum*** = int.**as_a(*enum*)**<br><br>(No checking is performed to make sure the int value is valid for the range of the enum.) |
| *enum* | int | int = ***enum*.as_a(**int**)**<br><br>(Truncation is automatic.) |
| *enum* | bool | ***enum*.as_a(**bool**)**<br><br>(Enumerated types with an associated unsigned integer value of 0 become FALSE; those with an associated non-0 values become TRUE. See "Enumerated Scalar Types" on page 77 for more information on values associated with enumerated types.) |
| bool | *enum* | bool.**as_a(*enum*)**<br><br>(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.) |

**Table 3-4—Type Conversion Between Scalars and Lists of Scalars *(continued)***

| From | To | Solutions |
|---|---|---|
| *enum* | *enum* | ***enum1*** = ***enum2*.as_a(*enum1*)**<br><br>(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) | listx.**as_a(**list of int(bits:y)**)**<br><br>(same number of items, each padded or truncated)<br><br>listy = **pack(packing.low**, listx**)**<br><br>(concatenated data, different number of items) |

### 3.3.1.2 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 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107 shows how to convert between strings and scalars or lists of scalars.

In Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, **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 cannot be used as the integer width.

The solutions assume that there is a variables 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 may have to be accomplished in two stages.

**Table 3-5—Type Conversion Between Strings and Scalars or Lists of Scalars**

| From | To | ASCII Convert? | Solutions |
|---|---|---|---|
| list of int<br><br>list of byte | string | yes | list_of_int.**as_a(**string**)**<br><br>(Each list item is converted to its ASCII character and the characters are concatenated into a single string. int[0] represents left-most character. If a list item is not a printable ASCII character, the string returned is empty.) |

**Table 3-5—Type Conversion Between Strings and Scalars or Lists of Scalars *(continued)***

| From | To | ASCII Convert? | Solutions |
|---|---|---|---|
| string | list of int<br><br>list of byte | yes | string.**as_a(**list of int**)**<br><br>(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 int[0]) |
| string | list of int | yes | list_of_int = **pack(packing.low**, string**)**<br><br>list_of_int = **%{**string**}**<br><br>(The numeric values of the characters are concatenated before assigning them to the list. Any pack option gives same result; null byte, 00, will be last item in list.) |
| string | int | yes | int = **%{**string**}**<br><br>int = **pack(packing.low**, string**)**<br><br>(Any pack option gives same result.) |
| int | string | yes | **unpack(packing.low**, %{8’b0, int}, string**)**<br><br>(Any pack option with scalar_reorder={} gives same result.) |
| string | int | no | string.**as_a(**int**)**<br>(Converts to decimal.)<br><br>append(“0b”, string).**as_a(**int**)**<br>(Converts to binary.)<br><br>append(“0x”, string).**as_a(**int**)**<br>(Converts to hexadecimal.) |
| int | string | no | int**.as_a(**string**)**<br>(Uses the current print radix.)<br><br>**append(**int**)**<br>(Converts int according to current print radix.)<br><br>**dec(**int**)**, **hex(**int**)**, **bin(**int**)**<br>(Converts int according to specific radix.) |
| string | bool | no | bool = string**.as_a(**bool**)**<br><br>(Only “TRUE” and “FALSE” can be converted to boolean; all other strings return an error.) |
| bool | string | no | string = bool**.as_a(**string**)** |

**Table 3-5—Type Conversion Between Strings and Scalars or Lists of Scalars *(continued)***

| From | To | ASCII Convert? | Solutions |
|---|---|---|---|
| string | *enum* | no | ***enum*** = string.**as_a(*enum*)** |
| *enum* | string | no | string = ***enum*.as_a(**string**)** |

### 3.3.1.3 Type Conversion Between Structs, Struct Subtypes, and Lists of Structs

Struct subtypes are automatically cast to their base struct type, so, for example, you can assign a variable of type “Ethernet packet” to a variable of type “packet” without using **as_a()**.

You can use **as_a()** 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 must have a common parent type. That is, the specified type parameter must be a subtype of the type of the list.

Assigning a struct subtype to a base struct type does not change the declared type. Thus, you have to use **as_a()** to cast the base struct type as the subtype in order to access any of the subtype-specific struct members. See Example 6 on page 112.

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

### 3.3.1.4 Type Conversion Between Simple Lists and Keyed Lists

You can convert simple lists to keyed lists and keyed lists to simple lists. When you convert a keyed list to a simple list, the hash key is dropped. When you convert a simple list to a keyed list, you must specify the key.

For example, if “sys.packets” is a simple list of packets and you want to convert it to a keyed list where the “len” field of the packet struct is the key, you can do so like this:

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

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)” returns 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. For example, it is a rule that if you use **as_a()** to convert an integer to a string, no ASCII conversion is performed. Similarly, if you use **as_a()** to convert a simple list of integers to a keyed list of strings, no ASCII conversion is performed:

```
var lk: list (key:it) of string
var l: list of int = {1;2;3;4;6;9}
```

```
lk = l.as_a(list (key:it) of string)
print lk.key_index("9")
lk.key_index("9") = 5
```

NOTE— 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.

### Example 1

In this example, the most significant bits of the 32-bit variable “i” are truncated when “i” is printed as a 16-bit variable. When “i” is printed as a 64-bit variable, it is sign-extended to fit.

```
extend sys {
    m() is {
        var i : int = 0xffff000f;
        print (i).as_a(int(bits:16)) using radix=HEX;
        print (i).as_a(int(bits:64)) using radix=HEX;
    };
};
```

### Result

```
(i).as_a(int(bits:16)) = 0x000f
(i).as_a(int(bits:64)) = 0xffffffffffff000f
```

### Example 2

No checking is performed when “c”, a variable of type color, is assigned a value outside its range. However, a message is issued when the “c” is accessed by the **print** statement.

```
type color: [red=2, blue=0, yellow=1];
extend sys{
    m() is {
        var c : color = blue;
        var i : int = 2;
        var u : uint = 0x74786574;
        print (i).as_a(color);
        print (c).as_a(int);
        c = u.as_a(color);      --no checking
        print c;                --message issued
    };
};
```

### Result

```
(i).as_a(color) = red
(c).as_a(int) = 0x0
c = (Bad enum value for 'color': 1954047348)
```

### Example 3

You can use the **as_a()** method to convert a boolean type to a numeric or an enumerated type or from one of those types to a boolean.

```
type color: [red=2, blue=0, yellow=1];
extend sys{
    m() is {
            var c : color = blue;
            var i : int = 2;
            var s : string = "hello";
            print (i).as_a(bool);
            print (c).as_a(bool);
        };
    };
```

### Result

```
    (i).as_a(bool) = TRUE
    (c).as_a(bool) = FALSE
```

### Example 4

You can cast between numeric types and strings with **as_a()**, but no ASCII conversion is performed. This example shows how to get ASCII conversion using **unpack()** and the bit concatenation operator **%{}**.

```
    extend sys{
        m() is {
            var i : int = 65;
            var s1 : string;
            var s2 : string = "B";
            print (i).as_a(string);
            unpack(packing.low, %{8'b0,i}, s1);
            print s1;
            --print (s2).as_a(int); --run-time error;
                                    --"B" is not a valid integer
            i = %{s2};
            print i;
        };
    };
```

### Result

```
    (i).as_a(string) = "65"
    s1 = "A"
    i = 66
```

### Example 5

You can cast between lists of numerics and strings with **as_a()**. As shown in the first print statement, each character in the string is converted to its numeric value and assigned to a separate element in the list. As shown in the second to last print statement, using **pack()** to convert a string concatenates the numeric values of the characters before assigning them to the list.

```
    extend sys {
        m() is {
            var s: string;
            s = "hello";
            var lint: list of int;
            lint = s.as_a(list of int);
            print lint;
            print lint.as_a(string);
            var lint2: list of int;
            lint2 = pack(packing.low, s);
            print lint2 using bin;
            print lint using bin;
        };
    };
```

### Result

```
    lint =
    0.      104
    1.      101
    2.      108
    3.      108
    4.      111
      lint.as_a(string) = "hello"
      lint2 =
    0.      0b1101100011011000110010101101000
    1.      0b1101111
      lint =
    0.      0b1101000
    1.      0b1100101
    2.      0b1101100
    3.      0b1101100
    4.      0b1101111
```

### Example 6

The "**print** pkt.**as_a(**foreign packet**)"** action below results in "pkt.as_a(foreign packet) = NULL" because "pkt" is of type "Ethernet packet".

The "**print** pkt.e_field" action in this example results in a compile-time error because the declared type of "pkt" does not have a field "e_field". However, the "**print** pkt.as_a(Ethernet packet).e_field" action prints the value of the field.

```
    type packet_protocol: [Ethernet, IEEE, foreign];
    struct packet {
        protocol: packet_protocol;
        size: int [0..1k];
        data[size]: list of byte;
        show() is undefined; // To be defined by children
    };
    extend Ethernet packet {
        e_field: int;
        show() is {out("I am an Ethernet packet")};
    };
    extend sys {
        m() is {
            var epkt: Ethernet packet = new;
            var pkt: packet = epkt;
            print pkt.type().name;
            print pkt.declared_type().name;
            print pkt.as_a(foreign packet);
    --      print pkt.e_field;              //compile-time error
            print pkt.as_a(Ethernet packet).e_field;
            print pkt.size;
        };
    };
```

### Result

```
    pkt.type().name = "Ethernet packet"
```

```
pkt.declared_type().name = "packet"
pkt.as_a(foreign packet) = (NULL)
pkt.as_a(Ethernet packet).e_field = 0
pkt.size = 0
```

### Example 7

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.

To pass a list of integer(bits: 4) as a parameter to a method that requires a list of integers, you can use explicit casting, as follows:

```
struct dtypes {
    increment_list (cnt: list of int) is {
        for each in cnt {
            cnt[index] = cnt[index] + 1;
        };
    };
};

extend sys {
    di:dtypes;
    m() is {
        var small_list: list of int (bits: 5) = {3;5;7;9};
        var big_list: list of int = {0;0;0;0;};
        big_list = small_list.as_a(list of int);
        di.increment_list(big_list);
        print big_list;
    };
};
```

### Result

The print statement gives the following results:

```
big_list =
0.      4
1.      6
2.      8
3.      10
```

### Example 8

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.

```
type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined; // To be defined by children
};
extend Ethernet packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};
extend sys {
    packets[5]: list of packet;
    post_generate() is also {
        print packets;
        print packets.as_a(list of IEEE packet);
    };
};
```

### Result

```
Doing setup ...
Generating the test using seed 1...
  packets =
item   type        protocol    size        data        Ethernet'*

0.     packet      Ethernet    872         (872 item*  -21166003*
1.     packet      Ethernet    830         (830 item*  -21443627*
2.     packet      Ethernet    834         (834 item*  1684201428
3.     packet      Ethernet    663         (663 item*  -15262725*
4.     packet      IEEE        213         (213 item*
  packets.as_a(list of IEEE packet) =
item   type        protocol    size        data        Ethernet'*

0.     packet      IEEE        213         (213 item*
```

### Example 9

You can use **as_a()** to convert a string to an enumerated type. The string has to match letter by letter one of the possible values of that type or a runtime error is issued.

This example sets a list of items of an enumerated type to the values read from a file.

```
type reg_address: [UARTCTL1, UARTDATA1, UARTCTL2, UARTDATA2];
extend sys {
    ctl_regs: list of reg_address;

    keep ctl_regs == (each line in file
        "~/data/enum_items.txt").apply(it.as_a(reg_address));
    run() is also {
        print sys.ctl_regs;
    };
};
```

### enum_items.txt

```
UARTCTL1
UARTDATA1
UARTCTL2
UARTDATA2
UARTDATA1
UARTCTL2
UARTCTL2
UARTCTL1
UARTDATA1
```

### Result

```
sys.ctl_regs =
0.      UARTCTL1
1.      UARTDATA1
2.      UARTCTL2
3.      UARTDATA2
4.      UARTDATA1
5.      UARTCTL2
6.      UARTCTL2
7.      UARTCTL1
8.      UARTDATA1
```

NOTE— If the file is not accessible, you will see a runtime error with the name of the missing file. If there is a typo in the file, you will see a runtime error message like the following:

```
*** Error: Enum type 'reg_address' has no item called 'UARTCTL'
```

### See Also

- "Automatic Type Casting" on page 96
- "e Data Types" on page 75
- "is [not] a" on page 67

## 3.3.2 all_values()

### Purpose

Access all values of a scalar type

### Category

Pseudo routine

### Syntax

**all_values(*scalar-type*:** type name**)**: list of scalar type

Syntax example:

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

### Parameters

| | |
|---|---|
| *scalar-type* | Any legal ***e*** scalar type. |

### Description

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.

NOTE— When the specified type has more than 1million legal values, this routine gives a compile time error to alert you to possible memory abuse.

### Example

```
type reg_address: [UARTCTL1, UARTDATA1, UARTCTL2, UARTDATA2];
extend sys {
    ctl_regs: list of reg_address;

    keep ctl_regs ==
      all_values(reg_address).all(it.as_a(string) ~"*CTL*");
    run() is also {
        print sys.ctl_regs;
    };
};
```

### Result

```
Running the test ...
  sys.ctl_regs =
0.      UARTCTL1
1.      UARTCTL2
```

# 4 Structs, Fields, and Subtypes

The basic organization of an ***e*** program is a tree of structs. A struct is a compound type that contains data fields, procedural methods, and other members. It is the ***e*** equivalent of a class in other object-oriented languages. A base struct type can be extended by adding members. Subtypes can be created from a base struct type which inherit the base type's members, and contain additional members.

This chapter contains the following sections:

**See Also**

— "Syntactic Elements" on page 11
— "Struct Hierarchy and Name Resolution" on page 19
— "Overview of e Data Types" on page 75
— "Defining and Extending Scalar Types" on page 98
— Chapter 7.1, "Basic Concepts of Generation"
— Chapter 15.1, "Rules for Defining and Extending Methods"
— Chapter 19, "List Pseudo-Methods Library"
— Chapter 8.1, "Events Overview"
— Chapter 10, "Temporal Struct Members"
— Chapter 12, "Coverage Constructs"

## 4.1 Structs Overview

Structs are used to define data elements and behavior of components of a test environment. A struct can hold all types of data and methods.

All user-defined structs inherit from the predefined base struct type, **any_struct**.

For reusability of ***e*** code, you can add struct members or change the behavior of a previously defined struct with **extend**.

Inheritance is implemented in ***e*** by either of two of aspects of a struct definition:

— "when" inheritance is specified by defining subtypes with **when** struct members
— "like" inheritance is specified with the **like** clause in new struct definitions

The best inheritance methodology for most applications is "when" inheritance. See "Comparison of When and Like Inheritance" on page 142 for more information.

## 4.2 Defining Structs: struct

### Purpose

Define a data struct

### Category

Statement

### Syntax

**struct *struct-type*** [**like *base-struct-type***] **{**
[***struct-member***; ...]**}**

Syntax example:

```
type packet_kind: [atm, eth];
struct packet {
    len: int;
    keep len < 256;
    kind: packet_kind;
};
```

## Parameters

| | |
|---|---|
| *struct-type* | The name of the new struct type. |
| *base-struct-type* | The type of the struct from which the new struct inherits its members. |
| ***struct-member***; ... | The contents of the struct. The following are types of struct members: |

- data fields for storing data
- methods for procedures
- events for defining temporal triggers
- coverage groups for defining coverage points
- **when**, for specifying inheritance subtypes
- declarative constraints for describing relations between data fields
- **on**, for specifying actions to perform upon event occurrences
- **expect**, for specifying temporal behavior rules

The definition of a struct can be empty, containing no members.

## Description

Structs are used to define the data elements and behavior of components and the test environment. Structs contain struct members of the types listed in the Parameters table. Struct members can be conditionally defined (see "Creating Subtypes with When" on page 133).

The optional **like** clause is an inheritance directive. All struct members defined in ***base-struct-type*** are implicitly defined in the new struct subtype, ***struct-type***. New struct members can also be added to the inheriting struct subtype, and methods of the base struct type can be extended in the inheriting struct subtype.

## Example 1

A struct type named "transaction" is defined in this example.

```
struct transaction {
    address: uint;
    data: list of uint;
    transform(multiple:uint) is empty;
};
```

The "transaction" struct contains three members:

— "address" field
— "data" field
— "transform()" empty method definition

### Example 2

In this example, a “pci_transaction” struct is derived from the “transaction” struct in the previous example, using **like** inheritance. The following struct members are added in this inherited struct:

— Fields named “command”, “dual_address”, and “bus_id”
(a **type** statement is included, to enumerate values for the “command” field)
— A **keep** constraint
— A **when** conditional subtype
— An **event** definition
— An **on** member
— A **cover** group definition

The “transform()” method, defined as empty in the “transaction” base type, is given a method body using the **is only** method extension syntax.

```
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                        MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction like transaction {
    command : PCICommandType;
    keep soft data.size() in [0..7];
    dual_address: bool;
    when dual_address pci_transaction {
        address2: uint;
    };
    bus_id: uint;
    event initiate;
    on initiate {
        out("An event has been initiated on bus ", bus_id);
    };
    cover initiate is {
        item command;
    };
    transform(multiple:uint) is only {
        address = address * multiple;
    };
};
```

### Example 3

Additional subtypes can, in turn, be derived from a subtype. In the following example, an “agp_transaction” subtype is derived from the “pci_transaction” subtype of the previous example. Each subtype can add fields to its base type, and place its own constraints on fields of its base type.

```
type AGPModeType: [AGP_2X, AGP_4X];
struct agp_transaction like pci_transaction {
    block_size: uint;
    mode: AGPModeType;
    when AGP_2X agp_transaction {
        keep block_size == 32;
    };
    when AGP_4X agp_transaction {
        keep block_size == 64;
    };
};
```

### See Also

— "Struct Members" on page 13
— "Extending Structs: extend type" on page 121
— "Extending Subtypes" on page 123
— "Creating Subtypes with When" on page 133
— "Comparison of When and Like Inheritance" on page 142
— Chapter 3, "Data Types"
— Chapter 15, "Methods"
— Chapter 7, "Generation Constraints"
— Chapter 8, "Events"
— Chapter 10, "Temporal Struct Members"
— Chapter 12, "Coverage Constructs"

## 4.3 Extending Structs: extend type

### Purpose

Extend an existing data struct

### Category

Statement

### Syntax

**extend** [***struct-subtype***] ***base-struct-type*** **{**
    [***struct-member***; ...]**}**

Syntax example:

```
type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend packet {
    keep len < 256;
};
```

### Parameters

| | |
|---|---|
| *struct-subtype* | Adds struct members to the specified subtype of the base struct type only. The added struct members are known only in that subtype, not in other subtypes. |
| *base-struct-type* | The base struct type to extend. |
| ***member***; ... | The contents of the struct. A struct member is one of the following types: |

- data fields for storing data
- methods for procedures
- events for defining temporal triggers
- coverage groups for defining coverage points
- **when**, for specifying inheritance subtypes
- declarative constraints for describing relations between data fields
- **on**, for specifying actions to perform upon event occurrences
- **expect**, for specifying temporal behavior rules

The extension of a struct can be empty, containing no members.

### Description

Adds struct members to a previously defined struct or struct subtype.

Members added to the base struct type in extensions apply to all other extensions of the same struct. Thus, for example, if you extend a method in a base struct with **is only**, it overrides that method in every one of the **like** children.

NOTE— If **like** inheritance has been used on a struct type, there are limitations on how the original base struct type definition can be further extended with **extend**. See "Restrictions on Like Inheritance" on page 149.

### Example 1

In the following example, a struct type named "pci_transaction" is defined in one module, which is then imported into another module where a field named "data_phases" and two constraints are added in an extension to the struct.

```
<'
// module pci_transaction_definition.e
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                        MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction {
    address: uint;
    data: list of uint;
    command : PCICommandType;
    bus_id: uint;
    event initiate;
    on initiate {
        out("An event has been initiated on bus ", bus_id);
    };
```

```
        cover initiate is {
            item command;
        };
    };
    '>

    <'
    // module pci_transaction_extension.e
    import pci_transaction_definition;
    extend pci_transaction {
        data_phases: uint;
        keep data_phases in [0..7];
        keep data.size() == data_phases;
    };
    '>
```

### Example 2

In the following, the “tx_packet” struct inherits its kind field from the “packet” struct definition, from which it is derived using like inheritance. The “keep kind == atm” constraint in the packet struct extension applies to both packet instances and tx_packet instances. The “keep len > 10” constraint in the tx_packet subtype applies only to tx_packet instances, reducing the range of len in tx_packet instances to [11..40]:

```
    type packet_kind: [atm, eth];
    struct packet {
        len: int;
        keep soft len <= 40;
        kind: packet_kind;
    };
    struct tx_packet like packet {
        send_delay: int [0..100];
        keep len > 10;
    };
    extend packet {
        keep kind == atm;
    };
```

### See Also

- “Defining Structs: struct” on page 118
- “Extending Subtypes” on page 123
- “Creating Subtypes with When” on page 133
- “Comparison of When and Like Inheritance” on page 142
- Chapter 3, “Data Types”
- Chapter 15, “Methods”
- Chapter 7, “Generation Constraints”
- Chapter 8, “Events”
- Chapter 10, “Temporal Struct Members”
- Chapter 12, “Coverage Constructs”

## 4.4 Extending Subtypes

A struct subtype is an instance of the struct in which one of its fields has a particular value. For example, the “packet” struct defined in the following example has “atm packet” and “eth packet” subtypes, depending on whether the “kind” field is “atm” or “eth”.

```
type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend packet {
    keep len < 256;
};
```

A struct subtype can optionally be specified with **extend**, so that the extension only applies to that subtype.

### Example 1

The following shows a definition of a struct type named “packet”, an extension that adds a field named “len” to the struct definition, and a second extension that adds a field named “transmit_size” only to packets whose “kind” is “transmit”.

```
type packet_kind: [transmit, receive];
struct packet {
    kind: packet_kind;
};
extend packet {
    len: int;
};
extend transmit packet {
    transmit_size: int;
};
The "extend transmit packet" syntax above is equivalent to:
extend packet {
    when transmit packet {
        transmit_size: int;
    };
};
```

### Example 2

The “packet” struct definition below is extended with a boolean field named “legal”. Two additional extensions add a field named “header” to the packet struct: for packets whose “legal” value is TRUE, the “header” field gets a “legal_header” struct instance. For packets whose “legal” values is FALSE, the “header” field gets a “bad_header” struct instance.

```
type packet_kind: [atm, eth];
struct packet {
    len: int;
    keep soft len == 40;
    kind: packet_kind;
};

extend packet{
    legal: bool;
};

struct legal_header {
    legal_ck: byte;
};
```

```
struct bad_header {
    bad_ck: byte;
};

extend legal packet {
    header: legal_header;
};
extend FALSE'legal packet {
    header: bad_header;
};
```

### See Also

— “Defining Structs: struct” on page 118
— “Extending Structs: extend type” on page 121
— “Creating Subtypes with When” on page 133
— “Extending When Subtypes” on page 136

## 4.5 Defining Fields: field

### Purpose

Define a struct field

### Category

Struct member

### Syntax

[**!**][**%**] ***field-name***[**:** ***type***] [[***min-val*** .. ***max-val***]] [((**bits** | **bytes**):***num***)]

Syntax example:

```
type NetworkType: [IP=0x0800, ARP=0x8060] (bits: 16);
struct header {
    address: uint (bits: 48);
    hdr_type: NetworkType;
    !counter: int;
};
```

### Parameters

| | |
|---|---|
| ! | Denotes an ungenerated field. The "!" and "%" options can be used together, in either order. |
| % | Denotes a physical field. The "!" and "%" options can be used together, in either order. |
| *field-name* | The name of the field being defined. |
| *type* | The type for the field. This can be any scalar type, string, struct, or list.<br><br>If the field name is the same as an existing type, you can omit the ": ***type***" part of the field definition. Otherwise, the type specification is required. |
| ***min-val*..*max-val*** | An optional range of values for the field, in the form. If no range is specified, the range is the default range for the field's type. |
| (bits \| bytes: *num*) | The width of the field in bits or bytes. This syntax allows you to specify a width for the field other than the default width.<br><br>This syntax can be used for any scalar field, even if the field has a type with a known width. |

### Description

Defines a field to hold data of a specific type. You can specify whether it is a physical field or a virtual field, and whether the field is to be automatically generated. For scalar data types, you can also specify the size of the field in bits or bytes.

### Physical Fields

A field defined as a physical field (with the "%" option) is packed when the struct is packed. Fields that represent data that is to be sent to the HDL device in the simulator or that are to be used for memories, need to be physical fields. Nonphysical fields are called virtual fields and are not packed automatically when the struct is packed, although they can be packed individually.

If no range is specified, the width of the field is determined by the field's type. For a physical field, if the field's type does not have a known width, you must use the **(bits | bytes : num)** syntax to specify the width.

### Ungenerated Fields

A field defined as ungenerated (with the "!" option) is not generated automatically. This is useful for fields that are to be explicitly assigned during the test, or whose values involve computations that cannot be expressed in constraints.

Ungenerated fields get default initial values (0 for scalars, NULL for structs, empty list for lists). An ungenerated field whose value is a range (such as [0..100]) gets the first value in the range. If the field is a struct, it will not be allocated and none of the fields in it will be generated.

### Assigning Values to Fields

Unless you define a field as ungenerated, a value is generated for it when the struct is generated, subject to any constraints that exist for the field. However, even for generated fields, you can always assign values in user-defined methods or predefined methods such as **init()**, **pre_generate()**, or **post_generate()**. The ability to assign a value to a field is not affected by either the "!" option or generation constraints.

### Example

The struct definitions below contain several types of fields.

```
type NetworkType: [IP=0x0800, ARP=0x8060] (bits: 16);
struct header {
    %address: uint (bits: 48);
    %length: uint [0 .. 32];
};
struct packet {
    hdr_type: NetworkType;
    %hdr: header;
    is_legal: bool;
    !counter: uint;
};
extend sys {
    packet;
};
```

The “header” struct contains two physical fields:

— A field named “address” which is a 48-bit field of data type **uint**
— A field named “length” of data type **uint**

The “packet” struct contains:

— An enumerated “hdr_type” field that can be either “IP” or “ARP”
— A physical field named “hdr” of type “header”, which will hold an instance of the “header” struct
— A boolean “is_legal” field
— An ungenerated **uint** field named “counter”

The **sys** struct extension contains a field for an instance of a “packet” struct. No type declaration is required for the “packet” field in the **sys** extension, since the field name is the same as the name of a type that was already defined.

### See Also

— Chapter 3, “Data Types”
— Chapter 7, “Generation Constraints”
— Chapter 17, “Packing and Unpacking”
— “list of” on page 127

## 4.6 Defining List Fields

This section shows the syntax and examples of lists in general, and of keyed lists. It contains these topics:

— “list of” on page 127
— “list(key) of” on page 129

### 4.6.1 list of

#### Purpose

Define a list field

### Category

Struct member

### Syntax

**[!][%]*list-name*[[*length-exp*]]: list of *type***

Syntax example:

```
packets: list of packet;
```

### Parameters

| | |
|---|---|
| ! | Do not generate this list. The “!” and “%” options can be used together, in either order. |
| % | Denotes a physical list. The “!” and “%” options can be used together, in either order. |
| *list-name* | The name of the list being defined. |
| *length-exp* | An expression that gives the initial size for the list. The expression must evaluate to a non-negative integer. |
| *type* | The type of items in the list. This can be any scalar type, string, or struct. It cannot be a list. |

### Description

Defines a list of items of a specified type.

An initial size can be specified for the list. The list initially contains that number of items. The size conforms to the initialization rules, the generation rules and the packing rules. Even if an initial size is specified, the list size can change during the test if the list is operated on by a list method that changes the number of items.

All list items are initialized to their default values when the list is created. For a generated list, the initial default values are replaced by generated values.

For information about initializing list items to particular values, see “Assignment of Lists” on page 93 and “Constraining Lists” on page 264.

### Example 1

Three list fields are defined in the struct definitions below. The “cell” struct contains a list of bytes, the “packet” struct contains a list of “cell” struct instances, and the **sys** struct extension contains a list of 16 “packet” struct instances.

```
struct cell {
    %data: list of byte;
    %length: uint;
};
struct packet {
    %is_legal: bool;
    cells: list of cell;
};
```

```
extend sys {
    packets[16]: list of packet;
};
```

### Example 2

Two lists of cells are defined in this following example, both with initial sizes specified using the [***length***] syntax. For the cells_1 list, the length expression is a value generated from within the 16 to 32 range specified for the num_cells field. For the cells_2 list, the length expression is the integer value of an item from the enumerated list named l_sel (sm has value 0, med has value 1, lge has value 3 due to their positions in the enumerated list).

```
struct cell {
    %data: list of byte;
    %length: uint;
};
struct packet {
    %is_legal: bool;

    num_cells: int;
    keep num_cells in [16..32];
    cells_1[num_cells]: list of cell;

    l_sel: [sm, med, lge];
    cells_2[l_sel.as_a(int)]: list of cell;
};
```

### See Also

— “Defining Fields: field” on page 125
— “Expressions” on page 19
— Chapter 3, “Data Types”
— Chapter 7, “Generation Constraints”
— Chapter 17, “Packing and Unpacking”
— Chapter 19, “List Pseudo-Methods Library”

## 4.6.2 list(key) of

### Purpose

Define a keyed list field

### Category

Struct member

### Syntax

**![%]*list-name*: list(key: *key-field*) of *type***

Syntax example:

```
!locations: list(key: address) of location;
```

## Parameters

| | |
|---|---|
| ! | Do not generate this list. For a keyed list, the “!” is required, not optional. |
| % | Denotes a physical list. The “%” option may precede or follow the “!”. |
| *list-name* | The name of the list being defined. |
| *key-field* | The key of the list. For a list of structs, it is the name of a field of the struct. For a list of scalar or string items, it is the item itself, represented by the **it** variable.<br><br>This is the field or value which the keyed list pseudo-methods will check when they operate on the list. |
| *type* | The type of items in the list. This can be any scalar type, string, or struct. It cannot be a list. |

## Description

Keyed lists are used to enable faster searching of lists by designating a particular field or value which is to be searched for. A keyed list can be used, for example, in the following ways:

- As a hash table, in which searching only for a key avoids the overhead of reading the entire contents of each item in the list.
- For a list that has the capacity to hold many items, but which in fact contains only a small percentage of its capacity, randomly spread across the range of possible items. An example is a sparse memory implementation.

Although all of the operations that can be done using a keyed list can also be done using a regular list, using a keyed list provides an advantage in the greater speed of searching a keyed list.

Besides the **key** parameter, the keyed list syntax differs from regular list syntax in the following ways:

- The list must be declared with the “**!**” do-not-generate operator. This means that you must build a keyed list item by item, since you cannot generate it.
- The “**[*exp*]**” list size initialization syntax is not allowed for keyed lists. That is, “***list*[*exp*]: list(key: *key*) of *type***” is not legal. Similarly, you cannot use a **keep** constraint to constrain the size of a keyed list.

The keyed list pseudo-methods (see “Keyed List Pseudo-Methods” on page 618) only work on lists that were defined and created as keyed lists. Conversely, restrictions apply when using regular list pseudo-methods or other operations on keyed lists. See “Restrictions on Keyed Lists” on page 624.

A keyed list is a distinct type, different from a regular list. This means that you cannot assign a keyed list to a regular list, nor assign a regular list to a keyed list: if list_a is a keyed list and list_b is a regular list, list_a = list_b is a syntax error.

If the same key value exists in more than one item in a keyed list. the keyed list pseudo-methods always use the item latest in the list (the one with the highest list index number). Other items with the same key value are ignored.

## Example 1

In the following example, the list named cl is declared to be a keyed list of four-bit uints, with the key being the list item itself. That is, the key is the value of a four-bit uint. A list of 10 items is built up by generating items and adding them to the keyed list in the for loop.

In the **if** action, the ***list.*****key_exists()** and ***list.*****key_index()** keyed list pseudo-methods are used to check for the existence of an item with the value of 8, and to print the list and the key value's index if it exists.

```
<'
extend sys {
    !cl: list(key: it) of uint(bits: 4);
    run() is also {
        var ch: uint(bits: 4);
        for i from 0 to 10 {
            gen ch;
            cl.add(ch);
        };
        if cl.key_exists(8) then {
            print cl;
            print cl.key_index(8);
        };
    };
};
'>
```

### Results

```
cl =  (10 items, dec):
                      13  5  4  11  9  14  3  8  5  4       .0

  cl.key_index(8) = 2
```

### Example 2

In the following example, the struct type named s has fields a and b. A keyed list of s structs, with the n field as the key, is declared in the **sys** extension, and the list is built by the bl() method.

In the **run()** method, the ***list*****.key_exists()** keyed list pseudo-method is used to check whether the value 98 occurs in the n field in any of the structs in the keyed list. It so happens that the n value in the fourth struct in the list (index 3) is 98. Other keyed list pseudo-methods are then used to print the struct instance and the list index number of the struct that has n equal to 98.

Note that two list instances, index 12 and index 15, have the value 95 for n. If 95 was entered as the key value for the ***list*****.key_exists()** and ***list*****.key_index()** pseudo-methods, those methods would use the last instance, that is index number 15, and ignore the instance with index 12.

```
<'
struct s {
    n: byte;
    b: bit;
};

extend sys {
    !sl: list(key: n) of s;
    bl() is {
        for i from 0 to 15 {
            var t: s;
            gen t;
            sl.add(t);
        };
    };
```

```
        run() is also {
            bl();
            print sl;
            var b: bool;
            b = sl.key_exists(98);
            print b;
            if b {
                print sl.key(98);
                print sl.key_index(98);
            };
        };
    };
    '>
```

## Results

```
    sl =
    item    type        n           b

    0.      s           109         0
    1.      s           122         0
    2.      s           133         1
    3.      s           98          0
    4.      s           163         0
    5.      s           196         0
    6.      s           159         0
    7.      s           223         1
    8.      s           118         1
    9.      s           192         1
    10.     s           22          1
    11.     s           170         1
    12.     s           95          0
    13.     s           153         1
    14.     s           169         0
    15.     s           95          0
      b = TRUE
      sl.key(98) = s-@0: s
                                                       @tmp
    0   n:                              98
    1   b:                              0
      sl.key_index(98) = 3
```

## Example 3

In the following example, a keyed list is used to model sparse memory. A struct type named location has address and value fields. A keyed list named locations, with address as the key, is used to hold instances of location structs generated in the **while** loop. For each new location struct generated, the ***list*.key_exists()** pseudo-method checks to see if the list already contains an instance with that address value. If it is not already in the list, the new instance is added to the list.This ensures that the keyed list will contain exactly LLEN (50) items, all with different address values.

```
    <'
    define LLEN 50;

    struct location {
        address: uint(bits: 8);
        value: int;
    };
```

```
extend sys {
    !locations: list(key: address) of location;
    post_generate() is also {
        var loc: location = new;
        while locations.size() < LLEN do {
            gen loc;
            if locations.key_exists(loc.address) == FALSE then {
                locations.add(loc);
            };
        };
    };
};
'>
```

### See Also

— "Keyed Lists" on page 85
— "Keyed List Pseudo-Methods" on page 618

## 4.7 Creating Subtypes with When

### 4.7.1 Overview

The **when** struct member creates a conditional subtype of the current struct type, if a particular field of the struct has a given value. This is called "when" inheritance, and is one of two techniques that ***e*** provides for implementing inheritance. The other is called "like" inheritance. When inheritance is described in this section. Like inheritance is described in "Defining Structs: struct" on page 118.

When inheritance is the recommended technique for modeling in ***e***. Like inheritance is more appropriate for procedural testbench programming. When and like inheritance are compared in "Comparison of When and Like Inheritance" on page 142.

### 4.7.2 when

#### Purpose

Create a subtype

#### Category

Struct member

#### Syntax

**when *struct-subtype base-struct-type***
    **{*struct-member*; ...}**

Syntax example:

```
struct packet {
    len: uint;
    good: bool;
```

```
        when FALSE'good packet {
            pkt_msg() is {
                out("bad packet");
            };
        };
    };
```

## Parameters

| | |
|---|---|
| *struct-subtype* | A subtype declaration in the form ***type-qualifier'field-name***.<br><br>The type-qualifier is one of the legal values for the field named by field-name. If the field-name is a boolean field, and its value is TRUE for the subtype, you can omit type-qualifier. That is, if "big" is a boolean field, "big" is the same as "TRUE'big".<br><br>The field-name is the name of a field in the base struct type. Only boolean or enumerated fields can be used. If the field type is boolean, the type qualifier must be TRUE or FALSE. If the field type is enumerated, the qualifier must be a value of the enumerated type. If the type qualifier can apply to only one field in the struct, you can omit **'*field-name***.<br><br>More than one ***type-qualifier'field-name*** combination can be stated, to create a subtype based on more than one field of the base struct type. |
| *base-struct-type* | The struct type of the current struct (in which the subtype is being created). |
| *struct-member* | Definition of a struct member for the struct subtype. One or more new struct members can be defined for the subtype. |

## Description

You can use the **when** construct to create families of objects, in which multiple subtypes are derived from a common base struct type.

A subtype is a struct type in which specific fields of the base struct have particular values. For example:

— If a struct type named "packet" has a field named "kind" that can have a value of "eth" or "atm", then two subtypes of "packet" are "eth packet" and "atm packet".
— If the "packet" struct has a boolean field named "good", two subtypes are "FALSE'good packet" and "TRUE'good packet".

Subtypes can also be combinations of fields, such as "eth TRUE'good packet" and "eth FALSE'good packet".

Struct members you define in a **when** construct can be accessed only in the subtype, not in the base struct. This provides a way to define a subtype that has some struct members in common with the base type and all of its other subtypes, but has other struct members that belong only to the current subtype.

NOTE— Once you have used like inheritance to create a subtype of a base struct type, you cannot extend the base type using **when**.

### Example 1

An instance of the “packet” struct below can have a “kind” of either “transmit” or “receive”. The **when** construct creates a “transmit packet” subtype. The “length” field and the **print()** method apply only to packet instances that have “kind” values of “transmit”.

```
type packet_kind: [transmit, receive];
struct packet {
    kind: packet_kind;
    when transmit packet {
        length: int;
        print() is {
            out("packet length is: ", length);
        };
    };
};
```

### Example 2

The “op1” field in the struct definition below can have one of the enumerated “reg_n” type values (REG0, REG1, REG2, or REG3). The “kind” field can have a value of “imm” or “reg”, and the “dest” field can have a value of “mm_1” or “reg”.

The “REG0'op1” subtype specification in the first **when** construct creates a subtype of instances in which the “op1” value is “REG0”. This subtype has all the “instr” struct fields plus a “print_op1()” method.

The “reg'kind” subtype specification in the second **when** construct creates a subtype of instances in which the “kind” value is “reg”. This subtype also has all the “instr” struct fields plus a “print_kind()” method.

It is necessary to add the “'kind” expression in the second **when** construct because the “dest” field can also have a value of reg, which means that “reg” is ambiguous without the further specification of the field name.

```
type reg_n : [REG0, REG1, REG2, REG3];
struct instr {
    %op1: reg_n;
    kind: [imm, reg];
    dest: [mm_1, reg];
};
extend instr {
    when REG0'op1 instr {
        print_op1() is {
            out("instr op1 is REG0");
        };
    };
    when reg'kind instr {
        print_kind() is {
            out("instr kind is reg");
        };
    };
};
```

### See Also

- “Defining Structs: struct” on page 118
- “Comparison of When and Like Inheritance” on page 142
- “is [not] a” on page 67

## 4.8 Extending When Subtypes

There are two general rules governing the extensions of when subtypes:

— If a struct member is declared in the base struct, it cannot be re-declared in any when subtype, but it can be extended.
— With the exception of coverage groups and the events associated with them, any struct member defined in a when subtype does not apply or is unknown in other subtypes, including:
  fields
  constraints
  events
  methods
  on
  expect
  assume

### 4.8.1 Coverage and When Subtypes

All coverage events must be defined in the base struct. Defining the ready3 event within the ADD subtype, for example, results in a load time error. Coverage groups can be defined in the base struct or in the subtype.

```
struct operation {
    opcode: [ADD, SUB];
    op1: uint;
    op2: uint;
    op3: uint;

    event ready is rise('top.ready');
    event ready3 is rise('top.op3ready'); // Must define here

    cover ready is {
        item op1;
        item op2;
        cross op1, op2;
    };
};
extend operation {
    when ADD operation {
//    event ready3 is rise('top.op3ready'); // Can't define here

    cover ready3 is {
        item op1;
        item op2;
        item op3;
        cross op1, op2, op3;
        };
    };

};
```

### 4.8.2 Extending Methods in When Subtypes

A method defined or extended within a **when** construct is executed in the context of the subtype and can freely access the unique struct members of the subtype with no need for any casting.

When a method is declared in a base type, each extension of the method in a subtype must have the same parameters and return type as the original declaration. For example, because do_op() is defined with two parameters in the base type, extending do_op() in the ADD subtype to have three parameters results in a load time error.

```
struct operation {
    opcode: [ADD, ADD3];
    op1: uint;
    op2: uint;

    do_op(op1: uint, op2: uint): uint is {
        return op1 + op2;
    };
};

extend operation {
    when ADD3 operation {
      op3: uint;
//    do_op(op1:uint,op2:uint,op3:uint): uint is { // Load time error
//        return op1 + op2 +op3;
//    };
    };
};
```

However, if a method is not declared in the base type, each definition of the method in a subtype can have different parameters and return type. The following variation of the example above loads without error.

```
struct operation {
    opcode: [ADD, ADD3];
    op1: uint;
    op2: uint;
};

extend operation {
    when ADD operation {
        do_op(op1: uint, op2: uint): uint is {
            return op1 + op2;
        };
    };
    when ADD3 operation {
      op3: uint;
      do_op(op1:uint,op2:uint,op3:uint): uint is {
          return op1 + op2 +op3;
      };
    };
};
```

If more than one method of the same name is known in a when subtype, any reference to that method is ambiguous and results in a load-time error. In the following example, the legal ethernet packet subtype inherits two definitions of the method show(). The error is not reported when the ambiguity becomes possible (when the legal ethernet packet subtype is extended) but when the reference to the show() method is made.

```
type protocol: [ethernet, ieee, foreign];
struct packet {
    legal: bool;
    protocol;
```

```
        when legal packet {
            show() is {out("it is a legal packet")};
        };

        when ethernet packet {
            show() is {out("it is a ethernet packet")};
        };

        when legal ethernet packet{
            le:uint;
        };
    };

    extend sys {
        packets: list of packet;
        post_generate() is {
    //      for each legal ethernet packet (p) in packets {
    //          p.show();                                   // Load-time error
    //      };
        };
    };
```

To remove the ambiguity from such a reference, use the **as_a()** type casting operator or the when subtype qualifier syntax:

```
    p.as_a(legal packet).show();
    break on call legal packet.show()
```

NOTE— Method calls are checked when the ***e*** code is parsed. If there is no ambiguity, the method to be called is selected and all similar references are resolved in the same manner. In the example above, the extension to ethernet packet could be placed in a separate file like this:

```
    extend packet {
        when ethernet packet {
            show() is {out("it is a ethernet packet")};
        };
    };
```

If this file is loaded after the rest of the ***e*** code has been loaded, no error is issued because the method call to p.show() was resolved when the first file was loaded. Any call to p.show() always prints:

```
    it is a legal packet
```

### See Also

— “Defining Structs: struct” on page 118
— “Extending Structs: extend type” on page 121
— “Extending Subtypes” on page 123
— “Creating Subtypes with When” on page 133
— “Rules for Defining and Extending Methods” on page 459

## 4.9 Defining Attributes

### 4.9.1 Overview

You can define attributes that control how a field behaves when it is copied or compared. These attributes are used by **deep_copy()**, **deep_compare()**, and **deep_compare_physical()**.

### 4.9.2 attribute field

#### Purpose

Define the behavior of a field when copied or compared

#### Category

Struct member

#### Syntax

**attribute *field-name attribute-name*** = ***exp***

Syntax example:

```
attribute channel deep_copy = reference;
```

#### Parameters

| | |
|---|---|
| *field-name* | The name of a field in the current struct. |
| ***attribute-name*** is one of the following: | |
| deep_copy | Controls how the field is copied by the **deep_copy()** routine. |
| deep_compare | Controls how the field is compared by the **deep_compare()** routine. |
| deep_compare_physical | Controls how the field is compared by the **deep_compare_physical()** routine. |
| deep_all | Controls how the field is copied by the **deep_copy()** routine or compared by the **deep_compare()** or **deep_compare_physical()** routines. |
| ***exp*** is one of the following: | |
| normal | Perform a deep (recursive) copy or comparison. |
| reference | Perform a shallow (non-recursive) copy or comparison. |
| ignore | Do not copy or compare. |

#### Description

Defines how a field behaves when copied or compared. For a full description of the behavior specified by each expression, see the description of the "deep_copy()" on page 713, "deep_compare()" on page 716, or "deep_compare_physical()" on page 720 routine.

The **attribute** construct can appear anywhere, including inside a **when** construct or an **extend** construct.

To determine which attributes of a field are valid, all extensions to a unit or a struct are scanned in the order they were loaded. If several values are specified for the same attribute of the same field, the last attribute specification loaded is the one that is used.

### Example

This example shows the effects of field attributes on the **deep_copy()** and **deep_compare()** routines. An instance of “packet”, which contains three fields of type “port” (also a struct type), is deep copied and then deep compared. Because each of the three “port” fields has a different attribute, the way each field is copied and compared is also different.

```
<’
struct port {
    %counter: int;
};

struct packet {
    %parent: port;
    attribute parent deep_all = reference;

    %origin: port;
    attribute origin deep_copy = ignore;

    %dest: port;
    attribute dest deep_copy = normal;
    attribute dest deep_compare = ignore;
    attribute dest deep_compare_physical = ignore;

    %length: int;
};

extend sys {
    run() is also {
        var port1: port = new port;
        var port2: port = new port;
        var port3: port = new port;

        var packet1: packet = new packet with {
            .parent = port1;
            .origin = port2;
            .dest = port3;
        };

        var packet2: packet = deep_copy(packet1);

        out("");
        out("parent of packet1 is       : ", packet1.parent);
        out("parent of packet1 should be: ",
             port1, " original copy");
        out("");

        out("parent of packet2 is       : ", packet2.parent);
        out("parent of packet2 should be: ", port1,
            " shallow  copy");
        out("");

        out("origin of packet1 is       : ", packet1.origin);
        out("origin of packet1 should be: ", port2,
            " original copy");
        out("");

        out("origin of packet2 is       : ", packet2.origin);
        out("origin of packet2 should be: \
            a NULL port, attribute: copy: ignore");
        out("");

        out("dest of packet1 is       : ", packet1.dest);
        out("dest of packet1 should be: ", port3);
        out("");

        out("dest of packet2 is       : ", packet2.dest);
        out("dest of packet2 should be: a different \
             port, attribute: copy: normal (deep)");
        out("");

        packet2.dest = new port; // force different field value

        var ldiff: list of string =
            deep_compare(packet1, packet2, UNDEF);
        out(ldiff, "\n");
        out("Notice a diff in the origin field, \
            attribute is normal for deep_compare");
        out("Notice no diff for the dest field, \
            attribute is ignore for deep_compare");
        out("");
    };
};

'>
```

**Result**

Here are the results of running the packet example:

```
Running the test ...

parent of packet1 is       : port-@0
parent of packet1 should be: port-@0 original copy

parent of packet2 is       : port-@0
parent of packet2 should be: port-@0 shallow  copy

origin of packet1 is       : port-@1
origin of packet1 should be: port-@1 original copy

origin of packet2 is       : (a NULL port)
origin of packet2 should be: a NULL port,
    attribute: copy: ignore

dest of packet1 is       : port-@2
dest of packet1 should be: port-@2

dest of packet2 is       : port-@3
dest of packet2 should be: a different port,
    attribute: copy: normal (deep)

```

```
23  Differences between packet-@4 and packet-@5
24  ---------------------------------------------
25  origin:    port-@1   !=   (a NULL port)
26
27  Notice a diff in the origin field,
28      attribute is normal for deep_compare
29  Notice no diff for the dest field,
30      attribute is ignore for deep_compare
```

Line 3-Line 7: Because the parent field has the **deep_all** attribute **reference**, the parent field of the packet2 instance contains a pointer to the parent field of packet1 (port-@0).

Line 9-Line 14: Because the origin field has the **deep_copy** attribute **ignore**, the origin field of the packet2 instance contains a NULL instance of type port.

Line 16-Line 21: Because the dest field has the **deep_copy** attribute **normal**, the dest field of the packet2 instance contains a new instance of type port (port-@3).

Line 23-Line 25: These lines show the results of a **deep_compare()** of packet1 and packet2. Note that just prior to this comparison, a new instance of type port was assigned to the dest field of packet2. However, no difference is reported for the dest fields of the two packet instances, because the **deep_compare** attribute of the dest field is **ignore**. A difference is reported for the origin field because the **deep_compare** attribute is **normal** and the fields are not the equal.

### See Also

— “deep_copy()” on page 713
— “deep_compare()” on page 716
— “deep_compare_physical()” on page 720

## 4.10 Comparison of When and Like Inheritance

There are two ways to implement object-oriented inheritance in ***e***:

— Like inheritance is the classical, single inheritance familiar to users of all object-oriented languages.
— When inheritance is a concept introduced by ***e***. It is less familiar initially, but lends itself more easily to the kind of modeling that people do in ***e***.

This section discusses the pros and cons of both these types of inheritance and recommends when to use each of them.

### 4.10.1 Summary of When versus Like

In general, “when” inheritance should be used for modeling all DUT-related data structures. It is superior from a knowledge representation point of view and from an extensibility point of view. When inheritance lets you:

— Explicitly reference a field that determines the when subtype
— Create multiple, orthogonal subtypes
— Use random generation to generate lists of objects with varying subtypes
— Easily extend the struct later

Although like inheritance has more restrictions than when inheritance, it is recommended in some special cases because:

— Like inheritance is somewhat more efficient than when inheritance.
— Generation of objects that use like inheritance can also be more efficient.

## 4.10.2 A Simple Example of When Inheritance

You can create a when subtype of a generic struct using any field in the struct that is a boolean or enumerated type. This field, which determines the when subtype of a particular struct instance, is called the when determinant. In the following example, the when determinant is “legal”.

```
struct packet {
    legal: bool;

    when legal packet {
        pkt_msg() is {
            out("good packet");
        };
    };
};
```

NOTE— The following syntax is used in this document because it looks closer to the “like” version:

```
extend legal packet {...}
```

This syntax is exactly equivalent to the **when** construct:

```
extend packet {when legal packet {...}}
```

The following example shows a generic packet struct with 3 fields, protocol, size and data, and an abstract method show(). In this example, the “protocol” field is the determinant of the when version of the packet. That is, this field determines whether the packet instance has a subtype of “IEEE”, “Ethernet”, or “foreign”. In this example. the Ethernet packet subtype is extended by adding a field and extending the show() method.

```
type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined; // To be defined by children
};
extend Ethernet packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};
```

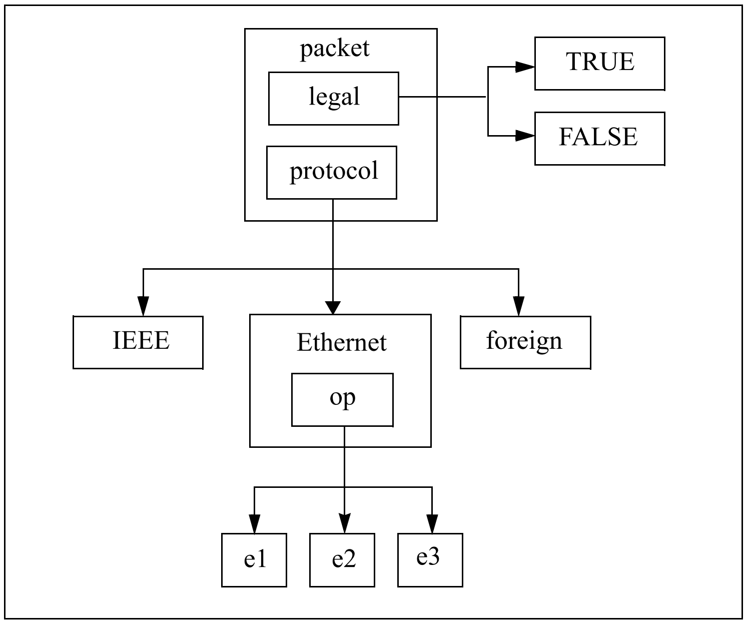
Of course, it is possible for a struct to have more than one when determinant. In the following example, the Ethernet packet subtype is extended with a field of a new enumerated type, Ethernet_op.

```
type Ethernet_op: [e1, e2, e3];
extend Ethernet packet { op: Ethernet_op; };
extend e1 Ethernet packet {
    e1_foo: int;
```

```
        show() is {out("I am an e1 Ethernet packet")};
    };
```

Because it is possible for a struct to have more than one when determinant, the inheritance tree for a struct using when inheritance consists of any number of orthogonal trees, each rooted at a separate enumerated or boolean field in the struct. Figure 4-1 on page 144 shows a when inheritance tree consisting of 3 orthogonal trees rooted in the legal, protocol, and op fields. Note that the when subtypes that have not been explicitly defined, such as IEEE packet, exist implicitly.

**Figure 4-1—When Inheritance Tree for Packet Struct Subtypes**



### 4.10.3 A Simple Example of Like Inheritance

You can create a like child of a generic struct using the **like** construct. In this example, a child Ethernet_packet is created from the generic struct packet and is extended by adding a field and extending the show() method.

```
    struct packet {
        size: int [0..1k];
        data[size]: list of byte;
        show() is undefined; // To be defined by children
    };
    struct Ethernet_packet like packet {
        e_field: int;
        show() is {out("I am an Ethernet packet")};
    };
    ...
```

In the same way, you can create an IEEE_packet from packet using like:

```
    struct IEEE_packet like packet {
        i_field: int;
        show() is {out("I am an IEEE packet")};
```

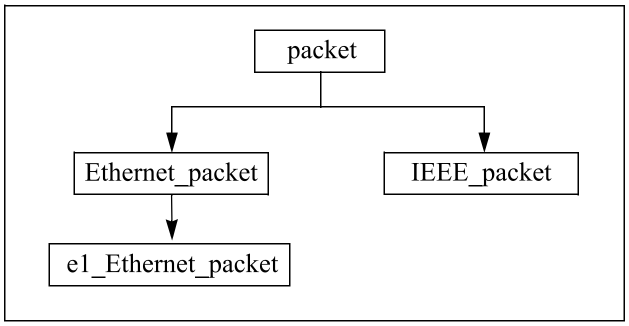
```
};
```

You can also easily create an e1_Ethernet_packet from Ethernet_packet using like inheritance.

```
struct e1_Ethernet_packet like Ethernet_packet {
    e1_foo: int;
    show() is {out("I am an e1 Ethernet packet")};
};
```

In contrast to the when inheritance tree, the like inheritance tree for the packet type is a single tree where each subtype must be defined explicitly, as shown in Figure 4-2. This difference between the like and when inheritance trees is the essential difference between like and when inheritance.

**Figure 4-2—Like Inheritance Tree for Packet Struct Subtypes**



## 4.10.4 Advantages of Using When Inheritance for Modeling

While the like version and the when version look similar, and the “like” version may seem more natural to people familiar with other object-oriented languages, the “when” version is much better for the kind of modeling typically done in ***e***. There are several reasons for this, which are explained in more detail below:

— “You can refer explicitly to the determinant fields” on page 145
— “You can create multiple orthogonal subtypes” on page 146
— “You can use random generation to create lists of objects with varying subtypes” on page 147
— “You can easily extend the struct later” on page 148
— “You can create a new type by simple extension” on page 148

### You can refer explicitly to the determinant fields

In the when version, the determinant of the when is an explicit field. In the like version, there is no explicit field that determines whether a packet instance is an Ethernet packet, an IEEE packet, or a foreign packet. The explicit determinant fields provide several advantages:

— Explicit determinant fields are more intuitive.
  Fields are more tangible than types and correspond better to the way hardware engineers perceive architectures. Having a field whose value determines what fields exist under it is familiar to engineers. (It is similar to C unions, for example.)
— You can specify the attributes of determinants that are physical fields.

If the determinant is a physical field, you probably want to specify its size in bits, the mapping of enumerated items to values, where it is in the order of fields, and so on. These things are done very naturally with when inheritance, because the determinant is just another field. For example:

```
%protocol: packet_protocol (bits: 2);
```

— With like inheritance, you can define the same field as the when determinant, but you also have to tie it to the type with code equivalent to the following:

```
var pkt: packet;

case protocol {

    Ethernet {var epkt: Ethernet packet; gen epkt; pkt = epkt;};

    IEEE {var ipkt: IEEE packet; gen ipkt; pkt = ipkt;};

};
```

There is an added inconvenience of having to generate or calculate protocol separately from the rest of the packet.

— You can constrain the when determinant.

Using when inheritance, it is very natural to write constraints like these in a test:

```
keep protocol in [Ethernet, IEEE];

keep protocol != IEEE;

keep soft protocol == select { 20: IEEE; 80: foreign; };

keep packets.is_all_iterations(.protocol, ...);
```

Constraining the value of fields in various ways is a main feature of generation. Doing the same with like inheritance is more complicated. For example, the first constraint above might be stated something like this:

```
keep me is an Ethernet_packet or me is an IEEE_packet;

// This pseudocode is not a legal constraint specification
```

However, constraints like this can become quite complex in like inheritance. Furthermore, there is no way to write the last two constraints.

### You can create multiple orthogonal subtypes

Suppose each packet (of any protocol) can be either a normal (data) packet, an ack packet or a nack packet, except that foreign packets are always normal:

```
type packet_kind: [normal, ack, nack];
extend packet {
    kind: packet_kind;
    keep protocol == foreign => kind == normal;
};
extend normal packet { n1: int; };
 ...
```

How do you do this in like inheritance? Disregard for now the issue of extending the packet struct later. Assume that you know the requirement stated above in advance, and you want to model it using like inheritance in the best possible way.

Here is one way:

```
struct normal_Ethernet_packet like Ethernet_packet {
    n1: int;
};
```

```
struct ack_Ethernet_packet like Ethernet_packet { ... };
struct nack_Ethernet_packet like Ethernet_packet { ... };
struct normal_IEEE_packet like IEEE_packet { ... };
// ...
```

This requires eight declarations.

Then, the Ethernet_op possibilities must be taken into account:

```
struct ack_e1_Ethernet_packet like e1_Ethernet_packet { ... }
// ...
```

This works, but requires ((N1 * N2 * ... * Nd) - IMP) declarations, where d is the number of orthogonal dimensions, Ni is the number of possibilities in dimension i, and IMP is the number of impossible cases.

Another issue is how to represent the impossible cases.

Multiple inheritance would solve some of these problems, but would introduce new complications.

With when inheritance all the possible combinations exist implicitly, but you do not have to enumerate them all. It is only when you want to say something about a particular one that you mention it, as in the following examples:

```
extend normal IEEE packet { ni_field: int; }; // Adds a field
extend ack e1 Ethernet packet { keep size == 0; };
// Adds a constraint
```

All in all, the when version is more natural from a knowledge representation point of view, because:

— It is immediately clear from the description what goes with what
— You only need to mention types if you have something to say about them

### You can use random generation to create lists of objects with varying subtypes

The job of the generator is to create (in this example) packet instances. By default, all possible packets should be generated. In both versions, you would create a list of packets. For example:

```
extend sys { packets: list of packet; };
```

However, the generator should only generate fully instantiated packets. In the when version, that happens automatically — there is no other way.

With like inheritance, if you generate a parent struct, only that parent struct is created; none of the like children are created. For example, the following gen action always creates a generic packet, never an Ethernet packet or an IEEE packet:

```
pkt: packet;
gen pkt;
```

Thus, in practice you should only generate fields whose type is a leaf in the like inheritance tree. For example, you normally write:

```
p: e1_Ethernet_packet;
gen p;
```

#### You can easily extend the struct later

There are some restrictions on extending structs that have like children. Details are in "Restrictions on Like Inheritance" on page 149.

#### You can create a new type by simple extension

You can extend the packet_protocol type and add new members to the packet subtype, for example:

```
extend packet_protocol: [brand_new];
extend brand_new packet {
    ...new struct members...
};
```

Automatically your old environment is able to generate brand_new packets. With like inheritance, you have to find all instances of the procedural generation code and add the new case to the case statement.

### 4.10.5 Advantages of Using Like Inheritance

Like inheritance is a shorthand notation for a subset of when inheritance. It is restricted but more efficient.

Like inheritance often has better performance than when inheritance for the following reasons:

- Method calling is faster for like inheritance.
- When generation is slower then like generation. This can be important if a large part of the total run time is attributable to generation.
- When inheritance uses more memory because all of the fields of all of the when subtypes consume space all the time.

NOTE— If this becomes a problem in a particular design, there is a workaround. Rather than having many separate fields under the **when**, put all the fields into a separate struct and put a single field for that struct under the **when**. For example, the following coding style may use a lot of memory if there are many fields declared under the Ethernet packet subtype.

```
type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    when Ethernet packet {
        e_field0: int;
        e_field1: int;
        e_field2: int;
        e_field3: int;
        // ...
    };
};
```

A more efficient coding style is shown below, where a single field is declared under the Ethernet packet subtype.

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

```
struct Ethernet_packet {
        e_field0: int;
        e_field1: int;
        e_field2: int;
        e_field3: int;
    // ...
};
struct packet {
    protocol: packet_protocol;
    when Ethernet packet {
        e_packet: Ethernet_packet;
    };
};
```

**When to Use Like Inheritance**

Like inheritance should be used for modeling only when the performance win is big enough to offset the restrictions, for example:

— Objects that use a lot of memory, such as a register file, where the number of distinct registers is very large, and for each such register a field of the register type must be generated, for example, “pc: pc_reg”, “psr: psr_reg” and so on.
— Objects that do not require randomization, such as a scoreboard or a memory.

Like inheritance should also be used for non-modeling, programming-like activities, such as implementing a generic package for a queue.

### 4.10.6 Restrictions on Like Inheritance

There are three types of restrictions on like inheritance:

— “Restrictions Due to Inherent Differences” on page 149
— “Restrictions Due to Implementation” on page 150
— “Generation Restrictions on Like Inheritance” on page 150

#### 4.10.6.1 Restrictions Due to Inherent Differences

Some of the restrictions on like inheritance derive from the inherent differences between when and like inheritance:

— You cannot explicitly reference the determinant fields.
— Creating multiple, orthogonal subtypes can be difficult with like inheritance.
— Generation of parent does not create like children.
— You cannot add when subtypes to a struct with like children. Similarly, you cannot create a like child from a struct that has when subtypes. See Example 1 on page 152 for more information.

For more information on the first 3 items in this list, see “Advantages of Using When Inheritance for Modeling” on page 145.

#### 4.10.6.2 Restrictions Due to Implementation

In addition, the following restrictions are implementation-based and may be removed in future releases:

— You cannot extend a struct with like children by:

- Extending or overriding a TCM, if the TCM has been modified by one of the like children.
- Adding fields, unless none of the like children have added fields either explicitly or implicitly. (Adding an event or a dynamic C routine might implicitly add a field, for example.)
- Adding an event, unless none of the like children have added fields either explicitly or implicitly.
- Adding or modifying an **expect** or **assume**, unless none of the like children have added fields either explicitly or implicitly.
- Adding a dynamic C routine, unless none of the like children have added fields either explicitly or implicitly.

— You cannot modify in a like child a cover group whose event is defined in the parent.

For more information see See “Examples of Like Inheritance Restrictions” on page 152.

#### 4.10.6.3 Generation Restrictions on Like Inheritance

This section describes restrictions on generation when like inheritance is used.

— Temporary fields in the parent cause problems.
Constraints that have expressions on one side of an equality or inequality create temporary fields. For example:

```
keep a > b * c;
```

gets translated internally into:

```
keep tmp == b * c; keep a > tmp;
```

If such constraints are specified in a parent, this may cause a crash during run time. (Note that there is no problem with constraints in a leaf child.)

```
<'
struct x {
   a:uint;
   b:uint;
   c:uint;

   keep a > b * c;
};

struct y like x {
   d:uint;
};
```

```
extend sys {
  x_list: list of x;
  y_list: list of y;
};
'>
```

A possible workaround is to use explicit temporary variables. That is, replace:

```
keep a > b * c;
```

with:

```
tmp: int;
keep tmp == b * c;
keep a > tmp;
```

— Unidirectional constraints in the parent do not induce generation order.
Unidirectional constraints in the parent struct do not induce the expected generation order in the child.
For example, suppose that the following constraint appears in packet:

```
keep size == f(b);
```

During generation of a like-inherited packet struct, such as Ethernet_packet, the constraint above does not cause b to be generated before size. This often leads to a contradiction.

```
<'
struct x {
   size:uint;
   b:uint;

   keep size == f(b);

   f(z:uint): uint is {
      return z * 5;
   };
};

struct y like x {
   c: uint;
};

extend sys {
   y_list: list of y;
};
'>
```

#### 4.10.6.4 Examples of Like Inheritance Restrictions

Restrictions on like inheritance are demonstrated in the following sample ***e*** code.

**Example 1**

You cannot add when subtypes to a struct with like children. Similarly, you cannot create a like child from a struct that has when subtypes.

```
<'
type protocol: [Ethernet, IEEE, foreign];
struct packet {
    p: protocol;
    data:list of byte;
};
struct tx_packet like packet {
    t:uint
};
extend packet {
// Load-time error
//    when Ethernet packet {
//        e:uint;
//    };
};
'>
```

**Example 2**

You cannot extend or override a TCM in a struct that has like children, if the TCM has been modified by one of the like children.

```
<'
struct packet {

   event clk is rise ('~/top.clk');
   zip()@clk is {wait [4]};
};

struct tx_packet like packet {
   t:uint;
   zip()@clk is also {wait [2]};
};

extend packet {
// Load-time error
//    zip()@clk is also {wait [5]};
};
'>
```

**Example 3**

You cannot add fields to a struct that has like children if those children have added fields, either implicitly or explicitly.

```
<'
struct packet {
      event clk is rise ('~/top.clk');
      zip()@clk is {wait [4]};

   };

   struct tx_packet like packet {
      t:uint;
   };

   extend packet {
   // Load-time error
   //    u:uint;
   };
   '>
```

### Example 4

You cannot add an event to a struct that has like children if those children have added fields, either implicitly or explicitly. It is OK to extend a parent to modify an event.

```
   <'
   struct packet {

      event clk is rise ('~/top.clk')@sim;
      zip()@clk is {wait [4]};
   };

   struct tx_packet like packet {
      t:uint;
   };

   extend packet {
       event clk is only fall ('~/top.clk')@sim;  // No load-time error
   //  event ready is rise ('~/top.ready');       // Load-time error
   };
   '>
```

### Example 5

You cannot add or modify an **expect** or **assume** to a struct that has like children if those children have added fields, either implicitly or explicitly.

```
   <'
   struct packet {

      event clk is rise ('~/top.clk')@sim;
      event ready is rise ('~/top.ready');
      event start_count;
      event stop_count;

      expect rule1 is @start_count => {[1..5];@ready}@clk;
   };

   struct tx_packet like packet {
      t:uint;
   };
```

```
extend packet {
// Load-time error for either of the following 2 lines
//    expect rule1 is only @start_count => {[2..6];@ready}@clk;
//     expect rule2 is @start_count => (eventually @stop_count);
};
'>
```

### Example 6

You cannot modify in a like child a cover group whose event is defined in the parent. It may load without error, but it will fail in unpredictable ways when run.

```
<'
struct packet {
   len: uint;
   addr:uint;

   event clk is rise ('~/top.clk')@sim;
   event packet_sent;

    cover packet_sent is {
      item len;
      item addr;
    };
};
struct tx_packet like packet {
   t:uint;

// cover packet_sent is {     // Error
//     item len;
//     item addr;
//     item t;
// };
};

'>
```

## 4.10.7 A When Inheritance Example

The following example contains the *e* code fragments in the section titled "A Simple Example of When Inheritance" on page 143.

```
<'
type packet_protocol: [Ethernet, IEEE, foreign];
struct packet {
    protocol: packet_protocol;
    size: int [0..1k];
    data[size]: list of byte;
    show() is undefined;
};
extend Ethernet packet {
    e_field: int;
    show() is {out("I am an Ethernet packet")};
};
extend IEEE packet {
    i_field: int;
    show() is {out("I am an IEEE packet")};
};
extend foreign packet {
    f_field: int;
    show() is {out("I am a foreign packet")};
};
type Ethernet_op: [e1, e2, e3];
extend Ethernet packet { op: Ethernet_op; };
extend e1 Ethernet packet {
    e1_foo: int;
    show() is {out("I am an e1 Ethernet packet")};
};
extend e2 Ethernet packet {
    e2_foo: int;
    show() is {out("I am an e2 Ethernet packet")};
};
extend e3 Ethernet packet {
    e3_foo: int;
    show() is {out("I am an e3 Ethernet packet")};
};
extend sys {
    packets: list of packet;
    post_generate() is also { for each in packets {.show()}; };
};
'>
```

# 5 Units

This chapter describes the constructs used to define units and explains how you can use units to implement a modular verification methodology. This chapter contains the following sections:

**See Also**

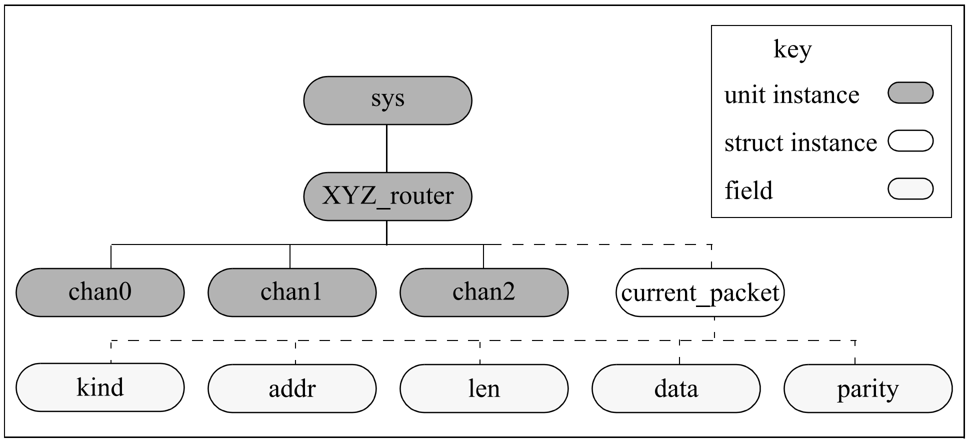
— Chapter 4, “Structs, Fields, and Subtypes”

## 5.1 Units Overview

Units are the basic structural blocks for creating verification modules (verification cores) that can easily be integrated together to test larger and larger portions of an HDL design as it develops. Units, like structs, are compound data types that contain data fields, procedural methods, and other members. Unlike structs, however, a unit instance is bound to a particular component in the DUT (an HDL path). Furthermore, each unit instance has a unique and constant place (an ***e*** path) in the runtime data structure of an ***e*** program. Both the ***e*** path and the complete HDL path associated with a unit instance are determined during pre-run generation.

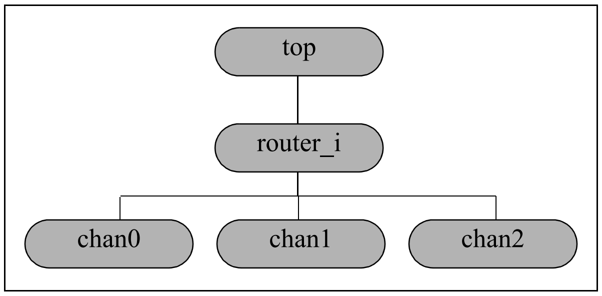
The basic runtime data structure of an ***e*** program is a tree of unit instances whose root is **sys,** the only predefined unit in ***e***. Additionally there are structs that are dynamically bound to unit instances. The runtime data structure of a typical ***e*** program is similar to that of the XYZ_router program shown in Figure 5-1.

**Figure 5-1—Runtime Data Structure of the XYZ_Router**



Each unit instance in the unit instance tree of the XYZ_router matches a module instance in the Verilog DUT, as shown in Figure 5-2. The one-to-one correspondence in this particular design between ***e*** unit instances and DUT module instances is not required for all designs. In more complex designs, there may be several levels of DUT hierarchy corresponding to a single level of hierarchy in the tree of ***e*** unit instances.

**Figure 5-2—DUT Router Hierarchy**



Binding an ***e*** unit instance to a particular component in the DUT hierarchy allows you to reference signals within that DUT component using relative HDL path names. When the units are integrated into a unit instance tree during pre-run generation,the complete path name for each referenced HDL object is determined by concatenating the complete HDL path of the parent unit to the path of the unit containing the referenced object.

This ability to use relative path names to reference HDL objects allows you to freely change the combination of verification cores as the HDL design and the verification environment evolve. Regardless of where the DUT component is instantiated in the final integration, the HDL path names in the verification environment remain valid.

### See Also

— "Units vs. Structs" on page 158
— "HDL Paths and Units" on page 159
— "Methodology Recommendations and Limitations" on page 160

### 5.1.1 Units vs. Structs

The decision of whether to model a DUT component with a unit or a struct often depends on your verification strategy. Compelling reasons for using a unit instead of a struct include:

— You intend to test the DUT component both standalone and integrated into a larger system.
Modeling the DUT component with a unit instead of a struct allows you to use relative path names when referencing HDL objects. When you integrate the component with the rest of the design, you simply change the HDL path associated with the unit instance and all the HDL references it contains are updated to reflect the component's new position in the design hierarchy.
This methodology eliminates the need for computed HDL names (for example, '(path_str).sig'), which impact runtime performance.

— Your ***e*** program has methods that access many signals at runtime.
The correctness of all signal references within units is determined and checked during pre-run generation.
If your ***e*** program does not contain user units, the absolute HDL references within structs are also checked during pre-run generation. However, if your ***e*** program does contain user units, the relative HDL references within structs are checked at run time. In this case, using units rather than structs can enhance runtime performance.

On the other hand, using a struct to model abstract collections of data, like packets, allows you more flexibility as to when you generate the data. With structs, you can generate the data either during pre-run generation, at runtime, or on the fly, possibly in response to conditions in the DUT. Unit instances, however, can only be generated during pre-run generation, because each unit instance has a unique and constant place (an ***e*** path) in the runtime data structure of an ***e*** program, just as an HDL component instance has a constant place in the DUT hierarchical tree.Thus you cannot modify the unit tree by generating unit instances on the fly.

Any allocated struct instance automatically establishes a reference to its parent unit. If this struct is a generated during pre-run generation it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct belonging to the enclosing method.

### See Also

- "HDL Paths and Units" on page 159
- "Methodology Recommendations and Limitations" on page 160

### 5.1.2 HDL Paths and Units

Relative HDL paths are essential in creating a verification module that can be used to test a DUT component either standalone or integrated into different or larger systems. Binding an ***e*** unit instance to a particular component in the DUT hierarchy allows you to reference signals within that DUT component using relative HDL path names. Regardless of where the DUT component is instantiated in the final integration, the HDL path names are still valid. To illustrate this, let's look at how the XYZ_router (shown in Figure 5-1 on page 157) is bound to the DUT router (shown in Figure 5-2 on page 158).

To associate a unit or unit instance with a DUT component, you use the **hdl_path()** method within a **keep** constraint. For example, the following code extends **sys** by creating an instance of the XYZ_router unit and binds the unit instance to the "router_i" instance in the DUT.

```
extend sys {
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() =="top.router_i";
};
```

Similarly, the following code creates three instances of XYZ_channel in XYZ_router and constrains the HDL path of the instances to be "chan0", "chan1", "chan2". These are the names of the channel instances in the DUT relative to the "router_i" instance.

```
unit XYZ_router {
    channels: list of XYZ_channel is instance;
    keep channels.size() == 3;
    keep for each in channels {.hdl_path() ==
      append("chan", index); };
};
```

The full HDL path of each unit instance is determined during generation, by appending the HDL path of the child unit instance to the full path of its parent, starting with **sys**. **sys** has the empty full path "". Thus the full path for the XYZ_router instance is "top.router_i" and that for the first channel instance is "top.router_i.chan0".

The full path for a unit instance is used to resolve any internal HDL object references that contain relative HDL paths.

By default, the **do_print()** method of any unit prints two predefined lines as well as the user-defined fields. The predefined lines display the ***e*** path and the full HDL path for that unit. The ***e*** path line contains a hyperlink to the parent unit.

### 5.1.3 Methodology Recommendations and Limitations

Each unit instance has a unique and constant place (an ***e*** path) in the runtime data structure of an ***e*** program that is determined during pre-run generation. Thus you cannot modify the unit tree created during pre-run generation by generating unit instances on the fly or making assignments of new values to existing unit instances. You can generate fields of unit type dynamically. However, when you generate a field of type unit, either on-the-fly or during pre-run generation, you must constrain the field to refer to an existing unit instance.

The following limitations are implied by the nature of unit instances and fields of unit type:

— Unit instances cannot be the object of a **new** or **gen** action or a call to **copy()**.
— Unit instances cannot be placed on the left-hand-side of the assignment operator.
— List methods which alter the original list, like **list.add()** or **list.pop()** cannot be applied to lists of unit instances.
— Units are intended to be used as structural components and not as data carriers. Therefore, using physical fields in unit instances, as well as packing or unpacking into unit instances is not recommended. Unpacking into a field of type unit when the field is NULL causes a runtime error.
— All instances of the same unit type must be bound to the same kind of HDL component.

If you intend to create a modular verification environment, the following recommendations are also important:

— Avoid setting global configuration options with **set_config()**. Instead, for numeric settings, use **set_config_max()**.
— Avoid global changes to the default packing options. Instead, define unit-specific options in the top-level unit and access them from lower-level units with **get_enclosing_unit()**.
— References to HDL objects should be placed in unit methods. If you need to access HDL objects from struct methods, you may declare additional methods in a unit. Because these access methods will probably be one line of ***e*** code, you can declare them as **inline** methods for maximum efficiency. For example, to access the following inline method declared in a struct,

```
get_reset_value() is inline { return 'reset'; };
```

you would use

```
get_enclosing_unit(CONTROLLER).get_reset_value();
```

— In structs that may be dynamically associated with more than one unit, it is recommended to use computed path names.
— Pre-run generation is performed before creating the stubs file. To minimize the time required to create a stubs file, you can move any pre-run generation that is not related to building the tree of unit instances into the procedural code, preferably as an extension of the **run()** method of the appropriate structs. For example, you probably want to avoid generating thousands of packets in order to create a stubs file.

## 5.2 Defining Units and Fields of Type Unit

The following sections describe the constructs for defining units and fields of type unit:

— "unit" on page 161
— "field: unit-type is instance" on page 165
— "field: unit-type" on page 166
— "field: list of unit instances" on page 167
— "field: list of unit-type" on page 169

### 5.2.1 unit

#### Purpose

Define a data struct associated with an HDL component or block

#### Category

Statement

#### Syntax

**unit** ***unit-type*** [**like** ***base-unit-type***] **{**
  [***unit-member*****;** ...]**}**

Syntax example:

```
unit XYZ_channel {
    event external_clock;
    event packet_start is rise('valid_out')@sim;
    event data_passed;

    verilog variable 'valid_out' using wire;

    data_checker() @external_clock is {
        while 'valid_out' == 1 {
        wait cycle;
        check that 'data_out' == 'data_in';
        };
    emit data_passed;
    };

    on packet_start {
        start data_checker();
    };
};
```

### Parameters

| | |
|---|---|
| *unit-type* | The type of the new unit. |
| *base-unit-type* | The type of the unit from which the new unit inherits its members. |
| ***unit-member***; ... | The contents of the unit. Like structs, units can have the following types of members: |

- data fields for storing data
- methods for procedures
- events for defining temporal triggers
- coverage groups for defining coverage points
- **when**, for specifying inheritance subtypes
- declarative constraints for describing relations between data fields
- **on**, for specifying actions to perform upon event occurrences
- **expect**, for specifying temporal behavior rules

Unlike structs, units can also have **verilog** members. This capability lets you create Verilog stub files for modular designs.

The definition of a unit can be empty, containing no members.

### Description

Units are the basic structural blocks for creating verification modules (verification cores) that can easily be integrated together to test larger designs. Units are a special kind of struct, with two important properties:

— Units or unit instances can be bound to a particular component in the DUT (an HDL path).
— Each unit instance has a unique and constant parent unit (an ***e*** path). Unit instances create a static tree, determined during pre-run generation, in the runtime data structure of an ***e*** program.

Because the base unit type (**any_unit**) is derived from the base struct type (**any_struct**), user-defined units have the same predefined methods. In addition, units can have **verilog** members and have several specialized predefined methods.

A unit type can be extended or used as the basis for creating unit subtypes. Extended unit types or unit subtypes inherit the base type's members and contain additional members.

See “Units vs. Structs” on page 158 for a discussion of when to use units instead of structs.

### Example

This example defines a unit type XYZ_router.

```
<'
unit XYZ_router {

    debug_mode: bool;
```

```
channels: list of XYZ_channel is instance;

keep channels.size() == 3;

keep for each in channels {
    .hdl_path() == append("chan", index);
    .router == me };

event pclk is rise('clock')@sim;

mutex_checker() @pclk is {
    while ('packet_valid') {
        var active_channel: int = UNDEF;
        for each in channels {
            if '(it).valid_out' {
                check that active_channel == UNDEF else
                    dut_error ("Violation of the mutual \
                        exclusion by channels ",
                        active_channel, " and ", index);
                active_channel = index;
                check that active_channel == 'addr' else
                    dut_error ("Violation of the \
                        correspondence between active \
                        channel and selected address");
            };
        };
        wait cycle;
    };
};

// transaction-level checking and coverage

!current_packet: XYZ_packet;

event packet_in is rise('packet_valid')@pclk;
on packet_in {
    current_packet = new XYZ_packet;
    current_packet.addr = 'addr';
    current_packet.len = 'len';
    out(current_packet.get_unit());
    start sample_data();
    start mutex_checker();
};

sample_data() @pclk is {
    if (debug_mode) {
        out("Start of sampling");
    };
    for j from 1 to current_packet.len {
        wait cycle;
        current_packet.data.add('data');
    };
    if (debug_mode) {
        out("End of sampling: packet data ",
            current_packet.data);
    };
    // Don't read parity yet
};
```

```
        event packet_out is fall('packet_valid')@pclk;

        expect (@packet_in => { [current_packet.len];
            cycle @packet_out}) @pclk
            else dut_error ("Violation of expected packet duration");

        event log is @packet_out;

        on packet_out {
            current_packet.parity = 'parity';

            // Check the last byte of the data
            current_packet.kind =
              ('data' == current_packet.parity_calc()) ? good : bad;

            if (debug_mode) {
                print current_packet;
            };

            if (current_packet.kind == good) then {
                check that 'err' == 0 else dut_error ("Err != 0 \
                    for good pkt");
            }
            else {
                check that 'err' == 1 else dut_error ("Err != 1 \
                    for bad pkt");
            };
        };

        event channel_data_passed;
        expect (@packet_out => [1] @channel_data_passed) @pclk
            else dut_error("Channel data pass and packet out \
                aren't synchronous");

        cover log using text = "End of package transaction" is {
            item addr : uint (bits : 2) = current_packet.addr
                using illegal = (addr == 3);
            item len : uint (bits : 6) = current_packet.len
                using ranges={
                range([0..3],"short");
                range([4..15],"medium");
                range([16..63],"long");
            };
            item kind : XYZ_kind_type = current_packet.kind ;
            item err : bool = 'err' ;
        };
    };
    '>
```

## See Also

— "Units Overview" on page 157
— Chapter 4, "Structs, Fields, and Subtypes"

### 5.2.2 field: unit-type is instance

#### Purpose

Define a unit instance field

#### Category

Unit member

#### Syntax

***field-name***[**:** ***unit-type***] **is instance**

Syntax example:

```
cpu: XYZ_cpu is instance;
```

#### Parameters

| | |
|---|---|
| *field-name* | The name of the unit instance being defined. |
| *unit-type* | The name of a unit type.<br><br>If the field name is the same as an existing type, you can omit the "**:** ***unit-type***" part of the field definition. Otherwise, the type specification is required. |

#### Description

Defines a field of a unit to be an instance of a unit type. Units can be instantiated within other units, thus creating a unit tree. The root of the unit tree is **sys**, the only predefined unit in ***e***.

A unit instance has to be bound to a particular component in the DUT (an HDL path). Each unit instance also has a unique and constant place (an ***e*** path) in the runtime data structure of an ***e*** program that is determined during pre-run generation.

#### Notes

— Instantiating a unit in a struct causes a compile-time error; units can only be instantiated within another unit.
— The do-not-generate operator (!) is not allowed with fields of type unit instance. Unit instances can be created only during pre-run generation.
— It is not recommended to use the physical field operator (%) with fields of type unit instance.

#### Example

This example creates an instance of the XYZ_router unit type in **sys**.

```
<'
extend sys {
    mntr: monitor;
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() =="top.router_i";
    keep unit_core.debug_mode == TRUE;
```

```
        setup() is also {
            set_check("...", WARNING);
            set_config(cover, mode, on);
        };
    };
    '>
```

### See Also

- — “Units Overview” on page 157
- — “field: unit-type” on page 166
- — “field: list of unit-type” on page 169
- — Chapter 4, “Structs, Fields, and Subtypes”

## 5.2.3 field: unit-type

### Purpose

Define a field of type unit

### Category

Struct or unit member

### Syntax

[**!**] ***field-name***[**:** ***unit-type***]

Syntax example:

```
extend XYZ_router{
    !current_chan: XYZ_channel;
};
```

### Parameters

| | |
|---|---|
| ! | Denotes an ungenerated field. If you generate this field on the fly, you must constrain it to an existing unit instance or a runtime error is issued. |
| *field-name* | The name of the field being defined. |
| *unit-type* | The name of a unit type.<br><br>If the field name is the same as an existing type, you can omit the “**:** ***unit-type***” part of the field definition. Otherwise, the type specification is required. |

### Description

Defines a field of unit type. A field of unit type is always either NULL or a reference to a unit instance of a specified unit type.

### Notes

- — It is not recommended to use the physical field operator (%) with fields of type unit.
- — If a field of type unit is generated it must be constrained to an existing unit instance.

### Example

In the example below, the XYZ_router is extended with an ungenerated field of type XYZ_channel, a unit type. It remains NULL until the “mutex_checker()” method is called. In this method the “current_chan” field is used as a pointer to each of the unit instances of type XYZ_channel in the channels list.

```
extend XYZ_router {
!current_chan: XYZ_channel;
  mutex_checker() @pclk is {
    while ('packet_valid') {
      var active_channel: int = UNDEF;
      for each in channels {
        current_chan = it;
        if '(current_chan).valid_out' {
          check that active_channel == UNDEF else
           dut_error ("Violation of the mutual exclusion by \
             channels ", active_channel, " and ", index);
          active_channel = index;
          check that active_channel == 'addr' else
            dut_error ("Violation of the correspondence \
             between active channel and selected address");
        };
      };
      wait cycle;
    };
  };
};
```

### See Also

— “field: unit-type is instance” on page 165
— “field: list of unit-type” on page 169
— Chapter 4, “Structs, Fields, and Subtypes”

## 5.2.4 field: list of unit instances

### Purpose

Define a list field of unit instances

### Category

Struct or unit member

### Syntax

***name***:[[***length-exp***]]: **list of *unit-type* is instance**

Syntax example:

```
channels: list of XYZ_channel is instance;
```

### Parameters

| | |
|---|---|
| *name* | The name of the list being defined. |
| *length-exp* | An expression that gives the initial size for the list. |
| *unit-type* | A unit type. |
| is instance | Creates a list of unit instances. |

### Description

Defines a list field of unit instances. A list of unit instances can only be created during pre-run generation and cannot be modified after it is generated.

### Notes

— List operations, such as **list.add()** or **list.pop()**, that alter the list created during pre-run generation are not allowed for lists of unit instances.
— It is not recommended to use the physical field operator (%) with lists of unit instances.

### Example

This example creates a list of unit instances of type XYZ_channel in XYZ_router.

```
<'
unit XYZ_channel {
    event external_clock;
    event packet_start is rise('valid_out')@sim;
    event data_passed;

    verilog variable 'valid_out' using wire;

    data_checker() @external_clock is {
        while 'valid_out' == 1 {
        wait cycle;
        check that 'data_out' == 'data_in';
        };
    emit data_passed;
    };

    on packet_start {
        start data_checker();
    };
};

unit XYZ_router {
    channels: list of XYZ_channel is instance;
    keep channels.size() == 3;
};
'>
```

### See Also

— "field: unit-type" on page 166
— "field: unit-type is instance" on page 165
— Chapter 4, "Structs, Fields, and Subtypes"

## 5.2.5 field: list of unit-type

### Purpose

Define a list field of type unit

### Category

Struct or unit member

### Syntax

**[!]*name*[[*length-exp*]]: list of *unit-type***

Syntax example:

```
var currently_valid_channels: list of XYZ_channel;
```

### Parameters

| | |
|---|---|
| ! | Do not generate this list. |
| *name* | The name of the list being defined. |
| *length-exp* | An expression that gives the initial size for the list. |
| *unit-type* | A unit type. |

### Description

Defines a list field of type unit.

NOTE— It is not recommended to use the physical field operator (%) with lists of unit type.

### Example

This example creates a list of unit type XYZ_channel, which is used to create a list of currently valid channels.

```
<'
unit XYZ_channel {
    router: XYZ_router;
};
unit XYZ_router {
    channels: list of XYZ_channel is instance;
    keep channels.size() == 3;

    validity_checker() is {
        var currently_valid_channels: list of XYZ_channel;

        for each in channels {
            if '(it).valid_in' {
                currently_valid_channels.add(it);
            };
        };
        print currently_valid_channels;
    };
    };
    '>
```

### See Also

— “field: unit-type” on page 166
— “field: unit-type is instance” on page 165
— Chapter 4, “Structs, Fields, and Subtypes”

## 5.3 Predefined Methods for Any Unit

There is a predefined generic type **any_unit**, which is derived from **any_struct**. **any_unit** is the base type implicitly used in user-defined unit types, so all predefined methods for **any_unit** are available for any user-defined unit. The predefined methods for **any_struct** are also available for any user-defined unit.

The predefined methods for any unit include:

— “hdl_path()” on page 170
— “full_hdl_path()” on page 172
— “e_path()” on page 173
— “agent()” on page 174
— “get_parent_unit()” on page 176

### See Also

— “Unit-Related Predefined Methods for Any Struct” on page 177
— “Unit-Related Predefined Routines” on page 184

### 5.3.1 hdl_path()

#### Purpose

Return a relative HDL path for a unit instance

#### Category

Predefined pseudo-method for any unit

#### Syntax

[***unit-exp***.]**hdl_path()**: string

Syntax example:

```
    extend dut_error_struct {
        write() is first {
            var channel: XYZ_channel =
              source_struct().try_enclosing_unit(XYZ_channel);
            if (channel != NULL) {
                out("Error in XYZ channel: instance ",
                  channel.hdl_path());
            };
        };
    };
```

### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

### Description

Returns the HDL path of a unit instance. The most important role of this method is to bind a unit instance to a particular component in the DUT hierarchy. Binding an ***e*** unit or unit instance to a DUT component allows you to reference signals within that component using relative HDL path names. Regardless of where the DUT component is instantiated in the final integration, the HDL path names are still valid. The binding of unit instances to HDL components is a part of the pre-run generation process and must be done in **keep** constraints.

Although absolute HDL paths are allowed, relative HDL paths are recommended if you intend to follow a modular verification strategy.

This method always returns an HDL path exactly as it was specified in constraints. If, for example, you use a macro in a constraint string, then **hdl_path()** returns the original and not substituted string.

### Notes

— All instances of the same unit must be bound to the same kind of HDL components.
— You cannot constrain the HDL path for **sys**.

### Example 1

This example shows how you can use relative paths in lower-level instances in the unit instance tree. To create the full HDL path of each unit instance, its HDL path is prefixed with the HDL path of its parent instance. In this example, because the HDL path of **sys** is “”, the full HDL path of “unit_core” is “top.router_i”. The full HDL path of “extra_channel” is “top.router_i.chan3”.

```
extend sys {
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() == "top.router_i";
};

extend XYZ_router {
    extra_channel: XYZ_channel is instance;
    keep extra_channel.hdl_path() == "chan3";
};
```

### Example 2

This example shows how **hdl_path()** returns the HDL path exactly as specified in the constraint. Thus the first **print** action prints “`TOP.router_i”. The second **print** action, in contrast, accesses “top.router_i.clk”.

```
verilog import macros.v;
extend sys {
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() == "`TOP.router_i";
    run() is also {
        print unit_core.hdl_path();
        print '(unit_core).clk';
    };
    };
```

### Result

```
    unit_core.hdl_path() = "‘TOP.router_i"
      ’top.router_i.clk’ = 0
```

### See Also

— “HDL Paths and Units” on page 159
— “full_hdl_path()” on page 172
— “e_path()” on page 173

## 5.3.2 full_hdl_path()

### Purpose

Return an absolute HDL path for a unit instance

### Category

Predefined method for any unit

### Syntax

[***unit-exp***.]**full_hdl_path()**: string

Syntax example:

```
    out ("Mutex violation in ", get_unit().full_hdl_path());};
```

### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

### Description

Returns the absolute HDL path for the specified unit instance. This method is used mainly in informational messages. Like the **hdl_path()** method, this method returns the path as originally specified in the **keep** constraint, without making any macro substitutions.

### Example

This example uses **full_hdl_path()** to display information about where a mutex violation has occurred.

```
    extend XYZ_router {
    !current_chan: XYZ_channel;
      mutex_checker() @pclk is {
        while ('packet_valid') {
          var active_channel: int = UNDEF;
          for each XYZ_channel(current_chan) in channels {
            if '(current_chan).valid_out' {
              if active_channel != UNDEF then {
                  out ("Mutex violation in ",
                    get_unit().full_hdl_path());};
              active_channel = index;
            };
          };
          wait cycle;
        };
      };
    };
```

### Result

```
Mutual exclusion violation in top.router_i
```

### See Also

— "hdl_path()" on page 170
— "e_path()" on page 173

## 5.3.3 e_path()

### Purpose

Returns the location of a unit instance in the unit tree

### Category

Predefined method for any unit

### Syntax

[***unit-exp***.]**e_path()**: string

Syntax example:

```
out("Started checking ", get_unit().e_path());
```

### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

### Description

Returns the location of a unit instance in the unit tree. This method is used mainly in informational messages.

### Example

```
<'
unit ex_u {
    run() is also {
        inst = get_unit().e_path();
        var inst: string;
        inst = get_unit().e_path();
        out("ex instance: ", inst);
    };
};

unit top_u {
    exlist[10]: list of ex_u is instance;
};

extend sys {
    top: top_u is instance;
};
'>
```

### Result

```
    ex instance: sys.top.exlist[0]
    ex instance: sys.top.exlist[1]
    ex instance: sys.top.exlist[2]
    ex instance: sys.top.exlist[3]
    ex instance: sys.top.exlist[4]
    ex instance: sys.top.exlist[5]
    ex instance: sys.top.exlist[6]
    ex instance: sys.top.exlist[7]
    ex instance: sys.top.exlist[8]
    ex instance: sys.top.exlist[9]
```

### See Also

— "full_hdl_path()" on page 172
— "hdl_path()" on page 170

## 5.3.4 agent()

### Purpose

Maps the DUT's HDL partitions into ***e*** code

### Category

Predefined pseudo-method for any unit

### Syntax

**keep** [*unit-exp*.]**agent() ==** ***string*****;**

Syntax example:

```
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
```

## Parameters

| | |
|---|---|
| unit-exp | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |
| string | One of the following predefined agent names: **verilog**, **vhdl**, **mti_vlog**, **mti_vhdl**, **ncvlog** and **ncvhdl**. Specifying the agent name as **verilog** or **vhdl** is preferred because it makes the ***e*** code portable between simulators. In contrast, if a unit is bound to a specific agent, for example to **mti_vhdl**, an error is issued if it is ported to NC Simulator. The predefined names are case-insensitive; in other words, **verilog** is the same as **Verilog.** |

## Description

Specifying an agent identifies the simulator that is used to simulate the corresponding DUT component. Once a unit instance has an explicitly specified agent name then all other unit instances instantiated within it are implicitly bound to the same agent name, unless another agent is explicitly specified.

An agent name may be omitted in a single-HDL environment but it must be defined implicitly or explicitly in a mixed HDL environment for each unit instance that is associated with a non-empty **hdl_path()**. If an agent name is not defined for a unit instance with a non-empty **hdl_path()** in a mixed HDL environment, an error message is issued.

Given the **hdl_path()** and **agent()** constraints, a correspondence map is established between the unit instance HDL path and its agent name. Any HDL path below the path in the map is associated with the same agent unless otherwise specified. This map is further used internally to pick the right adapter for each accessed HDL object.

It is possible to access Verilog signals from a VHDL unit instance code and vice-versa. Every signal is mapped to its HDL domain according to its full path, regardless of the specified agent of the unit that the signal is accessed from.

When the **agent()** method is called procedurally, it returns the agent of the unit. The spelling of the agent string is exactly as specified in the corresponding constraint.

## Notes

— Agents are bound to unit instances during the generation phase. Consequently, there is no way to map between HDL objects and agents before generation. As a result of this limitation, HDL objects in a mixed Verilog/VHDL environment cannot be accessed before generation from **sys.setup()** or from the command line.

— An unsupported agent name causes an error message during the test phase.

## Example 1

In the following example, the driver instance inherits an agent name implicitly from the enclosing router unit instance.

```
extend sys {
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
    keep router.hdl_path() == "top.rout";
};

extend XYZ_router {
    driver: XYZ_router_driver is instance;

};
```

### Example 2

In this example, the signal ‘top.rout.packet_valid’ is sampled using the Verilog PLI because the path “top.rout” is specified as a Verilog path. In contrast, the signal ‘top.rout.chan.mux.data_out’ is sampled using a VHDL foreign interface because the closest mapped path is “top.rout.chan” and it is mapped as a VHDL path.

```
extend sys {
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
    keep router.hdl_path() == "top.rout";
};
unit XYZ_router {
    channel: XYZ_channel is instance;
    keep channel.agent() == "VHDL";
    keep channel.hdl_path() == "chan";

    run() is also {
        print 'packet_valid';
    };
};
unit XYZ_channel {
    run() is also {
        print 'mux.data_out';
    };
};
```

## 5.3.5 get_parent_unit()

### Purpose

Return a reference to the unit containing the current unit instance

### Category

Predefined method for any unit

#### Syntax

[***unit-exp***.]**get_parent_unit()**: unit type

Syntax example:

```
out(sys.unit_core.channels[0].get_parent_unit());
```

#### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

#### Description

Returns a reference to the unit containing the current unit instance.

#### Example

```
out(sys.unit_core.channels[0].get_parent_unit())
  XYZ_router-@2
```

#### See Also

— “get_unit()” on page 177
— “get_enclosing_unit()” on page 180
— “try_enclosing_unit()” on page 182

## 5.4 Unit-Related Predefined Methods for Any Struct

The predefined methods for any struct include:

— “get_unit()” on page 177
— “get_enclosing_unit()” on page 180
— “try_enclosing_unit()” on page 182
— “set_unit()” on page 183

#### See Also

— “Predefined Methods for Any Unit” on page 170
— “Unit-Related Predefined Routines” on page 184

### 5.4.1 get_unit()

#### Purpose

Return a reference to a unit

#### Category

Predefined method of any struct

### Syntax

[***exp***.]**get_unit()**: unit type

Syntax example:

```
out ("Mutex violation in ", get_unit().full_hdl_path());};
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed. |

### Description

When applied to an allocated struct instance, this method returns a reference to the parent unit—the unit to which the struct is bound. When applied to a unit, it returns the unit itself.

Any allocated struct instance automatically establishes a reference to its parent unit. If this struct is generated during pre-run generation it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct the enclosing method belongs to. See Example 3 on page 179 for an illustration of this point.

This method is useful when you want to determine the parent unit instance of a struct or a unit. You can also use this method to access predefined unit members, such as **hdl_path()** or **full_hdl_path()**. To access user-defined unit members, use **get_enclosing_unit()**. See Example 1 on page 178 for an illustration of this point.

### Example 1

This example shows that **get_unit()** can access predefined unit members, while **get_enclosing_unit()** must be used to access user-defined unit members.

```
struct instr {
    %opcode     : cpu_opcode ;
    %op1        : reg ;
    kind        : [imm, reg];

    post_generate() is also {
--      get_unit().print_msg() ; -- COMPILE-TIME ERROR
        get_enclosing_unit(XYZ_cpu).print_msg();
        out("Destination for this instruction is ",
            get_unit().hdl_path()) ;
    };
};

unit XYZ_cpu {
    instrs[3] :  list of instr;
    print_msg() is {out("Generating instruction for \
        XYZ_cpu...");};
};

extend sys {
    cpu1: XYZ_cpu is instance;
    keep cpu1.hdl_path()=="'TOP/CPU1";
};
```

```
'>
```

### Result

```
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
```

### Example 2

The first call to **get_unit()** below shows that the parent unit of the struct instance “p” is **sys**. The second call shows that the parent unit has been changed to “XYZ_router”.

```
var p: XYZ_packet = new
out(p.get_unit())
  sys-@0
p.set_unit(sys.unit_core)
out(p.get_unit())
  XYZ_router-@1
```

### Example 3

In this example, the trace_inject() method displays the full HDL path of the “XYZ_dlx” unit (not the “XYZ_tb” unit) because “instr_list” is generated by the run method of “XYZ_dlx”.

```
extend sys {
    tb: XYZ_tb is instance;
    keep tb.hdl_path()=="'TOP/tb";
};
unit XYZ_tb {
    dlx: XYZ_dlx is instance;
        keep dlx.hdl_path()=="dlx_cpu";
    !instr_list: list of instruction;
    debug_mode: bool;
};
unit XYZ_dlx {
    run() is also {
        gen sys.tb.instr_list keeping { .size() < 30;};
    };
};
extend instruction {
    trace_inject() is {
        if get_enclosing_unit(XYZ_tb).debug_mode == TRUE {
            out("Injecting next instruction to ",
                get_unit().full_hdl_path());
        };
    };
};
```

### Result

```
sys.tb.instr_list[0].trace_inject()
Injecting next instruction to 'TOP/tb.dlx_cpu
```

### See Also

— “get_parent_unit()” on page 176
— “get_enclosing_unit()” on page 180
— “try_enclosing_unit()” on page 182

## 5.4.2 get_enclosing_unit()

### Purpose

Return a reference to nearest unit of specified type

### Category

Predefined pseudo-method of any struct

### Syntax

[***exp***.]**get_enclosing_unit(*unit-type***: exp**)**: unit instance

Syntax example:

```
unpack(p.get_enclosing_unit(XYZ_router).pack_config,
  'data', current_packet);
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed.<br><br>NOTE— If **get_enclosing_unit()** is called from within a unit of the same type as ***exp***, it returns the present unit instance and not the parent unit instance. |
| *unit-type* | The name of a unit type or unit subtype. |

### Description

Returns a reference to the nearest higher-level unit instance of the specified type, allowing you to access fields of the parent unit in a typed manner.

You can use the parent unit to store shared data or options such as packing options that are valid for all its associated subunits or structs. Then you can access this shared data or options with the **get_enclosing_unit()** method.

### Notes

— The unit type is recognized according to the same rules used for the **is a** operator. This means, for example, that if you specify a base unit type and there is an instance of a unit subtype, the unit subtype is found.
— If a unit instance of the specified type is not found, a runtime error is issued.

### Example 1

In the following example, **get_enclosing_unit()** is used to print fields of the nearest enclosing unit instances of type “XYZ_cpu” and “tbench”. Unlike **get_unit()**, which returns a reference only to its immediate parent unit, **get_enclosing_unit()** searches up the unit instance tree for a unit instance of the type you specify. A runtime error is issued unless an instance of type “XYZ_cpu” and an instance of type “tbench” are found in the enclosing unit hierarchy.

```
struct instr {
    %opcode     : cpu_opcode ;
    %op1        : reg ;
    kind        : [imm, reg];

    post_generate() is also {
       out("Debug mode for CPU is ",
           get_enclosing_unit(XYZ_cpu).debug_mode);
       out("Memory model is ",
           get_enclosing_unit(tbench).mem_model);
    };
};
unit XYZ_cpu {
    instr:  instr;
    debug_mode: bool;
};
unit tbench {
    cpu: XYZ_cpu is instance;
    mem_model: [small, big];
};

extend sys {
    tb: tbench is instance;
};
```

### Result

```
Debug mode for CPU is FALSE
Memory model is small
```

### Example 2

```
extend XYZ_router {
    pack_config:pack_options;

    keep pack_config == packing.low_big_endian;
};
```

### Result

```
var p: XYZ_packet = new
print p.data
  p.data = (empty)
out(p.get_unit())
  sys-@0
p.set_unit(sys.unit_core)
out(p.get_unit())
  XYZ_router-@1
unpack(p.get_enclosing_unit(XYZ_router).pack_config, data, p)
   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|  +0
                                                                   |
   0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 0|
                                                                  |
   data                                                            |

                                  |5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0|  +32
                                  +                               |
                                  |0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 0
                                  +                               |
                                  |                               |
```

### See Also

— “get_unit()” on page 177
— “set_unit()” on page 183
— “try_enclosing_unit()” on page 182
— “get_parent_unit()” on page 176

## 5.4.3 try_enclosing_unit()

### Purpose

Return a reference to nearest unit instance of specified type or NULL

### Category

Predefined method of any struct

### Syntax

[***exp***.]**try_enclosing_unit(*unit-type***: exp**)**: unit instance

Syntax example:

```
var MIPS := source_struct().try_enclosing_unit(MIPS);
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed.<br><br>NOTE— If **try_enclosing_unit()** is called from within a unit of the same type as ***exp***, it returns the present unit instance and not the parent unit instance. |
| *unit-type* | The name of a unit type or unit subtype. |

### Description

Like **get_enclosing_unit()**, this method returns a reference to the nearest higher-level unit instance of the specified type, allowing you to access fields of the parent unit in a typed manner.

Unlike **get_enclosing_unit()**, this method does not issue a runtime error if no unit instance of the specified type is found. Instead, it returns NULL. This feature makes the method suitable for use in extensions to global methods such as **dut_error_struct.write()**, which may be used with more than one unit type.

### Example

```
<'
extend dut_error_struct {
    write() is also {
        var MIPS := source_struct().try_enclosing_unit(MIPS);
        if MIPS != NULL then {
            out("Status of ", MIPS.e_path(),
              " at time of error:");
            MIPS.show_status();
        };
    };
};
'>
```

### See Also

— “get_unit()” on page 177
— “get_enclosing_unit()” on page 180
— “get_parent_unit()” on page 176

## 5.4.4 set_unit()

### Purpose

Change the parent unit of a struct

### Category

Predefined method of any struct

### Syntax

[***struct-exp***.]**set_unit(*parent***: exp**)**

Syntax example:

```
p.set_unit(sys.unit_core)
```

### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. If no expression is specified, the current struct is assumed. |
| *parent* | An expression that returns a unit instance. |

### Description

Changes the parent unit of a struct to the specified unit instance.

NOTE— This method does not exist for units because the unit tree cannot be modified.

### Example

```
var p: XYZ_packet = new
out(p.get_unit())
sys-@0
```

```
    p.set_unit(sys.unit_core)
    out(p.get_unit())
    XYZ_router-@1
```

## 5.5 Unit-Related Predefined Routines

The predefined routines that are useful for units include:

- "set_config_max()" on page 184
- "get_all_units()" on page 186

### 5.5.1 set_config_max()

#### Purpose

Increase values of numeric global configuration parameters

#### Category

Predefined routine

#### Syntax

**set_config_max(*category***: keyword**, *option***: keyword**, *value***: exp [**, *option***: keyword, ***value***: exp...]**)**

Syntax example:

**set_config_max(**memory, gc_threshold, 100m**);**

## Parameters

|  |  |
|---|---|
| *category* | Is one of the following: **cover**, **gen**, **memory**, and **run**. |
| *option* | The valid **cover** option is:<br>• absolute_max_buckets.<br>The valid **generate** options are:<br>• absolute_max_list_size<br>• max_depth<br>• max_structs<br>The valid **memory** options are:<br>• gc_threshold<br>• gc_increment<br>• max_size<br>• absolute_max_size<br>The valid **run** option is:<br>• tick_max<br>The options are described in "set_config()" on page 766. |
| *value* | The valid values are different for each option and are described in "set_config()" on page 766. |

## Description

Sets the numeric options of a particular category to the specified maximum values.

If you are creating a modular verification environment, it is recommended to use **set_config_max()** instead of **set_config()** in order to avoid possible conflicts that may happen in an integrated environment. For example, if two units are instantiated and both of them attempt to enlarge the configuration value of **absolute_max_size** then the recommended way to it is via set_config_max, so that no unit decrements the value set by another one.

## Example

```
<'
extend sys {
    setup() is also {
        set_config_max(memory, gc_threshold, 100m);
    };
};
'>
```

## See Also

— "Predefined Methods for Any Unit" on page 170

— "Unit-Related Predefined Methods for Any Struct" on page 177

### 5.5.2 get_all_units()

#### Purpose

Return a list of instances of a specified unit type

#### Category

Routine

#### Syntax

**get_all_units(*unit-type*:** exp**)**: list of unit instances

Syntax example:

print get_all_units(XYZ_channel);

#### Parameters

| | |
|---|---|
| *unit-type* | The name of a unit type. The type must be defined or an error occurs. |

#### Description

This routine receives a unit type as a parameter and returns a list of instances of this unit type as well as any unit instances contained within each instance.

#### Example

This example uses **get_all_units()** to print a list of the instances of XYZ_router. Note that the display also shows that this instance of XYZ_router contains "channels", which is a list of three unit instances.

```
<'
unit XYZ_router {
    channels: list of XYZ_channel is instance;

    keep channels.size() == 3;
    keep for each in channels {
        .hdl_path() == append("chan", index);
        .router == me
    };

};

unit XYZ_channel {
    router:XYZ_router;
};

extend sys {
    router:XYZ_router is instance;

    run() is also {
        print get_all_units(XYZ_router);
        };
    };
    '>
```

### Result

```
    get_all_units(XYZ_router) =
    item   type        channels
    ---------------------------------------------------------------
    0.     XYZ_router  (3 items)
```

### See Also

— “Predefined Methods for Any Unit” on page 170
— “Unit-Related Predefined Methods for Any Struct” on page 177

# 6 *e* Ports

This document describes ports, an ***e*** unit member that enhances the portability and inter-operability of verification environments by making separation between an ***e*** unit and its interface possible.

This document discusses the following topics:

## 6.1 Introduction to *e* Ports

A port is an ***e*** unit member that makes a connection between an ***e*** unit and its interface to another internal or external entity. There are two ways to use ports:

— Internal ports (***e***2***e*** ports) connect an ***e*** unit to another ***e*** unit.
— External ports connect an ***e*** unit to a simulated object.

External ports are a generic way to access simulated objects of various kinds. An external port is bound to a simulated object, for example an HDL signal in the DUT. Then all access to that signal is made via the port. The port can be used to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used.

NOTE— In this document, "simulator" means any hardware or software agent that runs in parallel with an ***e*** program, and models the behavior of any part of the design under test (DUT) or its environment.

### 6.1.1 Advantages of Using Ports

Although previous HDL access mechanisms are still supported, ports have the following advantages over the old access mechanisms:

— Ports support modularity and encapsulation by explicitly declaring interfaces to ***e*** units.
— They are typed.
— They improve performance of accessing DUT objects with configurable names.
— They can pass not only single values but also other kinds of information, such as events and queues.
— They can be accompanied in ***e*** with generic or simulator-specific attributes that let you specify information needed for enhanced access to DUT objects.
— They are suitable for use with a publicly available procedural External Simulator Interface (ESI).
— Some new simulator interfaces, such as SystemC, require the use of ports.

### 6.1.2 Creating Port Instances

Port type is defined by three aspects:

— The kind of port, either simple port, buffer port, or event port:

  - Simple ports access data directly.
  - Buffer ports implement an abstraction of queues with blocking get and put.
  - Event ports transfer events between ***e*** units or between an ***e*** unit and a simulator.

— Direction, either input or output (or inout for simple and event ports)
— Data element, the ***e*** type that can be passed through this port

You can instantiate ports only within units. Like units, port instances are generated during prerun generation and cannot be created, modified or removed during a run. When you instantiate a port, you must specify:

— A unique instance name
— The port type (direction, port kind, and a kind-specific type specifier)

The generic syntax for ports is as follows:

***port-instance-name*****:** [***direction***] ***port-kind*** **of** [***type-specifier***] **is instance;**

NOTE— Event ports do not allow a type specifier.

For example, the following unit member creates a port instance:

```
data_in: in buffer_port of packet is instance;
```

where:

— The port instance name is data_in.
— The port kind is a buffer port.
— The port direction is input.
— The data element the port accepts is “packet”.

As another example, the following line creates a list of simple ports which each pass data of type bit:

```
ports: list of simple_port of bit is instance;
```

### 6.1.3 Using Ports

A port’s behavior is influenced by port attributes, such as **hdl_path()** or **bind()**, which are applied to port instances using pre-run generation **keep** constraints. For example, the following lines of code create a port named “data” and connect (bind) it to an external simulator-related object whose HDL pathname is “data”.

```
data: inout simple_port of list of bit is instance;
  keep bind(data, external);
  keep data.hdl_path() == "data";
```

Each port kind has predefined methods that you use to access the port values. For example, buffer ports have a predefined method **put()**, which writes a value onto an output port:

```
data_out: out buffer_port of cell is instance;
drive_all() @sys.any is {
    var stimuli: cell;
    var counter: int=0;
    while counter < cells {
        wait [1]*cycle;
        gen stimuli;
        data_out.put(stimuli);
        counter+=1;
    };
};
```

### 6.1.4 Ports Example

The *e* code in this section shows examples of instantiating and using buffer ports. An output buffer port and an input buffer port are created, the ports are connected together, and data elements of type “cell” are generated and transmitted from the output buffer port to the input buffer port.

```
struct cell {
    header[2] : list of byte;
    data[50]  : list of byte;
};

unit trans {
    data_out: out buffer_port of cell is instance;
    !cells : int;
    keep cells == 100;
     drive_all() @sys.any is {
         var stimuli: cell;
         var counter: int=0;
         while counter < cells {
             wait [1]*cycle;
             gen stimuli;
             data_out.put(stimuli);
             counter+=1;
         };
     };
};

 unit rec {
     data_in: in buffer_port of cell is instance;
     keep data_in.buffer_size() == 20;
     get_all() @sys.any is {
         while TRUE {
             print data_in.get();
         };
     };
 };

 extend sys{
     transmitter: trans is instance;
     receiver: rec is instance;
     keep bind(transmitter.data_out, receiver.data_in);
     run() is also {
         start transmitter.drive_all();
         start receiver.get_all();
     };
39  };
```

Line 1 - Line 4 : define “cell”, the data element that is passed by the output buffer port.

Line 7 creates a port instance named “data_out”, whose type is “out buffer_port of cell”.

Line 10 - Line 19 : define a TCM that generates a variable named “stimuli” of type “cell” every cycle until 100 have been generated. This variable is written to the output buffer port by a predefined buffer port TCM, **put()**, in Line 16.

Line 23 creates a port instance named “data_in” of type “in buffer_port of cell”. This port complements the “data_out” port created in the trans unit, and is used to receive cell data written to the data_out port.

Line 24 constrains the maximum number of cells that can be held in the port queue to 20.

Line 25 - Line 29 : define a TCM that retrieves and prints, one by one, the cells that have been placed on the port queue by the drive_all() TCM. Another predefined buffer port method, **get()**, is used to do this.

Line 32 - Line 34 : create instances of the “rec” and “trans” units and connect the data_out port with the data_in port.

## 6.2 Using Simple Ports

You can use simple ports to transfer one data element at a time to or from either an external simulated object, such as a Verilog register, a VHDL signal or a SystemC method, or an internal object (another ***e*** unit). A simple port’s direction can be either input, output or inout.

Internal simple ports can transfer data elements of any type. External ports can transfer scalar types and lists of scalar types, including MVL data elements. Currently there is no support for passing structs or lists of struct through external simple ports.

You can read or write port values using the $ port access operator. To access multi-value logic (MVL) on simple ports, you can either declare a port’s data element to be mvl or list of mvl, or you can use the MVL methods. See “Accessing Simple Ports and Their Values” on page 193 and “Multi-Value Logic (MVL) on Simple Ports” on page 194 for more information.

Internal and external ports must have a bind() attribute that defines how they are connected. In addition, you can use the delayed() attribute to control whether new values are propagated immediately or at the next tick.

An external simple port must have an hdl_path() attribute to specify the name of the object that it is connected to. In addition, an external simple port can have several additional attributes that enable continuous driving of external signals.

See “Port Attributes” on page 210 for more information on attributes for simple ports.

### See Also

— “@sim Temporal Expressions with External Simple Ports” on page 196
— “An Internal Simple Ports Example” on page 197
— “An External Simple Ports Example” on page 198
— “simple_port” on page 202
— “any_simple_port, any_buffer_port, any_event_port” on page 207

### 6.2.1 Accessing Simple Ports and Their Values

Ports are containers, and the values they hold are separate entities from the port itself. The $ access operator distinguishes port value expressions from port reference expressions.

The $ access operator, for example p$, is used to access or update the value held in a simple port p. When used on the right-hand side, p$ refers to the port's value. On the left-hand side of an assignment, p$ refers to the value's location, so an assignment to p$ changes the value held in the port.

Without the $ operator an expression of any type port refers to the port itself, not to its value. In particular, an expression without the $ operator can be used for operations involving port references.

NOTE— You cannot apply the $ access operator to an item of abstract type, such as **any_simple_port**. This type does not have any access methods. The expression "port_arg$ == 0" in the following code causes a syntax error.

```
foo_tcm ( port_arg : any_simple_port )@clk is {
    if ( port_arg$ == 0) then { -- syntax error
        out (sys.time, " Testing port logic comparison.");
    };
};
```

#### Examples of Accessing Port Values

| | |
|---|---|
| print p$; | Prints the value of a simple port, p.<br><br>NOTE— Compare with "print p", which prints information about port p. |
| p$ = 0; | Assigns the value 0 to a simple port, p.<br><br>NOTE— Compare with "pref = NULL", which modifies a port reference so that it does not point to any port instance. |
| force p$ = 0; | Forces a simple external port to 0. |
| print q$[1:0]; | Prints the two least-significant bits of the value of q. |
| print q$[2:2]; | Prints the third least-significant bit of the value of q. |
| print sys.pp$; | Prints the value of port sys.pp. |
| print sys.plist[0]$; | Prints the value of port plist[0] from a list of ports, plist. |
| print blist$[0..1]; | Prints the first two elements of a list value. blist is defined as:<br><br>`blist: in simple_port of list of bit is instance;` |
| print listbl[0]$[1]; | Prints the second bit in a list value of the first element in a list of ports. Could be written (listbl[0])$[1]. listbl is defined as:<br><br>`listbl: list of in simple_port of list of bit is instance;` |

NOTE— Indexing, slicing, and field access for a port value on the left-hand-side of an expression are currently not supported.

### Examples of Accessing a Port

| | |
|---|---|
| print p; | Prints the information about port p. Port p is defined as:<br>`p: simple_port of int (bits:8) is instance;` |
| // p = 5; | An error, as it is an attempt to assign incompatible types. |
| keep q == p; | q refers to the port instance p. Port reference q is defined as:<br>`!q: simple_port of int (bits:8);` |
| r = q; | Port reference r refers to the port instance p too. It is defined as:<br>`var r: any_simple_port;` |
| keep plist.size() == 3; | plist is defined as:<br>`plist: list of in simple_port of int (bits:8) is instance;` |
| keep plist[0] == p; | plist[0] refers to the port instance p. |
| keep plist[1] == p2; | plist[1] refers to the port instance p2. p2 is defined as:<br>`p2: simple_port of int (bits:8) is instance;` |
| keep plist[2] == q; | plist[2] refers to the port instance p (because of q). |

### See Also

— “Multi-Value Logic (MVL) on Simple Ports” on page 194
— “simple_port” on page 202

## 6.2.2 Multi-Value Logic (MVL) on Simple Ports

There are two ways to read and write multi-value logic on simple ports:

— Define numeric ports (uint, int, and so on) and use the predefined MVL methods described in “Multi-Value Logic (MVL) Methods for Simple Ports” on page 235 to read and write values to the port.
— Define ports of type **mvl** or list of **mvl** and use the $ access operator to read and write values to the port.

Ports of type **mvl** or list of **mvl** (MVL ports) allow easy transformation between exact ***e*** values and multi-value logic, which is useful for communicating with objects that sometimes model bit values other than 0 or 1 during a test. Otherwise, using numeric ports is preferable, since numeric ports allow keeping the port values in a bit-by-bit representation, while MVL ports require having an ***e*** list for a multi-value logic vector.

The enumerated type **mvl** is defined as:

type mvl: [MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N]

### Notes

— You will get a syntax error if you use the Verilog comparison operators (=== or !==) with either numeric ports or MVL ports. These operators can be used only with the tick access syntax.

— Not all supported simulators need all the potential MVL values. All nine values are supported only for VHDL simulations. For Verilog simulations, only four values (MVL_U, MVL_X, MVL_0, MVL_1) are supported.

**Example 1: Numeric Port**

This example shows how tick access notation translates to MVL methods, assuming the following numeric port declaration:

```
data: inout simple_port of int is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";
d: int;
```

| | |
|---|---|
| d = 'data'; | d = data$; |
| 'data' = 32'bz; | data.put_mvl_list(32'bz); |
| check that 'data@x' == 0; | check that data.get_mvl_list().has(it == MVL_X) == FALSE; |
| | check that data.has_x() == FALSE; |
| d = 'data[31:10]@z'; | d = mvl_to_int(data.get_mvl_list(), {MVL_Z})[31:0]; |

**Example 2: MVL Port**

This example shows how tick access notation translates to use of an MVL port, assuming the following MVL port declaration:

```
data: inout simple_port of list of mvl is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";
```

| | |
|---|---|
| check that 'data@x' == 0; | check that data$.has(it == MVL_X} == FALSE; |
| | check that data.has_x() == FALSE; |
| 'data' = 32'bz; | data$ = 32'bz; |

**Example 3: Checking Numeric Ports for MVL Values**

If you have several ports that pass numeric data elements of different sizes, you might want to create a generic method that checks these ports for MVL values such as MVL_X or MVL_Z. For example, you can create a generic method for the following ports:

```
byte_port: in simple_port of byte is instance;
uint_port: in simple_port of uint is instance;
```

The correct way to create a generic method is to pass the port value, not the port itself, to the method. You must convert the port value to the desired type before passing it. For example:

```
    x_chk(m: list of mvl) is inline {
        check that m.has(it == MVL_X) == FALSE else
```

```
            dut_error("Bus has value of X!");
        };
        run() is also {
            x_chk(byte_port.get_mvl_list());
            x_chk(uint_port.get_mvl_list());
        };
```

### See Also

— “Multi-Value Logic (MVL) Methods for Simple Ports” on page 235
— “Methods for Simple Ports” on page 244
— “Global MVL Routines” on page 247

### 6.2.3 @sim Temporal Expressions with External Simple Ports

When you specify an event port, you cause ***e*** to be sensitive to the corresponding HDL signal during the entire simulation session. This might result in some unnecessary runtime performance cost if you need ***e*** to be sensitive only in certain scenarios. In such cases you can use an external simple port in temporal expressions with @sim, using the following syntax:

[**change**|**rise**|**fall**](*simple-port$*)**@sim**;

Normally you use this syntax in wait actions. For example:

```
transaction_complete: in simple_port of bit is instance;
    keep bind(transaction_complete, external);

write_transaction(data: list of byte) @clk$ is {
    ...
    data_port$ = data;
    wait rise(transaction_complete$)@sim;
};
```

This syntax might be also useful if you are interested in accessing a value of a signal, in addition to knowing if it changed. For example:

```
counter: in simple_port of uint is instance;
keep bind(counter, external);

event counter_change is change(counter$)@sim;
on counter_change {
  if (counter$ >= 255) {out("Counter is full")};
};
```

### Example

```
unit collector {
    pclk1: in simple_port of bit is instance;
    dataport: in simple_port of byte is instance;

    read_packet(pclk: in simple_port of bit) @sys.any is {
        var p: packet = new;
        var len := dataport$;
        for j from 0 to len - 1 {
            wait fall(pclk$)@sim;
            p.data.add(dataport$);
        };
```

```
            sys.packets.add(p);
        };
        run() is also {
            start read_packet(pclk1);
        };
    };
```

Trying to apply the **@sim** operator to a bound internal port causes an error when the corresponding temporal expression is evaluated, which occurs at runtime.

### See Also

— "simple_port" on page 202

## 6.2.4 An Internal Simple Ports Example

This example shows two units communicating through simple ports, with no external ports.

```
    unit u1 {
        p1: in simple_port of int(bits:64) is instance;
        // Define a simple port
        doit()@sys.any is {
            for i from 1 to 10 do {
                wait cycle;
                print p1$;    // Do a get from the port
                wait cycle;
            };
            stop_run();
        };
        run() is also {
            start doit();
        };
    };
    unit u2 {
        p2: out simple_port of int(bits:64) is instance;
        // Define another simple port
        doit()@sys.any is {
            var v: int(bits:64);
            while TRUE {
                gen v;
                p2$ = v;
                wait cycle;
                wait cycle;
            };
        };
        run() is also {
            start doit();
        };
    };
    extend sys {
        u1: u1 is instance;
        u2: u2 is instance;
        keep bind(u1.p1, u2.p2);    // Bind the two ports
    };
```

### 6.2.5 An External Simple Ports Example

The following ***e*** code describes a testbench component that drives data into an encoder and checks the output of the encoder for errors.

In this example the clk, data_length, data, address and rq ports are external ports associated with various Verilog signals. The name of the simulator is established by the pre-run generation constraint on Line 67 (keep e.agent() == “verilog”). You can re-direct the access to another simulator (and possibly, to another modeling language) by changing this constraint.

Verilog objects associated with the external ports are registers (clk, temp_address, data_width) and nets (data). On the ***e*** side, each port’s behavior corresponds to its specified type—event port, simple port, or buffer port. The event port clk, in Line 3, is used to synchronize the ***e*** program with the simulator. Port rq, in Line 21, illustrates the declaration of a buffer port. The other ports read and write the specified Verilog objects directly.

The postfix $ access operator, for example clk$ or data$ in Line 29 and Line 32, is used to access the event associated with an event port or to read or write to a simple port. Access to a buffer port, on the other hand, is performed using the predefined methods for buffer ports, **get()** and **put()**, as shown in Line 42.

```
1   unit encoder {
2
3       clk: in event_port is instance;
4         keep bind(clk, external);
5         keep clk.hdl_path() == "clk";
6
7       data_length: in simple_port of uint is instance;
8         keep bind(data_length, external);
9         keep data_length.hdl_path() == "data_width";
10
11       data: inout simple_port of list of bit is instance;
12         keep bind(data, external);
13         keep data.hdl_path() == "data";
14         keep data.verilog_wire() == TRUE; -- simple port attribute
15         keep data.declared_range() == "[31:0]"; -- simple port attribute
16
17       address: in simple_port of uint is instance;
18         keep bind(address, external);
19         keep address.hdl_path() == "PRIO/temp_address";
20
21       rq: in buffer_port of bool is instance;
22         keep bind(rq, external);
23         keep rq.buffer_size() == 8; -- buffer port attribute
24         keep rq.hdl_path() == "rq";
25
26       data_list: list of bit;
27         keep data_list.size() < 32;
28
29       inject()@clk$ is {
30           for j from 0 to 15 {
31               gen data_list;
32               data$ = data_list;
33               wait cycle;
34           };
35           stop_run();
36       };
37
```

```
        checker() @clk$ is {
            while TRUE {
                wait cycle;

                if not rq.get() {
                    check that address$ == 0;
                    check that data$.has(it != 0)== FALSE;
                } else {
                    check that address$ != 0;
                    var mask: uint = 0x10000000;
                    for {var i: byte = data_length$ - 1; i>0; i -= 1} {
                        if (data$[31:0] & mask) != 0 {
                            check that address$ == i;
                            break;
                        };
                        mask >>= 1;
                    };
                };
            };
        };
        run() is also {
            start inject();
            start checker();
        };
    };

    extend sys {
        e: encoder is instance;
          keep e.hdl_path() == "~/priority_encoder";
          keep e.agent() == "verilog";

    };
```

## 6.3 Using Buffer Ports

You can use buffer ports to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write access to the port is blocked. When the queue is empty, read access to the port is blocked.

The queue size is fixed during generation by the **buffer_size()** attribute and cannot be changed at runtime. The queue size may be set to 0 for rendezvous ports. See “buffer_size()” on page 217 and “Rendezvous-Zero Size Buffer Queue” on page 200 for more information.

A buffer port’s direction can be either input or output. Inout is not supported. Internal buffer ports can transfer data elements of any type..

You can read or write port values using the buffer port’s predefined **get()** and **put()** methods. These methods are time-consuming methods (TCMs). Use of the $ port access operator with buffer ports is not supported.

Buffer ports must have a bind() attribute that defines how they are connected. In addition, you can use the delayed() attribute to control whether new values are propagated immediately or at the next tick. The pass_by_pointer() attribute controls how data elements of composite type are passed. See “Port Attributes” on page 210 for more information on these attributes.

### See Also

- “An Internal Buffer Ports Example” on page 200
- “buffer_port” on page 204
- “any_simple_port, any_buffer_port, any_event_port” on page 207
- “Methods for Simple Ports” on page 244

## 6.3.1 Rendezvous-Zero Size Buffer Queue

In rendezvous-style handshaking protocol, access to a port is blocked after each **put()** until a subsequent **get()** is performed, and access is blocked after each **get()** until a subsequent **put()** is performed.

This style of communication is easily achieved by using buffer ports with a data queue size of 0. The following example shows how this is done.

### Example

```
unit consumer {
    in_p: in buffer_port of atm_cell is instance;
};

unit producer {
    out_p: out buffer_port of atm_cell is instance;
};

extend sys {
    consumer: consumer is instance;
    producer: producer is instance;
    keep bind(producer.out_p, consumer.in_p);
    keep producer.out_p.buffer_size() == 0;
};
```

### See Also

- “buffer_port” on page 204

## 6.3.2 An Internal Buffer Ports Example

This example shows two units communicating through buffer ports, with no external ports.

```
unit producer {
    p: out buffer_port of atm_cell is instance;
    producer()@sys.any is {
        var cell: atm_cell;
        for i from 1 to 100 do {
            gen cell;
            p.put(cell)    // Waits if the buffer is full
        };
        stop_run();
    };
};
unit consumer {
    p: in buffer_port of atm_cell is instance;
    consumer()@sys.any is {
        while (TRUE) do {
            var cell: atm_cell;
            cell = p.get();    // Waits if the buffer is empty
```

```
                // Inject the cell into the DUT
            };
        };
    };
    extend sys {
        consumer: consumer is instance;
        producer: producer is instance;
        keep bind(producer.p, consumer.p);
        keep producer.p.buffer_size() == 10;
    };
```

## 6.4 Using Event Ports

You can use event ports to transfer events between two ***e*** units or between an ***e*** unit and an external object. An internal event port's direction can be either input, output or inout.

You can read or write port values using the $ port access operator. See "Accessing Event Ports" on page 201 for more information.

Internal and external ports must have a bind() attribute that defines how they are connected.

An external port must have an hdl_path() attribute to specify the name of the object that it is connected to. The edge() attribute for an external input event port specifies the edge on which an event is generated.

See "Port Attributes" on page 210 for more information on these attributes.

### See Also

— "event_port" on page 205
— "any_simple_port, any_buffer_port, any_event_port" on page 207

### 6.4.1 Accessing Event Ports

The $ access operator is used to access the event associated with an event port. An expression of type event_port without the '$' operator refers to the port itself and not to its event.

#### Example 1

```
    emit me.ep$;
    monitor()@ep$ is { ... };
    wait @lep[0]$;
    event ep1 is @ep$;
    wait cycle @ep$;
    expect @a => { ... }@ep$;
```

#### Example 2

This example shows how to connect event ports, using a **bind()** constraint, and how to use the $ operator to access event ports in event contexts.

```
    unit u1 {
        in_ep: in event_port is instance;
        tcm1()@in_ep$ is {
        // ...
        };
```

```
};

unit u2 {
    out_ep: out event_port is instance;
    event clk is @sys.any;
    counter: uint;
    on clk {
        counter = counter + 1;
        if counter %10 == 0 {
            emit out_ep$
        };
    };
};

extend sys {
    u1: u1 is instance;
    u2: u2 is instance;
    keep bind(u1.in_ep,u2.out_ep);
};
```

### See Also

— "Methods for Simple Ports" on page 244

## 6.4.2 Defining and Referencing Ports

This section covers the following topics:

— "simple_port" on page 202
— "buffer_port" on page 204
— "event_port" on page 205
— "any_simple_port, any_buffer_port, any_event_port" on page 207
— "port$" on page 209

### 6.4.2.1 simple_port

#### Purpose

Access other port instances or external simulated objects directly

#### Category

Unit member

#### Syntax

***port-instance-name***: [**list of**] [***direction***] **simple_port of** ***element-type*** **is instance**;

Syntax example:

```
data: in simple_port of byte is instance;
```

### Parameters

| | |
|---|---|
| *port-instance-name* | A unique identifier you can use to refer to the port or access its value. |
| *direction* | One of **in**, **out**, or **inout**. The default is **inout**, which means that you can read values from and write values to this port. For an **in** port, you can only read values from the port, and for an **out** port you can only write values to the port. |
| *element-type* | Any predefined or user-defined ***e*** type except a port type or a unit type. |

### Description

You can use simple ports to transfer one data element at a time to or from either an external simulated object or an internal object (another ***e*** unit).

Internal simple ports can transfer data elements of any type. External ports can transfer scalar types and lists of scalar types, including MVL data elements. Currently there is no support for passing structs or lists of struct through external simple ports.

The port can be configured to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. Binding is fixed during generation.

A simple port's direction can be either input, output, or inout. The direction specifier in a simple port is not a when subtype determinant. This means, for example, that the following type:

```
data: simple_port of byte is instance;
```

is **not** the base type of:

```
data: out simple_port of byte is instance;
```

Furthermore, the following types are fully equivalent:

```
data: simple_port of byte is instance;
data: inout simple_port of byte is instance;
```

Thus, the following constraint is an error because the two types are not equivalent:

```
data: out simple_port of byte is instance;
!data_ref: simple_port of byte; // means inout simple_port of byte
keep data_ref == data; // error
```

### Example

```
<'
unit encoder {

    data: out simple_port of int(bits:64) is instance;
      keep bind(data, external);
      keep data.hdl_path() == "data";

    drive()@sys.any is {
        var v: int(bits:64);
        while TRUE {
            gen v;
            data$ = v;
            wait cycle;
            wait cycle;
        };
    };
    run() is also {
        start drive();
    };
};
'>
```

### See Also

— “Using Simple Ports” on page 192
— “any_simple_port, any_buffer_port, any_event_port” on page 207
— “Multi-Value Logic (MVL) Methods for Simple Ports” on page 235
— “Global MVL Routines” on page 247
— “Methods for Simple Ports” on page 244

### 6.4.2.2 buffer_port

### Purpose

Implement an abstraction of queues with blocking get and put

### Category

Unit member

### Syntax

***port-instance-name***: [**list of**] ***direction*** **buffer_port of** ***element-type*** **is instance**;

Syntax example:

```
rq: in buffer_port of bool is instance;
```

### Parameters

| | |
|---|---|
| *port-instance-name* | A unique identifier you can use to refer to the port or access its value. |
| *direction* | One of **in** or **out**. There is no default. For an **in** port, you can only read values from the port, and for an **out** port you can only write values to the port. See “Buffer Port Methods” on page 232 for information on how to read and write buffer ports. |
| *element-type* | Any predefined or user-defined ***e*** type except a unit or a port type. |

### Description

You can use buffer ports to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write access to the port is blocked. When the queue is empty, read access to the port is blocked.

The queue size is fixed during generation by the **buffer_size()** attribute and cannot be changed at runtime. The queue size may be set to 0 for rendezvous ports.

You can read or write port values using the buffer port's predefined **get()** and **put()** methods. These methods are time-consuming methods (TCMs). Use of the $ port access operator with buffer ports is not supported.

A typical usage of a buffer port is in a ***producer*** and ***consumer*** protocol, where one object puts data on an output port at possibly irregular intervals, and another object with the corresponding input port reads the data at its own rate.

#### Example

```
unit encoder {

    rq: in buffer_port of bool is instance;
      keep rq.buffer_size() == 8; -- buffer port attribute
};
```

#### See Also

— "Using Buffer Ports" on page 199
— "Buffer Port Methods" on page 232
— "Methods for Simple Ports" on page 244

### 6.4.2.3 event_port

#### Purpose

Transfer events between units or between simulators and units

#### Category

Unit member

#### Syntax

***event-port-field-name*: [list of] [*direction*] event_port is instance;**

Syntax example:

```
clk: in event_port is instance;
```

#### Parameters

| | |
|---|---|
| *event-port-field-name* | A unique identifier you can use to refer to the port or access its value. |
| *direction* | One of **in**, **out**, or **inout**. The default is **inout**, which means that events can be both emitted and sampled on the port. For a port with direction **in**, events can only be sampled. For a port with direction **out**, events can only be emitted. |

#### Description

You can use event ports to transfer events between two ***e*** units or between an ***e*** unit and an external object.

You can read or write port values using the $ port access operator. See "Accessing Event Ports" on page 201 for more information.

An internal event port's direction specifier can be either input, output or inout. The direction specifier is not a **when** subtype determinant. This means, for example, that the following type

```
clk: event_port is instance;
```

is **not** the base type of

```
clk: out event_port is instance;
```

Furthermore, the following types are fully equivalent:

```
clk: event_port is instance;
clk: inout event_port is instance;
```

### Notes

- — Currently, external out and inout event ports are unsupported.
- — The **on** struct member for event ports is not supported.
- — Coverage on event ports is currently unsupported.
- — It is impossible to specify a temporal formula (like "event_port is ...") for definition of an out event port.

In order to use any of the above unsupported capabilities (except the first in the list) it is possible to define an additional event and connect it to the event port as follows:

```
ep: in event_port is instance;
keep bind(ep,external);
event e is @ep$;
```

### Example 1

References to event ports are supported. In the following example, current_clk is an event port reference.

```
unit u {
    clks: list of in event_port is instance;
    events: list of out event_port is instance;
};

extend u {
    !current_clk: in event_port;
    keep current_clk == clks[0];
};
```

### Example 2

You can pass an event port as a parameter to a TCM. In this example, each event in a list of events is passed as a parameter to the drive() method.

```
extend u {
    drive(ep: out event_port) @current_clk$ is {
        emit ep$;
    };
```

```
        run() is also {
            for each in events do {
                start drive(it);
            };
        };
    };
```

### Example 3

The attribute **hdl_path()** must be specified for external event ports. In the following example, only a “cti” simulator can emit ext_ep. Presumably there is some DUT event related to a simulated item “~/top_s/transaction_done”.

```
    unit u {
        ext_ep: in event_port is instance;
          keep bind(ext_ep,external);
          keep ext_ep.hdl_path() == "transaction_done";
    };
    extend sys {
        u: u is instance;
          keep u.hdl_path() == "top_s";
    };
```

### See Also

- “Using Event Ports” on page 201
- “Methods for Simple Ports” on page 244

## 6.4.2.4 any_simple_port, any_buffer_port, any_event_port

### Purpose

Reference a port instance

### Category

Unit field, variable or method parameter

### Syntax

[**!** | **var**] ***port-reference-name*****:** [***direction***] ***port-kind*** [**of** ***element-type***]

[**!** | **var**] ***port-reference-name*****:** ***any-port-kind***

Syntax example:

```
        !last_printed_port: any_buffer_port;
        !in_int_buffer_port_ref: in buffer_port of int;
```

## Parameters

| | |
|---|---|
| *port-reference-name* | A unique identifier. |
| *direction* | One of **in**, **out**, or, for simple ports and event ports, **inout.** |
| *port-kind* | One of **simple_port**, **buffer_port** or **event_port**. |
| *any-port-kind* | One of **any_simple_port**, **any_buffer_port** or **any_event_port**. |
| *element-type* | Required if port-kind is **simple_port** or **buffer_port.** |

## Description

Port instances may be referenced by a field, a variable, or a method parameter of the same port type or of an abstract type:

- any_simple_port
- any_buffer_port
- **any_event_port**.

Abstract port types reference only the port kind, not the port direction or data element. Thus, a method parameter of type **any_simple_port** accepts all simple ports, including, for example:

```
data_length: in simple_port of uint is instance;
data: inout simple_port of list of bit is instance;
```

If a port reference is a field, then it must be marked as non-generated or it must be constrained to an existing port instance. Otherwise, a generation error results.

Port binding is allowed only for port instance fields, not for port reference fields. Trying to apply a **keep bind()** constraint to a port reference results in an error.

## Notes

- You cannot apply the **$** access operator to an item of type **any_simple_port** or **any_event_port**. Abstract types do not have any access methods. For example, the expression “port_arg$ == 0” in the following code causes a syntax error.

```
foo_tcm ( port_arg : any_simple_port )@clk is {

    if ( port_arg$ == 0) then { -- syntax error

        out (sys.time, " Testing port logic comparison.");

    };

};
```

- You cannot use an abstract type in a port instance; you must specify the element type.

## Example

The print_port() method in the following example can be called with any buffer port. The iterate() method shows an alternative way to print a list of ports.

```
unit u {
    plist: list of in buffer_port of int is instance;
    !last_printed_port: any_buffer_port; // A field, so must be
                                         // non-generated
```

```
        print_port(p: any_buffer_port) is {  // A method parameter
            print p;          // Prints the port's e path, agent name, and so on
            last_printed_port = p;
        };

        iterate() is {
            for each in plist {
               in_int_buffer_port_ref = it;
               print_port(in_int_buffer_port_ref);
            };
        };

    };
```

### 6.4.2.5 port$

#### Purpose

Read or write a value to a simple port or event port

#### Category

Operator

#### Syntax

*exp*$

Syntax example:

```
    p$ = 32'bz;          // Assigns an mvl literal to the port 'p'
```

#### Parameters

| | |
|---|---|
| *exp* | An expression that returns a simple port or event port instance. |

#### Description

The $ access operator is used to access or update the value held in a simple port or event port. When used on the right-hand side, p$ refers to the port's value. On the left-hand side of an assignment, p$ refers to the value's location, so an assignment to p$ changes the value held in the port.

Without the $ operator an expression of any type port refers to the port itself, not to its value. In particular, an expression without the $ operator can be used for operations involving port references.

NOTE— You cannot apply the $ access operator to an item of type **any_simple_port** or **any_event_port**. Abstract types do not have any access methods. For example, the expression "port_arg$ == 0" in the following code causes a syntax error.

```
        foo_tcm ( port_arg : any_simple_port )@clk is {

            if ( port_arg$ == 0) then { -- syntax error

                out (sys.time, " Testing port logic comparison.");
        };
    };
```

### Example

```
<'
unit u {
    free_port(p: inout simple_port of list of mvl) is {
        p$ = 32'bz;          // Assigns an mvl literal to the port
    };
};
'>
```

### See Also

- — "Accessing Simple Ports and Their Values" on page 193
- — "Accessing Event Ports" on page 201
- — "Multi-Value Logic (MVL) on Simple Ports" on page 194
- — "Methods for Simple Ports" on page 244

## 6.5 Port Attributes

Ports have attributes that affect their behavior and how they can be used. You assign port attributes using the ***attribute***() syntax in pre-generation constraints, as follows:

**keep** [**soft**] ***port_instance.attribute*() == *value*;**

You can use soft constraints for attributes that you might want to override later.

Most port attributes are ignored unless the port is an external port, but it does no harm to specify attributes for ports that are not external ports. Attributes intended for external ports may or may not be supported for a particular simulator. A particular adapter can also define additional port attributes that are required to enhance access to simulated objects.

### 6.5.1 Generic Port Attributes

Port attributes that are potentially valid for all simulators are described in Table 6-1. However, a particular simulator adapter might not implement some of these attributes.

NOTE— Depending on the simulator adapter you are using, port attributes might cause additional code to be written to the stubs file. In that case, if you add or change an attribute, you must rewrite the stubs file.

**Table 6-1—Generic Port Attributes**

| Attribute | Description | Applies to |
|---|---|---|
| bind() | Connects two internal ports or connect a port to an external object<br><br>Type: bool<br><br>Default: none<br><br>See also "bind()" on page 215. | All kinds of internal and external ports |
| buffer_size() | Specifies the maximum number of elements for a buffer port queue.<br><br>Type: uint<br><br>Default: none<br><br>See also "buffer_size()" on page 217. | Buffer ports |
| declared_range() | Specifies the bit width of an external multi-bit object.<br><br>Type: string<br><br>Default: none<br><br>See also "declared_range()" on page 219. | External output simple ports that are bound to some kinds of multi-bit objects |
| delayed() | Specifies whether propagation of a new port value assignment occurs immediately or is delayed to the tick boundary.<br><br>Type: bool<br><br>Default: TRUE<br><br>See also "delayed()" on page 219. | Internal and external simple ports |
| driver() | When TRUE, an additional resolved HDL driver is created for the corresponding simulator item, and that driver is written to instead of the port.<br><br>Type: bool<br><br>Default: FALSE<br><br>See also "driver()" on page 220. | External output simple ports |

**Table 6-1—Generic Port Attributes**

| Attribute | Description | Applies to |
|---|---|---|
| driver_delay() | Specifies the delay time for all assignments from ***e*** to the port.<br><br>Type: time<br><br>Default: 0<br><br>See also "driver_delay()" on page 221. | External output simple ports |
| edge() | Specifies the edge on which an event is generated.<br><br>Type: event_port_edge<br><br>Default: change<br><br>See also "edge()" on page 222. | External input event ports |
| hdl_path() | Specifies a relative path of the corresponding simulated item as a string.<br><br>Type: string<br><br>Default: none<br><br>See also "hdl_path()" on page 223. | External ports |
| pack_options() | Specifies how the port's data element is implicitly packed and unpacked.<br><br>Type: pack_options<br><br>Default: **global.packing.adapter**<br><br>See also "pack_options()" on page 225. | External simple ports whose data element is a composite type (lists and structs) |
| pass_by_pointer | When TRUE, composite data (structs or lists) are transferred by reference.<br><br>Type: bool<br><br>Default: FALSE (pass by value)<br><br>See also "pass_by_pointer()" on page 225. | Internal simple or buffer ports whose data element is a composite type (lists and structs) |

### 6.5.2 Port Attributes for HDL Simulators

Port attributes that are potentially valid for all HDL simulators are described in Table 6-2. However, a particular simulator adapter might not implement some of these attributes.

The port attributes in Table 6-2 enable extended functionality. They cause additional information to be written into the HDL stubs file to enhance user control over the driving of HDL signals. For this reason, if you add or change any attribute shown in Table 6-2, you must rewrite the stubs file.

Some of these attributes are similar to Verilog or VHDL unit members, such as **verilog variable** or **vhdl driver**.

### Example

The following **verilog variable** declaration

```
verilog variable 'sig[7:0]' using strobe="#1", drive="#5" ;
```

is equivalent to the following port attributes:

```
data : inout simple_port of uint(bits: 8) is instance;
keep bind(data, external);
keep data.hdl_path()=="sig";
keep data.declared_range() == "[7:0]";
keep data.verilog_strobe() == "#1";
keep data.verilog_drive() == "#5";
```

**Table 6-2—Port Attributes for Verilog or VHDL Agents**

| Attribute | Description | Applies to |
|---|---|---|
| driver_initial_value() | Applies an initial **mvl** value to the port.<br><br>Type: list of mvl<br><br>Default: {} (empty list)<br><br>See also "driver_initial_value()" on page 222. | External output simple ports |
| verilog_drive() | Specifies the event on which the data is driven to the Verilog object.<br><br>Type: string<br><br>Default: none<br><br>See also "verilog_drive()" on page 226. | External output simple ports |
| verilog_drive_hold() | Specifies an event after which the port data is set to Z.<br><br>Type: string<br><br>Default: none<br><br>See also "verilog_drive_hold()" on page 227. | External output simple ports |

**Table 6-2—Port Attributes for Verilog or VHDL Agents**

| Attribute | Description | Applies to |
|---|---|---|
| verilog_forcible() | Allows forcing of Verilog wires.<br><br>Type: bool<br><br>Default: FALSE<br><br>See also "verilog_forcible()" on page 227. | External output simple ports |
| verilog_strobe() | Specifies the sampling event for the Verilog signal that is bound to the port.<br><br>Type: string<br><br>Default: none<br><br>See also "verilog_strobe()" on page 228. | External output simple ports |
| verilog_wire() | Binds an external out port to a Verilog wire.<br><br>Type: bool<br><br>Default: FALSE<br><br>See also "verilog_wire()" on page 229. | External output simple ports |
| vhdl_delay_mode() | Specifies whether pulses whose period is shorter than the delay are propagated through the driver.<br><br>Type: sn_vhdl_delay_mode<br><br>Default: TRANSPORT (all pulses, regardless of length, are propagated)<br><br>See also "vhdl_delay_mode()" on page 229. | External output simple ports |
| ~~vhdl_disconnect_value()~~ | ~~Applies an **mvl** value to the port when you restore Specman Elite after issuing a **test** command but do not restart the simulator.~~<br><br>~~Type: list of mvl~~<br><br>~~Default: {} (empty list)~~<br><br>~~See also~~ "vhdl_disconnect_value()" on page 230~~.~~ | External output simple ports |

**Table 6-2—Port Attributes for Verilog or VHDL Agents**

| Attribute | Description | Applies to |
|---|---|---|
| vhdl_driver() | This is an alias for the **driver()** attribute.<br><br>Type: bool<br><br>Default: FALSE<br><br>See also "driver()" on page 220. | External output simple ports |

### 6.5.2.1 bind()

#### Purpose

Connect two internal ports or connect a port to an external object

#### Category

Generic port attribute

#### Syntax

**bind(*exp1*, *exp2*);**

**bind(*exp1*, external);**

**bind(*exp1*, empty | undefined);**

Syntax example:

```
buf_in1: in buffer_port of int(bits:16) is instance;
buf_out1: out buffer_port of int(bits:16) is instance;
keep bind(buf_in1, buf_out1);                         // Valid
```

#### Parameters

| | |
|---|---|
| *exp1, exp2* | One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected. |
| external | Defines a port as connected to a simulated object, such as a Verilog register, a VHDL signal, or a SystemC object. |
| empty | Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed. |
| undefined | Defines a disconnected port. Runtime accessing of a port with an undefined binding causes an error |

#### Description

Ports are connected to other ***e*** ports or to external simulated objects such as Verilog registers, VHDL signals, or SystemC methods using a pre-run generation constraint on the **bind()** attribute. Ports can also be left explicitly disconnected with **empty** or **undefined**.

### Rules

— All ports must be bound in one of the following ways:

- Bound in pairs, that is, one in or inout port bound to one out or inout port. It is illegal to bind together two input ports, two output ports, or two inout ports.
- Only ports of the same kind may be bound together. A simple port cannot be bound to a buffer port or to an event port and a buffer port cannot be bound to an event port.
- Bound to an external simulated item.
- Explicitly disconnected (empty or undefined).

  NOTE— Dangling ports (ports without **bind()** attributes) cause an error during elaboration. See "Checking of Ports" on page 216 for more information.

— Currently, no port may be connected to more than one other port. In other words, you can connect port A to port B or to port C but not to both.
— You can explicitly disconnect a port and then over-ride that disconnect with a binding to an internal or external object. No other multiple bindings are allowed. In other words, you cannot bind a port to an internal object and also bind it to an external object. Similarly, you cannot define a port's binding as both empty and undefined.
— Ports connected in a pair must have exactly the same element type.

NOTE— For Verisity adapters, if you add or change this attribute for an external port, you must rewrite the stubs file.

### Checking of Ports

Binding and checking of ports takes place automatically at the end of the predefined **generate_test()** test method. This process, called elaboration of ports, includes checking for dangling ports and binding consistency (directions, buffer sizes, and so on).

A port that has no **bind()** constraint is a dangling port. Since all ports must be bound, a dangling port causes an elaboration-time error.

### Disconnected Ports

A port that is bound using the **empty** or **undefined** keyword is called a disconnected port.

The **empty** or **undefined** keyword can only appear as the second argument of the **bind()** constraint, in place of a second port instance name.

The same port cannot be both empty and undefined. Attempting to apply such contradicting constraints to one port causes an elaboration-time error.

Empty binding allows you to define a port that is connected to nothing. Runtime accessing of an empty-bound port is allowed. Its effect depends on the operation and type of the port:

— Reading from an empty-bound simple port returns the last written value or the default of the port element type, if no value has been written so far.
— Writing to an empty-bound out or inout simple port stores the new value internally.
— Reading from an empty-bound buffer port causes the thread to halt.
— Writing to an empty-bound buffer port causes the thread to halt if the buffer is full.

— Waiting for an empty-bound event port causes the thread to halt. If the port direction is inout then emitting the port resumes the thread.
— An empty-bound event port can be emitted.

A subsequent constraint can be used to overwrite the empty binding constraint.

Like empty binding, undefined binding lets you define a port that is connected to nothing. The difference is that runtime accessing of a port with an undefined binding causes an error.

A subsequent constraint can be used to overwrite the undefined binding constraint.

**Example 1: Valid Bindings**

```
buf_in1: in buffer_port of int(bits:16) is instance;
buf_out1: out buffer_port of int(bits:16) is instance;
keep bind(buf_in1, buf_out1);                          // Valid

buf_in4: in buffer_port of int(bits:16) is instance;
buf_out4: out buffer_port of int(bits:16) is instance;
keep bind(buf_in4, empty);
keep bind(buf_in4, buf_out4); // Valid; buf_in4 will be bound to buf_out4

simple_in1: in simple_port of int(bits:16) is instance;
keep bind(simple_in1, empty);
keep bind(simple_in1, external);
keep simple_in1.hdl_path() == "foo"; // Valid; buf_in5 will be bound to foo
```

**Example 2: Invalid Bindings**

```
buf_in2: in buffer_port of int(bits:32) is instance;
buf_out2: out buffer_port of int(bits:16) is instance;
keep bind(buf_in2, buf_out2); // Invalid; different bit size

buf_in3: in buffer_port of packet is instance;
buf_out3: out buffer_port of small packet is instance;
keep bind(buf_in3, buf_out3); // Invalid; different subtypes

simple_in2: in simple_port of int(bits:16) is instance;
simple_out2: out simple_port of int(bits:16) is instance;
keep bind(simple_in2, simple_out2);
keep bind(simple_in2, external); // Invalid; multiple binding
```

`Example 3`

The **bind()** method can also be used in procedural code. It returns TRUE if the port in its argument is bound as specified. For example:

```
print bind(p, q);
```

### 6.5.2.2 buffer_size()

**Purpose**

Specify the size of a buffer port queue

### Category

Buffer port attribute

### Syntax

***exp*.buffer_size() == *num***

Syntax example:

```
keep u.p.buffer_size() == 20;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of type [**in** \| **out**] **buffer_port of *type***. |
| *num* | An integer specifying the maximum number of elements for the queue. |

### Description

This attribute determines the number of **put()** actions that can be performed before a **get()**. A **get()** action is required to remove data and make more room in the queue. Specifying a buffer size of 0 means rendezvous-style synchronization.

No default buffer size is provided. If a buffer size is not specified in a constraint, an error occurs. It is only necessary to specify a buffer size for one of the two ports in a pair of connected ports. That size applies to both ports. If the two ports have different buffer sizes specified, then both of them get the larger of the two sizes.

### Example

Like all port attributes, the buffer size can also be used as an expression.

```
unit consumer {
    in_p: in buffer_port of atm_cell is instance;
};

unit producer {
    out_p: out buffer_port of atm_cell is instance;
};

extend sys {
    consumer: consumer is instance;
    producer: producer is instance;
    keep bind(producer.out_p, consumer.in_p);
    keep producer.out_p.buffer_size() == 500;

    run() is also {
        // Print the size of the queue
        outf("Size of the queue is set to %u\n",
            consumer.in_p.buffer_size());
    };
};
```

### See Also

— “buffer_port” on page 204

### 6.5.2.3 declared_range()

#### Purpose

Specify the bit width of a multi-bit external object

#### Category

External port attribute

#### Syntax

***exp*.declared_range() == *string***

Syntax example:

```
keep u.p.declared_range() == "[31:0]";
```

#### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *string* | An expression in the form: "[***msb***:***lsb***]" |

#### Description

This string attribute is meaningful for external simple ports that are bound to multi-bit objects. Because it is legal to bind a port to an HDL object with a different size, the range information is not extracted from the port declaration. In order to implement access to multi-bit signals correctly in the stubs file, this attribute is required when using the **verilog_wire(), verilog_drive(), verilog_strobe()** or **driver()** attributes.

The interpretation of the string is adapter-specific. For Verisity adapters, the declared range must match the actual range of the signal; it cannot be a part select.

#### Example

```
unit u {
    p: simple_port of int is instance;
};
extend sys {
    u: u is instance;
    keep u.hdl_path() == "top";
    keep u.agent() == "Verilog";
    keep bind(u.p, external);
    keep u.p.hdl_path() == "shr";
    keep u.p.verilog_wire() == TRUE;
    keep u.p.declared_range() == "[31:0]";
};
```

### 6.5.2.4 delayed()

#### Purpose

Specify immediate or delayed propagation of new values

### Category

Simple port attribute

### Syntax

***exp*.delayed() == *bool***

Syntax example:

```
keep u.p.delayed() == FALSE;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *bool* | Either TRUE or FALSE. The default is TRUE. |

### Description

This boolean attribute specifies whether propagation of a new port value assignment occurs immediately or is delayed.

When the **delayed()** attribute is TRUE (the default), propagation of external ports is delayed until the next tick. Propagation of internal ports is delayed until the next tick at which the **sys.time** value changes. This behavior is consistent with the definition of delayed assignments in ***e*** and matches temporal ***e*** semantics with regard to the multiple ticks occurring at the same simulator time.

To make assigned values on ports visible immediately, constrain this attribute to be FALSE, for example:

```
keep u.p.delayed() == FALSE;
```

## 6.5.2.5 driver()

### Purpose

Create a resolved driver for an external object

### Category

External out simple port attribute

### Syntax

***exp*.driver() == *bool***

Syntax example:

```
keep u.p.driver() == TRUE;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *bool* | Either TRUE or FALSE. The default is FALSE. |

### Description

This boolean attribute is meaningful only for external out ports. When this attribute is set to TRUE, an additional resolved HDL driver is created for the corresponding simulator item, and that driver is written to instead of the port.

Every port instance associated with the same simulator may create a separate driver, thus allowing HDL resolution to be applied for multiple ***e*** resources.

### Notes

— For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.
— Verisity adapters require that you also use **declared_range()** if the object that is driven is multi-bit.
— Verisity Verilog adapters make use of this attribute only if it is applied to an external signal that can be driven contiguously and allows multiple drivers, such as Verilog wires (not registers or memories).
— Verisity VHDL adapters make use of this attribute only for MTI ModelSim and only if the VHDL signals are of a resolved type (not VHDL variables or signals of unresolved type).
— The Verisity OSCI (SystemC) adapter requires this attribute to be specified in order to drive SystemC ports.

### 6.5.2.6 driver_delay()

### Purpose

Specify the delay for assignments to a port

### Category

External out simple port attribute

### Syntax

***exp*.driver_delay() == *time***

Syntax example:

```
keep u.p.driver_delay() == 2;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *time* | A value of type **time** (64 bits). The default is 0. |

### Description

This attribute of type **time** is meaningful only for external out ports. It specifies the delay time for all assignments from ***e*** to the port. This attribute is silently ignored unless the **driver()** attribute or the **vhdl_driver()** attribute is set to TRUE.

NOTE— For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.

### 6.5.2.7 driver_initial_value()

### Purpose

Specify an initial value for an HDL object

### Category

HDL port attribute

### Syntax

***exp*.driver_initial_value() == *mvl-list***

Syntax example:

```
keep u.p.driver_initial_value() == {MVL_X;MVL_X;MVL_1;MVL_1};
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a port instance. |
| *mvl-list* | A lists of mvl values. Possible values are MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N. The default is {} (an empty list). |

### Description

This **mvl** list type attribute applies an initial **mvl** value to an external Verilog or VHDL object. This attribute is silently ignored unless the **driver()** attribute or the **vhdl_driver()** attribute is set to TRUE.

When an ***e*** program is driving a std_logic signal that is also driven from VHDL, unless an initial value is specified, the adapter creates a VHDL driver that is initialized by MVL_X.

### 6.5.2.8 edge()

### Purpose

Specify the edge on which an event is generated

### Category

Event port attribute

### Syntax

***exp*.edge() == *edge-option***

Syntax example:

keep e.edge() == any_change;

### Parameters

| | |
|---|---|
| *exp* | An expression of an event port type. |
| *edge-option* | Possible values are of type event_port_edge: |

- **change**, **rise**, **fall** — equivalent to the behavior of **@sim** temporal expressions. This means that transitions between x and 0, z and 1 are not detected, x to 1 is considered a rise, z to 0 a fall, and so on.
- **any_change** — any change within the supported MVL values is detected, including transitions from x to 0 and 1 to z.
- **MVL_0_to_1** — transitions from 0 to 1 only.
- **MVL_1_to_0** — transitions from 1 to 0 only.
- **MVL_X_to_0** — transitions from X to 0 only.
- **MVL_0_to_X** — transitions from 0 to X only.
- **MVL_Z_to_1** — transitions from Z to 1 only.
- **MVL_1_to_Z** — transitions from 1 to Z only.

The default is **change**.

### Description

This attribute of type **event_port_edge** for an external event port specifies the edge on which an event is generated.

### Example

```
e: in event_port is instance;
  keep bind(e,external);
  keep e.hdl_path() == "clk";
  keep e.edge() == any_change;
```

### 6.5.2.9 hdl_path()

### Purpose

Map port instance to an external object

### Category

Generic port attribute

### Syntax

***exp*.hdl_path() == *string***

Syntax example:

```
        clk: in event_port is instance;
          keep clk.hdl_path() == "clk";
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a port type. |
| *string* | The path to the external object, enclosed in double quotes. The default is an empty string. |

### Description

To access an external, simulated object, you must provide a path to the object with the **hdl_path()** attribute. This path is a concatenation of the partial paths you provide for the port itself and for its enclosing units. The partial paths can contain any separator that is supported by the adapter for the simulator you are using.

To allow portability between simulators, you can use the ***e*** canonical path notation. (See the documentation for the adapter for a description of supported separators.)

NOTE— For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.

### Example

In this example, all ports inherit the Verilog simulator specified as the agent for the encoder instance. The clk, data_width, data and rq ports access Verilog signals of the same name in the top-level module "priority_encoder". The address port accesses a signal with the path priority_encoder.PRIO.temp_address.

```
    unit encoder {

        clk: in event_port is instance;
          keep bind(clk, external);
          keep clk.hdl_path() == "clk";

        data_length: in simple_port of uint is instance;
          keep bind(data_length, external);
          keep data_length.hdl_path() == "data_width";

        data: inout simple_port of list of bit is instance;
          keep bind(data, external);
          keep data.hdl_path() == "data";
          keep data.verilog_wire() == TRUE; -- simple port attribute
          keep data.declared_range() == "[31:0]"; -- simple port attribute

        address: in simple_port of uint is instance;
          keep bind(address, external);
          keep address.hdl_path() == "PRIO/temp_address";
        rq: in buffer_port of bool is instance;
          keep bind(rq, external);
          keep rq.buffer_size() == 8; -- buffer port attribute
          keep rq.hdl_path() == "rq";


    extend sys {
        e: encoder is instance;
          keep e.hdl_path() == "~/priority_encoder";
    };
```

#### 6.5.2.10 pack_options()

**Purpose**

Specify how an external port's data element is implicitly packed and unpacked

**Category**

External simple port attribute

**Syntax**

***exp*.pack_options() == *pack-option***

Syntax example:

```
keep u.p.pack_options() == packing.low_big_endian;
```

**Parameters**

| | |
|---|---|
| *exp* | An expression of a simple or buffer port type. |
| *pack-option* | A predefined or user-defined pack option. The default is **global.packing.adapter**. |

**Description**

This attribute of type **pack_options** is meaningful only for external ports whose data element is a composite type (lists and structs). It affects the way a port's data element is implicitly packed and unpacked. This attribute exists both for units and ports and may be propagated downwards from an enclosing unit instance to its ports and other unit instances.

NOTE— None of the existing simulator adapters supports external simple port of structs.

#### 6.5.2.11 pass_by_pointer()

**Purpose**

Specify how composite data is transferred by internal ports

**Category**

Internal port attribute

### Syntax

***exp*.pass_by_pointer() == *bool***

Syntax example:

```
keep u.p.pass_by_pointer() == TRUE;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple or buffer port type. |
| *bool* | Either TRUE or FALSE. The default is FALSE. |

### Description

This boolean attribute specifies how composite data (structs or lists) are transferred by internal simple ports or buffer ports.

By default, this attribute is FALSE and complex objects are deep-copied upon an internal port access operation. To pass data by reference and speed up the test, you can set this attribute to TRUE. If you do so, you must write your code such that it does not result in test correctness violations.

There is also a global **config misc** option, **ports_data_pass_by_pointer**. Setting this option influences all internal ports.

## 6.5.2.12 verilog_drive()

### Purpose

Specify timing control for data driven to the Verilog object

### Category

Verilog port attribute

### Syntax

***exp*.verilog_drive() == *timing-control***

Syntax example:

```
keep u.p.verilog_drive() == "@posedge clk2";
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *timing-control* | A string specifying any legal Verilog timing control (event or delay). |

### Description

This string attribute tells an external output port to drive its data to the Verilog signal when the specified timing occurs. It can be either a Verilog temporal expression such as "@(posedge top.clk)" or a simple delay of kind "#1". This attribute is functionally equivalent to a **verilog variable using drive** declaration.

### Notes

— For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.
— Verisity adapters require that you also use **declared_range()** if the object that is driven is multi-bit.

### 6.5.2.13 verilog_drive_hold()

### Purpose

Specify when to set the port to Z

### Category

Verilog port attribute

### Syntax

***exp*.verilog_drive_hold() ==** ***event***

Syntax example:

```
keep u.p.verilog_drive_hold() == "@negedge clk2";
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *event* | A string specifying any legal Verilog timing control. |

### Description

On the first occurrence of the specified event after the port data is driven, the value of the corresponding Verilog signal is set to Z. The event is a string specifying any legal Verilog timing control. This attribute requires that you also specify the **verilog_drive()** attribute.

### 6.5.2.14 verilog_forcible()

### Purpose

Specifies that a Verilog object can be forced

### Category

Verilog port attribute

### Syntax

***exp*.verilog_forcible() == *bool***

Syntax example:

```
keep u.p.verilog_forcible() == TRUE;
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *bool* | Either TRUE or FALSE. The default is FALSE. |

### Description

This boolean attribute allows forcing of Verilog wires. By default Verilog wires are not forcible. This attribute requires that you also specify the **verilog_wire()** attribute.

## 6.5.2.15 verilog_strobe()

### Purpose

Specify the sampling event for a Verilog object

### Category

Verilog port attribute

### Syntax

***exp*.verilog_strobe() == *event***

Syntax example:

```
keep u.p.verilog_strobe() == "@posedge clk1";
```

### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *event* | A string specifying any legal Verilog timing control. |

### Description

This string attribute specifies the sampling event for the Verilog signal that is bound to an external input port. This attribute is equivalent to the **verilog variable ... using strobe** declaration.

### Notes

- For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.
- Verisity adapters require that you also use **declared_range()** if the object that is driven is multi-bit.

### 6.5.2.16 verilog_wire()

#### Purpose

Create a single driver for a port (or multiple ports)

#### Category

Verilog port attribute

#### Syntax

***exp*.verilog_wire() == *bool***

Syntax example:

```
keep u.p.verilog_wire() == TRUE;
```

#### Parameters

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *bool* | Either TRUE or FALSE. The default is FALSE. |

#### Description

This boolean attribute allows an external out port to be bound to a Verilog wire, in a manner similar to a **verilog variable using wire** declaration.

The main difference between this attribute and the **driver()** attribute is that, being backward compatible, the **verilog_wire()** attribute merges all of the ports that have this attribute into a single Verilog driver, while the **driver()** attribute creates a separate driver for each port.

#### Notes

- For Verisity adapters, if you add or change this attribute, you must rewrite the stubs file.
- Verisity adapters require that you also use **declared_range()** if the object that is driven is multi-bit.

### 6.5.2.17 vhdl_delay_mode()

#### Purpose

Specify whether short pulses are propagated through driver

#### Category

HDL port attribute

#### Syntax

***exp*.vhdl_delay_mode() == *mode-option***

Syntax example:

```
keep u.p.vhdl_delay_mode() == INERTIAL;
```

**Parameters**

| | |
|---|---|
| *exp* | An expression of a simple port type. |
| *mode-option* | Either TRANSPORT (the default) or INERTIAL. |

**Description**

This **sn_vhdl_delay_mode** type attribute applies a VHDL delay mode value to an external out port. This attribute specifies whether pulses whose period is shorter than the delay specified by the **driver_delay()** attribute are propagated through the driver. INERTIAL specifies that such pulses are not propagated. TRANSPORT specifies that all pulses, regardless of length, are propagated.

This attribute also influences what happens if another driver (either VHDL or another unit) schedules a signal change and before that change occurs, this driver schedules a different change. With INERTIAL, the first change never occurs.

This attribute is silently ignored unless the **driver_delay()** attribute is also specified.

#### ~~6.5.2.18 vhdl_disconnect_value()~~

**~~Purpose~~**

~~Specify value to apply on Specman Elite restore~~

**~~Category~~**

~~HDL port attribute~~

**~~Syntax~~**

~~***exp*.vhdl_disconnect_value() == *mvl-value-list***~~

~~Syntax example:~~

```
~~keep u.p.vhdl_disconnect_value() == {MVL_Z};~~
```

**~~Parameters~~**

| | |
|---|---|
| *exp* | An expression that returns a port instance. |
| *mvl-value-list* | A list of one or more of the following: MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N. |

**~~Description~~**

~~This **mvl** type attribute applies an **mvl** value to an external output port when you restore Specman Elite after issuing a **test** command but do not restart the simulator. This value should be set to a value that does not affect the overall value of the resolved signal. In Verisity's ModelSim VHDL adapter the default value for std_logic signals is MVL_Z.~~

~~This attribute is silently ignored unless the **driver()** attribute or the **vhdl_driver()** attribute is set to TRUE.~~

### 6.5.3 Using Port Values and Attributes in Constraints

Like units, port instances can be created only during pre-run generation. They cannot be created with **new**, nor generated at runtime. Consequently, a port value cannot be initialized or sampled in pre-run generation constraints. Port values can be used in on-the-fly generation constraints in accordance with the basic constraint principles, such as the bidirectional nature of constraints. See Example 1 on page 231.

Another methodological requirement is that you must explicitly specify attribute values in hard constraints if the attributes are used anywhere in bidirectional constraints, including implication constraints. See Example 2 on page 231.

#### Example 1

This example shows the correct way to initialize an out port.

```
<'
extend sys {
    inport: in simple_port of int is instance;
    keep bind(inport, external);
    outport: inout simple_port of int is instance;
    keep bind(outport, external);
    !startval: int;

    run() is also {
        gen startval;
        outport$ = startval;  // Use port$ to set a value
    };
};
'>
```

Trying to constrain the generation of startval to equal the value of the out port does not work because outport$ in this context samples the port value, but does not affect it:

```
gen startval keeping { outport$ == startval}; // does not work
```

#### Example 2

This example shows how using port attribute values in bidirectional constraints can have undesired effects.

```
<'
extend sys {
    pclk: buffer_port of packet is instance;
    keep synthesized() == FALSE => pclk.pass_by_pointer() == TRUE;
};
'>
```

The implication constraint above requires the following constraint to be set in every specific non-synthesized test, instead of relying on the default value:

```
extend sys {
    keep synthesized() == FALSE;
};
```

Adding a constraint such as

```
keep pclk.pass_by_pointer()==FALSE
```

silently sets **synthesized()** to TRUE.

### 6.5.4 Buffer Port Methods

The methods in this section are used to read from or write to buffer ports and to check whether a buffer port queue is empty or full. The methods are:

— "get()" on page 232
— "put()" on page 233
— "is_empty()" on page 234
— "is_full()" on page 235

#### 6.5.4.1 get()

##### Purpose

Read and remove data from an input buffer port queue

##### Category

Predefined TCM for buffer ports

##### Syntax

***in-port-instance-name*.get()**: port element type

Syntax example:

```
rec_cell = in_port.get();
```

##### Description

Reads a data item from the buffer port queue and removes the item from the queue.

Since buffer ports use a FIFO queue, **get()** returns the first item that was written to the port.

The thread blocks upon **get()** when there are no more items in the queue.

If the queue is empty, or if it has a buffer size of 0 and no **put()** has been done on the port since the last **get()**, then the **get()** is blocked until a **put()** is done on the port.

The number of consecutive **get()** actions that is possible is limited to the number of items inserted by **put()**.

##### Example

```
unit consumer {
    cell_in: in buffer_port of atm_cell is instance;
    current_cell: atm_cell;
    update_cell() @clk$ is {
        current_cell = cell_in.get();
    };
};
```

### See Also

— "put()" on page 233
— "is_empty()" on page 234
— "is_full()" on page 235

### 6.5.4.2 put()

#### Purpose

Write data to an output buffer port queue

#### Category

Predefined TCM for buffer ports

#### Syntax

***out-port-instance-name*****.put(*****data***: port-element-type**)**

Syntax example:

```
out_port.put(trans_cell);
```

#### Parameters

| | |
|---|---|
| data | A data item of the port element type. |

#### Description

Writes a data item to the output buffer port queue. The sampling event of this TCM is **sys.any**.

The new data item is placed in a FIFO queue in the output buffer port.

If the queue is full, or if it has a buffer size of 0 and no **get()** has been done on the port since the last **put()**, then the **put()** is blocked until a **get()** is done on the port.

The number of consecutive **put()** actions that is possible is limited to the buffer size.

The thread blocks upon **put()** when there is no more room in the queue, that is, when the number of consequent **put()** operations exceeds the **buffer_size()** of the port instance.

#### Example

```
unit producer {
    clk: in event_port is instance;
    cell_out: out buffer_port of atm_cell is instance;
    write_cell_list(atm_cells: list of atm_cell) @clk$ is {
        for each in atm_cells do {
            cell_out.put(it);
        };
    };
};
```

### See Also

- — "get()" on page 232
- — "is_empty()" on page 234
- — "is_full()" on page 235

### 6.5.4.3 is_empty()

#### Purpose

Check if an input buffer port queue is empty

#### Category

Pseudo-method for buffer ports

#### Syntax

***in-port-instance-name*.is_empty()**: bool

Syntax example:

```
var readable: bool;
readable = not cell_in.is_empty();
```

#### Description

Returns TRUE if the input port queue is empty.

Returns FALSE if the input port queue is not empty.

#### Example

```
unit consumer {
    cell_in: in buffer_port of atm_cell is instance;
    clk: in event_port is instance;
    check_and_read(atm_cell): atm_cell @clk$ is {
        if cell_in.is_empty() {
            // No data is available - avoid blocking:
            dut_error("No atm cell is available");
        }
        else {
        // Read data from the port:
            return cell_in.get();
        };
    };
};
```

#### See Also

- — "get()" on page 232
- — "put()" on page 233
- — "is_full()" on page 235

### 6.5.4.4 is_full()

#### Purpose

Check if an output buffer port queue is full

#### Category

Pseudo-method for buffer ports

#### Syntax

***out-port-instance-name*.is_full()**: bool

Syntax example:

```
var overflow: bool;
overflow = cell_out.is_full();
```

#### Description

Returns TRUE if the output port queue is full.

Returns FALSE if the output port queue is not full.

#### Example

```
unit producer {
    cell_out: out buffer_port of atm_cell is instance;
    clk: in event_port is instance;
    check_and_write(cell: atm_cell)@clk$ is {
        if cell_out.is_full() {
            // Cannot write to the port without being blocked
            dut_error("Overflow in atm cells queue");
        }
        else {
           // Write data to the port
          cell_out.put(cell);
       };
    };
};
```

#### See Also

— “get()” on page 232
— “put()” on page 233
— “is_empty()” on page 234

### 6.5.5 Multi-Value Logic (MVL) Methods for Simple Ports

The predefined port methods in this section are for reading and writing MVL data between ports, to facilitate communication with objects where MVL values occur.

These methods operate on data of type **mvl**, which is defined as follows:

type mvl: [MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N]

The enumeration literals are the same as those of VHDL, except for MVL_N, which corresponds to the VHDL ‘-’ (“don’t care”) literal.

NOTE— Mixed access—accessing a port with MVL methods and accessing it through the **$** operator—is allowed.

The MVL methods are applicable in accordance to the port direction. Methods that write a value to a port are accessible for out and inout simple ports, while methods that read a value from a port are accessible for in and inout simple ports.

The predefined methods for simple ports are:

— “put_mvl()” on page 236
— “get_mvl()” on page 237
— “put_mvl_list()” on page 238
— “get_mvl_list()” on page 239
— “put_mvl_string()” on page 240
— “get_mvl_string()” on page 241
— “get_mvl4()” on page 241
— “get_mvl4_list()” on page 242
— “get_mvl4_string()” on page 243

### 6.5.5.1 put_mvl()

#### Purpose

Put an **mvl** data on a port of a non-**mvl** type

#### Category

Predefined method for simple ports

#### Syntax

***exp*.put_mvl(*value*:** mvl**)**

Syntax example:

```
p.put_mvl(MVL_Z)
```

#### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |
| *value* | A multi-value logic value. |

#### Description

Place an **mvl** value on an output or inout simple port, to initialize an object to a “disconnected” value, for example.

Placing an **mvl** value on a port whose element type is list places the value in the LSB of the list.

### Example

```
unit uo {
    pbo: out simple_port of bit is instance;
    keep bind(pbo, external);
    disconnect_pbo() is {
        pbo.put_mvl(MVL_Z);
    };
};
```

### See Also

— "get_mvl()" on page 237
— "put_mvl_list()" on page 238
— "put_mvl_string()" on page 240
— "string_to_mvl()" on page 247
— "int_to_mvl()" on page 250
— "bits_to_mvl()" on page 252
— "mvl_to_mvl4()" on page 252

### 6.5.5.2 get_mvl()

### Purpose

Read **mvl** data from a port of a non-**mvl** type

### Category

Predefined method for simple ports

### Syntax

***exp*.get_mvl()**: mvl

Syntax example:

```
check that pbi.get_mvl() != MVL_X else dut_error("Bad value");
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |

### Description

Reads an **mvl** value from an input or inout simple port, to check that there are no undefined "x" bits, for example.

Getting an **mvl** value from a port whose element type is list reads the LSB of the list.

### Example

```
unit ui {
    pbi: in simple_port of bit is instance;
    keep bind(pbi, external);
    chk_pbi() is {
        check that pbi.get_mvl() != MVL_X else
```

```
            dut_error("Bad value");
        };
    };
```

### See Also

- — "put_mvl()" on page 236
- — "get_mvl_list()" on page 239
- — "get_mvl_string()" on page 241
- — "get_mvl4()" on page 241
- — "mvl_to_string()" on page 248
- — "mvl_to_int()" on page 249
- — "mvl_to_bits()" on page 251
- — "mvl_to_mvl4()" on page 252

## 6.5.5.3 put_mvl_list()

### Purpose

Put a list of **mvl** values on a port of a non-**mvl** type

### Category

Predefined method for simple ports

### Syntax

***exp*.put_mvl_list(*values***: list of mvl**)**

Syntax example:

```
pbo.put_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0});
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |
| *values* | A list of **mvl** values |

### Description

Writes a list of **mvl** values to an output or inout simple port.

Putting a list of **mvl** values on a port whose element type is a single bit writes only the LSB of the list.

### Example

```
unit ui {
    pbi: in simple_port of uint(bits:4) is instance;
};

unit uo {
    uin: ui is instance;
    pbo: out simple_port of uint(bits:4) is instance;
    keep bind(pbo, uin.pbi);
    wr_pbo() is {
        pbo.put_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0});
    };
};
```

### See Also

— "put_mvl()" on page 236
— "get_mvl()" on page 237
— "get_mvl_list()" on page 239

## 6.5.5.4 get_mvl_list()

### Purpose

Get a list of **mvl** values from a port of a non-**mvl** type

### Category

Predefined method for simple ports

### Syntax

***exp*.get_mvl_list()**: list of mvl

Syntax example:

```
    check that pbil.get_mvl_list().has(it == MVL_U) == FALSE else
      dut_error("Bad list");
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |

### Description

Reads a list of **mvl** values from an input or inout simple port.

### Example

```
    unit uo {
        pbol: out simple_port of list of bit is instance;
    };

    unit ui {
        uout: uo is instance;
        pbil: in simple_port of list of bit is instance;
        keep bind(uout.pbol, pbil);
        chk_pbil() is {
            check that pbil.get_mvl_list().has(it == MVL_U) == FALSE else
               dut_error("Bad list");
        };
    };
```

### See Also

— "put_mvl()" on page 236

— "get_mvl()" on page 237
— "put_mvl_list()" on page 238
— "get_mvl4_list()" on page 242

### 6.5.5.5 put_mvl_string()

#### Purpose

Put an **mvl** value on a port of a non-**mvl** type when a value is represented as a string

#### Category

Predefined method for simple ports

#### Syntax

***exp*.put_mvl_string(*value*:** string**)**

Syntax example:

```
pbol.put_mvl_string("32'hxxxxllll");
```

#### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |
| *value* | An **mvl** value in the form of a base and one or more characters, entered as a string. The **mvl** values in the string must be lowercase. Use 1 for MVL_1, 0 for MVL_0, z for MVL_Z, and so on. |

#### Description

Writes a string representing a list of **mvl** values to a simple output or inout port. The **mvl** value consists of any legal base, for example, 32'b, followed by one or more characters, for example xxxxzzzz. The string representation follows the same rules as Verilog literals. The difference is that Verilog literals support only 4-value logic digits (1,0,x and z) while ***e*** allows also the characters u, l, h, w and n.

#### Example

```
unit uo {
    pbol: out simple_port of uint(bits:4) is instance;
    keep bind(pbol, external);
    wr_pbol() is {
        pbol.put_mvl_string("32'hxxxxllll");
    };
};
```

#### See Also

— "put_mvl()" on page 236
— "get_mvl()" on page 237
— "get_mvl_string()" on page 241

### 6.5.5.6 get_mvl_string()

#### Purpose

Get a value in form of a string from a port of a non-**mvl** type

#### Category

Predefined method for simple ports

#### Syntax

***exp*.get_mvl_string(*radix***: radix**)**: string

Syntax example:

```
print pbis.get_mvl_string(BIN);
```

#### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |
| radix | One of BIN, OCT, or HEX. |

#### Description

Returns a string in which each character represents an **mvl** value. The characters are lowercase. HDL value '1' is represented by the character 1, 'Z' by z, '-' by character n. The returned string always includes all the bits, with no implicit extensions. For example, a port of type uint returns a string of 32 characters, since an int is a 32-bit data type.

#### Example

```
unit ui {
    pbis: in simple_port of uint(bits:4) is instance;
    keep bind(pbis,external);
    chk_pbis() is {
        print pbis.get_mvl_string(BIN);
    };
};
```

#### See Also

— "put_mvl()" on page 236
— "get_mvl()" on page 237
— "put_mvl_string()" on page 240
— "get_mvl4_string()" on page 243

### 6.5.5.7 get_mvl4()

#### Purpose

Get an **mvl** value from a port, converting 9-value logic to 4-value logic

### Category

Predefined method for simple ports

### Syntax

***exp*.get_mvl4()**: mvl

Syntax example:

```
check that pbi.get_mvl4() != MVL_Z else dut_error("Bad value");
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |

### Description

Reads a 9-value **mvl** value from an input simple port and converts it to 4-value subset **mvl**.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z
```

### Example

```
unit ui {
    pbi: in simple_port of bit is instance;
    keep bind(pbi, external);
    chk_pbi() is {
        check that pbi.get_mvl4() != MVL_X else
           dut_error("Bad value");
    };
};
```

### See Also

— "put_mvl()" on page 236
— "get_mvl()" on page 237
— "get_mvl4_list()" on page 242
— "get_mvl4_string()" on page 243
— "mvl_to_mvl4()" on page 252

## 6.5.5.8 get_mvl4_list()

### Purpose

Get a list of **mvl** values from a port, converting from 9-value logic to 4-value logic

### Category

Predefined method for simple ports

### Syntax

***exp*.get_mvl4_list()**: list of mvl

Syntax example:

```
check that pbi4l.get_mvl4_list().has(it == MVL_X) == FALSE else
  dut_error("Bad list");
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |

### Description

Reads a list of 9-value **mvl** values from an input simple port and converts them to 4-value MVL.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z
```

### Example

```
unit ui {
    pbi4l: in simple_port of list of bit is instance;
    keep bind(pbi4l, external);
    chk_pbi4l() is {
        check that pbi4l.get_mvl4_list().has(it == MVL_X) == FALSE else
           dut_error("Bad list");
    };
};
```

### See Also

— "put_mvl()" on page 236
— "get_mvl()" on page 237
— "get_mvl4()" on page 241
— "mvl_list_to_mvl4_list()" on page 253

## 6.5.5.9 get_mvl4_string()

### Purpose

Get a 4-state value in form of a string from a port of a non-**mvl** type

### Category

Predefined method for simple ports

### Syntax

***exp*.get_mvl4_string(*radix*)**: string

Syntax example:

```
print pbi4s.get_mvl4_string(BIN);
```

### Parameters

| | |
|---|---|
| exp | An expression that returns a simple port instance. |
| radix | One of BIN, OCT, or HEX. |

### Description

Reads a string in which each character represents a 4-value logic digit from a subset of **mvl**, converted from 9-value logic. The characters are lowercase.

The predefined mapping from 9-value logic to 4-value logic is the same as it is commonly used when converting from VHDL std_logic to Verilog:

```
U, W, X, N -> x
L, 0 -> 0
H, 1 -> 1
Z -> z
```

The returned string always includes all the bits, with no implicit extensions. For example, a port of type int returns a string of 32 characters, since an int is a 32-bit data type.

### Example

```
unit ui {
    pbi4s: in simple_port of list of int(bits:4) is instance;
    keep bind(pbi4s,external);
    chk_pbi4s() is {
        print pbi4s.get_mvl4_string(HEX);
    };
};
```

### See Also

— “put_mvl()” on page 236
— “get_mvl()” on page 237
— “get_mvl4()” on page 241
— “string_to_mvl4()” on page 254

## 6.5.6 Methods for Simple Ports

These methods are defined for all simple ports, regardless of the type of data element:

— “has_x()” on page 245
— “has_z()” on page 245
— “has_unknown()” on page 246

### 6.5.6.1 has_x()

#### Purpose

Determine if port has X

#### Category

Predefined method for simple ports

#### Syntax

***exp*.has_x()**: bool

Syntax example:

```
print pbi4s.has_x();
```

#### Parameters

| | |
|---|---|
| exp | An expression of a simple port type. |

#### Description

Returns TRUE if at least one bit of the port is MVL_X.

#### Example

```
unit ui {
    pbi4s: in simple_port of uint(bits:4) is instance;
    keep bind(pbi4s,external);
    chk_pbi4s() is {
        print pbi4s.has_x();
    };
};
```

#### See Also

- "has_z()" on page 245
- "has_unknown()" on page 246

### 6.5.6.2 has_z()

#### Purpose

Determine if port has Z

#### Category

Predefined method for simple ports

#### Syntax

***exp*.has_z()**: bool

Syntax example:

```
print pbi4s.has_z();
```

### Parameters

| | |
|---|---|
| exp | An expression of a simple port type. |

### Description

Returns TRUE if at least one bit of the port is MVL_Z.

### Example

```
unit ui {
    pbi4s: in simple_port of uint(bits:4) is instance;
    keep bind(pbi4s,external);
    chk_pbi4s() is {
        print pbi4s.has_z();
    };
};
```

### See Also

— “has_x()” on page 245
— “has_unknown()” on page 246

### 6.5.6.3 has_unknown()

### Purpose

Determine if port has U

### Category

Predefined method for simple ports

### Syntax

***exp*.has_unknown()**: bool

Syntax example:

```
print pbi4s.has_unknown();
```

### Parameters

| | |
|---|---|
| exp | An expression of a simple port type. |

### Description

Returns TRUE if at least one bit of the port is one of the following:

- MVL_U

- MVL_X
- MVL_Z
- MVL_W
- MVL_N

### Example

```
unit ui {
    pbi4s: in simple_port of uint(bits:4) is instance;
    keep bind(pbi4s,external);
    chk_pbi4s() is {
        print pbi4s.has_unknown();
    };
};
```

### See Also

— “has_x()” on page 245
— “has_z()” on page 245

## 6.5.7 Global MVL Routines

The global routines for manipulating MVL values are:

— “string_to_mvl()” on page 247
— “mvl_to_string()” on page 248
— “mvl_to_int()” on page 249
— “int_to_mvl()” on page 250
— “mvl_to_bits()” on page 251
— “bits_to_mvl()” on page 252
— “mvl_to_mvl4()” on page 252
— “mvl_list_to_mvl4_list()” on page 253
— “string_to_mvl4()” on page 254

### 6.5.7.1 string_to_mvl()

### Purpose

Convert a string to a list of **mvl** values

### Category

Predefined routine

### Syntax

**string_to_mvl(*value-string***: string**)**: list of mvl

Syntax example:

```
mlist = string_to_mvl("8'bxz1");
```

### Parameters

| | |
|---|---|
| value-string | A string representing **mvl** values, consisting of a width and base followed by a series of characters corresponding to **mvl** values. Format of the input string is the same as in Verilog literals, except there are additional 9-value logic digits: u, l, h, w and n. |

### Description

Converts each character in the input string to an **mvl** value.

### Example

```
var mlist: list of mvl;
mlist = string_to_mvl("8'bz");
// returns {MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z};
mlist = string_to_mvl("8'bxz1");
// returns {MVL_1; MVL_Z; MVL_X; MVL_X; MVL_X; MVL_X; MVL_X; MVL_X};
```

### See Also

— "mvl_to_string()" on page 248

### 6.5.7.2 mvl_to_string()

### Purpose

Convert a list of **mvl** values to a string

### Category

Predefined routine

### Syntax

**mvl_to_string(*mvl-list*:** list of mvl, ***radix***: radix**)**: string

Syntax example:

```
mstring = mvl_to_string({MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_X; MVL_X; MVL_X;
    MVL_X}, BIN);
```

### Parameters

| | |
|---|---|
| mvl-list | A list of **mvl** values. |
| radix | One of BIN, OCT, or HEX. |

### Description

Converts a list of **mvl** values to a string. The mapping is done in the following way:

```
MVL_U is converted to character "u" (lowercase)
MVL_X - "x"
MVL_0 - "0"
MVL_1 - "1"
```

```
MVL_Z - "z"
MVL_W - "w";
MVL_L - "l"
MVL_H - "h"
MVL_N - "n"
```

NOTE— This routine always returns a sized number as a string.

### Example 1

```
var mstring: string;
mstring = mvl_to_string({MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_X; MVL_X; MVL_X;
    MVL_X}, BIN);
// returns "8b'zzzzxxxx"
```

### Example 2

```
var l: list of mvl = {MVL_1;MVL_0};
print mvl_to_string(l, BIN); --prints 2'b10
print mvl_to_string(l, HEX); --prints 2'h2
```

### See Also

— “string_to_mvl()” on page 247
— “mvl_to_int()” on page 249
— “mvl_to_bits()” on page 251
— “mvl_to_mvl4()” on page 252
— “mvl_list_to_mvl4_list()” on page 253

### 6.5.7.3 mvl_to_int()

### Purpose

Convert an **mvl** value to an integer

### Category

Predefined routine

### Syntax

**mvl_to_int(*mvl-list***: list of mvl, ***mask***: list of mvl**)**: uint

Syntax example:

```
var ma: uint = mvl_to_int(l, {MVL_X});
```

### Parameters

| | |
|---|---|
| mvl-list | A list of **mvl** values to convert to an integer value. |
| mask | A list of **mvl** values that are to be converted to 1. |

### Description

Converts each value in a list of **mvl** values into a binary integer (1 or 0), using a list of **mvl** mask values to determine which **mvl** values are converted to 1.

When the list is less than 32 bits, it is padded with 0. When it is greater than 32 bits, it is truncated, leaving the 32 least significant bits.

### Example

```
var l: list of mvl = {MVL_X; MVL_X; MVL_0; MVL_1};
var ma: uint = mvl_to_int(l,{MVL_X});
// returns 12 (0b1100)
var mb: uint = mvl_to_int(l, {MVL_Z})
// returns 0
```

### See Also

— “int_to_mvl()” on page 250
— “mvl_to_bits()” on page 251
— “mvl_to_mvl4()” on page 252
— “mvl_list_to_mvl4_list()” on page 253

## 6.5.7.4 int_to_mvl()

### Purpose

Convert an integer value to a list of **mvl** values

### Category

Predefined routine

### Syntax

**int_to_mvl(*value*:** uint, ***mask***: mvl**)**: list of mvl

Syntax example:

```
var mlist: list of mvl = int_to_mvl(12, MVL_X)
```

### Parameters

| | |
|---|---|
| value | An integer value to convert to a list of **mvl** values. |
| mask | An **mvl** value that replaces each bit in the integer that has the value 1. |

### Description

Maps each bit that has the value 1 to the mask **mvl** value, retains the 0 bits as MVL_0, and returns a list of 32 **mvl** values. The returned list always has a size of 32.

### Example

```
var mlist: list of mvl = int_to_mvl(12, MVL_X)
// returns MVL_0;.........MVL_X;MVL_X;MVL_0;MVL_0
```

### See Also

— “mvl_to_int()” on page 249

## 6.5.7.5 mvl_to_bits()

### Purpose

Convert a list of **mvl** values to a list of bits

### Category

Predefined routine

### Syntax

**mvl_to_bits(*mvl-list***: list of mvl, ***mask***: list of mvl**)**: list of bit

Syntax example:

```
var bl: list of bit = mvl_to_bits({MVL_Z; MVL_Z; MVL_X; MVL_L}, {MVL_Z; MVL_X})
```

### Parameters

| | |
|---|---|
| mvl-list | A list of **mvl** values to convert to bits. |
| mask | A list of **mvl** values that specifies which **mvl** values are to be converted to 1. |

### Description

Converts a list of **mvl** values to a list of bits, using a mask of **mvl** values to indicate which **mvl** values are converted to 1 in the list of bits.

### Example

```
var bl: list of bit = mvl_to_bits({MVL_Z; MVL_Z; MVL_X; MVL_L}, {MVL_Z; MVL_X})
// returns {1; 1; 1; 0}
```

### See Also

- — "mvl_to_string()" on page 248
- — "mvl_to_int()" on page 249
- — "mvl_to_mvl4()" on page 252

### 6.5.7.6 bits_to_mvl()

#### Purpose

Convert a list of bits to a list of **mvl** values

#### Category

Predefined routine

#### Syntax

**bits_to_mvl(*bit-list*:** list of bit, ***mask***: mvl**)**: list of mvl

Syntax example:

```
var ml: list of mvl = bits_to_mvl({1; 0; 1; 0}, MVL_Z)
```

#### Parameters

| | |
|---|---|
| bit-list | A list of bits to convert to **mvl** values. |
| mask | An **mvl** value that replaces each bit in the list that has the value 1. |

#### Description

Maps each bit with the value 1 to the mask **mvl** value, retains the 0 bits as MVL_0, and returns an **mvl** list that is ***bit-list*** size.

#### Example

```
var ml: list of mvl = bits_to_mvl({1; 0; 1; 0}, MVL_Z)
// returns {MVL_Z;MVL_0;MVL_Z;MVL_0}
```

#### See Also

- — "mvl_to_bits()" on page 251

### 6.5.7.7 mvl_to_mvl4()

#### Purpose

Convert an **mvl** value to a 4-value logic value

#### Category

Predefined routine

### Syntax

**mvl_to_mvl4(*value***: mvl**)**: mvl

Syntax example:

```
var m4: mvl = mvl_to_mvl4(MVL_U)
```

### Parameters

| | |
|---|---|
| value | An **mvl** value to convert to a 4-value logic value |

### Description

Converts an **mvl** value to the appropriate 4-value logic subset value.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L ,MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z
```

### Example

```
var m4: mvl = mvl_to_mvl4(MVL_U)
// returns MVL_X
```

### See Also

- "mvl_to_string()" on page 248
- "mvl_to_int()" on page 249
- "mvl_to_bits()" on page 251
- "mvl_list_to_mvl4_list()" on page 253

## 6.5.7.8 mvl_list_to_mvl4_list()

### Purpose

Convert a list of **mvl** values to a list of 4-value logic subset values

### Category

Predefined routine

### Syntax

**mvl_list_to_mvl4_list(*mvl-list***: list of mvl**)**: list of mvl

Syntax example:

```
var m4l: list of mvl = mvl_list_to_mvl4_list({MVL_N; MVL_L; MVL_H; MVL_1})
```

### Parameters

| | |
|---|---|
| mvl-list | A list of **mvl** values to convert to a list of 4-value logic subset values |

### Description

Converts each value in a list of **mvl** values to the corresponding 4-value logic value.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> MVL_0
MVL_H, MVL_1 -> MVL_1
MVL_Z -> MVL_Z
```

### Example

```
var m4l: list of mvl = mvl_list_to_mvl4_list({MVL_N; MVL_L; MVL_H; MVL_1})
// returns {MVL_X; MVL_0; MVL_1; MVL_1;}
```

### See Also

- Line 6.5.7.2
- "mvl_to_int()" on page 249
- "mvl_to_bits()" on page 251
- "mvl_to_mvl4()" on page 252

## 6.5.7.9 string_to_mvl4()

### Purpose

Convert a string to a list of 4-value logic **mvl** subset values

### Category

Predefined routine

### Syntax

**string_to_mvl4(*value-string*:** string**)**: list of mvl

Syntax example:

```
mlist = string_to_mvl("8'bxz");
```

### Parameters

| | |
|---|---|
| value-string | A string representing MVL values, consisting of a width and base followed by a series of characters corresponding to 9-value logic values. |

### Description

Converts each character in the string to the corresponding 4-value logic value. If the string contains characters other than '0', '1', 'x', 'z', 'h', 'l', 'u', 'w' or 'n' a runtime error is issued.

### Example

```
var mlist: list of mvl;
mlist = string_to_mvl4("8'bz");
// returns {MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_Z};
mlist = string_to_mvl("8'bxz");
// returns {MVL_Z; MVL_X; MVL_X; MVL_X; MVL_X; MVL_X; MVL_X; MVL_X};
```

### See Also

— “string_to_mvl()” on page 247
— “mvl_to_string()” on page 248
— “mvl_to_mvl4()” on page 252
— “mvl_list_to_mvl4_list()” on page 253

# 7 Generation Constraints

Test generation is the process that produces values for fields and variables (data items). Constraints are directives that influence the behavior of the test generator. They are declared within a struct and influence the generation of values for data items within the struct and its subtree. There are two basic types of constraints:

1) Value constraints restrict the range of possible values that the generator produces for data items, and they constrain the relationship between multiple items.
2) Order constraints influence the sequence in which data items are generated. Generation order is important because it affects the distribution of values and the success of generation.

Both value and order constraints can be hard or soft:

— Hard constraints (either value or order) must be met or an error is issued.
— Soft value constraints suggest default values but can be overridden by hard value constraints.
— Soft order constraints suggest modifications to the default generation order, but they can be overridden by dependencies between data items or by hard order constraints.

You can define constraints in many ways:

— By defining a range of legal values in the field or variable declaration
— By defining a list size in the list declaration
— By using one of the **keep** construct variations within a struct definition
— By using a **gen...keeping** action within a method

You can generate values for particular struct instances, fields, or variables during simulation (on-the-fly generation) with the **gen** action. You can also set values procedurally before or after generation within the **pre_generate()** or **post_generate()** methods.

This chapter contains the following sections:

## 7.1 Basic Concepts of Generation

The following introduce basic concepts related to constraints and generation.

### 7.1.1 Generation Order

The fields in a struct are generated one by one, starting with the first field defined and progressing through the fields in the order in which they appear in the ***e*** code. A struct item is always fully generated, including all substructs, before the next item is generated. Similarly, within a list, each item is fully generated, including all substructs, before the next item is generated. Lists are generated in ascending index order.

User-defined constraints can affect generation order. A constraint for a particular item might create a dependency that requires a field to be generated before some other fields. For example, the **keep** constraint shown below requires that the “kind” field be generated first and passed to the “get_size()” method in order to determine the value of “size”. Value constraints that induce a generation order are called unidirectional constraints.

```
type kind: [tx, rx];
struct packet {
    kind;
    size: byte;
    keep size == get_size(kind);
};
```

Generation order is important because it influences the distribution of values. For example, in the **keep** constraint shown below, if “kind” is generated first, “kind” is “tx” about 1/2 the time because there are only two legal values for “kind”:

```
struct packet {
    kind: [tx, rx];
    size: byte;
    keep size > 15 => kind == rx;
};
```

On the other hand, if “size” is generated first, there is only a 1 in 16 chance that “size” will be less than or equal to 15, so “kind” will be “tx” about 1/16 of the time.

## 7.1.2 Subtype Generation Optimization Constraints

In a subtype generation optimization constraint like the **keep gen_before_subtypes(**kind**)** constraint shown below, you specify a field that has at least one value that is used as a **when** determinant for creation of a subtype of the struct. In this case, the **when** determinant is the tx value of the kind field, since that is the value that determines when a subtype (that is, a tx packet) will be created.

```
type kind: [tx, rx];
struct packet {
    kind;
    offset: uint;
    keep gen_before_subtypes(kind);
    when tx packet {
        len: uint;
        keep size > 0 => offset == size - 1;
    };
    size: byte;
};
```

A subtype generation optimization constraint may change the order of generation by delaying analysis of constraints under the **when** until a **when** determinant value is actually generated. When no subtype generation optimization constraint is present, the generator analyzes all of the constraints and fields in the struct before it generates the struct, even fields and constraints that are defined under subtypes. When a subtype optimization constraint is present, then the generator initially analyzes and generates only the base type of the struct. It is not until it encounters a subtype optimization **when** determinant field that the generator analyzes the fields and constraints in the associated subtype, and then generates the subtype.

In the example below, the **keep gen** (size) **before** (offset) constraint might be ignored if, due to subtype optimization, the “offset” field is generated before the “kind” field is generated.

```
type kind: [tx, rx];
struct packet {
    kind;
    offset: uint;
    keep gen_before_subtypes(kind);
    when tx packet {
        len: uint;
        keep gen (size) before (offset);
    };
    size: byte;
};
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.1.3 Unidirectional Constraints

Value constraints that induce a generation order are called unidirectional constraints. For example, the **keep** constraint shown below requires that the “kind” field be generated first. The test generator cannot determine the value of the expression “get_size(kind)” without first generating the value of “kind”.

```
type kind: [tx, rx];
struct packet {
    kind;
    size: byte;
    keep size == get_size(kind);
};
```

Expressions like “get_size(kind)” are treated like constants within the context of a constraint boolean expression. That is, any parameters in these expressions are first generated, the operation is performed on the generated values, and the returned value can be used to constrain other generatable items in the constraint boolean expression. In the example above, the field “size” is constrained by return value of “get_size(kind)”.

Other expressions that are treated as constants within the context of a constraint boolean expression are:

| | |
|---|---|
| list slicing | lob[7..15] |
| bitwise operations | ~sigA, sigA \| sigB |
| most method calls | my_method(), b.as_a(int), value() |
| multiplication, division, and modulo operations | i.address % 2, 3*b, c/4 |
| in | cellA in cellList, cellListA in cellListB |

The only method calls that are not treated as constants are:

— my_list.size()

— my_list.is_all_iterations()
— my_list.is_a_permutation()

When “my_list” is a generatable list, these expressions are also generatable.

A unidirectional constraint can cause a runtime contradiction error if it selects a value for a parameter that turns out to conflict with a subsequent constraint. In the example below, the first constraint is unidirectional for b, c, and d because of the multiplication operator. Thus the generator first selects a value for b from the range [0-16], then selects a value for c from the range [0-4], and for d from the range [0-1]. Finally, the generator applies the constraint (a + 3*b + 12*c + 48*d) == 48. Most of the time this constraint results in a contradiction error because the values for three of the integers are selected before the constraint defining the required relationship between the integers is applied.

```
a: uint;
b: uint;
c: uint;
d: uint;

keep a + 3*b + 12*c + 48*d == 48; // Usually results in a contradiction error
keep a <= 48;
keep b <= 16;
keep c <= 4;
keep d <= 1;
```

In some cases you can rewrite the constraints to avoid the contradiction error. To avoid the contradiction illustrated above, for example, you need define the generation order so that the integers in the multiplication expressions (d, c, and b) are generated before a. You also must define each integer based only on constants and the values of the previously generated integers. Modifying the constraints in this manner avoids the contradiction error. To change the distribution of values (50% of the time d is 1 and a, b, and c are all 0), you can either add **keep soft select** constraints or you can constrain one of the other integers (a, b, or c) to a constant, and then constrain the others based only on constants and products of previously generated integers.

```
d: uint;
keep d <= 1;
c: uint;
keep c <= (48 - 48*d)/12;
b: uint;
keep b <= (48 - 48*d - 12*c)/3;
a: uint;
keep a + 3*b + 12*c + 48*d == 48;
```

Unidirectional constraints can also cause a constraint cycle, which results in a runtime contradiction error. A constraint cycle occurs when two or more unidirectional constraints impose conflicting requirements on the generation order. For example, the first constraint shown below requires that the “kind” field be generated first. The second constraint, however, requires that the “size” field be generated first.

```
type kind: [tx, rx];
struct packet {
    kind;
    size: byte;
    keep size == get_size(kind);
    keep kind == get_kind(size);    // Constraint cycle
};
```

### See Also

- “Basic Concepts of Generation” on page 257
- “Defining Constraints” on page 270
- “Invoking Generation” on page 296

## 7.1.4 Enforceable Expressions

An enforceable constraint boolean expression is an expression for which the test generator can choose a value that satisfies the constraint. Expressions that are not enforceable often involve non-generatable items or expressions that are treated as constants.

The following expressions are not enforceable:

- An expression that contains no generatable item
- An expression that restricts the legal values of an expression that is treated as a constant

### Example 1

For a compound constraint boolean expression that uses **and**, both subexpressions must be enforceable. The expression in this example is not enforceable because “sys.x” is not generatable. A runtime error is issued.

```
struct cons {
    y: int;
    z: int;

    keep sys.x > 100 and z < 100; // Not enforceable

};
extend sys {
    ci: cons;
    x: int;
};
```

### Example 2

For a compound constraint boolean expression that uses **or**, only one subexpression has to be enforceable. In this example, the second expression (z < 100) is enforceable, so no runtime error occurs. The first expression (sys.x > 100) is ignored because “sys.x” is not generatable.

```
struct cons {
    y: int;
    z: int;

    keep sys.x > 100 or z < 100;

};
extend sys {
    ci: cons;
    x: int;
};
```

#### Example 3

This expression is not enforceable because the test generator first generates “address”, then performs the modulo operation, and then cannot constrain the resulting value to zero.

```
    i.address % 2 == 0    // Not enforceable
```

#### Example 4

This expression is not enforceable because the test generator first generates “y”, then passes “y” to the **value()** method, and then cannot constrain the returned value to zero.

```
    value(y) == 0    // Not enforceable
```

#### Example 5

This expression is enforceable because the test generator first generates “y”, then passes “y” to the **value()** method, and then generates “x”.

```
    value(y) == x
```

#### Example 6

This expression is enforceable because the test generator first generates “y”, extracts the least significant bit of “y”, and then generates “x”.

```
    x == y[0:0]
```

#### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

### 7.1.5 Order of Evaluation of Soft Value Constraints

Soft value constraints on a data item are considered only at the time the data item is generated, after the hard value constraints on the data item are applied. Soft constraints are evaluated in reverse order of definition. If a soft constraint conflicts with the constraints that have already been applied, it is skipped.

NOTE— If a soft constraint does not contradict a hard constraint, it will be applied. If your intent is to over-ride a soft constraint with a hard constraint, use **reset_soft()**. See Example 2 on page 263.

#### Example 1

```
    keep x in [1..10];
    keep soft x > 3;
    keep soft x==8;
    keep soft x < 6;
```

The evaluation of the constraints is as follows:

1) The hard constraint is applied, so the range is [1..10].
2) The last soft constraint in the code order, x < 6, is considered. It does not conflict with the current range, so it is applied. The range is now [1..5].

3) The next to last soft constraint, x == 8, conflicts with the current range, so it is skipped. The range is still [1..5].
4) The first soft constraint in the code order, x > 3, does not conflict with the current range, so it is applied. The final range of legal values is [4, 5].

### Example 2

The constraint shown below sets the default value for num to the range [1..10].

```
<'
struct x {
    num: uint;

    keep soft num in [1..10];
};
'>
```

In order to override the default and change the range with a hard constraint to [10.20] for a particular test, you must also reset the soft constraint. Because there is one value (10) in the intersection of the soft and the hard constraint, both constraints are applied and num will always be 10. The example below shows how to override the soft constraint with **reset_soft()**.

```
<'
extend sys {
    xlist[10]: list of x;

    keep for each (n) in xlist {
        n.num.reset_soft();
        n.num in [10..20];
    };

    run() is also {
        print sys.xlist;
    };
};
'>
```

### See Also

- "keep gen-item.reset_soft()" on page 283
- "Basic Concepts of Generation" on page 257
- "Defining Constraints" on page 270
- "Invoking Generation" on page 296

## 7.1.6 Constraining Struct Instances

You can constrain two struct instances of the same type to have the same contents. The constraint causes the two struct instances to refer to the same memory location. As a result, changing one of the struct instances also changes the other struct instance immediately. For example, in the code below, when scons1.x is set to 5, the value of scons2.x also becomes 5.

```
struct scons {
    x: uint;
    blist: list of byte;
};
```

```
extend sys {
    scons1: scons;
    scons2: scons;
    keep scons2 == scons1;

    run() is also {
        scons1.x = 5;
    };
};
```

### 7.1.7 Constraining Lists

There are several ways that you can constrain a list or its elements. See the following sections for more information:

- "List Size" on page 264
- "List Item" on page 264
- "One List to Another List" on page 265
- "Multiple List Items" on page 265
- "List of Structs" on page 266
- "Multiple Lists" on page 266

#### 7.1.7.1 List Size

You can constrain the list size of a field either by using a size expression in a field declaration or by using a **keep** constraint. The following statements both constrain the number of elements in the "pacs" list to 10:

```
pacs[10]: list of pac;
keep pacs.size() == 10;
```

The key difference between these two methods is that the **keep** constraint affects only generation, whereas the field declaration also initializes the list automatically. Note, however, that if you use the field declaration approach and the size expression cannot be evaluated when **init()** is called, you will see an error. For example, if the size expression is ***struct-field.field*** and ***struct-field*** is NULL when init() is called, you get an error.

If you unpack data into a field declared as a list, it is better to use the size expression in a field declaration. That way, the list's size is always exactly as specified. See "Packing and Unpacking Lists" on page 503 for more information.

To constrain the list size of a variable, you must use the **keep** constraint. A size expression in a variable declaration is not allowed.

If there are no explicit constraints on the size of a list, the generated list will have a size between zero and the value of the configuration variable, **default_max_list_size**. This variable is set initially to 50.

#### 7.1.7.2 List Item

You can constrain an individual item in a list of scalar items using the **keep** constraint as follows.

```
keep me.data[0] == 0x9a;
```

You can constrain an individual item in a list of structs using the **keep** constraint as follows.

```
keep dstructs[0].data == 0xff;
```

NOTE— Neither multiple list indexing nor index expressions may be used in constraints. For example, top[0].dstruct[0].data is not legal, and dstruct[n+1].data is not legal.

### 7.1.7.3 Item in List

You can constrain a list to keep a specific item in the list. For example:

```
<'
extend sys {
    x: uint;
    keep x == 5;
    lu: list of uint;
    keep x in lu;
};
'>
```

This constraint is bidirectional, meaning that it does not imply a generation order for the item and list. However, the item is always at the last place in the list, regardless of which is generated first, the item or the list).

In this example, x is generated before lu and therefore the last item in lu is 5.

Therefore, the following code results in a contradiction:

```
<'
extend sys {
    x: uint;
    y: uint;
    lu: list of uint;
    keep x in lu; // last item in lu is x
    keep y in lu;  // last item in lu is y
    keep x != y;
};
'>
```

### 7.1.7.4 One List to Another List

You can constrain one list to contain the same items as another list, using the **keep** constraint.

```
data1: list of byte;
data2: list of byte;
keep data2 == data1;
```

This results in two references to two separate lists which initially contain the same values. Changing one of the lists does not affect the other list unless one list is assigned to the other, which results in the references to the two lists pointing to the same memory location.

### 7.1.7.5 Multiple List Items

You can constrain multiple items in a list, using the **keep for each** constraint.

```
keep for each in pacs {
    index == 0 => it.kind == control;
};
```

#### 7.1.7.6 List of Structs

You can constrain a list of structs to have all legal values of one or more fields, using the **.is_all_iterations()** method.

```
keep pacs.is_all_iterations(.kind);
```

#### 7.1.7.7 Multiple Lists

You can constrain a list to be a subset of another list, using the **in** construct. In this example, all the elements in the "pacs_sub" list are contained in the "pacs" list, but not necessarily in the same order. The "pacs" list can have elements that are not in "pacs_sub".

```
pacs_sub[10]: list of pac;
keep pacs_sub in pacs;
```

You can constrain a list to have the same elements as another list using the **is_a_permutation()** pseudo-method. In this example, the "pacs_dup" list and the "pacs" list have exactly the same elements, but not necessarily in the same order.

```
pacs_dup[10]: list of pac;
keep pacs_dup.is_a_permutation(pacs);
```

**See Also**

— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

### 7.1.8 Constraining Bit Slices

You can use the bit slice operator in constraints to achieve a variety of purposes. A simple example is using the bit slice operator to constrain the fields of a CPU instruction:

```
struct cpu_env {
    instr: uint (bits: 16);

    keep instr[15:13] == 0b100;
    keep instr[12:8] == 0b11001;
    keep instr[7:0] == 0b00001111;
};
```

Another simple but useful application of the bit slice constraint is to generate a list of even integers:

```
struct cpu_env {
    lint: list of int;

    keep for each in lint {
        it[0:0] == 0;
    };
};
```

NOTE— Using “it%2 == 0” to generate a list of even integers does not work. Since the “%” operator makes the constraint unidirectional, “it” is generated before the constraint is checked, and a contradiction occurs about 50% of the time.

You can also use a bit constraint to constrain particular bits in relation to each other. For example, the following constraint ensures that only one of the lower four bits of “x” is 1:

```
keep x[3:0] in [1,2,4,8];
```

You can use non-constant bit indices in bit slice constraints, as in the following example, which generates a 4-bit integer with 1s in two consecutive bits:

```
i: int [0..3];
j: int [0..3];
l: int (bits: 4);

keep j - i == 1;
keep l[j:i] == 0b11;
```

### See Also

— “Bit Slice Constraints and Generation Order” on page 267
— “Bit Slice Constraints and Signed Entities” on page 268
— “Bit Slice Constraints and Soft Constraints” on page 269
— “Limitations of Bit Slice Constraints” on page 269
— “Debugging Bit Slice Constraints” on page 269

## 7.1.8.1 Bit Slice Constraints and Generation Order

A generatable item can contain a bit slice reference; however, there are implications for generation order:

### Non-constant Bit Indices

Non-constant bit indices must be generated before other entities in the constraint. You cannot override this order.

For example, the following constraint

```
keep x[j:i] == y;
```

implies

```
keep gen (j, i) before (x, y);
```

NOTE— A further implication is that constraints like the following, where the bit indices are non-constant and the other items are constant, cannot be solved.

```
    keep 125[j:i] == 0b101;
```

### Generation of Bit Sliced Items

By default, bit sliced items are generated after other items in the same constraint. You can override this default with a **keep gen** constraint.

For example, the following constraint

```
keep x[j:i] == y;
```

implies

```
keep soft gen (y) before (x);
```

There can be cases where you need to override this default generation order with a **keep gen** constraint. For example, to meet the following constraints, “x” must be generated before “y”:

```
keep y == x[1:0];
keep x in [1,2,5,6];
```

In order to make this happen, you can add the constraint:

```
keep gen (x) before (y);
```

or you can add the value() routine to the existing constraint:

```
keep y == value(x[1:0])
```

### 7.1.8.2 Bit Slice Constraints and Signed Entities

Bit slices in ***e*** are treated as unsigned. It is possible, however, to constrain the value of a bit slice (or any unsigned entity) relative to a signed entity. In the example below, a bit slice of “x” is constrained by a signed entity, “y”:

```
x: int;
y: int (bits:5);
keep x[4:0] == y;
```

There are several implications of constraints that relate a bit slice to a signed entity:

— The value of the bit slice is treated as an unsigned integer; in other words, none of the bits in the slice is treated as a sign bit. In the example above, although “x” can be a negative number, x[4:0] is treated as a positive value.
— The value of the signed entity is generated as a non-negative. In the example above, “y” will always be generated as a non-negative integer.
— The value of both the bit slice and the signed entity must fit into the smaller of

- The bit width of the bit slice
- The bit width of the highest possible value of the signed entity (This width excludes one bit used to store the sign.)

#### Example

Given the following integers, “x” and “y”,

```
x: int;
y: int (bits:5);
```

any one of the following constraints requires the value of “y” to be a non-negative number no larger than four bits (the bit width of “y”, minus one bit to store the sign). In other words, the value of both “y” and the specified bit slice of “x” is generated in the range [0..15]. Any upper bits of the bit slice not required to store the value are set to 0:

```
keep x[7:0] == y;    // x[7:4] is 0
keep x[4:0] == y;    // x[4] is 0
keep x[3:0] == y;
```

By contrast, the value of “y” in the following constraint must fit into only three bits (the bit width of the bit slice), so “y” and “x[2:0]” are generated in the range [0..7]:

```
keep x[2:0] == y;
```

#### 7.1.8.3 Bit Slice Constraints and Soft Constraints

A hard constraint on a bit slice of a variable always overrides a soft constraint on that variable. For example, the intention of the following constraints is to make all the bits of a scalar be zero by default, then set individual bits with bit slice constraints:

```
keep soft x == 0;
keep x[7:7] == 1;    // Doesn't have desired effect
```

These constraints will not have the desired effect as the soft constraint will always be overridden. The only way to achieve this purpose is to apply the soft constraint to each individual bit explicitly:

```
keep soft x[0:0] == 0;
...
keep soft x[31:31] == 0;
keep x[7:7] == 1;
```

#### 7.1.8.4 Limitations of Bit Slice Constraints

If a bit slice is a function of another bit slice of the same field or variable, in many cases a contradiction occurs.

In the following example, “x” is an argument to the “bit_parity()” function and must be generated before the function is called:

```
keep x[8:8] == bit_parity(x[7:0]); // Usually a contradiction error
```

The result of the function call is then compared to “x[8:8]” and will fail in 50% of the cases.

The workaround is to assign a new virtual field for “x[7:0]”.

```
y: uint (bits:8);
keep y == x[7:0];
keep x[8:8] == bit_parity(y);
```

These constraints cause “y” to be generated first, “x[7:0]” to be constrained to have the value of “y” and “x[8:8]” to be constrained to have the return value from the bit_parity() method.

#### ~~7.1.8.5 Debugging Bit Slice Constraints~~

~~For bit slice constraints, the **collect gen** command displays the item’s range list (enclosed in square brackets) together with the item’s bit value representation (enclosed in angle brackets) as shown below:~~

~~`[range-list]: <bit-value-representation>`~~

~~The bit value representation has a single character, either 0, 1, or X, that represents each bit. The characters 0 and 1 indicate that a particular bit must be a 0 or a 1, respectively. The character X indicates that a bit can be either 0 or 1.~~

~~For example, the following display describes an 8-bit odd integer within the range 10 to 20 or 50 to 60:~~

```
~~[11..19, 51..59] : <00XXXXX1>~~
```

## 7.2 Defining Constraints

For information on the constructs used to define constraints, see:

— "keep" on page 270
— "keep all of {...}" on page 272
— "keep struct-list.is_all_iterations()" on page 274
— "keep for each" on page 275
— "keep soft" on page 278
— "keep soft... select" on page 279
— "keep gen-item.reset_soft()" on page 283
— "keep gen ... before" on page 284
— "keep soft gen ... before" on page 285
— "keep gen_before_subtypes()" on page 287
— "keep reset_gen _before_subtypes()" on page 289
— "value()" on page 290
— "constraint-bool-exp" on page 292
— "gen-item" on page 294

In addition, see the following for helpful information.

— "Constraining Lists" on page 264
— "Constraining Bit Slices" on page 266
— "Comparison Operators" on page 42

### 7.2.1 keep

#### Purpose

Define a hard value constraint

#### Category

Struct member

#### Syntax

**keep *constraint-bool-exp***

Syntax example:

```
keep kind != tx or len == 16;
```

### Parameters

| | |
|---|---|
| *constraint-bool-exp* | A simple or a compound boolean expression. See "constraint-bool-exp" on page 292 for a full description of this parameter. |

### Description

States restrictions on the values generated for fields in the struct or the struct subtree, or describes required relationships between field values and other items in the struct or its subtree.

Hard constraints are applied whenever the enclosing struct is generated. For any **keep** constraint in a generated struct, the generator either meets the constraint or issues a constraint contradiction message.

NOTE— If the **keep** constraint appears under a **when** construct, the constraint is considered only if the **when** condition is true.

### Example 1

This example describes a required relationship between two fields, "kind" and "len". If the current "pkt" is of kind "tx", then "len" must be 16.

```
struct pkt {
    kind: [tx, rx];
    len: uint;
    keep kind == tx => len == 16;
};
```

This constraint is translated internally into an **or** constraint:

```
keep kind != tx or len == 16;
```

### Example 2

This example shows a required relationship between two fields, "kind" and "len", using a local variable, "p", to represent "pckt" instances of kind "tx":

```
struct pckt {
    kind: [tx, rx];
    len: uint;
};
struct top {
    packet: pckt;
    keep packet is a tx pckt (p) => p.len in [128..255];
};
```

### Example 3

This example shows another way to describe the required relationship between the two fields, "kind" and "len". This constraint is also translated into an **or** constraint:

```
struct pkt {
    kind: [tx, rx];
    len: uint;
    when tx pkt {
        keep len == 16;
    };
};
```

### Example 4

This example shows how to call the ***list*.is_a_permutation()** method to constrain a list to have a random permutation of items from another list. In this example, “l_1” and “l_2” will have exactly the same elements. The elements will not necessarily appear in the same order.

```
    struct astr {
        l_1: list of int;
        l_2: list of int;
        keep l_2.is_a_permutation(l_1);
    };
```

### Example 5

This example shows a constraint on a single list item (“data[0]”) and the use of path names to identify the item to be constrained.

```
    type transaction_kind: [good, bad];
    struct transaction {
       kind: transaction_kind;
       address: uint;
       length: uint;
       data: list of byte;
    };

    extend transaction {
        keep length < 24;
        keep data[0] == 0x9a;
        keep address in [0x100..0x200];
        keep me.kind == good;
    };

    extend sys {
        t: transaction;
        keep me.t.length != 0;
    };
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.2.2 keep all of {...}

### Purpose

Define a constraint block

### Category

Struct member

### Syntax

**keep all of {*constraint-bool-exp*; ...}**

Syntax example:

```
keep all of {
    kind != tx;
    len == 16;
};
```

### Parameters

| | |
|---|---|
| *constraint-bool-exp* | A simple or a compound boolean expression. See "constraint-bool-exp" on page 292 for a full description of this parameter. |

### Description

A **keep** constraint block is exactly equivalent to a **keep** constraint for each constraint boolean expression in the block. For example, the following constraint block

```
keep all of {
    kind != tx;
    len == 16;
};
```

is exactly equivalent to

```
keep kind != tx;
keep len == 16;
```

The **all of** block can be used as a constraint boolean expression itself, as is shown in Example on page 273.

### Example

```
type transaction_kind: [VERSION1, VERSION2, VERSION3];
struct transaction {
    kind: transaction_kind;
    address: uint;
    length: uint;
    data: list of byte;

    keep kind in [VERSION1, VERSION2] => all of {
        length < 24;
        data[0] == 0x9a;
        address in [0x100..0x200];
    };
};
```

### See Also

— "Basic Concepts of Generation" on page 257

— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

### 7.2.3 keep struct-list.is_all_iterations()

#### Purpose

Cause a list of structs to have all iterations of a field

#### Category

Constraint-specific list method

#### Syntax

**keep *gen-item*.is_all_iterations(*.field-name*:** exp, ...**)**

Syntax example:

```
keep packets.is_all_iterations(.kind,.protocol);
```

#### Parameters

| | |
|---|---|
| *gen-item* | A generatable item of type list of struct. See "gen-item" on page 294 for more information. |
| *field-name* | The name of a scalar field of a struct. The field name must be prefixed by a period. The order of fields in this list does not affect the order in which they are iterated. The specified field that is defined first in the struct is the one that is iterated first. |

#### Description

Causes a list of structs to have all legal, non-contradicting iterations of the fields specified in the field list. Fields not included in the field list are not iterated; their values can be constrained by other relevant constraints. The highest value always occupies the last element in the list.

Soft constraints on fields specified in the field list are skipped. For example, given the following constraints, packet_list will have all legal iterations of the length field, not just iterations within 10 and 100:

```
keep soft len in [10..100];
keep packet_list.is_all_iterations(.len)
```

All other relevant hard constraints on the list and on the struct are applied. If these constraints reduce the ranges of some of the fields in the field list, then the generated list is also reduced.

#### Memory Usage and Performance Considerations

The number of iterations in a list produced by ***list*.is_all_iterations()** is the product of the number of possible values in each field in the list. For example, if you list all iterations of a struct with the following fields:

```
i: int [0..4]           // 5 possible values
j: int [0..3, 5..7]     // 7 possible values
k: int (bits: 8)        // 256 possible values
```

The number of iterations for the list is:

```
5 * 7 * 256 = 8960
```

The **absolute_max_list_size** generation configuration option sets the maximum number of iterations allowed in a list. The default is 524,288. If the number of iterations in your list exceeds this number, you can set **absolute_max_list_size** to a larger number with ~~the **config gen** command~~.

### Notes

- — The *list*.**is_all_iterations()** method can only be used in a constraint boolean expression.
- — The fields to be iterated must be of a scalar type, not a list or struct type.

### Example

The "**sys**.packets" list will have six elements (2 "kinds" * 3 "protocols"). The "len" field is not iterated on; it will get any value from its legal range for each of the list items.

```
type p_kind: [tx, rx];
type p_protocol: [atm, eth, other];
struct packet {
    kind: p_kind;
    protocol: p_protocol;
    len: int [0..4k];
};
extend sys {
    packets: list of packet;
    keep packets.is_all_iterations(.kind,.protocol);
};
```

### See Also

- — "Basic Concepts of Generation" on page 257
- — "Defining Constraints" on page 270
- — "Invoking Generation" on page 296

## 7.2.4 keep for each

### Purpose

Constrain list items

### Category

Struct member

### Syntax

**keep for each** [(***item-name***)] [**using** [**index** (***index-name***)] [**prev** (***prev-name***)]] **in**
***gen-item*** {***constraint-bool-exp*** | ***nested-for-each***; ...}

Syntax example:

```
keep for each (p) in pkl {
    soft p.protocol in [atm, eth];
};
```

### Parameters

| | |
|---|---|
| *item-name* | An optional name used as a local variable referring to the current item in the list. The default is **it**. |
| *index-name* | An optional name referring to index of the current item in the list. The default is **index**. |
| *prev-name* | An optional name referring to the previous item in the list. The default is **prev**. |
| *gen-item* | A generatable item of type list. See "gen-item" on page 294 for more information. |
| *constraint-bool-exp* | A simple or a compound boolean expression. See "constraint-bool-exp" on page 292 for a full description of this parameter. |
| *nested-for-each* | A nested **for each** block, with the same syntax as the enclosing **for each** block, except that "keep" is omitted. |

### Description

Defines a value constraint on multiple list items.

### Notes

- You must refer to the items you want to generate using a path name that starts either with **it**, such as "**it**.pk" or with the name that you assigned to the list item (***item-name***). Items whose pathname does not start with **it** can only be sampled; their generated values cannot be constrained.
- Within a **for each** constraint, **prev** and **index** are predefined constants and cannot be constrained or generated.
- Items in lists are generated in ascending order starting with index zero. Constraints that use an index expression to refer to other items in a list can only refer to items with lower index values.
- Referencing **prev** while in the first item of the list causes an error.
- You can nest **for each** constraints.
- If a **for each** constraint is contained in a **gen ... keeping** action, you must name the iterated variable. See Example 3 on page 298 for more information.

### Example 1

In this example, the "**keep for each in** dat" constraint in the "pstr" struct constrains all the "dat" fields to be less than 64. Note that referring to the list items in the boolean expression "**it** < 64" as "dat[index]" rather than "**it**" generates an error.

```
struct pstr {
    dat: list of uint;
    keep for each in dat {
        it < 64;
    };
};
```

### Example 2

The following example uses an item name "p" and an index name "pi" to constrain the generation of values for the variable "indx":

```
struct packet {
    indx: uint;
};
extend sys {
    packets: list of packet;
    keep for each (p) using index (pi) in packets {
        p.indx == pi;
    };
};
```

## Example 3

The following example shows the use of **index** in a nested **for each** block. The “x” field receives the value of the outer index and each byte of “payload” receives the value of the inner index.

```
struct packet {
    x: int;
    %payload: list of byte;
    keep payload.size() == 10;
};
extend sys {
    packets: list of packet;
    keep packets.size() == 5;
    keep for each (p) in packets {
        p.x == index;
        for each in p.payload {
            it == index;
        };
    };
    post_generate() is also {
        for i from 0 to 4 {
            print packets[i].x;
            print packets[i].payload;
        };
    };
};
```

## Result

```
Generating the test using seed 1...
  packets[i].x = 0
  packets[i].payload =  (10 items, dec):
                9   8    7   6   5   4    3   2   1   0       .0

  packets[i].x = 1
  packets[i].payload =  (10 items, dec):
                9   8    7   6   5   4    3   2   1   0       .0

  packets[i].x = 2
  packets[i].payload =  (10 items, dec):
                9   8    7   6   5   4    3   2   1   0       .0

  packets[i].x = 3
  packets[i].payload =  (10 items, dec):
                9   8    7   6   5   4    3   2   1   0       .0

  packets[i].x = 4
```

```
    packets[i].payload =  (10 items, dec):
                        9   8    7   6   5   4    3   2   1   0       .0
```

### See Also

— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

## 7.2.5 keep soft

### Purpose

Define a soft value constraint

### Category

Struct member

### Syntax

keep soft *constraint-bool-exp*

Syntax example:

```
keep soft legal == TRUE;
```

### Parameters

| | |
|---|---|
| *constraint-bool-exp* | A simple boolean expression. See "constraint-bool-exp" on page 292 for a full description of this parameter. |

### Description

Suggests default values for fields or variables in the struct or the struct subtree, or describes suggested relationships between field values and other items in the struct or its subtree.

Soft constraints are order dependent and will not be met if they conflict with hard constraints or soft constraints that have already been applied. See "Order of Evaluation of Soft Value Constraints" on page 262 for more information on this topic.

NOTE— The **soft** keyword can be used in simple boolean expressions, but not in compound boolean expressions. Thus the first constraint below is valid, but the second generates a compile-time error:

```
keep x > 0 => soft y < 0;
keep soft x > 0 => y < 0;    // Compile-time error
```

### Example 1

Because soft constraints only suggest default values, it is better not to use them to define architectural constraints, such as "keep opcode in [ADD, SUB, AND, XOR, RET, NOP]". If you want to be able to explicitly override the architectural constraints in order to generate illegal instructions for a particular test, then you can define a boolean field for legal instructions and place a soft constraint on that:

```
    struct instr {
        %opcode: cpu_opcode;
        legal: bool;
        keep soft legal == TRUE;
        keep legal => opcode in [ADD, SUB, AND, XOR, RET, NOP];
    };
```

### Example 2

Individual constraints inside a constraint block can be soft constraints.

```
    extend sys {
        packets: list of packet;
        keep for each in me.packets {
            soft .len == 2k;
            .kind != tx;
        };
    };
```

### See Also

— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

## 7.2.6 keep soft... select

### Purpose

Constrain distribution of values

### Category

Struct member

### Syntax

**keep soft *gen-item*==select {*weight*: *value*; ...}**

Syntax example:

```
    keep soft me.opcode == select {
        30: ADD;
        20: ADDI;
        10: [SUB, SUBI];
    };
```

## Parameters

| | |
|---|---|
| *gen-item* | A generatable item. See "gen-item" on page 294 for a full description of this parameter. |
| *weight* | Any **uint** expression. Weights are proportions; they do not have to add up to 100. A relatively higher weight indicates a greater probability that the value is chosen. |

***value*** is one of the following:

| | |
|---|---|
| *range-list* | A range list such as [2..7]. A select expression with a range list selects the portion of the current range that intersects with the specified range list. |
| *exp* | A constant expression. A select expression with a constant expression (usually a single number) selects that number, if it is part of the current range. |
| others | Selects the portions of the current range that do not intersect with other select expressions in this constraint.<br><br>Using a weight of 0 for **others** causes the constraint to be ignored. That is, the effect is the same as if the **others** option were not entered at all. |
| pass | Ignores this constraint and keeps the current range as is. |
| edges | Selects the values at the extreme ends of the current range(s). |
| min | Selects the minimum value of the ***gen-item***. |
| max | Selects the maximum value of the ***gen-item***. |

## Description

Specifies the relative probability that a particular value or set of values is chosen from the current range of legal values. The current range is the range of values as reduced by hard constraints and by soft constraints that have already been applied.

A weighted value will be assigned with the probability of

- weight/(sum of all weights)

Weights are treated as integers. If you use an expression for a weight, take care to avoid a situation where the value of the expression is larger than the maximum integer size (MAX_INT).

Like other soft constraints, **keep soft select** is order dependent and will not be met if it conflicts with hard constraints or soft constraints that have already been applied. See "Order of Evaluation of Soft Value Constraints" on page 262 for more information on this topic.

## Example 1

The following soft select constraint specifies that there is a 3/6 probability that ADD is selected from the current range, a 2/6 probability for ADDI, and a 1/6 probability that either SUB or SUBI is selected.

```
struct instr {
    %opcode: cpu_opcode;
    keep soft me.opcode == select {
        30: ADD;
        20: ADDI;
        10: [SUB, SUBI];
        };
    };
```

### Example 2

In the following example, “address” is generated in the range [0..49] 10% of the time, as exactly [50] 60% of the time, and in range [51..99] 30% of the time, assuming that the current range includes all these values.

```
    struct transaction {
        address: uint;
        keep soft address == select {
            10: [0..49];
            60: 50;
            30: [51..99];
        };
    };
```

### Example 3

This particular test uses the distribution described in the original definition of “transaction” only 10% of the time and uses the range [200..299] 90% of the time.

```
    extend transaction {
        keep soft address == select {
            10: pass;
            90: [200..299];
        };
    };
```

The final distribution is 90% [200..299], 1% [0..49], 6% [50], 3% [51..99].

### Example 4

This extension to “transaction” sets the current range with a hard constraint. 50% of the time the extreme edges of the range are selected (0, 50, 100, and 150). 50% of the time other values in the range are chosen.

```
    extend transaction {
        keep address in [0..50,100..150];
        keep soft address == select {
            50: edges;
            50: others;
        };
    };
```

### Example 5

This extension to “transaction” sets the current range with a hard constraint. About 10% of the values are to be 10 and about 30% of the values are to be 50. The remaining 60% of the values are to be distributed between 10 and 50.

```
    extend transaction {
        keep address in [10..50];
        keep soft address == select {
            10: min;
            60: others;
            30: max;
        };
    };
```

### Example 6

This example shows how to weight the values of generated elements of a list. The **it** variable is used to represent a list element in the **keep for each** construct. The values A, B, and C are given equal weights of 20, and all other possible values (D through L) are given a collective weight of 40. About 20% of the generated list elements will be A, 20% will be B, 20% will be C, and the remaining 40% will get random values in the range D through L.

```
    <'
    type alpha: [A, B, C, D, E, F, G, H, I, J, K, L];
    struct top {
        my_list[50]: list of alpha;

        keep for each in my_list {
            soft it == select  {
                20: A;
                20: B;
                20: C;
                40: others;
            };
        };
    };

    extend sys {
        top;
    };
    '>
```

### Example 7

This example shows how a runtime value from the simulation can be used to weight the selection of a value. In this case, the generation of the JMPC opcode is controlled by the value of the 'top.carry' signal.

```
    extend instr {
        keep soft opcode == select {
                          40: [ADD, ADDI, SUB, SUBI];
                          20: [AND, ANDI, XOR, XORI];
                          10: [JMP, CALL, RET, NOP];
            'top.carry' * 90: JMPC;
        };
    };
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

### 7.2.7 keep gen-item.reset_soft()

#### Purpose

Quit evaluation of soft constraints for a field

#### Category

Struct member

#### Syntax

**keep *gen-item*.reset_soft()**

Syntax example:

```
keep c.reset_soft();
```

#### Parameters

| | |
|---|---|
| *gen-item* | A generatable item. See "gen-item" on page 294 for a full description of this parameter. |

#### Description

Causes the program to quit the evaluation of soft value constraints for the specified field. Soft constraints for other fields are still evaluated..

#### Example 1

It is important to remember that soft constraints are considered in reverse order to the order in which they are defined in the ***e*** code. If the following constraints are defined in the order shown, then the program applies the "**keep soft** c > 5 and c < 10" constraint (the last one defined) and then quits the evaluation of soft value constraints for "c" when it encounters the "**keep** c.**reset_soft()**" constraint. It never considers the "**keep soft** c < 3" constraint. It does evaluate the "**keep soft** d< 3" constraint:

```
struct adder {
    c: uint;
    d: uint;
    keep soft c < 3;    // Is never considered
    keep soft d < 3;    // Is considered
};

extend adder {
    keep c.reset_soft();
};

extend adder {
    keep soft c > 5 and c < 10;
};
```

### Example 2

This example shows the use of **reset_soft()** in the situation where a soft constraint is written with the intent that it will be ignored if other constraints are added in an extension. Normally, “address” should be less than 64. The test writer needs to do nothing additional to get this behavior.

In a few tests, “address” should be any value less than 128. The test writer needs to remove the effect of the soft constraint so that it does not reduce the range [0..128] to [0..64].

```
struct transaction {
   address: uint;
   keep soft address < 64;
};
extend transaction {
   keep address.reset_soft();
   keep address < 128;
};
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.2.8 keep gen ... before

### Purpose

Modify the generation order

### Category

Struct member

### Syntax

**keep gen (*gen-item*:** exp, ...**) before (*gen-item*:** exp, ...**)**

Syntax example:

```
keep gen (y) before (x);
```

### Parameters

| | |
|---|---|
| *gen-item, ...* | An expression that returns a generatable item. The parentheses are required. See “gen-item” on page 294 for more information. |

### Description

Requires the generatable items specified in the first list to be generated before the items specified in the second list. You can use this constraint to influence the distribution of values by preventing soft value constraints from being consistently skipped. Before using this constraint for this purpose, read “Basic Concepts of Generation” on page 257 to be sure that you understand how soft constraints are evaluated.

### Notes

- This constraint itself can cause constraint cycles. If a constraint cycle involving one of the fields in the **keep gen** ... **before** constraint exists and if the **resolve_cycles** generation configuration option is TRUE, the constraint can be ignored if the program cannot satisfy both it and other constraints that conflict with it.
- This constraint cannot appear on the left-hand side of a implication operator (=>).

### Example

In the following example, the constraint requires the test generator to generate values for “length” and “data” before generating “crc”.

```
struct packet {
  good: bool;
  length: byte [1..24];
  data [length]: list of byte;
  crc: uint;
  keep good => crc == crc_calc();
  keep gen (length, data) before (crc);

  crc_calc() : uint is {
    result = pack(packing.low,length,data).crc_32(0,length);
  };
};

extend sys {
  p: list of packet;
  run() is also {
    print sys.p.crc;
  };
};

extend packet {
  keep good == TRUE; // just to show interesting case
};
```

### See Also

- “Basic Concepts of Generation” on page 257
- “Defining Constraints” on page 270
- “Invoking Generation” on page 296

## 7.2.9 keep soft gen ... before

### Purpose

Suggest order of generation

### Category

Struct member

### Syntax

**keep soft gen (*gen-item*:** exp, ...**) before (*gen-item*:** exp**,** ...**)**

Syntax example:

```
keep soft gen (y) before (x);
```

### Parameters

| | |
|---|---|
| *gen-item, ...* | An expression that returns a generatable item. See "gen-item" on page 294 for more information. |

### Description

Modifies the "soft" generation order by recommending that the fields specified in the first field list to be generated before the fields specified in the second field list. This soft generation order is second in priority to the hard generation order created by dependencies between parameters and **keep gen before** constraints.

You can use this constraint to suggest a generation order that you can later override for particular purposes in individual tests with a hard order constraint.

NOTE— This constraint cannot appear on the left-hand side of a implication operator (=>).

### Example

This example shows how you can use a soft order constraint to get the distribution of values you want. In the example below, there is a hard value constraint on "length" and "address".

```
struct transaction {
   address: uint;
};

extend transaction {
   length: uint [1..10];
   keep length == 5 => address < 50;
};
```

However, because "address" is generated first (based on coding order), "length" is generated to 5 only a small percentage of the time (50 out of MAX_UINT). If you want 5 to be as likely as any other value, the default ordering must be changed with a soft order constraint.

```
extend transaction {
   keep soft gen (length) before (address);
};
```

Since the order constraint is soft, it can be overridden by a hard constraint, such as one that uses a method. The following hard value constraint requires "address" to be generated before "length", overriding the soft generation order suggested by the previous extension to "transaction".

```
extend transaction {
   keep length == value(address);
};
```

### See Also

— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

## 7.2.10 keep gen_before_subtypes()

### Purpose

Specify a **when** determinant field for deferred generation

### Category

Struct member

### Syntax

**keep gen_before_subtypes(*determinant-field*:** field**, ...)**

Syntax example:

```
keep gen_before_subtypes(format);
```

### Parameters

| | |
|---|---|
| *determinant-field* | An expression that evaluates to the name of a field in the struct type. The field must be one that has at least one value that is used as a **when** determinant for a subtype definition. If the field is not a **when** determinant field, a warning is issued and the constraint is ignored.<br><br>Multiple field expressions can be entered, separated by commas. |

### Description

To speed up generation of structs with multiple **when** subtypes, this type of constraint, called a subtype optimization constraint, causes the generator engine to wait until a **when** determinant value is generated for a specified field before it analyzes constraints and generates fields under the **when** subtype.

When no subtype optimization constraints are present in a struct, the generator analyzes all of the constraints and fields in the struct before it generates the struct, even those constraints and fields that are defined under **when** subtypes. When a subtype optimization constraint is present, the generator initially analyzes only the constraints and fields of the base struct type. Only when a subtype optimization **when** determinant is encountered does the generator analyze the associated **when** subtype and then generate it.

### Notes

— Subtype optimization can handle multiple determinants. Subtypes are analyzed and generated in the order in which their **when** determinants are encountered.

— If multiple determinants are specified, and some of them are subtype optimization determinants while others are not, then a subtype that is a result of multiple inheritance of a subtype optimization determinant and a non-subtype optimization determinant will be treated the same as a other subtype optimization determinant subtype.

— The generator engine's ability to resolve contradictions is diminished somewhat by subtype optimization constraints. Specifically, the generator might not be able to resolve contradictions arising from constraints under subtypes that involve fields of the base type.

— The analysis and generation is recursive. If a subtype contains another determinant that is specified in a subtype optimization constraint, then that sub-subtype is analyzed and generated as soon as its determinant field is generated under the higher-level subtype.

### Example 1

This example shows a subtype optimization constraint on the field named format in the instr_s struct. The generator defers analysis and generation of constraints and fields under the FMT_A and FMT_B subtypes, since those are **when** determinants.

```
type format_t: [FMT_A, FMT_B, FMT_C];
struct instr_s {
    intrpt: bool;
    format: format_t;
    keep gen_before_subtypes(format);
    keep format == FMT_A => intrpt != FALSE;

    when FMT_A'format instr_s {
        a_intrp: bool;
        keep intrpt != a_intrp;
        keep gen (size) before (offset);
        keep offset == 0x10;
    };

    when FMT_B'format instr_s {
        b_intrp: bool;
        keep intrpt == TRUE;
    };

    offset: int;
    size: int;
};
```

The generation order for the example above is:

1) All constraints in the base struct concerning intrpt and format are analyzed.
2) A value is generated for intrpt.
3) A value is generated for format.
4) When format is FMT_A,

- All constraints under subtype FMT_A are analyzed.
- A value is generated for a_intrp.

5) A value is generated for size.
6) A value is generated for offset.

### Notes

— The **gen...before** constraint in the FMT_A subtype can be satisfied because neither offset nor size has been generated when that constraint is encountered.

— The constraint between intrpt and a_intrp can be satisfied even though it is unidirectional, because intrpt is generated before a_intrp.

— If the **gen...before** constraint under FMT_A was between intrpt and offset, for example, then it would be ignored because intrpt is generated before any subtypes are analyzed (unless an explicit order constraint was added between the format determinant and intrpt).

### Example 2

In the following example, the keep op != SUB constraint under the FMT_A subtype might cause a contradiction, since it involves a field in the base struct. This is because the generator initially generates the base struct (the op and format fields) before it analyzes the constraints in the subtype. There are no constraints involving op and format in the base struct, so the generator is free to choose FMT_A and SUB for those fields. However, once the format determinant is fixed, the generator analyzes the FMT_A subtype, and finds that op is not allowed to be SUB. This results in a contradiction.

```
type format_t: [FMT_A, FMT_B, FMT_C];
type opcode_t: [ADD, SUB, MUL, DIV];
struct instr_s {
    op: opcode_t;
    format: format_t;
    keep gen_before_subtypes(format);

    when FMT_A'format instr_s {
        a_intrp: bool;
        keep op != SUB;    // Might cause a contradiction
    };
};
```

To avoid the possibility of a contradiction described above, you can elevate the constraint from the subtype to the base struct:

```
keep format == FMT_A => op != SUB;
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.2.11 keep reset_gen _before_subtypes()

### Purpose

Disable all previous **keep gen_before_subtypes()** subtype optimization constraints

### Category

Struct member

### Syntax

keep reset_gen_ before_subtypes()

Syntax example:

```
keep reset_gen_before_subtypes();
```

### Description

When subtype optimization is turned off by default, this constraint causes the generator to ignore all previously defined **gen_before_subtypes()** constraints for the enclosing struct or unit. Any **gen_before_subtypes()** constraints you define after the reset will be followed.

When subtype optimization is turned on by default, this constraint turns off subtype optimization for the enclosing struct or unit.

When subtype optimization is forced on or off, this constraint has no effect.

NOTE— You can define other subtype optimization constraints following a **keep reset_gen_before_subtypes()** constraint.

### Example

This example shows a **reset_gen_before_subtypes()** constraint, which disables all previous **gen_before_subtypes()** constraints, followed by a new **gen_before_subtypes()** constraint which is still effective.

```
type format_t: [FMT_A, FMT_B, FMT_C];
struct instr_s {
    intrpt: bool;
    format: format_t;
    keep gen_before_subtypes(format);
};
extend instr_s {
    keep reset_gen_before_subtypes();
    // Disables previous subtype optimization constraints
    keep gen_before_subtypes(intrpt);
    // This new subtype optimization constraint is still in effect
};
```

### See Also

— “keep gen_before_subtypes()” on page 287
— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.2.12 value()

### Purpose

Modify generation sequence

### Category

Pseudo-method

### Syntax

**value(*item*:** exp**)**

Syntax example:

```
keep i < value(j);
```

### Parameters

| | |
|---|---|
| *item* | A legal ***e*** expression. |

### Description

Generates values for any data items that are contained in the expression and returns the value of the expression.

This method affects generation order and also makes the constraint unidirectional.

```
keep a == value(b + c);
```

The constraint shown above has two results:

— “b” and “c” will be generated before “a”.
— You cannot otherwise constrain the value of “a”.

The **value()** method is similar to a **gen before** constraint in that it affects generation order. It can also provide some performance improvement. The test generator has less work to do because it does not need to propagate constraint information from “a” to “b” and “c”.

NOTE— Like most other method calls, the **value()** method cannot be used to constrain the legal values for any data item that it contains. A constraint such as “keep value(j) == 16”, which appears to require the test generator to keep the value of “j” equal to 16, is not enforceable.

Thus, although the following example loads without error, a contradiction almost always occurs during generation because “j” is generated before the constraint is applied:

```
extend sys {
    j:int;


    keep value(j) == 16;
};
```

### Result

In these sample results, “j” was generated as 536940611, and then the constraint was applied, reducing the valid range to [].

```
*** Error: Contradiction:
    A contradiction has occurred when generating sys-@0.(j) :
       Previous constraints reduced its range of possible values,
        then the following constraint contradicted these values:
            keep 16 == value(j)                          at line 5 in @test4
        Reduced: sys-@0.(j) into []
          Using: sys-@0.j == [536940611]
    To see details, reload and rerun with "col gen"
```

See “Enforceable Expressions” on page 261 for more information.

### Example

"j" is generated first, then "i". "i" is always less than the value of "j" shifted one bit to the right.

```
extend sys {
    i: int;
    j: int;
    keep i < value(j >> 1);
};
```

The code shown below is equivalent:

```
extend sys {
    i: int;
    j: int;
    keep gen (j) before (i);
    keep i < (j >> 1);
};
```

### See Also

— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296

## 7.2.13 constraint-bool-exp

### Purpose

Define a constraint on a generatable item

### Category

Expression

### Syntax

***bool-exp*** [**or** | **and** | => ***bool-exp***] ...

Syntax example:

```
z == x + y
```

### Parameters

| | |
|---|---|
| *bool-exp* | An expression that returns either TRUE or FALSE when evaluated at run time. |

### Description

A constraint boolean expression is a simple or compound boolean expression that describes the legal values for at least one generatable item or constrains the relation of one generatable item with others.

A compound boolean expression is composed of two or more simple expressions joined with the **or**, **and** or implication (=>) operators.

***e*** has several special constructs that are useful in constraint boolean expressions:

| | |
|---|---|
| soft | A keyword that indicates that the constraint is either a soft value constraint or a soft order constraint. See “Generation Constraints” on page 257 for a definition of these types of constraints. |
| **soft**...**select** | An expression that constrains the distribution of values. |
| **.reset_soft()** | A pseudo-method that causes the test generator to quit evaluation of soft constraints for a field, in effect, removing previously defined soft constraints. |
| **.is_all_iterations()** | A list method used only within constraint boolean expressions that causes a list of structs to have all legal, non-contradicting iterations of the specified fields. |
| **.is_a_permutation()** | A list method that can be used within constraint boolean expressions to constrain a list to have the same elements as another list. |
| [**not**] **in** | An operator that can be used within constraints boolean expressions to constrain an item to a range of values or a list to be a subset of another list, or, with **not**, to be outside the range or absent from another list. |
| **is** [**not**] **a** | An operator that checks the subtype of a struct. |

## Notes

— The **soft** keyword can be used in simple boolean expressions, but not in compound boolean expressions. Thus the first constraint below is valid, but the second generates a compile-time error:

```
keep x > 0 => soft y < 0;

keep soft x > 0 => y < 0;    // Compile-time error
```

— The order of precedence for boolean operators is: and, or, =>. A compound expression containing multiple boolean operators of equal precedence is evaluated from left to right, unless parentheses are used to indicate expressions of higher precedence. See Example 3 on page 294.

— Any ***e*** operator can be used in a constraint boolean expression. However, certain operators can affect generation order or can create an constraint that is not enforceable. See “Unidirectional Constraints” on page 259 and “Enforceable Expressions” on page 261 for more information.

## Example 1

The following are examples of simple constraint boolean expressions:

```
  long == TRUE
  soft x > y
  x == WIDTH
  x + z == y + 7
  x in [0..10]
  (list_1) in (list_2)
  list_3.is_a_permutation(list_4)
  list_of_packets.is_all_iterations(.protocol)
  packet.reset_soft()
  packet is a legal packet
  soft me.opcode == select {
     30: ADD;
     20: ADDI;
     10: [SUB, SUBI];
};
```

### Example 2

The following are examples of compound constraint boolean expressions:

```
x > 0 and soft x < y
is_a_good_match(x, y) => z < 1024
color != red or resolution in [900..999]
packet is a good packet => length in [0..1023]
```

### Example 3

In compound expressions where multiple implication operators are used, the order in which the operations are performed is significant. For example, in the following constraint, the first expression (a => b) is evaluated first by default:

```
keep a => b => c;                // is equivalent to
keep (not a or b) => c;          // is equivalent to
keep a and (not b) or c;
```

However, adding parentheses around the expression (b => c) causes it to be evaluated first, with very different results.

```
keep a => (b => c);              // is equivalent to
keep a => (not b) or c;          // is equivalent to
keep (not a ) or (not b)  or c;
```

### See Also

— "keep soft" on page 278
— "keep soft gen ... before" on page 285
— "keep gen-item.reset_soft()" on page 283
— "keep struct-list.is_all_iterations()" on page 274
— "Basic Concepts of Generation" on page 257
— "Defining Constraints" on page 270
— "Invoking Generation" on page 296
— "Scalar Types" on page 75

## 7.2.14 gen-item

### Purpose

Identifies a generatable item

### Category

Expression

### Syntax

[**me.**]***field1-name***[.***field2-name*** ...]

| **it** | [**it**].***field1-name***[.***field2-name*** ...]

Syntax example:

```
me.protocol
```

## Parameters

| | |
|---|---|
| *field-name* | The name of a field in the current struct or struct type. |

## Description

A generatable item is an operand in a boolean expression that describes the legal values for that generatable item or constrains its relation with another generatable item. Every constraint must have at least one generatable item or an error is issued.

In a **keep** constraint, the syntax for specifying a generatable item is a path starting with **me** of the struct containing the constraint and ending with a field name. In a **gen** action, the syntax for specifying a generatable item is a path starting with **it** of the struct containing the constraint and ending with a field name.

NOTE— A generatable item cannot have an indexed reference in it except as the last item in the path. Thus, constraints such as “**keep** a.keys[i] > 10” are legal, while constraints such as “**keep** packets[0].len > 10” are illegal.

To work around this restriction, use a **keep for each** constraint with an implication constraint:

```
keep for each (p) in packets {
    index == 0 => p.len > 10;
};
```

## Example 1

This example illustrates generatable and non-generatable items within a **keep** constraint. In the constraint boolean expression “x > **sys**.yy + y” shown in the example below, “x” is a generatable item. “y” is also generatable, even though it is marked with **!** as do-not-generate. On the other hand, “**sys**.yy” is not a generatable item because its path does not start with **me**.

```
extend sys {
    yy: int;
};
struct tmp {
    !y: int;
    x: int;
    keep x > sys.yy + y;
};
```

## Example 2

This example illustrates generatable and non-generatable items within a **gen** action. In the **gen** action shown below, **it** is the only generatable item. “i” is a local variable, and the paths of the other variables do not start with **it**.

```
extend sys {
    yy: int;
};
struct tmp {
    !y: int;
    x: int;
```

```
        m() is {
            var i: int;
            gen y keeping {
                it > i - me.x + sys.yy;
            };
        };
    };
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296
— “Implicit Variables” on page 24

## 7.3 Invoking Generation

For information on constructs used to control or invoke generation, see:

— ! described in “Defining Fields: field” on page 125
— “gen” on page 296
— “pre_generate()” on page 299
— “post_generate()” on page 300

### 7.3.1 gen

#### Purpose

Generate values for an item

#### Category

Action

#### Syntax

**gen** ***gen-item*** [**keeping** {[**it**].***constraint-bool-exp***; ...}]

Syntax example:

```
    gen next_packet keeping {
        .kind in [normal, control];
    };
```

## Parameters

| | |
|---|---|
| *gen-item* | A generatable item. If the expression is a struct, it is automatically allocated, and all fields under it are generated recursively, in depth-first order. |
| *constraint-bool-exp* | A simple or a compound boolean expression. See "constraint-bool-exp" on page 292 for more information. |

## Description

Generates a random value for the instance of the item specified in the expression and stores the value in that instance, while considering all the constraints specified in the **keeping** block as well as other relevant constraints at the current scope on that item or its children. Constraints defined at a higher scope than the enclosing struct are not considered. See Example 1 on page 297.

You can generate values for particular struct instances, fields, or variables during simulation (on-the-fly generation) with the **gen** action.

This constraint allows you to specify constraints that apply only to one instance of the item.

## Notes

— You can use the **soft** keyword in the list of constraints within a **gen** action.
— The earliest the **gen** action can be called is from a struct's **pre_generate()** method.
— The generatable item for the **gen** action cannot include an index reference. For example, "gen sys.pckts[index].dat;" and "gen sys.pckts.dat[index];" are both illegal.
— If a **gen ... keeping** action contains a **for each** constraint, you must name the iterated variable. See Example 3 on page 298 for more information.

## Example 1

This example uses the **gen** action within a TCM called "gen_next()" to create packets to send to the device under test. A constraint within the **gen** action keeps "len" with a range of values. A constraint defined at a lower scope level, "packet", is also applied when the **gen** action is executed, keeping the size of the "data" field equal to the "len" field. The constraint defined at the **sys** level "**keep** sp.next_packet.len == 4;" is not considered because it is not at the current scope of the **gen** action.

```
extend sys {
    event event_clk is @sys.any;
    sp: send_packet;
    keep sp.next_packet.len == 4;
};

struct packet {
    len: int [0..10];
    kind: [normal, control, ack];
    data: list of int;
    keep me.data.size() == len;
};

struct send_packet {
    num_of_packets_to_send: int [0..20];
    !next_packet: packet;

    gen_next() @sys.event_clk is {
        gen num_of_packets_to_send; // Random loop delimiter
        for i from 0 to num_of_packets_to_send - 1 do {
            wait ([100]*cycle);
            gen next_packet keeping {
                .len in [5..10];
                .kind in [normal, control];
            };
        };
    };
    run() is also {
        start gen_next();
    };
};
```

### Example 2

This example shows a **keep soft ... select** constraint on the **gen** action, which weights the distribution of values for the len field of a packet.

```
    struct packet {
        len: int [0..10];
        kind: [normal, control, ack];
    };
    struct top {
        !packet_list: list of packet;
        pkt_gen() is {
            var pkt: packet;
            for i from 0 to 100 {
                gen pkt keeping {
                    soft it.len == select {
                            20: [0..3];
                            60: [4..6];
                            20: [7..10];
                    };
                };
                packet_list.add(pkt);
            };
        };
    };
```

### Example 3

This example shows how to generate a struct containing a list on the fly, while constraining each item in the list.

NOTE— You must provide a name for the list item that is iterated. In other words substituting “for each in .data” for “for each (item) in .data” causes an error.

```
    <'
    struct packet {
        addr: uint;
        data: list of byte;
    };

    extend sys {

        send_packet()@sys.any is {
            var p: packet;
            gen p keeping {
                it.addr > 450000;
                for each (item) in .data {    // name is required
                    item > 10 and item < 30;
                };
            };
        };
    };
    '>
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.3.2 pre_generate()

### Purpose

Method run before generation of struct

### Category

Method of any struct

### Syntax

[***struct-exp***.]**pre_generate()**

Syntax example:

```
    pre_generate() is also {
        m = 7;
    };
```

### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. The default is the current struct. |

### Description

The **pre_generate()** method is run automatically after an instance of the enclosing struct is allocated but before generation is performed. This method is initially empty, but you can extend it to apply values procedurally to prepare constraints for generation. The **pre_generate()** method allows you to simplify constraint expressions before they are analyzed by the constraint solver.

The order of generation is recursively as follows:

1) Allocate the new struct.
2) Call pre_generate().
3) Perform generation
4) Call post_generate().

NOTE— Prefix the ! character to the name of any field whose value is determined by **pre_generate()**. Otherwise, normal generation will overwrite this value.

### Example

```
struct a {
    !m: int;
    m1: int;
    keep m1 == m + 1;
    pre_generate() is also {
        m = 7;
    };
};
extend sys {
    A: a;
};
```

### See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

## 7.3.3 post_generate()

### Purpose

Method run after generation of struct

### Category

Predefined method of any struct

### Syntax

[***struct-exp***.]**post_generate()**

Syntax example:

```
post_generate() is also {
    m = m1 + 1;
};
```

### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. The default is the current struct. |

### Description

The **post_generate()** method is run automatically after an instance of the enclosing struct is allocated and both pre-generation and generation have been performed. You can extend the predefined **post_generate()** method for any struct to manipulate values produced during generation. The **post_generate()** method allows you to derive more complex expressions or values from the generated values.

The order of generation is recursively as follows:

1) Allocate the new struct.
2) Call pre_generate().
3) Perform generation
4) Call post_generate().

## Example

```
struct a {
    !m: int;
    m1: int;
    post_generate() is also {
        m = m1 + 1;
    };
};
extend sys {
     A: a;
};
```

## See Also

— “Basic Concepts of Generation” on page 257
— “Defining Constraints” on page 270
— “Invoking Generation” on page 296

# 8 Events

The ***e*** language provides temporal constructs for specifying and verifying behavior over time. All ***e*** temporal language features depend on the occurrence of events, which are used to synchronize activity with a simulator and within the ***e*** program.

This chapter contains the following sections:

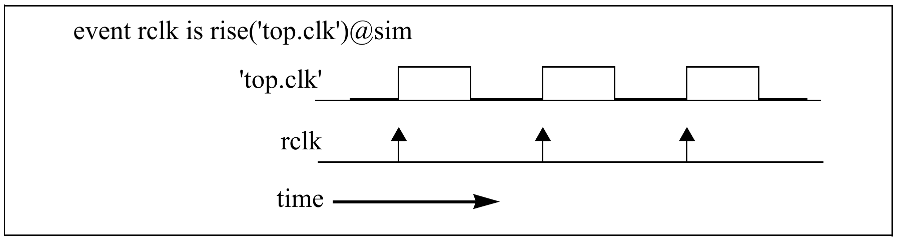
**See Also**

— “Defining Structs: struct” on page 118
— “Rules for Defining and Extending Methods” on page 459
— “Invoking Methods” on page 474
— Chapter 9, “Temporal Expressions”
— Chapter 10, “Temporal Struct Members”
— Chapter 11, “Time-Consuming Actions”

## 8.1 Events Overview

An example of an event definition is shown in Figure 8-1. An event named “rclk” is defined to be the rising edge of a signal named “top.clk” at another event named “sim”. The **@** symbol is used with an event name, ***@event,*** to mean “when the event is true”. The special @**sim** syntax means at a callback from the simulator. The **rise()** expression always causes a callback when the signal rises. Therefore, this event definition means “a rise of top.clk causes rclk to occur”. Occurrences of the “rclk” event are represented by arrows.

**Figure 8-1—Event Definition Example**



Once an event has been defined, it can be used in as the sampling event in temporal constructs such as temporal expressions (see Chapter 9, “Temporal Expressions”) like the following:

```
expect {[1]; true(chk)@rclk};
on rclk { ... };
```

Events also are used as the sampling points in time-consuming methods (see “Rules for Defining and Extending Methods” on page 459):

```
set_chk()@rclk is { ... };
```

The occurrence of any event is counted as a tick. Ticks are the means by which the ***e*** program marks the passage of time.

Many events are predefined in ***e***. You can use **event** struct members to define your own events, called named events, like the “rclk” example above.

All user-defined events are automatically included in functional coverage. A field named **events** in the **session** struct holds the user-defined event coverage data.

### See Also

— “Causes of Events” on page 304
— “Scope of Events” on page 304
— “Defining and Emitting Named Events” on page 305
— “Sampling Events Overview” on page 308
— “Predefined Events Overview” on page 309

## 8.1.1 Causes of Events

The ways in which an event are made to occur are described below.

| Syntax | Cause of the Event |
|---|---|
| **event** a **is** (@b and @c)@d | Derived from other events (see Chapter 9, “Temporal Expressions”). |
| **event** a **is rise(**'top.b')**@sim** | Derived from behavior of a simulated device (see Chapter 9, “Temporal Expressions”). |
| **event** a **is** { @b**;** @c**;** @d }@e | A sequence of other events (see Chapter 9, “Temporal Expressions”) |
| **event** a**;**<br>meth_b()@c **is** { ... ; **emit** a; ... }; | By the **emit** action in procedural code (see “emit” on page 307). |

You can use the **emit** action in any method to cause an event to occur, whether it has an attached temporal expression or not.

### See Also

— “Events Overview” on page 303

## 8.1.2 Scope of Events

The scoping rules for events are similar to other struct members, such as fields.

Events are defined as a part of a struct definition. When a struct is instantiated, each instance has its own event instances. Each event instance has its own schedule of occurrences. There is no relation between occurrences of event instances of the same type.

All references to events are to event instances. The scoping rules are as follows:

— If a path is provided, use the event defined in the struct instance pointed to by the path.
— If no path is provided, the event is resolved at compile time. The current struct instance is searched.
— If the event instance is not found, a compile-time error is issued.

#### See Also

— "Events Overview" on page 303

## 8.2 Defining and Emitting Named Events

This section describes the following constructs:

— "event" on page 305
— "emit" on page 307

#### See Also

— "Events Overview" on page 303
— "Sampling Events Overview" on page 308
— "Predefined Events Overview" on page 309

### 8.2.1 event

#### Purpose

Define a named event

#### Category

Struct member

#### Syntax

**event *event-type***[**is** [**only**] ***temporal-expression***]

Syntax example:

```
event clk is rise('top.cpu_clk') @sim;
```

#### Parameters

| | |
|---|---|
| *event-type* | The name you give the event type. It can be any legal ***e*** identifier. |
| *temporal-expression* | An event or combination of events and temporal operators.<br><br>To use an event name alone as a temporal expression, you must prefix the event name with the @ sign. For example, to define event A to succeed whenever event D succeeds, you must use the @ in front of D: "event A is @D". For more information about temporal expressions, see Chapter 9, "Temporal Expressions". |

#### Description

Events can be attached to temporal expressions, using the option **is** [**only**] ***temporal-expression*** syntax, or they can be unattached. An attached event is emitted automatically during any tick in which the temporal expression attached to it succeeds.

Events, like methods, can be redefined in struct extensions. The **is only *temporal-expression*** syntax in a struct extension is used to change the definition of an event. You can define an event once, and then attach it to several different temporal expressions under different **when** struct subtypes, using the **is only** syntax.

### Example 1

In the following, "start_ct" is an unattached event, and "top_clk" and "stop_ct" are attached events. The "m_str" extension contains a redefinition of the "top_ct" event.

```
struct m_str {
    event start_ct;
    event top_clk is fall('top.r_clk') @sim;
    event stop_ct is {@start_ct; [1]} @top_clk;
};
extend m_str {
    event stop_ct is only {@start_ct; [3]}@top_clk;
};
```

### Example 2

One way to cause a callback from the simulator is to sample a change, rise, or fall of a simulator object using **@sim**. The following causes a callback and a "sim_ready" event whenever the value of the simulator object "top/ready" changes.

```
event sim_ready is change('top/ready') @sim;
```

### Example 3

The **emit** action can be used to force any event to occur. The **emit** action in the following forces the "sim_ready" event to occur even if the "change('top/ready') @sim" temporal expression has not succeeded.

```
struct sys_ready {
    event sim_ready is change('top/ready') @sim;
    bar() @sys.clk is {
        while TRUE {
            wait until @sys.ok;
            wait [1] *cycle;
            emit sim_ready;
        };
    };
};
```

### See Also

— Chapter 9, "Temporal Expressions"
— "Rules for Defining and Extending Methods" on page 459
— "emit" on page 307
— "Events Overview" on page 303
— "Sampling Events Overview" on page 308
— "Predefined Events Overview" on page 309

### 8.2.2 emit

#### Purpose

Cause a named event to occur

#### Category

Action

#### Syntax

**emit** [***struct-exp*.**]***event-type***

Syntax example:

```
emit ready;
```

#### Parameters

| | |
|---|---|
| *struct-exp* | An expression referring to the struct instance in which the event is defined. |
| *event-type* | The type of event to emit. |

#### Description

Causes an event of the specified type to occur.

The simplest usage of **emit** is to synchronize two TCMs, where one TCM waits for the named event and the other TCM emits it.

Emitting an event causes the immediate evaluation of all temporal expressions that contain that event.

The **emit** event does not consume time. It can be used in regular methods and in TCMs.

#### Example

```
<'
struct xmit_recv {
    event rec_ev;
    transmit() @sys.clk is {
        wait cycle;
        emit rec_ev;
        out("rec_ev emitted");
    };
    receive() @sys.clk is {
        wait until @rec_ev;
        out("rec_ev occurred");
        stop_run();
    };
    run() is also {
        start transmit();
        start receive();
    };
};

extend sys {
    event clk is @sys.any;
    xmtrcv_i: xmit_recv;
};
'>
```

### See Also

— “event” on page 305
— “Events Overview” on page 303
— “Sampling Events Overview” on page 308
— “Predefined Events Overview” on page 309

## 8.3 Sampling Events Overview

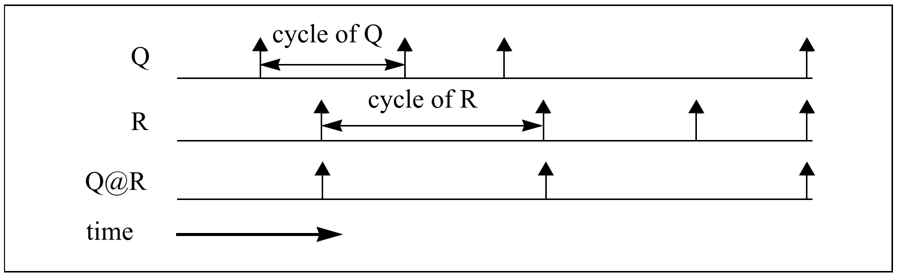
Events are used to define the points at which temporal expressions and TCMs are sampled. An event attached to a temporal expression becomes the sampling event for the temporal expression. The event is attached using the ***@sampling-event*** syntax:

***temporal-expression @sampling-event***

The temporal expression is evaluated at every occurrence of the sampling event. The sampling period is the time from after one sampling event up to and including the next sampling event. All event occurrences within the same sampling period are considered simultaneous. Multiple occurrences of a particular event within one sampling period are considered to be one occurrence of that event.

In Figure 8-2, Q and R are previously defined events that occur at the points shown. The temporal expression “Q@R” means “evaluate Q every time the sampling event R occurs”. If Q has occurred since the previous R event, then “Q@R” succeeds upon the next occurrence of R. The final “Q@R” success happens because the sampling period for the expression includes the last R event, which occurs at the same time as the last Q.

**Figure 8-2—Sampling Event for a Temporal Expression**



If “Q” in the figure above is a temporal expression that includes other events, “R” is the default sampling event of the temporal expression. The default sampling event of a temporal expression applies to all subexpressions within the expression, except where it is overridden explicitly by another event embedded in the expression.

For a TCM, the sampling event is written as:

***time-consuming-method*(...)*@sampling-event* is {...}**

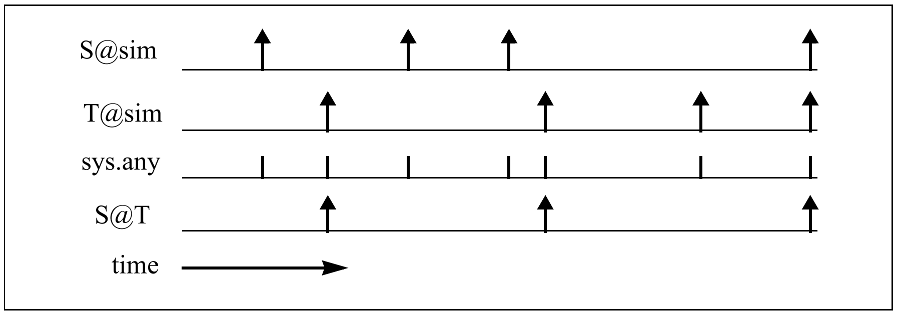
The default sampling event specified for a TCM drives or synchronizes actions within the TCM. It is not a trigger that launches the TCM, but is rather the event to which temporal actions and expressions in the TCM relate.

The predefined **sys.any** event occurs every time any other event occurs in the test. Expressions that need the highest time resolution can be attached to the **sys.any** event. The **sys.any** event is the default sampling event for all temporal expressions.

When a callback to is needed upon a change in a simulator variable, you can attach the name **sim** to the simulator variable. A change in a simulator variable to which **sim** is attached always causes a callback. The term **@sim** just represents a callback from the simulator; no **sim** event ever actually occurs.

In Figure 8-3, “S” and “T” are previously defined events. Attaching **@sim** to each of these event names causes the event to occur when its conditions are true at the time of a callback. The **sys.any** event occurs at every “S” event and every “T” event. The temporal expression “S@T” succeeds at every occurrence of T where “S” has occurred since the last T event.

**Figure 8-3—Occurrences of the sys.any Event**



### See Also

— “Events Overview” on page 303
— “Defining and Emitting Named Events” on page 305
— “Predefined Events Overview” on page 309

## 8.4 Predefined Events Overview

Predefined events are emitted at particular points in time. They are described in the following sections:

— “General Predefined Events” on page 310
— “Events for Aiding Debugging” on page 312
— “Simulation Time and Ticks” on page 312

### See Also

— “Events Overview” on page 303
— “Defining and Emitting Named Events” on page 305
— “Sampling Events Overview” on page 308

### 8.4.1 General Predefined Events

Table 8-1 lists the general predefined events. The events are described in more detail after the table.

**Table 8-1—Predefined Events**

| Predefined Event | Description |
|---|---|
| sys.any | Emitted on every tick. |
| sys.tick_start | Emitted at the start of every tick. |
| sys.tick_end | Emitted at the end of every tick. |
| session.start_of_test | Emitted once at test start. |
| session.end_of_test | Emitted once at test end. |
| ***struct***.quit | Emitted when a struct's **quit()** method is called. Only exists in structs that contain events or have members that consume time (for example, time-consuming methods and **on** struct members). |
| sys.new_time | In stand-alone operation (no simulator), this event is emitted on every **sys.any** event. When a simulator is being used, this event is emitted every time a callback occurs, if the attached simulator's time has changed since the previous callback. |

#### sys.any

Emitted on every tick.

This is a special event that defines the highest granularity of time. The occurrence of any event in the system causes an occurrence of the **any** event at the same tick. For any temporal expression used without an explicit sampling event, **sys.any** is used by default.

In stand-alone ***e*** program operation (that is, with no simulator attached), the **sys.any** event is the only one that occurs automatically. It typically is used as the clock for stand-alone operation, as in the following example.

Original clock definition for simulation:

```
<'
extend sys {
    event clk is rise('top.clk')@sim; // clk drives the system
};
'>
```

Extension to override the clock to tie it to **sys.any** for stand-alone operation:

```
<'
extend sys {
    event clk is only cycle @sys.any;
};
'>
```

### sys.tick_start

Emitted at the start of every tick.

This event is provided mainly for visualizing and debugging the program flow in the event viewer.

### sys.tick_end

Emitted at the end of every tick.

This event is provided mainly for visualizing and debugging the program flow in the event viewer. It also can be used to provide visibility into changes of values that are computed during the tick, such as the values of coverage items.

### session.start_of_test

Emitted once at the start of the test.

The first action the predefined **run()** method executes is to emit the **session.start_of_test** event. This event is typically used to anchor temporal expressions to the beginning of a test.

For example, in the following, the "watchdog" time is anchored to the beginning of the test by **session.start_of_test**:

```
<'
extend sys {
    event clk is cycle @sys.any;
    event watchdog is {@session.start_of_test; [100]}@clk;
    on watchdog {
        out("Watchdog triggered");
        stop_run();
    };
};
'>
```

### session.end_of_test

Emitted once at the end of the test.

This event is typically used to sample data at the end of the test. This event cannot be used in temporal expressions as it is emitted after evaluation of temporal expression has been stopped. The **on session.end_of_test** struct member is typically used to prepare the data sampled at the end of the test.

### *struct*.quit

Exists only in structs that contain temporal members (events, **on**, **expect**, TCMs). Emitted when the struct's **quit()** method is called, to signal the end of time for the struct.

The first action executed during the check test phase is to emit the **quit** event for each struct that contains it. It can be used to cause the evaluation of temporal expressions that contain the **eventually** temporal operator. This allows you to check for **eventually** temporal expressions that have not been satisfied.

### See Also

— "Events Overview" on page 303

- “Defining and Emitting Named Events” on page 305
- “Sampling Events Overview” on page 308
- “Predefined Events Overview” on page 309
- “Events for Aiding Debugging” on page 312
- “Simulation Time and Ticks” on page 312

## 8.4.2 Events for Aiding Debugging

Table 8-2 shows predefined events intended as aids in debugging. By setting **trace on** ***event***, you can see occurrences of events as the program runs. For example, entering “**trace on** tcm_start” before a test displays every occurrence of the **session.tcm_start** event.

NOTE— The predefined events in the **session** struct are not graded until the end of the test run, unlike user-defined events which are graded during the run. If you look at the session events grades during the run, you will see grades of 0.

**Table 8-2—Predefined Debugging Events**

| Debugging Events | Description |
|---|---|
| session.tcm_start | Emitted when any TCM is started |
| session.tcm_end | Emitted when any TCM finishes |
| session.tcm_call | Emitted when any TCM is called |
| session.tcm_return | Emitted when any TCM returns |
| session.call | Emitted when any method is called |
| session.return | Emitted when any method returns |
| session.output | Emitted when any output is issued |
| session.line | Shows line numbers for all traced events |
| session.tcm_wait | Emitted when a wait action occurs |
| session.tcm_state | Emitted when a state change occurs |
| session.sim_read | Emitted when a simulator variable is read |
| session.sim_write | Emitted when a simulator variable is written to |
| session.check | Emitted when a check action is performed. |
| session.dut_error | Emitted when a dut_error action is performed. |

### See Also

- “Events Overview” on page 303
- “Defining and Emitting Named Events” on page 305
- “Predefined Events Overview” on page 309

## 8.4.3 Simulation Time and Ticks

Using any of the following expressions causes the DUT to be monitored for a change in that expression:

— **rise** | **fall** | **change** (***HDL expression***) **@sim**
— **wait delay** ***expression***
— Verilog event

For each simulation delta cycle in which a change in at least one of these monitored expressions occurs, the simulator passes control to the ***e*** program. If simulation time has advanced since the last time control was passed to the ***e*** program, a **new_time** event is issued. In any case, **tick_start** and **any** events are issued. Then, after emitting all events initiated by changes in monitored expressions in that simulation delta cycle, a **tick_end** event is issued.

Thus, the **new_time** event corresponds to a new simulation time slot, and a tick corresponds to a simulation delta cycle in which at least one monitored expression changes.

Multiple ticks can occur in the same simulation time slot under the following conditions:

— When a new value is driven into the DUT and that value causes a change in a monitored HDL object, as in a clock generator
— When a monitored event is derived from another monitored event, as in a clock tree
— When a zero delay HDL subprogram is call from ***e***

NOTE— Glitches that occur in a single simulation time slot are ignored. Only the first occurrence of a particular monitored event in a single simulation time slot is recognized. For example, if a signal transitions from 0 to 1 and back to 0 in the same time slot, only the 0 to 1 transition is seen; the 1 to 0 transition is ignored. See Example 3 on page 316 for ways to handle glitches.

For an explanation of when values are assigned, see "<=" on page 493.

## Example 1

This example shows a clock generator and illustrates how two ticks can occur at the same simulation time.

### clock_gen.v

```
module top;
  reg clk;

  initial clk = 0;

  reg [31:0] data;
endmodule
```

### clock_gen.e

```
<'
extend sys {

    event clk is change('top.clk')@sim;

    event boot;

    clk_gen()@boot is {
        var clk_value: bit = 0;

        for j from 0 to 9 {
            'top.clk' = clk_value;
            wait delay(5);
```

```
                clk_value = ~clk_value;
            };
            stop_run();
        };

        run() is also {
            start clk_gen();
            emit boot;
        };
    };
    '>
```

### Result

There are two ticks in each simulation time slot. The first tick is caused by the elapse of the wait delay; the second, by the new value applied to top.clk.

```
-> 0 sys-@0.tick_start
-> 0 sys-@0.any
-> 0 sys-@0.tick_end

-> 5 sys-@0.new_time
-> 5 sys-@0.tick_start
-> 5 sys-@0.any
-> 5 sys-@0.tick_end
-> 5 sys-@0.tick_start
-> 5 sys-@0.any
-> 5 sys-@0.clk
-> 5 sys-@0.tick_end

-> 10 sys-@0.new_time
-> 10 sys-@0.tick_start
-> 10 sys-@0.any
-> 10 sys-@0.tick_end
-> 10 sys-@0.tick_start
-> 10 sys-@0.any
-> 10 sys-@0.clk
-> 10 sys-@0.tick_end

-> 15 sys-@0.new_time
-> 15 sys-@0.tick_start
-> 15 sys-@0.any
-> 15 sys-@0.tick_end
-> 15 sys-@0.tick_start
-> 15 sys-@0.any
-> 15 sys-@0.clk
-> 15 sys-@0.tick_end
...
```

### Example 2

This example shows a clock tree and illustrates how two ticks can occur at the same simulation time.

### tree.v

```
module top;

  reg prim_clk;
```

```
      reg sec_clk;

      initial begin
          prim_clk = 0;
          sec_clk = 0;
      end

      always #10 prim_clk = ~prim_clk;

      always @(prim_clk)
         sec_clk <= prim_clk;

    endmodule
```

### tree.e

In this example, the **wait** statement marked with a comment is expected to miss one edge of the secondary clock because the old value of top.sec_clk is sampled— one delta cycle before it changes. Note that **wait** action cannot resume in zero time, even if there are multiple delta cycles.

```
    <'
    extend sys {
        event prim_clk is change('top.prim_clk')@sim;
        event sec_clk is change('top.sec_clk')@sim;

        verify() @prim_clk is {
            var sec_clk_value: int;

            wait cycle;
            print sys.time;
            sec_clk_value = 'top.sec_clk';

            // Following wait misses one edge of secondary clock
            wait change('top.sec_clk')@sim;
            print sys.time;
            out("TCM reacts at edge of 'top.sec_clk': ",        'top.sec_clk');
            check that 'top.sec_clk' != sec_clk_value;
            wait change('top.sec_clk')@sim;
            print sys.time;
            stop_run();
        };

        on sec_clk {
            out("The program observes an edge of 'top.sec_clk': ",
               'top.sec_clk');
        };

        run() is also {
            start verify();
        };
    };

    '>
```

### Result

There are two ticks in each simulation time slot. The first tick is caused by the primary clock event; the second, by the secondary clock event.

```
---------------------------------------------------
-> 0 sys-@0.tick_start
-> 0 sys-@0.any
-> 0 sys-@0.tick_end
   ---------------------------------------------------
-> 10 sys-@0.new_time
-> 10 sys-@0.tick_start
-> 10 sys-@0.any
-> 10 sys-@0.prim_clk
-> 10 sys-@0.tick_end
-> 10 sys-@0.tick_start
-> 10 sys-@0.any
The program observes an edge of ’top.sec_clk’: 1
-> 10 sys-@0.sec_clk
-> 10 sys-@0.tick_end
   ---------------------------------------------------
-> 20 sys-@0.new_time
-> 20 sys-@0.tick_start
-> 20 sys-@0.any
-> 20 sys-@0.prim_clk
  sys.time = 20
-> 20 sys-@0.tick_end
-> 20 sys-@0.tick_start
-> 20 sys-@0.any
The program observes an edge of ’top.sec_clk’: 0
-> 20 sys-@0.sec_clk
-> 20 sys-@0.tick_end
   ---------------------------------------------------
-> 30 sys-@0.new_time
-> 30 sys-@0.tick_start
-> 30 sys-@0.any
-> 30 sys-@0.prim_clk
-> 30 sys-@0.tick_end
-> 30 sys-@0.tick_start
-> 30 sys-@0.any
The program observes an edge of ’top.sec_clk’: 1
-> 30 sys-@0.sec_clk
-> 30 sys-@0.tick_end
   ---------------------------------------------------
-> 40 sys-@0.new_time
-> 40 sys-@0.tick_start
-> 40 sys-@0.any
-> 40 sys-@0.prim_clk
-> 40 sys-@0.tick_end
-> 40 sys-@0.tick_start
-> 40 sys-@0.any
The program observes an edge of 'top.sec_clk': 0
-> 40 sys-@0.sec_clk
  sys.time = 40
TCM reacts at edge of 'top.sec_clk': 0
-> 40 sys-@0.tick_end
   ---------------------------------------------------
```

### Example 3

If multiple edges occur in a single simulation time slot on a signal monitored with @sim, the ***e*** program sees only the first one. For example, if you have an event defined as

```
event my_event is change('~/top/wireA')@sim;
```

and wireA transitions from 1 to 0 and back to 1 in the same time slot, my_event is emitted and stores 0 as the value of wireA. Then, if wireA changes to 0 in a subsequent time slot, the ***e*** program does not see this as a change and does not emit my_event.

There are two ways to handle this situation.

1) Change the sampling event of wireA to a clock sampling event:

```
event clk is change('~/top/clk')@sim;
event my_event is change('~/top/wireA')@clk;
```

2) Use the strobe option of **verilog variable** to filter out the glitch:

```
verilog variable '~/top/wireA' using wire, strobe = "#1";
```

### See Also

— "Events Overview" on page 303
— "Defining and Emitting Named Events" on page 305
— "Predefined Events Overview" on page 309
— "<=" on page 493

# 9 Temporal Expressions

Temporal expressions provide a declarative way to describe temporal behavior. The ***e*** language provides a set of temporal operators and keywords that you can use to construct temporal expressions.

This chapter contains the following sections:

— “Temporal Expressions Overview” on page 319
— “Temporal Operators and Constructs” on page 327

### See Also

— “Invoking Methods” on page 474
— Chapter 8, “Events”
— Chapter 10, “Temporal Struct Members”
— Chapter 11, “Time-Consuming Actions”

## 9.1 Temporal Expressions Overview

A temporal expression (TE) is a combination of events and temporal operators that describes behavior. A TE expresses temporal relationships between events, values of fields, variables, or other items during a test.

Temporal expressions are used to define the occurrence of events, specify sequences of events as checkers, and suspend execution of a thread until the given sequence of events occurs. Temporal expressions are only legal in the following constructs:

— **wait** and **sync** actions in time-consuming methods
— **event** definitions and **expect** or **assume** struct members.

The following sections provide more information about temporal expressions, how they are evaluated over time, and how the context in which they are used affects their interpretation.

— “Evaluating Temporal Expressions” on page 319
— “Using HDL Objects in Temporal Expressions” on page 322
— “Selected Applications of Temporal Expressions” on page 323
— “Forms for Common Temporal Expressions” on page 324
— “Translation of Temporal Expressions” on page 326

### See Also

— “Temporal Operators and Constructs” on page 327
— Chapter 8, “Events”
— Chapter 11, “Time-Consuming Actions”

### 9.1.1 Evaluating Temporal Expressions

Evaluating a temporal expression is more difficult than evaluating an arithmetic or boolean expression since it might require several clock cycles to complete. Every temporal expression has a default clock, called its sampling event, either stated specifically for the TE, or inherited from the context in which it appears. In the following example three cycles of the clk sampling event are needed before the TE terminates:

```
{rise('top.req'); [1]; @ack} @clk;
```

The "@clk" syntax means "evaluate when clk occurs". The above expression is a sequence temporal expression containing three simpler temporal expressions:

— **rise**('top.req') - a rise in the level of an HDL signal, followed by
— [1] - one cycle of the clk sampling event, followed by
— @ack - an occurrence of the ack event

Evaluation of this temporal expression commences with a rise of the top.req signal when sampled at the clk event. Evaluation then continues for two more clk cycles, and terminates successfully in the third cycle if, in that cycle, there is an occurrence of the ack event. Evaluation of the temporal expression terminates without success in the third clk cycle if the ack event does not occur in that cycle.

An evaluation of a TE thus succeeds, fails, or remains open on every occurrence of the sampling event. The period between occurrences of the sampling event is called the sampling period.

The context in which a TE is used determines when TE evaluation commences. In general a new evaluation commences on every occurrence of the sampling event, so that there may be several open evaluations of the TE at any one time. The context also determines the effect of success or failure of TE evaluation.

— A **wait** or **sync** action in a TCM operates on a temporal expression. See "Synchronization Actions" on page 365.

For example, to wait three pclk cycles after a rise on the request line before driving the data:

```
drive(data: byte) @pclk is {
    wait {rise('top.req'); [3]};
    'top.inbuf' = data;
};
```

A **wait** TE is first evaluated on the next pclk after the drive() TCM is started or called. When the temporal expression succeeds, the **wait** or **sync** construct terminates any open evaluations before resuming the thread. The TE will not be evaluated again until the drive() method is reinvoked. See "Invoking Methods" on page 474.

— An **event** definition is a struct member. An event definition binds a named event to a temporal expression:

```
event drive is {rise('top.req'); [3]} @pclk;
```

The TE in an event definition commences evaluation when the **run**() method of the struct in which it is declared is invoked, at the time the struct is initialized. Thereafter a new evaluation of the TE commences on every occurrence of the sampling event. Whenever the TE ***succeeds*** the event will be emitted. No action is taken when an **event** temporal expression fails. See Chapter 8, "Events".

— An **expect** or an **assume** is a struct member that binds a named behavioral rule to a temporal expression. For example a done event must occur no more than three pclk cycles after a drive event occurs:

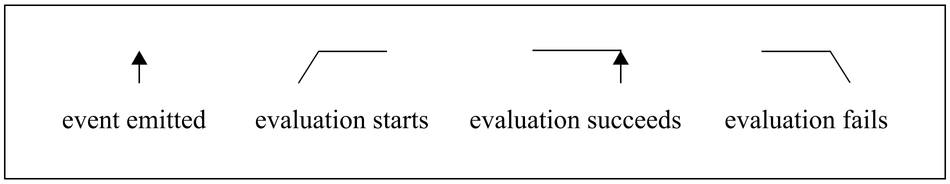
```
expect drive_check is @drive => {[..2]; @done} @pclk

    else dut_error{"drive_check failed at ", sys.time, ".");
```

The TE in the declaration of an **expect** commences evaluation when the **run**() method of the struct in which it is declared is invoked, at the time the struct is initialized. Thereafter a new evaluation of the TE commences on every occurrence of the sampling event. Whenever the TE ***fails*** the exception associated with the rule is invoked causing the test to be aborted, or a warning message to be printed. No action is taken when an **expect** TE succeeds. See "expect | assume" on page 360

A struct is terminated by calling the ***struct*****.quit**() method, which is predefined for any struct that contains time-consuming constructs. When the **quit**() method is called any TE evaluation that succeeds or fails at the same time is allowed to do so, but all open temporal expression evaluations are terminated.

Figure 9-1 shows the graphical notation used in illustrations in the ***e*** language temporals documentation. The illustrations do not show evaluations that start and immediately fail.

**Figure 9-1—Legend for Temporals Graphics**



An example of the graphical notation to represent evaluation of temporal sequences is shown in Figure 9-2.

**Figure 9-2—Evaluation of Three Sequences**

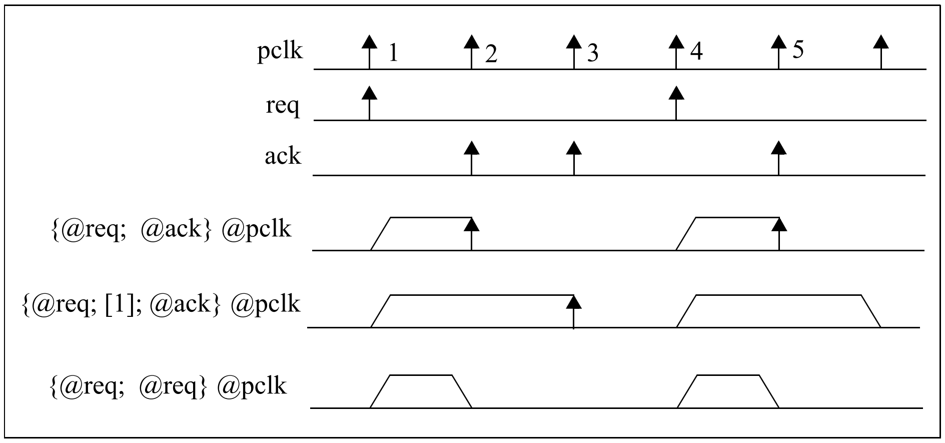


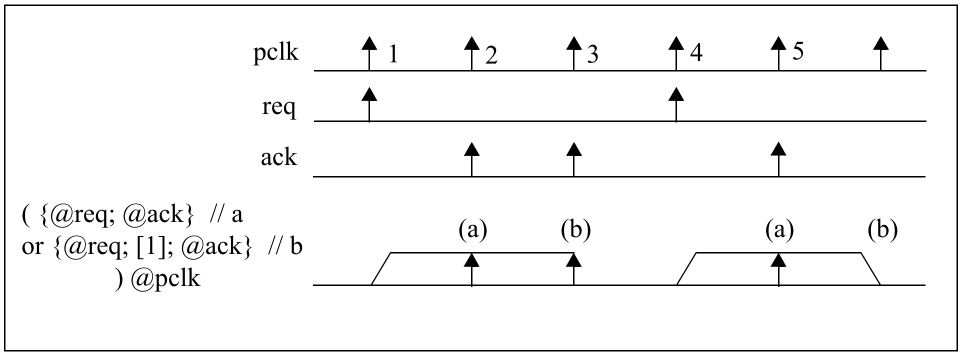
Figure 9-2 shows occurrences of three events, pclk, req, and ack. The pclk event is the sampling event for three sequences involving req and ack.

— The {@req; @ack} sequence starts evaluating each time req occurs at pclk. When ack occurs one pclk after req, the sequence succeeds.
— The {@req; [1]; @ack} sequence likewise starts evaluating at the first req occurrence at pclk, and shifts at the next pclk occurrence. When ack occurs at the third pclk, the sequence succeeds.
— The {@req; @req} sequence starts evaluating each time req occurs at pclk, and fails when req does not occur again at the next pclk.

Some sequences can succeed more than once during a particular evaluation. Figure 9-3 shows an evaluation of a temporal expression that is the OR of two sequences. The first sequence (a), {@req; @ack}@pclk, succeeds at the first ack occurrence (second pclk occurrence). The second sequence (b), {@req; [1]; @ack}@plck, succeeds at the second ack occurrence (third pclk).

When req occurs again at the fourth pclk occurrence, a new evaluation of the sequence starts. This evaluation succeeds upon the occurrence of ack at the fifth pclk cycle. Evaluation of the (b) branch continues at the sixth pclk, where it fails and terminates.

**Figure 9-3—Evaluation of the OR of Two Sequences**



### 9.1.2 Using HDL Objects in Temporal Expressions

To synchronize an ***e*** program to a simulator define an event that depends on a simulator variable (typically a clock), and use the event as the sampling event for a TCM, or as part of a temporal expression.

To create an event dependent on an HDL object or expression, use the following syntax:

**event *event-name* is (rise | fall | change) ('*HDL-expression*') @sim**

Using the **@sim** syntax activates the ***e*** program whenever the ***HDL-expression*** changes in the designated way (rises, falls, or changes value). The event is emitted at that time.

NOTE— For HDL expressions that contain vectors or bit selects of vectors, ***e*** detects changes on all bits of the vector. Thus, if the HDL expression is a bit select of a multibit clock signal, for example '/clockbus(1)', a callback occurs whenever any bit of clockbus changes, not just when clockbus(1) changes.

HDL expressions can be used in TEs sampled by any ***e*** event, not just **@sim**. The HDL values are sampled at each occurrence of the given sampling event.

In the following example, an event named clk is defined as the fall of a simulator signal named xor_top.clk. The clk event is used as the sampling event for a TCM named body() so that every time-consuming action in the TCM is synchronized to clk.

```
struct verify {
    event clk is fall('xor_top.clk')@sim;
                            // Causes simulator callback
    event rdy is fall('xor_top.ready')@clk;
                            // Does not cause callback
    body() @clk is {
        for each operation (op) in sys.ops {
            'xor_top.a' = op.a;
            'xor_top.b' = op.b;
            wait @rdy;
            op.result_from_dut = 'xor_top.out';
            print sys.time;
            print op;
            check that op.result_from_dut == (op.a ^ op.b);
            wait [1]*cycle;
        };
        };
    };
```

Verilog events can also be used to create events. To create an event from a Verilog event, use **change('*VL-event*') @sim**, as in the following example:

```
    event vl_event1 is change('xor_top.verilog_event1')@sim;
```

### See Also

— "Temporal Expressions Overview" on page 319

## 9.1.3 Selected Applications of Temporal Expressions

This section describes the following:

— "Handling Overlapping Transactions" on page 323
— "Restricting TE Matches" on page 323

### 9.1.3.1 Handling Overlapping Transactions

Transactions can overlap in the sense that many of them can be active at the same time. These can be purely pipelined transactions or transactions that are identified by some key.

Handling pipelined transactions is easy in ***e***. For example, the following is a behavioral rule for a buffer with a latency of three cycles:

```
    expect @buf.in => {[2]; @buf.out};
```

Often data need to be carried with the transaction. These may be input our output data associated with the transaction, or some identification of the specific transaction.

In such cases the solution is to create a "transaction" struct that carries the data. The struct also contains the temporal rule describing the expected behavior of the struct. A new transaction struct needs to be created every time a transaction starts.

### 9.1.3.2 Restricting TE Matches

A temporal expression is re-evaluated at every occurrence of the sampling event to see if there is any possible match of the behavior it describes. Sometimes a different behavior is expected, where not all matches are considered.

For example, consider a component that handles fixed length transactions. The basic behavior we want to check for is "every transaction that starts will end after N cycles":

```
    expect @trans.start => {[N]; @trans.end} @clk;
```

However, suppose that the design under test can only handle one transaction at a time. If a new transaction starts while the previous one is being processed, the component cannot handle the second transaction. The **expect** check fails because the component does not emit the expected transaction end. Since an **expect** automatically traces multiple transactions, you must explicitly rule out such cases. For example:

```
    expect @trans.start =>
```

```
                {[N]* not @trans.start;@trans.end}@clk;
```

This formulation explicitly states that a transaction start must not occur while a transaction is being processed.

### See Also

— “Temporal Expressions Overview” on page 319

## 9.1.4 Forms for Common Temporal Expressions

The natural way to specify future behavior is in terms of “cause yields effect”, which is done with the temporal yield operator (=>). The following is equivalent to the statement, “A transaction start is followed within 1 to 4 cycles by a transaction end”:

```
   @transaction.start => {[..3]; @transaction.end};
```

The language also provides a way to maintain information about past events, which you can then use in yield expressions like the above. This is done with the **detach** operator. The following implements the requirement that “a transaction end is preceded by a transaction start within the previous 1 to 4 cycles:

```
   @transaction.end => detach({@transaction.start; ~[2..5]});
```

The **detach**() operator causes the embedded temporal expression to be evaluated in parallel with the main temporal expression. See “detach” on page 343.

Temporal expressions for many situations are shown below. The desired conditions are stated, and then an expression is shown for those conditions. In these expressions, TE***n*** is a temporal subexpression, which can be an event.

### See Also

— “Examples of Sequence Expressions” on page 324
— “Examples of Behavioral Rule Checks” on page 325

### 9.1.4.1 Examples of Sequence Expressions

— TE1 and TE2 at the same time:

```
TE1 and TE2
```

— TE1 followed by TE2:

```
{TE1; TE2}
```

— TE1 any number of times, then TE2:

```
{[..] * TE1; TE2}
```

— TE2 in the nth cycle after TE1:

```
{TE1; [n - 1]; TE2}
```

— TE2 within n cycles after TE1:

```
{TE1; [..n-1]; TE2}
```

— TE2 within n1 to n2 cycles after TE1:

```
{TE1; [n1-1..n2-1]; TE2}
```

— TE2 any number of cycles after TE1:

```
{TE1; [..]; TE2}
```

— No TE2 before TE1:

```
{[..] * not TE2; TE1 and not TE2}
```

— TE2 after n cycles of no TE1:

```
{[n] * not TE1; TE2}
```

— TE2 any number of cycles after TE1, with no TE3 in between:

```
{TE1; [..] * not TE3; TE2}
```

— TE2 after TE1, repeated n times:

```
{@session.start_of_test; [n] * {[..]; detach({TE1; TE2})}
```

— TE1 after the last TE2 and before TE3:

```
{TE2; [..] * not TE2; (TE1 and not TE3 and not TE2);
    [..] * not TE2; TE3}
```

### 9.1.4.2 Examples of Behavioral Rule Checks

— If TE1 succeeds TE2 must follow:

```
expect TE1 => TE2
    else dut_error("TE2 did not occur 1 cycle after TE1")
```

— TE2 must succeed within n1 to n2 cycles of TE1:

```
expect TE1 => {[n1-1..n2-1]; TE2}
    else dut_error("No TE2 ",n1," to ",n2," cycles after TE1")
```

— If TE1 succeeds then TE2 should eventually succeed:

```
expect TE1 => (eventually TE2)
    else dut_error("TE1 occurred but not TE2")
```

— If TE1, then TE2 must not succeed for n cycles:

```
expect TE1 => [n] * not TE2
    else dut_error("TE2 less than ",n," cycles after TE1")
```

— If TE2, then TE1 must have succeeded n cycles earlier:

```
expect TE2 => detach({TE1; [n + 1]})
    else dut_error("TE2 without TE1 ",n," cycles earlier")
```

— If TE2, then TE1 must have succeeded sometime earlier:

```
expect TE2 => detach({TE1; ~[2..]})
    else dut_error("TE2 without a previous TE1")
```

— TE2 should succeed after TE1 and no more that n occurrences of TE3:

```
expect {TE1; [n] * {[..]; TE3}} => {[..] * not TE3; TE2}
    else dut_error("TE1 not followed by TE2")
```

— TE must not succeed more than n times during the test:

```
expect @session.start_of_test => fail{[n + 1] * {[..]; TE}}
    else dut_error("TE occurred more than ",n," times")
```

### See Also

— “Temporal Expressions Overview” on page 319

## 9.1.5 Translation of Temporal Expressions

Certain temporal expressions that describe unusually complex temporal behavior cannot be processed by the static analysis of the temporal engine. Errors can result from:

— Analysis capacity overflow
— Limited analysis capability

If the ***e*** program identifies a failure to translate a complex temporal expression at load or compile time, you will have to decompose the expression into several smaller expressions. If you have difficulty decomposing complex temporal expressions, contact ***support@verisity.com***.

### Examples of Analysis Capacity Overflow

The following types of temporal expressions may produce complexity beyond the capacity of the static analysis:

— Multiple bounded repeats (such as [3], [..i] or [n..]) combined by the **and** or **or** temporal operator. The complexity is much higher when using such a temporal expression within **expect** struct members, **fail** or **not** temporal operators. For example:

```
[3] or [4] or [7]  or [17]
```

```
expect @a => {[..i];@b} or {[..j];@c}  or {[n..];@f}
```

— Nested sampling of multiple temporal expressions combined by temporal **and** operator. For example:

```
(@a and @b and @c and @d .. and @k) @q
```

— Use of long temporal sequence or complex nested sampling (as described above) in the match part of a first-match repeat operator. For example:

```
{[..];(@a and @b and @c   and @k) @q} @sys.any
```

NOTE— There is no complexity issue if the repeat part and the match part are mutually exclusive. For example:

```
{[..]*fail @a;(@a and @b and @c   and @k)@q} @sys.any
```

This may also hold when the match part uses a small constant in a simple repeat temporal expression. For example:

```
{[..n]; [3] *@a}.
```

— Combinations of smaller examples of the above.

### Examples of Limited Analysis Capability

The following types of temporal expressions are certain to produce complexity beyond the capabilities of the static analysis:

— A bounded repeat (such as [m]) in the match expression of a first match repeat. For example:

```
{[..n]; [m] *@a}
```

NOTE— There is no complexity issue if the repeat part and the match part are mutually exclusive. For example:

```
{[..n] * fail @a ; [m] *@a}
```

There is no complexity issue if the match part uses a small constant bound in a simple repeat as described in 1.c above. For example:

```
{[..n]; [2] *@a}
```

— A temporal expression that has multiple successes (such as @a or {@b;@b}) as the repeat expression of a bounded first match repeat. For example:

```
{[..n]*{@a or {@b;@b}};@c}
```

— A true match repeat (such as `(~[..m]*@a))` as the repeat expression of a bounded first match repeat. For example:

```
{[..n]*(~[..m]*@a);@c}
```

— A bounded true match repeat in an **expect** struct member or with **fail** or **not** temporal operators. For example:

```
not {~[..n]*{@a or {@b;@b}; @c}
```

## 9.2 Temporal Operators and Constructs

This section describes the constructs you can use in temporal expressions:

In addition, it describes:

— “Precedence of Temporal Operators” on page 328

### See Also

— “Temporal Expressions Overview” on page 319

### 9.2.1 Precedence of Temporal Operators

The precedence of temporal operators is shown in Table 9-1, listed from highest precedence to lowest.

**Table 9-1—Precedence of Temporal Operators**

| Operator Name | Operator Example |
|---|---|
| named event | @event-name |
| exec<br>consume | TE exec action-block<br>consume (@event-name) |
| repeat | [ ] * TE |
| fail<br>not | fail TE<br>not TE |
| and | TE1 and TE2 |
| or | TE1 or TE2 |
| sequence | {TE1 ; TE2} |
| yield | TE1 => TE2 |
| sample event | TE @ event-name |

#### See Also

— “Temporal Expressions Overview” on page 319

### 9.2.2 not

#### Purpose

Temporal expression inversion operator

#### Category

Temporal expression

#### Syntax

**not** *temporal-expression*

Syntax example:

not {@ev_b;@ev_c}

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

### Description

The **not** temporal expression succeeds if the evaluation of the subexpression does not succeed during the sampling period. Thus **not** TE succeeds on every occurrence of the sampling event if TE does not succeed.

NOTE— If an event is explicitly *emitted* (using "emit" on page 307), a race condition can arise between the event occurrence and the **not** of the event when used in some temporal expression.

### Example 1

In the following, the event ev_d occurs every time there is an occurrence of ev_c that is not preceded by an occurrence of ev_a and then two consecutive occurrences of ev_b.

```
    event n_e is {not{ @ev_a; @ev_b; @ev_b}}; @ev_c} @clk;
```

See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.

### Example 2

The **fail** operator (see "fail" on page 329) differs from the **not** operator. Figure 9-4 on page 331 illustrates the differences in behavior of **not** and **fail** for the sequence of ev_a, ev_b, and ev_c events shown at the top of the figure. (See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.)

### See Also

— "Sampling Events Overview" on page 308
— "fail" on page 329
— "Temporal Operators and Constructs" on page 327

## 9.2.3 fail

### Purpose

Temporal expression failure operator

### Category

Temporal expression

### Syntax

**fail *temporal-expression***

Syntax example:

fail{@ev_b; @ev_c}

## Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

## Description

A **fail** succeeds whenever the temporal expression fails. If the temporal expression has multiple interpretations (for example, **fail** (TE1 **or** TE2)), the expression succeeds if and only if all the interpretations fail.

The expression **fail** TE succeeds at the point where all possibilities to satisfy TE have been exhausted. Any TE can fail at most once per sampling event.

NOTE— The **not** operator differs from the **fail** operator.

## Example

The expression

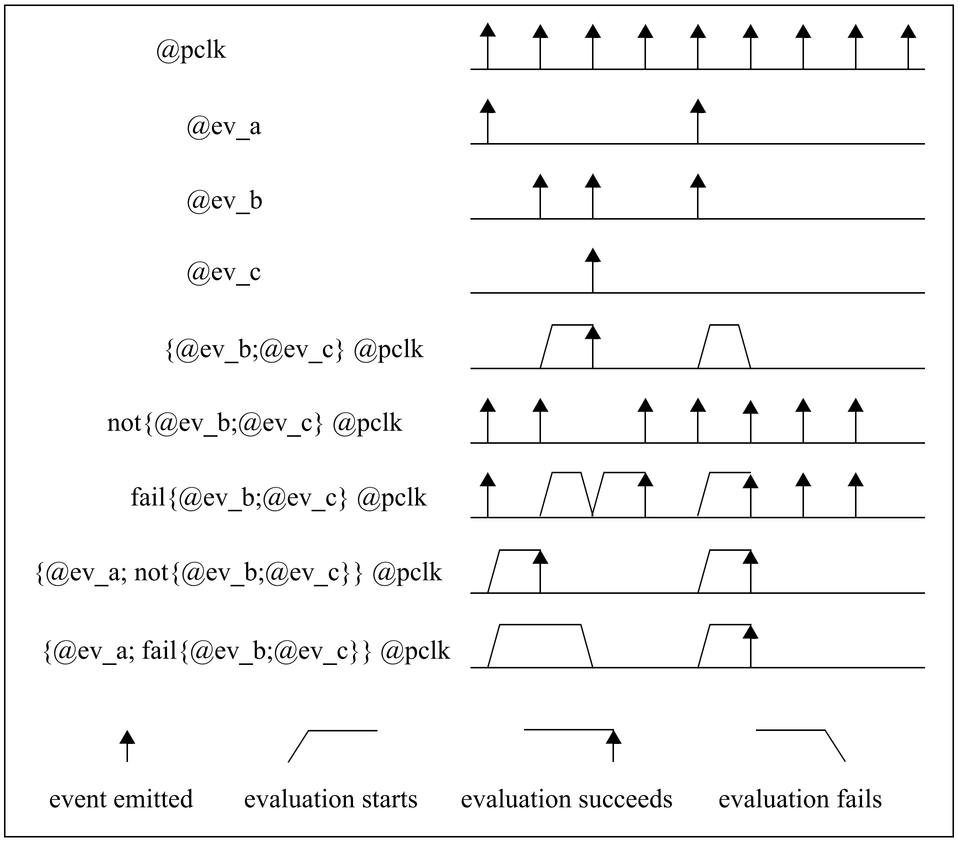
```
fail {@ev_b;@ev_c}
```

succeeds for any of the following conditions:

— event ev_b does not occur in the first cycle
— ev_b succeeds in the first cycle, but event ev_c does not occur in the second cycle

Figure 9-4 on page 331 illustrates the differences in behavior of **not** and **fail**.

**Figure 9-4—Comparison of Temporal not and fail Operators**



### See Also

— "not" on page 328
— "Temporal Operators and Constructs" on page 327

### 9.2.4 and

#### Purpose

Temporal expression and operator

#### Category

Temporal expression

#### Syntax

***temporal-expression*** **and** ***temporal-expression***

Syntax example:

```
(@TE1 and @TE2)@clk
```

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

### Description

The temporal **and** succeeds when both temporal expressions start evaluating in the same sampling period, and succeed in the same sampling period.

### Example 1

Evaluation of the **and** temporal expression:

```
event TE3 is (TE1 and TE2) @qclk
```

for the following conditions:

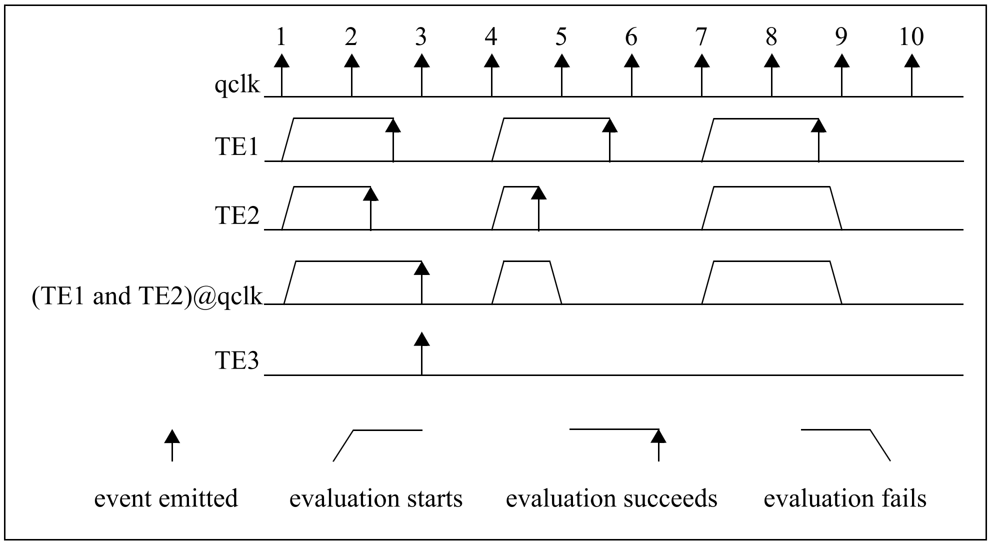
— Evaluation of both TE1 and TE2 begins on the first qclk. Both TE1 and TE2 succeed between the second and third qclk so the event TE3 is emitted at the third qclk.

— The evaluations of TE1 and TE2 that begin on the fourth qclk eventually result in success of both TE1 and TE2, but TE1 succeeds before the fifth qclk, and TE2 succeeds before the sixth qclk. Therefore, TE1 **and** TE2 does not succeed.

— On the seventh qclk, evaluation of TE1 begins, and it succeeds before the eighth qclk. However, the corresponding evaluation of TE2 fails during that period, so TE3 fails.

is shown in Figure 9-5 on page 332.

**Figure 9-5—Example 1 of Temporal and Operator Behavior**



### Example 2

Evaluation of the **and** temporal expression:

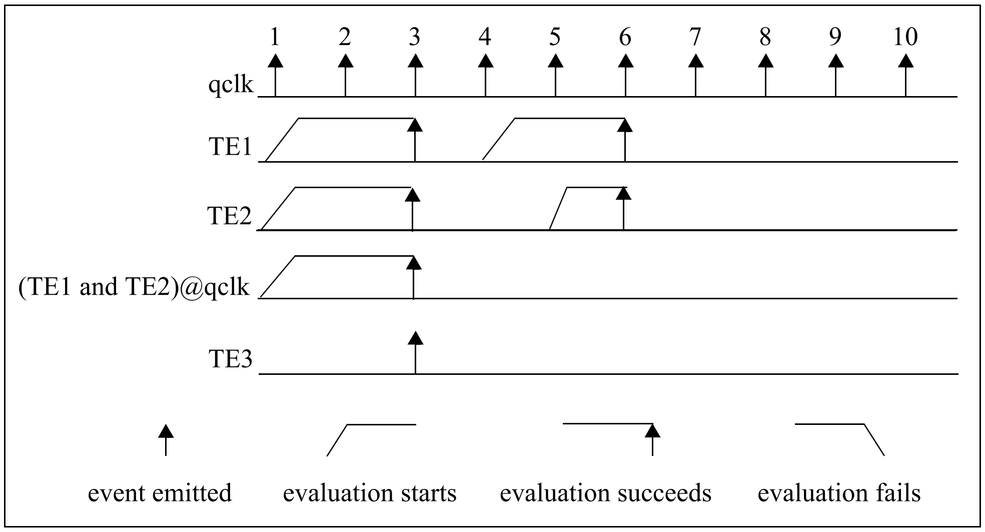
```
event TE3 is (TE1 and TE2) @qclk
```

for the following conditions:

— TE1 and TE2 both start evaluating at the first qclk.
— TE1 and TE2 both succeed at the third qclk.
The **and** succeeds because both sides started evaluating at the same time and both succeeded at the same time.
— TE1 starts evaluating at the fourth qclk.
— TE2 starts evaluating at the fifth qclk.
— TE1 and TE2 both succeed at the sixth qclk.
The **and** does not succeed because the two sides started evaluating at different time.

is shown in Figure 9-6 on page 333,

**Figure 9-6—Example 2 of Temporal and Operator Behavior**



### See Also

— "Sampling Events Overview" on page 308
— "Temporal Operators and Constructs" on page 327

## 9.2.5 or

### Purpose

Temporal expression or operator

### Category

Temporal expression

### Syntax

***temporal-expression*** **or** ***temporal-expression***

Syntax example:

```
    (@TE1 or @TE2)@clk
```

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

### Description

The **or** temporal expression succeeds when either temporal expression succeeds.

An **or** operator creates a parallel evaluation for each of its subexpressions. It can create multiple successes for a single temporal expression evaluation.

### Example

Evaluation of the temporal **or** operator:

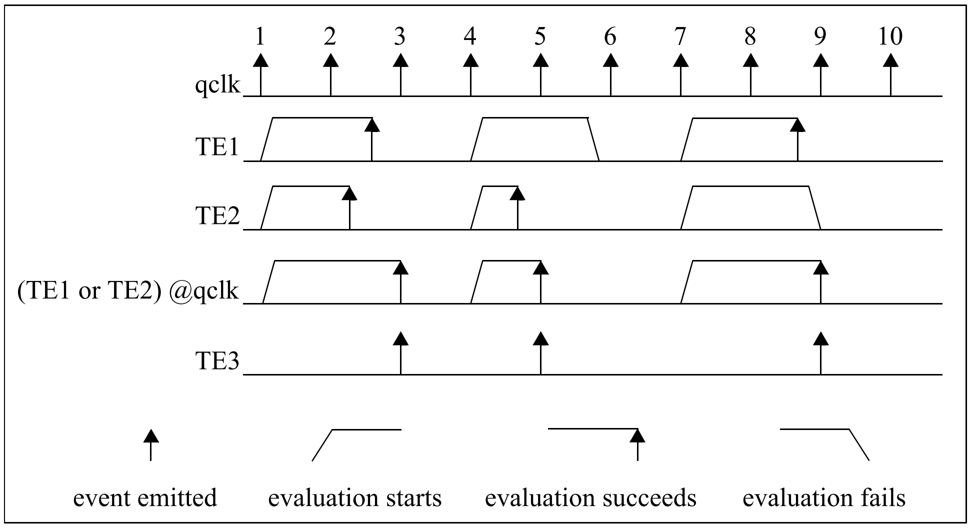
```
event TE3 is (@TE1 or @TE2) @qclk;
```

for the following conditions:

— Evaluation of both TE1 and TE2 begins on the first qclk, and succeed between the second and third qclk, so TE1 **or** TE2 succeeds at the third qclk.
— The evaluations of TE1 and TE2 that begin on the fourth qclk result in success of TE2 before the fifth qclk, so TE3 succeeds at the fifth qclk.
— Evaluation of TE1 or TE2 begins again at the seventh qclk, and TE1 succeeds before the ninth qclk, so TE3 succeeds at the ninth qclk.

is shown in Figure 9-7 on page 334.

**Figure 9-7—Example of Temporal or Operator Behavior**



### See Also

— "Sampling Events Overview" on page 308
— "Temporal Operators and Constructs" on page 327

### 9.2.6 { exp ; exp }

#### Purpose

Temporal expression sequence operator

#### Category

Temporal expression

#### Syntax

{***temporal-expression***; ***temporal-expression***; ...}

Syntax example:

```
{@ev_d; @ev_e} @ev_f
```

#### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

#### Description

The semicolon (;) sequence operator evaluates a series of temporal expressions over successive occurrences of a specified sampling event. Each temporal expression following a ";" starts evaluating in the sampling period following that in which the preceding temporal expression succeeded. The sequence succeeds whenever its final expression succeeds.

NOTE— Curly braces ({}) in the scope of a temporal expression define a sequence. They should not be used in any other way.

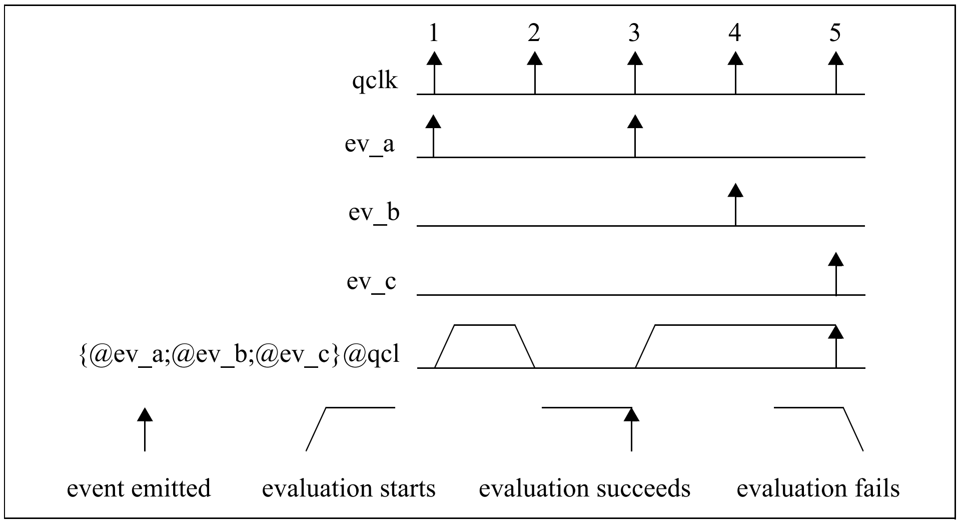
#### Example

Figure 9-8 on page 336 shows the results of evaluating the temporal sequence:

```
{@ev_a; @ev_b; @ev_c} @qclk;
```

over the series of ev_a, ev_b, and ev_c events shown at the top of the figure. Evaluation of the sequence starts whenever event ev_a occurs.

**Figure 9-8—Example Evaluations of a Temporal Sequence**



### See Also

— “Sampling Events Overview” on page 308
— “Temporal Operators and Constructs” on page 327

## 9.2.7 eventually

### Purpose

Temporal expression success check

### Category

Temporal expression

### Syntax

**eventually *temporal-expression***

Syntax example:

```
    {@ev_d; eventually @ev_e}
```

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |

### Description

Used to indicate that the temporal expression should succeed at some unspecified time.

Typically, **eventually** is used in an **expect** struct member to specify that a temporal expression is expected to succeed sometime before the **quit** event occurs for the struct.

### Example

The following instance of the temporal yield operator (see "=>" on page 342) succeeds after the event ev_c occurs only if event ev_a occurs in the next cycle, and then event ev_b occurs sometime before the example struct instance is terminated. See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.

```
struct example {
    event ev_a;
    event ev_b;
    event ev_c;
    expect @ev_c => {@ev_a; eventually @ev_b};
};
```

### See Also

— "The quit() Method of any_struct" on page 659
— "Temporal Operators and Constructs" on page 327

## 9.2.8 [ exp ]

### Purpose

Fixed repetition operator

### Category

Temporal expression

### Syntax

**[*exp*]** [***** ***temporal-expression***]

Syntax example:

```
wait [2]*cycle;
```

### Parameters

| | |
|---|---|
| *exp* | A 32-bit, non-negative integer expression, which specifies the number of times to repeat the evaluation of the temporal expression. This cannot contain functions. |
| *temporal-expression* | A temporal expression. If "***** temporal-expression" is omitted, "*** cycle**" is automatically used in its place. |

### Description

Repetition of a temporal expression is frequently used to describe cyclic or periodic temporal behavior. The [***exp***] fixed repeat operator specifies a fixed number of occurrences of the same temporal expression.

If the numeric expression evaluates to zero, the temporal expression succeeds immediately.

### Examples

The {...;...} syntax used in the examples below specifies a temporal sequence. The expressions are evaluated one after another, in consecutive sampling periods. See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.

In the following example, the **wait** action proceeds after the sequence event ev_a, then three occurrences of event ev_b, then event ev_c, all sampled at the default sampling event:

```
wait {@ev_a; [3]*@ev_b; @ev_c};
```

In the following example, the **wait** action proceeds after M+2 consecutive pclk cycles in which sys.interrupt occurs. If there is a pclk cycle without a sys.interrupt, the count restarts from 0:

```
wait ([M+2] * @sys.interrupt)@pclk;
```

In the following example, the **wait** action proceeds on the occurrence of the ev_a event:

```
wait {@ev_a; [0]*@ev_b};
```

In the following example, the **wait** action proceeds five sampling event cycles after event ev_a:

```
wait {@ev_a; [5]};
```

The numeric expression cannot include any functions. The following two examples show how to substitute temporary variables for functions in repeat expressions.

In a TCM, this is not legal:

```
wait [my_func()] * cycle; // illegal
```

To overcome this restriction, use a variable to hold the function value:

```
var t: int = my_func();
wait [t] * cycle;
```

In **expect**, assume or **event** struct members, this is not legal:

```
event my_ev is { @ev_a; [my_func()] } @clk; // illegal
```

In this situation, use a field to hold the function value and an **exec** expression to execute the function:

```
!temp: int;
event my_ev is { @ev_a exec {temp=my_func()}; [temp] } @clk;
```

### See Also

— "Temporal Operators and Constructs" on page 327

## 9.2.9 [ exp..exp ]

### Purpose

First match variable repeat operator

### Category

Expression

### Syntax

**{ ... ; [[*from-exp*]..[*to-exp*] ] [* *repeat-expression*]; *match-expression*; ... }**

Syntax example:

```
{[2..4]*@pclk;@reset}
```

### Parameters

| | |
|---|---|
| *from-exp* | An optional non-negative 32 bit numeric expression that specifies the minimum number of repetitions of the ***repeat-expression***. If the ***from-exp*** is missing, zero is used. |
| *to-exp* | An optional non-negative 32 bit numeric expression that specifies the maximum number of repetitions of the ***repeat-expression***. If the ***to-exp*** is missing, infinity is used. |
| *repeat-expression* | The temporal expression that is to be repeated a certain number of times within the ***from-exp..to-exp*** range. If the "****repeat-expression***" is omitted, "*cycle" is assumed. |
| *match-expression* | The temporal expression to match. |

### Description

The first match repeat operator is only valid in a temporal sequence {TE; TE; TE}. The first match repeat expression succeeds on the first success of the ***match-expression*** between the lower and upper bounds specified for the ***repeat-expression***.

First match repeat also enables specification of behavior over infinite sequences by allowing an infinite number of repetitions of the ***repeat-expression*** to occur before the ***match-expression*** succeeds.

Where @ev_a is an event occurrence, {[..]*TE1;@ev_a} is equivalent to:

- {@ev_a} or {[1]*TE1; @ev_a} or {[2]*TE1; @ev_a} or {[3]*TE1; @ev_a}...

### Examples

The following examples all make use of the {...;...} syntax for sequence temporal expressions since the first match repeat operator is only allowed inside a sequence. See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.

In the following example, the **wait** action proceeds after the first occurrence of ev_a followed by ev_b at pclk:

```
wait {[..]; {@ev_a; @ev_b}}@pclk
```

In the following example, the **wait** action proceeds after one or more occurrences of ev_a at consecutive pclk events, followed by one occurrence of ev_b at the next pclk event:

```
wait {[1..]*@ev_a; @ev_b}@pclk
```

In the following example, the **wait** action proceeds after between zero and three occurrences of the sequence {ev_a; ev_b} (sampled by pclk), followed by an occurrence of ev_c at the next pclk event:

```
wait {[..3]*{@ev_a; @ev_b}; @ev_c}@pclk
```

In the following example,

```
wait {@ev_a; [0..2]*@ev_b; @ev_c}@pclk
```

the **wait** action proceeds after any one of the three sequences sampled at consecutive sampling events:

- {@ev_a; @ev_c}
- {@ev_a; @ev_b; @ev_c}
- {@ev_a; @ev_b; @ev_b; @ev_c}

### See Also

— "~[ exp..exp ]" on page 340
— "cycle" on page 349
— "[ exp ]" on page 337
— "Temporal Operators and Constructs" on page 327

## 9.2.10 ~[ exp..exp ]

### Purpose

True match variable repeat operator

### Category

Expression

### Syntax

**~[[*from-exp*]..[*to-exp*]]** [***** ***temporal-expression***]

Syntax example:

```
~[2..4]*@pclk
```

## Parameters

| | |
|---|---|
| *from-exp* | An optional non-negative 32 bit numeric expression that specifies the minimum number of repetitions of the temporal expression. If the ***from-exp*** is missing, zero is used. |
| *to-exp* | An optional non-negative 32 bit numeric expression that specifies the maximum number of repetitions of the temporal expression. If the ***to-exp*** is missing, infinity is used. |
| *temporal-expression* | The temporal expression that is to be repeated some number of times within the ***from-expr**..**to-exp*** range. If "****temporal-expression***" is omitted, "* cycle" is assumed. |

## Description

You can use the true match repeat operator to specify a variable number of consecutive successes of a temporal expression.

True match variable repeat succeeds every time the subexpression succeeds. This expression creates a number of parallel repeat evaluations within the range.

True match repeat also enables specification of behavior over infinite sequences by repeating an infinite number of occurrences of a temporal expression. The expression ~[..]*TE is equivalent to:

- [0] or [1]*TE or [2]*TE...

This construct is mainly useful for maintaining information about past events. See "[ exp ]" on page 337.

The following are examples of both forms of variable repeats, using implicit and explicit from - to range expressions:

## Example 1

In the examples below, the {...;...} syntax specifies a temporal sequence. See "{ exp ; exp }" on page 335 for information about the ";" sequence operator.

The following temporal expression succeeds if A has occurred sometime during an earlier cycle:

```
{@A;~[..]}
```

The following temporal expression succeeds after any of the sequences {A}, {A; B}, {A; B; B}, or {A; B; B; B}:

```
{@A;~[..3]*@B}
```

## Example 2

The following temporal expression succeeds three pclk cycles after reset occurs, again at four pclk cycles after reset, and again five pclk cycles after reset (with reset also sampled at pclk):

```
{@reset; ~[3..5]} @pclk
```

### Example 3

The following temporal expression using the **and** temporal operator succeeds if A is followed at any time by B, or if A and B both occur during the same initial cycle:

```
{@A; ~[..]} and {[..]; @B}
```

NOTE— A more efficient way to write the above example is:

```
(@A and @B) or {@A; [..]; @B}
```

### See Also

— “[ exp..exp ]” on page 338
— “cycle” on page 349
— “[ exp ]” on page 337
— “Temporal Operators and Constructs” on page 327

## 9.2.11 =>

### Purpose

Temporal yield operator

### Category

Temporal expression

### Syntax

***temporal-expression1* => *temporal-expression2***

Syntax example:

```
@A => {[1..2]*@clk; @B}
```

### Parameters

| | |
|---|---|
| *temporal-expression1* | The first temporal expression. The second temporal expression is expected to succeed if this expression succeeds. |
| *temporal-expression2* | The second temporal expression. If the first temporal expression succeeds, this expression is also expected to succeed. |

### Description

The yield operator is used to assert that success of one temporal expression depends on the success of another temporal expression. The yield expression TE1 => TE2 is equivalent to (**fail** TE1) **or** {TE1 ; TE2}.

The yield expression succeeds without evaluating the second expression if the first expression fails. If the first expression succeeds, then the second expression must succeed in sequence.

Yield is typically used in conjunction with the **expect** struct member to express temporal rules.

The sampling event from the context applies to both sides of the yield operator expression. The entire expression is essentially a single temporal expression, so that

```
(TE1 => TE2)@sampling_event
```

is effectively

```
(TE)@sampling_event
```

where TE is the temporal expression made up of TE1 => TE2.

### Example

The following temporal expression succeeds if acknowledge occurs 1 to 100 cycles after request occurs. (The {...;...} syntax specifies a temporal sequence. See "{ exp ; exp }" on page 335 for information about the ";" sequence operator).

```
expect @request => {[..99]; @acknowledge};
```

### See Also

— "expect | assume" on page 360
— "Temporal Operators and Constructs" on page 327

## 9.2.12 detach

### Purpose

Detach a temporal expression

### Category

Temporal expression

### Syntax

detach(*temporal-expression*)

Syntax example:

```
@trans.end => detach({@trans.start; ~[2..5]})@pclk
```

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression to be independently evaluated. |

### Description

A detached temporal expression is evaluated independently of the expression in which it is used. It starts evaluation when the main expression does. Whenever the detached TE succeeds it emits an "implicit" event which will only be recognized by the main TE. The detached temporal expression inherits the sampling event from the main temporal expression.

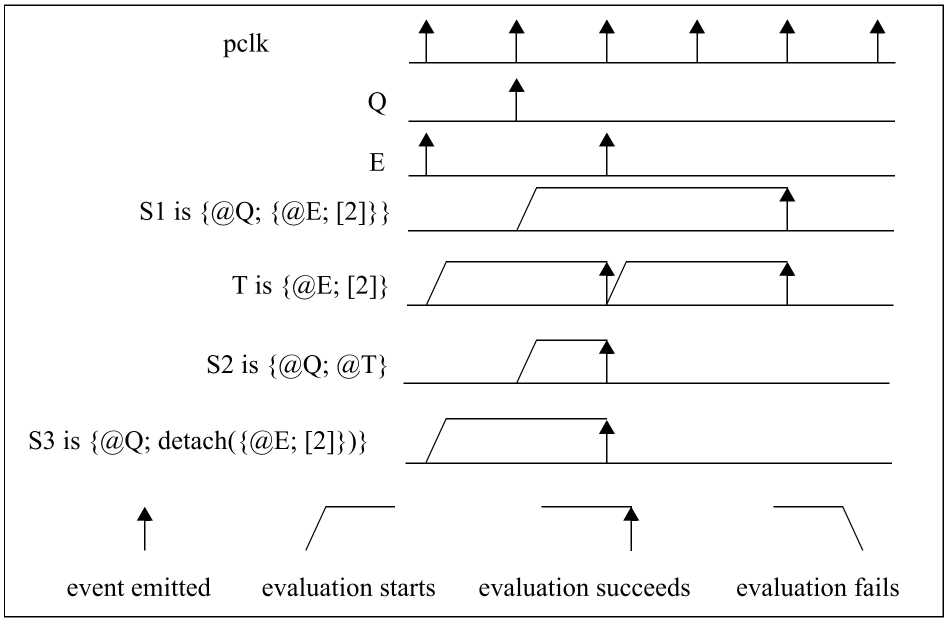
### Example 1

In the following example, both S1 and S2 start with @Q. However, the S1 temporal expression expects E to follow Q, while the S2 temporal expression expects E to precede Q by one cycle.

The **detach()** construct causes the temporal expressions to be evaluated separately. As a result, the S3 temporal expression is equivalent to the S2 expression. See Figure 9-9.

```
struct s {
    event pclk is @sys.pclk;
    event Q;
    event E;
    event T is {@E; [2]} @pclk;
    event S1 is {@Q; {@E; [2]}} @pclk;
    event S2 is {@Q; @T} @pclk;
    event S3 is {@Q; detach({@E; [2]})} @pclk;
};
```

**Figure 9-9—Examples Illustrating Detached Temporal Expressions**



### Example 2

Since a detached expression is evaluated independently and in parallel with the main temporal expression the two events below are not the same:

```
event ev_a is {@TE1; {@TE1; @TE2}};
event ev_b is {@TE1; detach({@TE1; @TE2})};
```

The first expression is equivalent to:

```
event ev_a is {@TE1; @TE1; @TE2};
```

While the second is equivalent to:

```
event ev_b is {@TE1; @ev_c};
event ev_c is {@TE1; @TE2};
```

### Example 3

The following two expressions are equivalent:

```
fail detach({@ev_a; @ev_b})
not({@ev_a; @ev_b})
```

### See Also

— "Temporal Operators and Constructs" on page 327

## 9.2.13 delay

### Purpose

Specify a simulation time delay

### Category

Temporal expression

### Syntax

**delay(*int*:** exp**)**

Syntax example:

```
wait delay(3);
```

### Parameters

| | |
|---|---|
| *int* | An integer expression or time expression no larger than 64 bits. The number specifies the amount of simulation time to delay. The time units are in the timescale used in the HDL simulator. |

### Description

Succeeds after a specified simulation time delay elapses. A callback occurs after the specified time. A delay of zero succeeds immediately.

Attaching a sampling event to **delay** has no effect. The **delay** ignores the sampling event and succeeds as soon as the delay period elapses.

NOTE— This expression is not legal in standalone mode. It can only be used if the ***e*** porgram is being run with an attached HDL simulator.

— The **delay** temporal expression is only supported for the cases:

```
wait delay(x);
```

```
    event e is {@a;delay(x)};
```

### Example 1

The following specifies a delay of 20 simulator time units:

```
wait delay(20);
```

### Example 2

The following specifies a delay of df*5 simulator time units:

```
wait delay(df*5);
```

### Example 3

The following use of the delay expression generates an error:

```
//    event del_b is {@set_a; delay(10)} @clk;    // Load-time error
```

### See Also

— "wait" on page 367
— "verilog time" on page 803
— "vhdl time" on page 829
— "Temporal Operators and Constructs" on page 327

## 9.2.14 @ unary event operator

### Purpose

Use an event as a temporal expression

### Category

Temporal expression

### Syntax

**@**[*struct-exp***.**]*event-type*

Syntax example:

```
wait @rst;
```

### Parameters

| | |
|---|---|
| *struct-exp.event-type* | The name of an event. This can be either a predefined event or a user-defined event, optionally including the name of the struct instance in which the event is defined. |

### Description

An event can be used as the simplest form of a temporal expression. The temporal expression @***event-type*** succeeds every time the event occurs. Success of the expression is simultaneous with the occurrence of the event.

The ***struct-exp*** is an expression that evaluates to the struct instance that contains the event instance. If no struct expression is specified, the default is the current struct instance.

NOTE— If a struct expression is included in the event name, the value of the struct expression must not change throughout the evaluation of the temporal expression.

### Examples

In the following, pclk is a temporal expression:

```
@pclk
```

The predefined sys.any event occurs at every tick. As a sampling event, use it as follows:

```
@sys.any
```

### See Also

— "event" on page 305
— "Predefined Events Overview" on page 309
— "Temporal Operators and Constructs" on page 327

## 9.2.15 @ sampling operator

### Purpose

Specify a sampling event for a temporal expression

### Category

Temporal expression

### Syntax

***temporal-expression @event-name***

Syntax example:

```
wait cycle @sys.reset;
```

## Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression. |
| *event-name* | The sampling event. |

## Description

Used to specify the sampling event for a temporal expression. The specified sampling event overrides the default sampling event.

Every temporal expression has a sampling event. The sampling event applies to all subexpressions of the temporal expression. It can be overridden for a subexpression by attaching a different sampling event to the subexpression.

A sampled temporal expression succeeds when its sampling event occurs with or after the success of the temporal expression.

The sampling event for a temporal expression is one of the following, in decreasing precedence order:

1) For any expression or subexpression, a sampling event specified with the @ binary event operator.
2) For a subexpression, the sampling event inherited from its parent expression.
3) For an expression in a TCM, the default sampling event of the TCM.
4) If none of the above applies, the predefined **sys.any** event.

## Examples

The reset event is sampled at the pclk event:

```
@reset @pclk
```

The reset event is sampled by the predefined **sys.any** event:

```
@reset @sys.any
```

Event ev_a occurs when the reset event occurs, sampled at the rclk event:

```
event ev_a is @reset @rclk;
```

The following is the same as **event** ev_b **is** @reset @sys.any:

```
event e_b is @reset;
```

## See Also

— "true(exp)" on page 350
— "event" on page 305
— "Predefined Events Overview" on page 309
— "Sampling Events Overview" on page 308
— "Temporal Operators and Constructs" on page 327

### 9.2.16 cycle

#### Purpose

Specify an occurrence of a sampling event

#### Category

Temporal expression

#### Syntax

cycle

Syntax example:

```
wait cycle;
```

#### Description

Represents one cycle of some sampling event. With no explicit sampling event specified, this represents one cycle of the sampling event from the context (that is, the sampling event from the overall temporal expression, or the sampling event for the TCM that contains the temporal expression). When a sampling event is specified, as in **cycle@*sampling-event,*** this is equivalent to **@*sampling-event*@*sampling-event***.

In the following, the event named sys.pclk is the sampling event for the TCM named proc(). The **wait cycle** action is the same as **wait** @**sys**.pclk.

```
proc() @sys.pclk is { wait cycle; };
```

#### Example 1

The following event definition replicates **sys**.clk as the local clk for the struct:

```
event clk is cycle @sys.clk;
```

This is equivalent to "**event** clk **is** @**sys**.clk @**sys**.clk". It is also equivalent to "**event** clk **is** @**sys**.clk", but more efficient.

#### Example 2

The following expression succeeds as soon as ev_a occurs:

```
m_tcm() @ev_s is {
    wait cycle @ev_a;
    out("Done");
};
```

#### See Also

— "event" on page 305
— "Predefined Events Overview" on page 309
— "Sampling Events Overview" on page 308
— "@ sampling operator" on page 347

— “Temporal Operators and Constructs” on page 327

### 9.2.17 true(exp)

#### Purpose

Boolean temporal expression

#### Category

Temporal expression

#### Syntax

**true(*bool*:** exp**)**

Syntax example:

```
event rst is true(reset == 1) @clk;
```

#### Parameters

| | |
|---|---|
| *bool* | A boolean expression. |

#### Description

Use a boolean expression as a temporal expression. Each occurrence of the sampling event causes an evaluation of the boolean expression. The boolean expression is evaluated only at the sampling point.

The temporal expression succeeds each time the expression evaluates to TRUE.

NOTE— The expression exp will be evaluated after pclk. Changes in ***exp*** after **true(*exp*)** @pclk has been evaluated will be ignored.

#### Example 1

The following causes the TCM to suspend until reset is high. The condition is checked for the first time at the first occurrence of clk after the **wait** is encountered; it is then checked every clk cycle after that. See “wait” on page 367.

```
notify_reset() @clk is {
    wait true(reset == 1);
    out("reset is 1");
};
```

#### Example 2

The temporal expression below succeeds when the boolean condition **sys**.number_of_packets == 5 evaluates to TRUE at the default sampling event. Execution of the TCM containing the **wait** action suspends until the boolean condition is true.

```
wait true(sys.number_of_packets == 5);
```

### See Also

— Boolean scalar type in “Scalar Types” on page 75
— “Sampling Events Overview” on page 308
— “Temporal Operators and Constructs” on page 327

## 9.2.18 change(exp), fall(exp), rise(exp)

### Purpose

Transition or edge temporal expression

### Category

Temporal expression

### Syntax

**change** | **fall** | **rise(*scalar***: exp**)** [**@*event-type***]

**change** | **fall** | **rise(’*HDL-pathname*’) @sim**

Syntax example:

```
event hv_c is change('top.hold_var')@sim;
```

### Parameters

| | |
|---|---|
| *scalar* | A boolean expression or an integer expression. |
| *event-type* | The sampling event for the expression. |
| *’HDL-pathname’* | An HDL object enclosed in single quotes (' '). |
| @sim | A special annotation used to detect changes in HDL signals. |

### Description

Detects a change in the sampled value of an expression.

The behavior of each of the three temporal expressions (**change**, **fall**, and **rise**) is described in Table 9-2, “Edge Condition Options”, on page 351.

**Table 9-2—Edge Condition Options**

| Edge Condition | Meaning |
|---|---|
| rise(*exp*) | Triggered when the expression changes from FALSE to TRUE. If it is an integer expression, the **rise()** temporal expression succeeds upon any change from x to y>x. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed. |
| fall(*exp*) | Triggered when the expression changes from TRUE to FALSE. If it is an integer expression, the **fall()** temporal expression succeeds upon any change from x to y<x. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed. |

**Table 9-2—Edge Condition Options *(continued)***

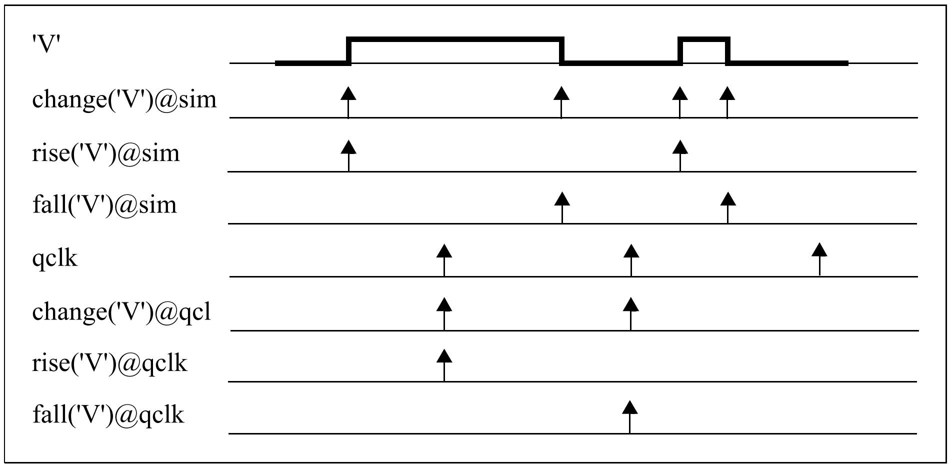
| Edge Condition | Meaning |
|---|---|
| change(*exp*) | Triggered when the value of the expression changes. The **change()** temporal expression succeeds upon any change of the expression. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed. |

The expression is evaluated at each occurrence of the sampling event, and is compared to the value it had at the previous sampling point. Only the values at sampling points are detected. The value of the expression between sampling points is invisible to the temporal expression.

A special notation, @**sim**, can be used in place of a sampling event for **rise**, **fall**, or **change** of HDL objects. If **@sim** is used, the HDL object is watched by the simulator. The **@sim** notation does not signify an event, but is used only to cause a callback any time there is a change in the value of the HDL object to which it is attached.

When a sampling event other than **@sim** is used, changes to the HDL object are detected only if they are visible at the sampling rate of the sampling event. In Figure 9-10 on page 352, evaluations of **rise**, **fall**, and **change** expressions for the HDL signal V are shown, with the sampling event **@sim** and with the sampling event **@qclk**. The qclk event is an arbitrary event that is emitted at the indicated points. The V signal rises and then falls between the second and third occurrences of event qclk. Since the signal's value is the same at the third qclk event as it was at the second qclk event, the **change('V')@qclk** expression does not succeed at the third qclk event.

**Figure 9-10—Effects of the Sampling Rate on Detecting HDL Object Changes**



When applied to HDL variables, the expressions examine the value after each bit is translated from the HDL four-value or nine-value logic representation to ***e*** two-value logic representation. Table 9-3, "Transition of HDL Values", on page 353 describes the default translation of HDL values to ***e*** values. The '@x' and '@z' HDL value modifiers can be used to override the default translation.

**Table 9-3—Transition of HDL Values**

| HDL Values | *e* Value |
|---|---|
| 0, X, U, W, L, - | 0 |
| 1, Z, H | 1 |

### Notes

— You can use the === operator to detect a change between two-state and four-state logic, as in the following:

```
change('top.hold_var' === two_state_hold_var)@clk;
```

— An ***e*** program ignores glitches that occur in a single simulation time slot. Only the first occurrence of a particular monitored event in a single simulation time slot is recognized by the ***e*** program. For example, if a signal transitions from 0 to 1 and back to 0 in the same time slot, the ***e*** program sees only the 0 to 1 transition; the 1 to 0 transition is ignored. For information on how to handle glitches, see "Simulation Time and Ticks" on page 312.

### Examples

The following defines an event that occurs at any change in the value of an HDL signal named top.clk:

```
event clk_c is change('top.clk')@sim;
```

The following defines an event that occurs when the boolean expression pkt_size > 20 changes from FALSE to TRUE:

```
event big_pkt is rise(pkt.size > 20);
```

The following defines an event that occurs at a fall in the value of an HDL signal named ~/top/reset, sampled by an event named rsmp:

```
event rst is fall('~/top/reset')@rsmp;
```

### See Also

— "Sampling Events Overview" on page 308
— "Using HDL Objects in Temporal Expressions" on page 322
— "'HDL-pathname'" on page 838
— "Temporal Operators and Constructs" on page 327

## 9.2.19 consume

### Purpose

Consume an occurrence of an event

### Category

Temporal expression

### Syntax

consume(@*event-type*)

Syntax example:

```
sync consume(@counter);
wait consume(@done)@sys.any;
```

### Parameters

| | |
|---|---|
| *event-type* | The name of the event that is to be consumed. |

### Description

Removes the occurrence of an event so that it is not available for other temporal expressions. The **consume** expression succeeds whenever the event occurs. If the event occurs more than once during any given cycle, all occurrences are consumed.

After an event occurrence is consumed, that occurrence will not be recognized by any temporal expression during the current tick, unless the event is emitted again.

An event cannot be consumed by more then one consume expression. Care should be used to avoid creating race conditions between multiple events that use an event that is consumed.

### Notes

— The **consume(@*event-type*)** temporal expression can only be used in one of the following time consuming actions:

**sync consume(@*event-type*) [@*sampling_event*]**;

**wait consume(@*event-type*) [@*sampling_event*]**;

When an optional sampling event is specified, then the **sync** or **wait** action finishes with the first occurrence of the sampling event after the **consume** succeeds.
If no sampling event is specified, the default sampling event of the TCM applies.

— An event can either be used in a **consume(@*event-type*)** expression or be used in another temporal expression, but not in both.

### Example

The following code shows how you can use consume() to handle concurrent requests from multiple clients in an orderly manner.

The example enables the following behaviors:

— The requests are granted on a First-In-First-Out basis.
— A client may hold the grant for several cycles.
— The server can accumulate requests during several cycles.
— Multiple clients requests can be granted sequentially at the same ***e*** time.

In this example there are four client structs and one server struct. The server ensures that all requests are granted and that there are no simultaneous grants.

When multiple clients issue a request at the same time the server is using a counter to keep track of the number of requests. The server consumes all the requests, and then issues a grant event. The first client to issue a request consumes the grant, making it unavailable to the other clients. When this client is done with the grant it issues a done event. The server consumes the done event and issues a release event. The release event is consumed by the client. The server repeats this process until the request counter is zero.

```
<'
struct server {
    event clk is rise('top.clock');
    event request;
    event grant;
    event done;
    event release;
    !req_counter: int;
    on request {
        req_counter += 1;
            out("req_counter = ", req_counter);
    };
    serv() @clk is {
        while TRUE {
            if (req_counter == 0 || now @request) {
                sync consume(@request);
                out("Requests consumed...");
            };
            req_counter -= 1;
            emit grant;
            sync consume(@done);
            emit release;
        };
    };

    run() is also {start serv();};
};
struct client {
    id: string;
    s: server;
    handshake() @s.clk is {     // Zero delay handshake
        out(id, ": Starting handshake at time ", sys.time);
        emit s.request;
        sync consume(@s.grant);
        out(id, ": Granted, releasing");
        emit s.done;
        sync consume(@s.release);
    };

    run() is also {start handshake();};
};

extend sys {
    event clk;

    s: server;
    c1: client; keep { c1.s==s; c1.id=="client 1" };
    c2: client; keep { c2.s==s; c2.id=="client 2" };
    c3: client; keep { c3.s==s; c3.id=="client 3" };
    c4: client; keep { c4.s==s; c4.id=="client 4" };

    go()@any is {
        for i from 0 to 40 do {
        wait cycle;
        emit clk;
        };
    stop_run();
    };

    run() is also {start go()};
};
'>
```

### Result

```
Running the test ...
client 1: Starting handshake at time 1
req_counter = 1
client 2: Starting handshake at time 1
req_counter = 2
client 3: Starting handshake at time 1
req_counter = 3
client 4: Starting handshake at time 1
req_counter = 4
Requests consumed...
client 1: Granted, releasing
client 2: Granted, releasing
client 3: Granted, releasing
client 4: Granted, releasing
```

### See Also

— "event" on page 305
— "Temporal Operators and Constructs" on page 327

## 9.2.20 exec

### Purpose

Attach an action block to a temporal expression

### Category

Temporal expression side effect

### Syntax

*temporal-expression* **exec** ***action*****;** ...

Syntax example:

```
wait @watcher_on exec {print watcher_status_1;};
```

## Parameters

| | |
|---|---|
| *temporal-expression* | The temporal expression that invokes the action block. |
| *action* | A series of actions to perform when the expression succeeds. |

## Description

Invokes an action block when a temporal expression succeeds. The actions are executed immediately upon the success of the expression, but not more than once per tick.

To support extensibility of your ***e*** code, use method calls in the **exec** action block rather than calling the actions directly.

The usage of **exec** is similar to the **on** struct member, except that:

— Any temporal expression can be used as the condition for **exec**, while only an event can be used as the condition for **on.**
— **exec** must be used in the context of a temporal expression in a TCM or an event definition, while **on** can only be a struct member.

You cannot attach an **exec** action to a first match variable repeat expression:

```
([7..10] *@b) exec {out("in exec")}  -- is not allowed
```

However, you can attach an **exec** action to the repeat expression of a first match variable repeat expression as follows:

```
[7..10] *@b exec {out("in exec")}    -- is allowed
[7..10] * (@b exec {out("in exec")}) -- is allowed
```

NOTE— The two expressions above are equivalent. They will both execute the **exec** action once for each occurrence of b.

An **exec** action cannot be attached to an implicit repeat expression:

```
expect @e => {
    [..20];    -- exec is not allowed here
    @a;
};
```

You must make the implicit repeat expression explicit in order to attach an **exec** action:

```
expect @e => {
    [..20] * (cycle exec {out("in exec")}); -- is allowed
    @a;
};
```

NOTE— The action block cannot contain any time-consuming actions.

## Example

The following code maintains a pipeline of instruction instances.

```
struct pipelined_alu {
    instructions: list of inst;
```

```
        ck_inst: inst;
        event alu_watcher is {
            rise('inst_start') exec {
                    var i: inst = new;
                    instructions.add(i);
                };
            [..];
            rise('inst_end') exec {
                    ck_inst = instructions.pop()
                }
        }@sys.alu.aclk;
    };
```

### See Also

— Action blocks, described in "Actions" on page 14
— "on" on page 359
— "Temporal Operators and Constructs" on page 327

# 10 Temporal Struct Members

In addition to the **event** struct member (see "event" on page 305), there are two struct members used for temporal coding. These struct members are described in this chapter:

— "on" on page 359
— "expect | assume" on page 360

## See Also

— "Invoking Methods" on page 474
— Chapter 8, "Events"
— Chapter 9, "Temporal Expressions"
— Chapter 11, "Time-Consuming Actions"

## 10.1 on

### Purpose

Specify a block of actions that execute on an event

### Category

Struct member

### Syntax

**on** *event-type* **{***action***;** ...**}**

Syntax example:

```
on xmit_ready {transmit();};
```

### Parameters

| | |
|---|---|
| *event-type* | The name of an event that invokes the action block. |
| *action; ...* | A block of non-time-consuming actions. |

### Description

Defines a struct member that executes a block of actions immediately whenever a specified event occurs. An **on** struct member is similar to a regular method except that the action block for an **on** struct member is invoked immediately upon an occurrence of the event. An **on** action block is executed before TCMs waiting for the same event.

The **on** action block is invoked every time the event occurs. The actions are executed in the order in which they appear in the action block.

You can extend an **on** struct member by repeating its declaration, with a different action block. This has the same effect as using **is also** to extend a method.

The **on** struct member is implemented as a method, named **on**_*event-type*(). You can invoke the action block without the occurrence of the event by calling the **on**_*event-type*() method. You can extend the **on**_*event-type*() method like any other method, using **is**, **is also**, **is only**, or **is first**.

### Notes

— The named event must be local to the struct in which the **on** is defined.
— The **on** action block must not contain any time-consuming actions.

### Example 1

```
struct cnt_e {
    event ready;
    event start_count;
    on ready {sys.req = 0};
    on start_count {
        sys.count = 0;
        sys.counting = 1;
        out("Starting to count");
    };
};
```

### Example 2

The following example shows how to invoke an **on** action block with an event that is defined in a different struct, by defining a local event that uses the nonlocal event as its temporal expression.

```
<'
extend sys {
    event global_clk is change('top.clk') @sim;
                                // system clock
    card_i: card;
};
struct card {
    event clk is cycle @sys.global_clk;
                               // replicates the global
    on clk {                   // clock locally
        out("local clock tick");
    };
};
'>
```

### See Also

— “event” on page 305
— Action block, in “Actions” on page 14
— “method [@event] is also | first | only | inline only” on page 467

## 10.2 expect | assume

### Purpose

Define a temporal behavioral rule

### Category

Struct member

### Syntax

**expect** | **assume** [*rule-name* **is** [**only**]] *temporal-expression*
[**else dut_error(***string-exp***)**]

or

**expect** | **assume** *rule-name*

Syntax example:

```
expect @a => {[1..5];@b} @clk;
```

### Parameters

| | |
|---|---|
| *rule-name* | An optional name that uniquely identifies the rule from other rules or events within the struct. You can use this name to override the temporal rule later on in the code or change from **expect** to **assume** or vice versa. |
| *temporal-expression* | A temporal expression that is always expected to succeed. Typically involves a temporal yield (=>) operation. |
| *string-exp* | A string or a method that returns a string. If the temporal expression fails, the string is printed, or the method is executed and its result is printed. |

### Description

Both the **expect** and **assume** struct members define temporal rules. If the temporal expression fails at its sampling event, the temporal rule is violated and an error is reported. If there is no **dut_error()** clause, the rule name is printed.

If you are not using an ***e*** program linked with a formal verification tool, you can use **expect** and **assume** interchangeably to define temporal rules with no difference in behavior.

When using an ***e*** program linked with a formal verification (FV) tool, you can use **assume** to identify temporal sequences that the FV tool must not apply to the DUT. Use **expect** to identify temporal sequences that the FV tool must verify. In this manner, you avoid applying illegal inputs to the DUT and you improve the performance of the FV tool by requiring it to verify only a subset of the temporal rules defined.

Once a rule has been defined, it can be modified using the **is only** syntax and it can be changed from an **expect** to an **assume** or vice versa. You can perform multiple verification runs either varying the rules slightly or using the same set of rules in different **expect**/**assume** combinations. See the examples below for information on how to do this.

NOTE— The **is also**, **is undefined**, and **is empty** forms are not supported for this construct.

### Example 1

This example defines an **expect**, “bus_cycle_length”, which requires that the length of the bus cycle be no longer than 1000 cycles.

```
struct bus_e {
    event bus_clk is change('top.b_clk') @sim;
    event transmit_start is rise('top.trans') @bus_clk;
    event transmit_end is rise('top.transmit_done') @bus_clk;
    event bus_cycle_length;
    expect bus_cycle_length is
        @transmit_start => {[0..999];@transmit_end} @bus_clk
        else dut_error("Bus cycle did not end in 1000 cycles");
};
```

### Result

If the bus cycle is longer than 1000 cycles, the following message will be issued.

```
-------------------------------------------------------
   *** Dut error at time 1000
        Checked at line 7 in @expect_msg
        In bus_e-@0:

bus_cycle_length: Bus cycle did not end in 1000 cycles
-------------------------------------------------------
Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred
```

### Example 2

In this example, the “bus_e” struct from Example 1 on page 361 is extended and two subtypes are created, “Slow” and “Fast”. For the “Fast” subtype, the bus_cycle_length is modified to be shorter. ***e*** inheritance allows subtypes to override rules defined in the base struct using the **is only** syntax.

```
extend bus_e {
   type: [Slow, Fast];
   when Fast bus_e {
      expect bus_cycle_length is only
         @transmit_start => {[0..99]; @transmit_end} @bus_clk
         else dut_error("Bus cycle did not end \
           within 100 cycles");
   };
};
```

### Example 3

In this example, the “bus_e” struct from Example 1 on page 361 and Example 2 on page 362 is extended. The bus cycle rule is changed from an **expect** rule to an **assume** rule.

```
<'
struct bus_e {
    event bus_clk is change('top.b_clk') @sim;
    event transmit_start is rise('top.trans') @bus_clk;
    event transmit_end is rise('top.transmit_done') @bus_clk;
    expect bus_cycle_length is
        @transmit_start => {[0..999];@transmit_end} @bus_clk
        else dut_error("Bus cycle did not end in 1000 cycles");
};
```

```
extend bus_e {
    assume bus_cycle_length;
};
extend sys {
    bus_e;
};
'>
```

### Example 4

In the following example, two **expect** statements are used to specify that the “transmit_end” event must occur within three to six “bus_clk” cycles after the “transmit_start” event. If fewer than three cycles occur, the “DRV_SHORT” err_id is passed to the “m_error()” method. If more than six cycles occur, the “DRV_LONG” err_id is passed to the method. The “m_error()” method adds one to the “error_count” value, and returns a string that states which type of error occurred.

```
<'
type watcher_errors :[DRV_LONG, DRV_SHORT];
    // Enumerated error conditions

extend sys {
    event clk is @sys.any;
    event start_drive;
    event stop_drive;
    !error_count: int; // Counts number of errors
    my_watcher: watcher;
};

struct watcher {
    expect @sys.start_drive =>
        [2] * not @sys.stop_drive @sys.clk
        else dut_error("Drive Rule 1: ",
        m_error(DRV_SHORT));
    expect @sys.start_drive =>
        {[..5];@sys.stop_drive}@sys.clk
        else dut_error("Drive Rule 1: ",
        m_error(DRV_LONG));
    m_error(err_id: watcher_errors):string is {
        case err_id {
            DRV_LONG: {
                result = "Driving strobe too long";
                sys.error_count += 1;
            };
            DRV_SHORT: {
                result = "Driving strobe too short";
                sys.error_count += 1;
            };
            default: { result = "No error"; };
        };
    };
};
'>
```

### See Also

— Chapter 9, “Temporal Expressions”, in particular, “=>” on page 342

# 11 Time-Consuming Actions

This chapter contains the following sections:

— “Synchronization Actions” on page 365
— “Concurrency Actions” on page 368

### See Also

— “Invoking Methods” on page 474
— Chapter 8, “Events”
— Chapter 9, “Temporal Expressions”
— Chapter 10, “Temporal Struct Members”

## 11.1 Synchronization Actions

The following actions are used to synchronize temporal test activities within an ***e*** program and between the DUT and the ***e*** program:

— “sync” on page 365
— “wait” on page 367

### 11.1.1 sync

#### Purpose

Synchronize an executing TCM

#### Category

Action

#### Syntax

**sync** [***temporal-expression***]

Syntax example:

```
sent_data_tcm();
sync;
```

#### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression that specifies what the TCM synchronizes to. |

#### Description

Suspends execution of the current TCM until the temporal expression succeeds. Evaluation of the temporal expression starts immediately when the **sync** action is reached. If the temporal expression succeeds within the current tick, the execution continues immediately.

If no temporal expression is provided, the TCM synchronizes to its default sampling event. The TCM suspends until the occurrence of its sampling event, or continues immediately if the sampling event succeeds in the current tick.

You can use the **sync** action after a call to another TCM to align the continuation of the calling TCM with its sampling event when the called TCM returns.

Execution of a thread is atomic: it cannot be interrupted except by a **sync** action or a **wait** action. When one of those actions is encountered, control can be passed from the TCM to other TCMs.

The **sync** action is similar to the **wait** action, except that a **wait** action always requires at least one cycle of the TCM's sampling event before execution can continue. With a **sync** action, execution can continue in the same tick.

### Example

In the following example, the **wait** action in the "driver()" TCM causes at least a one-cycle delay, since the **true()** temporal expression is evaluated for the first time at the next occurrence of the sampling event. The **wait** consumes one occurrence of the "clk" event, and then execution of the TCM continues at the second occurrence of "clk".

On the other hand, the **sync** action in the "shadow()" TCM does not result in a delay if its **true** temporal expression succeeds immediately. Execution of the TCM continues at the next occurrence of the "clk" event.

```
<'
struct data_drive {
    event clk is rise('top.clk') @sim;
    data: list of int;
    driver() @clk is {
        for each in data {
            wait true('top.data_ready'== 1);
            // Will not fall through, even if the condition
            // holds when the wait is reached.
            'top.in_reg' = it;
        };
        stop_run();
    };
    shadow() @clk is {
        while TRUE {
            sync true('top.data_ready' == 0);
            // If the condition holds, the sync falls through.
            out("Shadow read ", 'top.in_reg');
            wait cycle;
            // This wait is necessary to prevent
            // an infinite loop.
        };
    };
    run() is also {
        start driver();
        start shadow();
    };
};
'>
```

### See Also

— "event" on page 305

- “Sampling Events Overview” on page 308
- Chapter 9, “Temporal Expressions”
- “wait” on page 367

## 11.1.2 wait

### Purpose

Wait until a temporal expression succeeds

### Category

Action

### Syntax

**wait** [[**until**] ***temporal-expression***]

Syntax example:

```
wait [3]*cycle;
```

### Parameters

| | |
|---|---|
| *temporal-expression* | A temporal expression that specifies what the TCM is to wait for. |

### Description

Suspend execution of the current time-consuming method until a given temporal expression succeeds. If no temporal expression is provided, the TCM waits for its default sampling event. The **until** option is for users who find that it clarifies what the **wait** action does. The option has no effect on the results of the action.

When a VHDL or Verilog simulator is linked to an ***e*** program, the syntax **wait delay**(***exp***) can be used to wait for a specific simulation time period. Because not all simulators support delay values greater than 32 bits, the value of the expression in **wait delay**(***exp***) cannot exceed 32 bits. A **wait delay**(***exp***) is influenced by the timescale. See “verilog time” on page 803 and “vhdl time” on page 829 for more information on how the ***e*** program determines the timescale.

The TCM cannot continue during the same cycle in which it reaches a **wait**, unless the temporal expression evaluates to 0. That is, if the temporal expression evaluates to “[0] * something”, execution can continue in the same cycle.

If the **wait** action’s temporal expression contains a variable subexpression, such as “wait [var1 + var2] * cycle”, the subexpression is only evaluated once, when the wait is encountered. Any changes in the value of the subexpression during subsequent cycles are ignored.

Execution of a thread is atomic: it cannot be interrupted except by a **wait** action or a **sync** action. When one of those actions is encountered, control can be passed from the TCM to other TCMs.

The **wait** action is similar to the **sync** action, except that a **wait** action always requires at least one cycle of the TCM’s sampling event before execution can continue (unless a wait of zero is specified). With a **sync** action, execution can continue immediately upon encountering the **sync**, if the temporal expression succeeds at that time See“sync” on page 365 for an example comparing the behavior of **sync** and **wait**.

NOTE— The cycle-based simulator SpeedSim does not support the **wait delay**(***exp***) action.

### Example 1

Several examples of temporal expressions for **wait** actions are shown below:

```
wait [3]*cycle;
    // Continue on the fourth cycle from now
wait delay(30);
    // Wait 30 simulator time units
wait [var1 + var2]*cycle;
    // Calculate the number of cycles to wait
wait until [var1 + var2]*cycle;
    // Same as wait [var1 + var2]*cycle
wait true(sys.time >= 200);
    // Continue when sys.time is greater than or equal to 200
wait cycle @sys.reset;
    // Continue on reset even if it is not synchronous with
    // the TCMs default sampling event
wait @sys.reset;
    // Continue on the next default sampling event after reset
```

### Example 2

In the following example, the **wait** action in the “driver()” TCM causes a one-cycle delay even if the **true** temporal expression succeeds immediately. The **wait** consumes one occurrence of the “clk” event, and then execution of the TCM continues at the second occurrence of “clk”.

```
<'
struct data_drive {
    event clk is rise('top.clk') @sim;
    data: list of int;
    driver() @clk is {
        for each in data {
            wait true('top.data_ready'== 1);
            'top.in_reg' = it;
        };
        stop_run();
    };
    run() is also {
        start driver();
    };
};
'>
```

### See Also

- “event” on page 305
- “Sampling Events Overview” on page 308
- Chapter 9, “Temporal Expressions”
- “sync” on page 365

## 11.2 Concurrency Actions

The actions that control concurrent execution of time-consuming methods are described in this section:

— "all of" on page 369
— "first of" on page 370

Both of these actions create parallel action blocks which might start or call TCMs. The first action awaits completion of all "branches", while the second terminates at the first completion of any "branch".

Control of individual branches or TCMs can also be accomplished using predefined methods of the predefined **scheduler** struct.

## 11.2.1 all of

### Purpose

Execute action blocks in parallel

### Category

Action

### Syntax

**all of {{*action*; ...}; ... }**

Syntax example:

```
all of { {block_a}; {block_b}; };
```

### Parameters

| | |
|---|---|
| **{*action*; ...}; ...** | Action blocks that are to execute concurrently. Each action block is a separate branch. |

### Description

Execute multiple action blocks concurrently, as separate branches of a fork. The action following the **all of** action will be reached only when all branches of the **all of** have been fully executed. All branches of the fork are automatically joined at the termination of the **all of** action block.

### Example 1

Execute the following three TCMs concurrently, and continue after they all have finished:

```
all of {
    {check_bus_controller();};
    {check_memory_controller();};
    {wait cycle; check_alu();};
};
```

### Example 2

The **all of** construct can be used to wait for several events, no matter what order they arrive in. This can be used as an AND relation between events as shown below.

```
<'
extend sys {
    event aclk is @any;
    event a;
    event b;
    detect_all() @aclk is {
        all of {
            { wait @a; };
            { wait @b; };
        };
        out("Both a and b occurred");
    };
};
'>
```

### See Also

— "first of" on page 370

## 11.2.2 first of

### Purpose

Execute action blocks in parallel

### Category

Action

### Syntax

**first of {{*action*; ...}; ... }**

Syntax example:

```
first of { {wait [3]*cycle@ev_a}; {wait @ev_e; }; };
```

### Parameters

| | |
|---|---|
| **{*action*; ...}; ...** | Action blocks that are to execute concurrently. Each action block is a separate branch. |

### Description

Execute multiple action blocks concurrently, as separate branches of a fork. The action following the **first of** action will be reached when any of the branches in the **first of** has been fully executed. All branches of the fork are automatically joined at the termination of the **first of** action block.

The parallel branches can be thought of as racing each other until one completes. Once one branch terminates, the ***e*** program terminates the execution of each of the other branches.

When two branches finish executing during the same cycle, it is not possible to determine which will prevail. One will complete successfully and the other will terminate.

### Example 1

The **first of** construct can be used in order to wait for one of several events. This can be used as an OR relation between events:

```
<'
extend sys {
    event c;
    event d;
    event fclk is @any;
    detect_first() @fclk is {
        first of {
            { wait @c; };
            { wait @d; };
        };
        out("Either c or d occurred");
    };
};
'>
```

### Example 2

The **all of** and **first of** actions can be used together to combine **wait** actions. In the following, the **first of** action block terminates when “event2” is seen:

```
<'
struct tcm_struct {
    event clk is @sys.any;
    event event1;
    event event2;
    main() @clk is {
        all of {
            {wait [10] * cycle;
             emit event1;
             emit event2;
             out("Branch #1 done");};
            {first of {
                {wait @event1; out("Branch #2-1 done"); };
                {wait @event2; out("Branch #2-2 done"); };
                 // One of the branches will never print
                };
            };
        };
        stop_run();
    };
};
'>
```

### Example 3

In the following example, **first of** is used to create two branches, one of which continues after a “sys.reset” event, and the other of which calls a method named “sys.pdata()” and then waits one cycle. The number of cycles required by the “pdata()” method is the main factor in determining which branch finishes first.

```
<'
struct mv_data {
    data: int;
```

```
        mdata() @sys.clk is {
            first of {
                {wait cycle @sys.reset;};
                {sys.pdata(); wait cycle;};
            };
            stop_run();
        };
        run() is also {
            start mdata();
        };
    };
    '>
```

### See Also

— “all of” on page 369

# 12 Coverage Constructs

This chapter contains the following sections:

## 12.1 Defining Coverage Groups: cover

### Purpose

Define a coverage group

### Category

Struct member

### Syntax

**cover** ***event-type*** [**using** ***coverage-group-option***, ...] **is** {***coverage-item-definition***; ...}**;**

**cover** ***event_type*** **is empty**;

Syntax example:

```
cover inst_driven is {
    item opcode;
    item op1;
    cross opcode, op1;
};
```

## Parameters

| | |
|---|---|
| *event-type* | The name of the group. This must be the name of an event type defined previously in the struct. The event must not have been defined in a subtype.<br><br>The event is the sampling event for the coverage group. Coverage data for the group is collected every time the event occurs.<br><br>The full name of the coverage group is ***struct-exp.event-name***. The full name must be specified for the coverage methods. |
| *coverage-group-option* | The coverage group options listed in Table 12-1 can be specified with the **using** keyword.<br><br>Each coverage group can have its own set of options. The options can appear in any order after the **using** keyword. |
| *coverage-item-definition* | The definition of a coverage item. Coverage items are described in "Defining Basic Coverage Items" on page 378. |
| **is also** | See "Extending Coverage Groups" on page 412. |
| **is empty** | The **empty** keyword can be used to define an empty coverage group that will be extended later, using a **cover is also** struct member with the same name. |

**Table 12-1—Coverage Group Options**

| Option | Description |
|---|---|
| **no_collect** | This coverage group is not displayed in coverage reports and is not saved in the coverage files. This option enables tracing of coverage information and enables event viewing with **echo event**, without saving the coverage information. |
| **count_only** | This option reduces memory consumption because the data collected for this coverage group is reduced. You cannot do interactive, post-processing cross coverage of items in **count_only** groups. The coverage configuration option **count_only** sets this option for all coverage groups. |
| **text**=*string* | A text description for this coverage group.<br><br>This can only be a quoted string, not a variable or expression. The text is shown at the beginning of the information for the group in the coverage report. |
| **when**=***bool-exp*** | The coverage group is sampled only when ***bool-exp*** is TRUE. The ***bool-exp*** is evaluated in the context of the parent struct. |
| **global** | A global coverage group is a group whose sampling event is expected to occur only once. If the sampling event occurs more than once, a DUT error is issued. If items from a global group are used in interactive cross coverage, no timing relationships exist between the items. |

**Table 12-1—Coverage Group Options *(continued)***

| Option | Description |
|---|---|
| **radix=DEC\|HEX\|BIN** | Buckets for items of type **int** or **uint** are given the item value ranges as names.This option specifies which radix the bucket names are displayed in.<br><br>The global **print** radix option does not affect the bucket name radix.<br><br>Legal values are DEC (decimal), HEX (hexadecimal), and BIN (binary). The value must be in upper case letters.<br><br>If the **radix** is not used, **int** or **uint** bucket names are displayed in decimal. |
| **weight=*uint*** | This option specifies the grading weight of the current group relative to other groups. It is a nonnegative integer with a default of 1. |

## Description

Defines a coverage group. A coverage group is struct member that contains a list of data items for which data is collected over time.

NOTE— Unless you turn on coverage mode, no coverage results are collected even if cover groups and items are defined. Use the cover configuration option to turn on coverage, as in the following example.

```
extend sys {
    setup() is also {
        set_config(cover, mode, on_interactive);
    };
};
```

Once coverage items have been defined in a coverage group, you can use them to define special coverage group items called **transition** and **cross** items. See "Defining Transition Coverage Items" on page 403 and "Defining Cross Coverage Items" on page 396 for information about those coverage items.

The **is** keyword is used to define a new coverage group. See "Extending Coverage Groups" on page 412 for information on using **is also** to extend an existing coverage group.

All basic items in a coverage group are enabled for **echo event**.

Coverage groups should not be initially defined in **when** constructs, although they can be extended in **when** constructs.

If you extend a coverage group in a **when** construct by adding a **per_instance** item, then the instances refer only to the when subtype. If you define a **per_instance** item in a base type and then add additional items under **when** construct, then the cover group instances refer to the base type, and the cover item values refer to the when subtype. See "Coverage Per Instance" on page 384.

### Example 1

A coverage group named “inst_driven” is defined in the example below. The sampling event “inst_driven” is declared earlier in the same struct. The coverage group contains definitions of three basic coverage items named “opcode”, “op1”, and “op2”.

```
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
type cpu_reg: [reg0, reg1, reg2, reg3];
struct inst {
    opcode: cpu_opcode;
    op1: cpu_reg;
    op2: byte;
    event inst_driven;
    cover inst_driven is {
        item opcode;
        item op1;
        item op2;
    };
};
```

### Example 2

The code below contains examples of the coverage group options.

```
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
type cpu_reg: [reg0, reg1, reg2, reg3];
struct inst {
    cache: c_mem;
    opcode: cpu_opcode;
    event info;
    event data_change;
    cover data_change using no_collect is {
        item data: uint(bits:16) = cache.data;
    };
    cover info is {
        item opcode;
    };
};
type memory_mode: [full, partial];
type cpu_state: [START, FETCH1, FETCH2, EXEC];
struct cpu {
    memory: memory_mode;
    init_complete: bool;
    event state_change;
    event reset_event;
    cover state_change using text = "Main state-machine",
        when = (init_complete == TRUE) is {
        item st: cpu_state = 'top.cpu.main_cur_state';
    };
    cover reset_event using global is {
        item memory;
    };
};
```

The effects of the options in the example above are:

— **no_collect** option: For the “data_change” group, do not save coverage data, but provide data for **show event**.
— **text** option: The text “Main state machine” appears at the beginning of the data for the group in the ASCII coverage report.
— **when** option: Coverage is collected for “st” when the “state_change” event occurs and “init_complete” is TRUE.
— **global** option: The “reset_event” is expected to occur only once. If it occurs more than once, a DUT error is issued.

### Example 3

The code below shows the **radix** coverage group option.

```
cover done using radix = HEX is {
    item len: uint (bits: 3) = sys.len;
    item data: byte = data using
        ranges = {range([0..0xff], "", 4)},
        radix = HEX;
    item mask: uint (bits: 2) = sys.mask using radix = BIN;
};
```

For the “len” item, the bucket names are: 0x0, 0x1, ... 0x7 (using the HEX radix specified for the group).

For the “data” item, the bucket names are: [0x0..0x03], [0x04..0x07], ... [0xfc..0xff] (using the HEX radix specified for the item).

For the “mask” item, the bucket names are 0b00, 0b01, 0b10, and 0b11 (since the **radix = BIN** option is used for this item to override the group’s HEX radix.)

### Example 4

The code below shows the **weight** coverage group option.

```
cover done using weight = 3 is {
    item len: uint (bits: 3) = sys.len;
    item data: byte = data;
    item mask;
};
```

The “done” coverage group is assigned a weight of 3. If there are 10 other coverage groups that all have default weights of 1, the “done” group contributes (3/13)*grading(done) to the “all” grade.

### Example 5

The code below shows how to use the **empty** coverage group keyword.

```
<'
struct inst {
    cover done is empty;
};
'>
<'
extend inst {
    cover done is also {
        item len: uint (bits: 3) = sys.len;
        item data: byte = data;
```

```
            item mask;
        };
    };
    '>
```

### Example 6

The code below shows how to define coverage items in a **when** construct, by extending a coverage group defined previously but initially left empty.

```
    <'
    struct inst {
        size: [WIDE, REG];
        event done;
        cover done is {};
    };
    '>
    <'
    extend inst {
        when WIDE inst {
            cover done is also {
                item len: uint (bits: 3) = sys.len;
                item data: byte = data;
                item mask;
            };
        };
    };
    '>
```

### See Also

- "Defining Basic Coverage Items" on page 378
- "Defining Cross Coverage Items" on page 396
- "Defining Transition Coverage Items" on page 403
- "Extending Coverage Groups" on page 412
- Chapter 8, "Events"
- Chapter 4, "Structs, Fields, and Subtypes"
- The **coverage** option of "set_config()" on page 766

## 12.2 Defining Basic Coverage Items

### 12.2.1 Overview

The **item** constuct is used to

### 12.2.2 item

#### Purpose

Define a coverage item

#### Category

Coverage group item

### Syntax

**item *item-name***[**:*type***=***exp***] [**using *coverage-item-option***, ...]

Syntax example:

```
cover inst_driven is {
    item op1;
    item op2;
    item op2_big: bool = (op2 >= 64);
    item hdl_sig: int = 'top.sig_1';
};
```

### Parameters

| | |
|---|---|
| *item-name* | The name you assign to the coverage item.<br><br>If you do not specify the optional ***type***=***exp***, the value of the field named ***item-name*** is used as the coverage sample value. The field may be a scalar not larger than 32 bits, or a string.<br><br>If you specify the optional ***type***=***exp***, the value of the expression is used as the coverage sample value. |
| *type* | The type of the item. The type expression must evaluate to a scalar not larger than 32 bits, or a string. |
| *exp* | The expression is evaluated at the time the whole coverage group is sampled. This value is used for the item. |
| *coverage-item-option* | Coverage item options are listed in Table 12-2. The options can appear in any order after the **using** keyword. |

**Table 12-2—Coverage Item Options**

| Option | Description |
|---|---|
| **per_instance** | Coverage data is collected and graded for all the other items in a separate listing for each bucket of this item. This option can only be used for basic items (not for cross or transition items, or items whose ranges are not known at generation time). |
| **no_collect** | This coverage item is not displayed in coverage reports and is not saved in the coverage files. This option enables tracing of coverage information and enables event viewing with **echo event**, without saving the coverage information. |
| **text=***string* | A text description for this coverage item. This can only be a quoted string, not a variable or expression. In the ASCII coverage report the text is shown along with the item name at the top of the coverage information for the item. |

**Table 12-2—Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **when**=***bool-exp*** | The item is sampled only when ***bool-exp*** is TRUE. The ***bool-exp*** is evaluated in the context of the parent struct.<br><br>The sampling is done at run time. |
| **at_least**=***num*** | The minimum number of samples for each bucket of the item. Anything less than ***num*** is considered a hole.<br><br>This option cannot be used with string items or for unconstrained integer items (items that do not have specified ranges).<br><br>You cannot specify a negative number. The default is 1. |

**Table 12-2—Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **ranges** = {**range(*parameters*);**…} | Create buckets for this item's values or ranges of values. This option cannot be used for string items.<br><br>The **range()** has up to four parameters. The parameters specify how the values are separated into buckets. The first parameter, ***range***, is required. The other three are optional. The syntax for range options is:<br><br>**range(*range***: range, ***name***: string, ***every-count***: int, ***at_least-num***: int**)**<br><br>The parameters are:<br><br>• *range*<br><br>The range for the bucket. It must be a literal range such as "[1..5]", of the proper type. Even a single value must be specified in brackets (for example "[7]"). If you specify ranges that overlap, values in the overlapping region go into the first of the overlapping buckets. The specified range for a bucket is the bucket name. That is, the buckets above are named "[1..5]" and "[7]".<br><br>• *name*<br><br>A name for the bucket.<br>If you use the ***name*** parameter, you cannot use an ***every-count*** value. You must enter UNDEF for the ***every-count*** parameter.<br><br>• *every-count*<br><br>The size of the buckets to create within the range.<br>If you use the ***every-count*** parameter, you cannot use a ***name***. You must enter an empty string ("") as a placeholder for the ***name*** parameter.<br><br>• *at-least-num*<br><br>A number that specifies the minimum number of samples required for a bucket. If the item occurs fewer times than this, a hole is marked. This parameter overrides the global **at_least** option and the per-item **at_least** option. The value of ***at-least-num*** can be set to zero, meaning "do not show holes for this range". |

**Table 12-2—Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **ignore**=***item-bool-exp*** | Define values that are to be completely ignored. They do not appear in the statistics at all. The expression is a boolean expression that can contain a coverage item name and constants.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a cross, that means any of the participating items. In a transition, that means the item or **prev__*item***.<br><br>For example, if “i” is a coverage item and “j” is a reference to a struct field, the expression “i > 5” is a valid expression, but “i > me.j” is not legal.<br><br>If the **ignore** expression is TRUE when the data is sampled, the sampled value is ignored (that is, not added to the bucket count).<br><br>If you want to achieve the first effect (ignore specific samples), but you do not want to hide buckets containing holes and you want the grade to reflect all generated values, use the **when** option instead. |
| **illegal**=***item-bool-exp*** | Define values that are illegal. An illegal value causes a DUT error. If the **check_illegal_immediately** coverage configuration option is FALSE, the DUT error occurs during the **check_test** phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on the fly). Note that checking illegal values immediately has a significant negative impact on ***e*** program performance.<br><br>See “illegal Example” on page 388 for an example of how to set the error effect of this check to WARNING instead of ERROR.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a cross, that means any of the participating items. In a transition, that means the item or **prev__*item***.<br><br>For example, if “i” is a coverage item and “j” is a reference to a struct field, the expression “i > 5” is a valid expression, but “i > me.j” is not legal.<br><br>If you want the coverage grades to reflect all bucket contents, use the **when** option instead to specify the circumstances under which a given value is counted. |

**Table 12-2—Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **radix=DEC\|HEX\|BIN** | For items of type **int** or **uint**, specifies the radix used in coverage reports for implicit buckets. If the **ranges** option is not used to create explicit buckets for an item, a bucket is created for every value of the item that occurs in the test. Each different value sampled gets its own bucket, with the value as the name of the bucket. These are called implicit buckets.<br><br>Legal values are DEC (decimal), HEX (hexadecimal), and BIN (binary). The value must be in upper case letters. If the **radix** is not used, **int** or **uint** bucket names are displayed in decimal.<br><br>The global **print** radix option does not affect the bucket name radix.<br><br>If no radix is specified for an item, but a radix is specified for the item's group, the group's radix applies to the item. |
| **no_trace** | This item will not be traced by the simulator. Use this option to collect data for **echo event**. |
| **weight**=***uint*** | Specifies the weight of the current item relative to other items in the same coverage group. It is a non-negative integer with a default of 1. |
| **name** | Assign an alternative name for a cross or transition item. For example:<br><br>`transition len using name = t_len;`<br>`transition ptype using name = t_ptype;`<br>`cross t_len, t_ptype;`<br><br>This option cannot be modified by **using also**. |

## Description

Defines a new basic coverage item with an optional type. Options specify how coverage data is collected and reported for the item. The item can be an existing field name, or a new name. If you use a new name for a coverage item, you must specify the item's type and the expression that defines it.

If a value for an item falls outside all of the buckets for the item, that value does not count toward the item's grade. The **ranges** option determines the number and size of buckets into which values for the item will be placed. If **ranges** is not specified, the default number of buckets is 16 (set by the **max_int_buckets** coverage configuration option). If buckets are not created for all possible values of the item, the values for which buckets do not exist are ungradeable. Those values are given goals of 0, and do not affect the grade for the item. For example, a randomly generated item of type **uint** has $2^{32}$ -1 possible values. If no ranges are specified for a **uint** item, then buckets are created by default for only the first 16 possible values (0 through 15). Since the odds that a **uint** value will be less than 16 are very small, it is almost certain that none of the values will fall into one of the 0 to 15 buckets, which are the only buckets for which a grade is calculated. This means that the item will not receive a grade, and will not contribute to the grade for the group.

By default, basic items are enabled for **echo event**. You can use the **no_trace** option to disable tracing for an item.

Below are some general examples of coverage item definitions. For examples of each of using the coverage item options, see "Coverage Item Options Examples" on page 386.

NOTE— Unless you turn on coverage mode, no coverage results are collected even if cover groups and items are defined. Use the cover configuration option to turn on coverage, as in the following example.

```
extend sys {
    setup() is also {
        set_config(cover, mode, on_interactive);
    };
};
```

## Coverage Per Instance

The coverage per instance feature (**per_instance** option) allows you to collect coverage information for separate instances of structs or units, and to see the coverage data and grade associated with each particular instance.

When you use the **per_instance** option in a cover item definition, that item becomes a "**per_instance** item". Each bucket of that item gets its own coverage grade and is shown separately in the coverage report. For example, if a struct has a field named packet_type and the value of the packet_type field can be either Ethernet or ATM, then making that field a **per_instance** item results in a grade and a coverage report listing for Ethernet instances and a separate grade and coverage report listing for ATM instances.

Typically, you use per-instance coverage on one item and transition or cross coverage on other items to see transitions or crosses of values within the different subtypes determined by the **per_instance** item. See Example 5 on page 391 and Example 6 on page 394.

An instance is created for every valid bucket of the **per_instance** item. Any instance that is not sampled is marked as a hole.

Along with the **per_instance** item data, coverage data is also collected for the original, **per_type** item as if it were not a **per_instance** item. This coverage data for the **per_type** item is the accumulated information for all the instances, using the coverage options defined for the item.

Grading is calculated for each instance separately. The grade of the cover group is the weighted grades of all the **per_instance** items. The **per_type** item receives the same grade it would get if there were no **per_instance** items.

An instance item name is the name of the **per_type** item followed by "==" and the name of the instance bucket. For example, the instance item names for the case above are:

packet_type==Ethernet
packet_type==ATM

An item my_b of type **boolean** will have the following instance names:

my_b==TRUE
my_b==FALSE

An item my_u of type **uint(bits:2)** will have the following instance names:

my_u==0
my_u==1
my_u==2
my_u==3

An item my_num of type **uint** with ranges={range([0], "Zero"); range, [1..1000], "", 500)} will have the following instance names:

my_num==Zero
my_num==[1..500]
my_num==[501..1000]
my_num==others

where others is the bucket for all uint values higher than 1000.

For integer instances, the decimal radix is used regardless of what the radix is for the cover group.

You can define more than one **per_instance** item in the same cover group. In this case, the total number of instances is the sum of all valid buckets for all the **per_instance** items plus one (the **per_instance** bucket).

If a **per_instance** item definition is changed in an extension, then the coverage data for the original **per_type** item might not accurately reflect nor agree with the coverage data collected per instance.

You cannot define a **per_instance** item under a specific instance.

You can define items with the same name under two different instances, with the condition that they must have the same definition (type and expression).

If a **per_instance** item is participating in a cross item or a transition item, then the cross or transition item is not added to the instances created by the **per_instance** item.

You can use the **ignore** option to ignore a particular instance or the **illegal** option to define a particular instance as illegal. For example, if an item named port_id has a bucket PORT_3, you can use "**ignore** = port_id == PORT_3".

To cancel per instance coverage collection in an extension, use the **also per_instance** = FALSE option. For example: **item** my_item **using also per_instance** = FALSE.

## Per_Instance Item Errors

Table 12-3 lists errors that might occur when coverage per instance is used.

**Table 12-3—Coverage Per Instance Errors**

| Error | Description |
|---|---|
| Using a non-gradeable item as a **per_instance** item | User defines a **per_instance** item option for a non-gradeable item.<br><br>Runtime error, "Items used with per_instance option must be gradable". |
| Using a cross or transition item as a **per_instance** item | User defines a **per_instance** item for a cross item or a transition item.<br><br>Load fails with the message "Not supported". |

**Table 12-3—Coverage Per Instance Errors *(continued)***

| Error | Description |
|---|---|
| Trying to extend an invalid instance | User tries to extend (using **cover** ... **is also**) a group instance that does not exist.<br><br>Load fails. |
| Recursively split instances | User defines a **per_instance** item option for an instance group extension.<br><br>Load fails with the message "Not supported". |
| Trying to extend specific instances without using **is also** | User tries to extend a specific group instance using **is** instead of **is also**. |
| Trying to define multiple items with the same name but different definitions under different instances | User tries to define an item with the same name under two different instances. |
| Specifying an invalid instance name | User specifies an invalid instance name (possibly using wild cards). No matching instance is found.<br><br>Command is ignored. |

## Coverage Item Options Examples

Examples of all the coverage item options are shown below. More examples of coverage item definitions are shown in "Additional Examples" on page 390.

### **per_instance** Example

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    event stimulus;
    cover stimulus is {
        item alu using per_instance;
        item opcode;
    };
};
```

For alu bucket ALU_0, coverage information is collected and graded for all other items, and listed under instance *stimulus(alu==ALU_0)* in the coverage report. Likewise, for ALU_1, coverage is collected and graded for all other items and listed under ***stimulus(alu==ALU_1)*** in the report.

### **no_collect** Example

```
struct sm {
    cpu: top_cpu;
    event cs;
    cover cs is {
        item cb: bit = cpu.carry using no_collect;
    };
};
```

Coverage information is not collected for item “cb”, but the item can be used in cross coverage, and Verilog tracing can be done on it.

### **text** Example

```
type state_name: [S1, S2];
struct sm {
    st: state_name;
    event state_change;
    cover state_change is {
        item st using text = "The CPU state";
    };
};
```

The text is displayed with the data for item “st” in the coverage report.

### **when** Example

```
type state_name: [S1, S2];
struct sm {
    cpu: top_cpu;
    st: state_name;
    event state_change;
    cover state_change is {
        item st using when = (cpu.init_complete == TRUE);
    };
};
```

Coverage information is collected for item “st” only when the boolean expression is TRUE at the time the “state_change” event occurs.

### **at_least** Example

```
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
struct inst {
    opcode: cpu_opcode;
    op1: byte;
    op2: byte;
    event inst_driven;
    cover inst_driven is {
        item op1;
        item op2;
    };
    when JMP inst {
        op3: byte;
        cover inst_driven is also {
            item op3 using ranges =
                {range([0..255], "", 16)}, at_least = 10;
        };
    };
};
```

The **ranges** option creates a bucket for each set of 16 values (0-15, 16 -31, ... , 240-255) for item “op3”. Any of those buckets for which fewer than 10 samples is collected is a hole.

### **ranges** Example

```
struct pcc {
    pc_on_page_boundary: uint (bits: 15);
```

```
        pc: uint (bits: 15);
        stack_change: byte;
        event pclock;
        cover pclock is {
            item pc_on_page_boundary using
                ranges = {
                    range([0], "0"); range([4k], "4k");
                    range([8k], "8k"); range([12k], "12k");
                    range([16k], "16k"); range([20k], "20k");
                    range([24k], "24k"); range([28k], "28k");
                    range([0..32k-1], "non-boundary");
                };
            item pc using radix = HEX,
            ranges = {
                    range([0..4k-1], "page_0", UNDEF, 4);
                    range([4k..32k-1], "", 8k, 2);
                };
            item stack_change using
                ranges = { range( [0..32], "", 1); };
        };
    };
```

The **range** specifications in this example create the following buckets:

- Item pc_on_page_boundary:
  Bucket names: 0, 4k, 8k, 12k, 16k, 20k, 24k, 28k
  Each of these buckets will hold the given value.
  Bucket name: non-boundary:
  This bucket will hold all values from 0 to 32k-1 that are not put into one of the buckets above.
- Item pc:
  Bucket name: page_0
  This bucket will hold values from 0 to 4095, and must contain at least four samples (because ***at_least_num*** is 4).
  Bucket names: 0x1000..ex2fff, 0x3000..0x4fff, 0x5000..0x6fff, 0x7000..0x7cff
  Each of these buckets will hold values in the given range and must contain at least two samples (because ***every_count*** is 8k, and ***at_least_num*** is 2).
- Item stack_change:
  Bucket names: 0, 1, 2, ... , 32
  Each of these buckets will hold the given value (because ***every_count*** is 1).

### **ignore** Example

```
    struct packet {
        len: uint (bytes: 2);
        event xfer;
        cover xfer is {
            item len using ignore = (len > 32k);
        };
    };
```

Any “len” value greater than 32,768 is ignored.

### **illegal** Example

```
    struct packet {
        packet_len: uint (bits: 12);
        event rcv_clk;
        cover rcv_clk is {
            item len: uint (bits: 12) = packet_len using
                ranges = {
                            range( [16..255], "small");
                            range( [256..3k-1], "medium");
                            range( [3k..4k], "big");
                         },
                illegal = (len < 16 or len > 4000);
        };
    };
```

Any “len” value less than 16 or greater than 4,096 is illegal. If an illegal “len” value occurs, a DUT error is issued during the **check_test** phase of the test. The **ranges** option creates buckets for values from 16 to 4,096.

### radix Example

```
    cover done is {
        item len: uint (bits: 3) = packet_len;
        item data: byte = data using
            ranges = {range([0..0xff], "", 16)};
        item mask: uint (bits: 2) = mask using radix = BIN;
    };
```

For the “len” item, the bucket names are: 0, 1, ... 7 (using the default decimal radix).

For the “data” item, the bucket names are: [0..15], [16..31], ... [240..255] (using the default decimal radix).

For the “mask” item, the bucket names are 0b00, 0b01, 0b10, and 0b11 (using the **radix** = BIN option specified for this item).

### weight Example

```
    cover done is {
        item len: uint (bits: 3) = packet_len;
        item data: byte = data;
        item mask using weight = 2;
    };
```

The “mask” item is assigned a weight of 2. Since there are two other items, “len” and “data”, with default weights of 1, the “mask” item contributes (2/4)*grading(done) to the grade for the group.

### no_trace Example

```
    struct packet {
        event done;
        mask: uint (bits: 2);
        packet_data: byte;
        cover done is {
            item data: byte = packet_data;
            item mask using no_trace;
        };
    };
```

The “done” event is marked for tracing, but the “mask” item in the “done” coverage group is marked **no_trace**, so it is not traced in the simulator and is not displayed by **echo event**.

### Additional Examples

### Example 1

The following example uses ***type=exp*** to define coverage for a list element. In the b0 item definition, the ***type*** is byte and ***exp*** is b_list[0]. This collects coverage data for the value of the first byte in b_list.

This example also shows a predefined list method, b_list.**size()**, used in the item definition expression.

```
struct mem {
    b_list: list of byte;
    keep b_list.size() in [2..16];
    event mem_ch;
    cover mem_ch is {
        item b0: byte = b_list[0];
        item b_list_size: uint (bits: 4) = b_list.size();
    };
};
```

### Example 2

The following example uses the ***type=exp*** parameter in the mem_mode item definition to define coverage for a struct field which is instantiated through a hierarchy of struct instances. The ***type*** is memory_type. The ***exp*** is the hierarchical path sys.config.mem_type.

```
struct mem {
    event mem_ch;
    cover mem_ch is {
        item mem_mode: memory_mode = sys.config.mem_mode;
    };
};
struct config {
    mem_mode: memory_mode;
    keep mem_mode == 'top.mem_mode';
};
```

### Example 3

The following example demonstrates a way to cover different combinations of values for particular bits of an item. It uses ***type=exp*** and the **when** coverage item option to collect coverage for bit 14 = 0, bit 15 = 0 versus bit 14 = 0, bit 15 = 1 of a 16-bit unit item named opcode.

```
struct inst {
    opcode: uint (bits: 16);
    len: int [1..3];
    event fetch;
    cover fetch is {
        item opcode using radix = BIN;
        item opcode0: uint (bits: 2) = opcode
            using when = (opcode[15:14] == 2’b00);
        item opcode1: uint (bits: 2) = opcode
            using when = (opcode[15:14] == 2’b01);
        item len;
    };
    run() is also {
        emit fetch;
    };
};
```

### Example 4

In the following example, coverage data is collected for a field generated on the fly. The field is addr_tmp, which has been added just to serve as a coverage item. The addr_tmp values replace the addr field values in a previously generated list of packets structs.

For each addr_tmp value generated in the **for** loop, the cov_addr event is emitted to take a coverage sample of the addr_tmp value. The the new addr_tmp value is then placed in the addr field in the current packet instance.

The ranges option is used in the addr_tmp coverage item definition to create four buckets, for values from 0 to 63, 64 to 127, 128 to 191, and 192 to 255. The at_least option is also used, to specify that any bucket that does not get at least three values is a hole.

```
    struct packet {
        %addr: byte;
        %len: uint (bytes: 2);
        %data[len]: list of byte;
    };
    struct pkts {
        packets: list of packet;
        keep packets.size() == 12;
    };
    extend pkts {
        addr_tmp: byte;
        add_addr()@sys.clk is {
            for i from 0 to packets.size() - 1 do {
                gen addr_tmp;
                emit cov_addr;
                packets[i].addr = addr_tmp;
                wait cycle;
            };
            stop_run();
        };
        event cov_addr;
        cover cov_addr is {
            item addr_tmp using ranges = {range([0..255], "", 64)},
            at_least = 3;
        };
        run() is also {
            start add_addr();
        };
    };
```

### Example 5

The following example specifies that the alu item is a **per_instance** item. For each of the buckets of the alu item, ALU_0 and ALU_1, coverage data will be collected and grades will be calculated for the opcode, operand1, and operand2 items in each of the two alu buckets. The coverage report follows the sample code.

```
    struct instruction {
        alu: [ALU_0, ALU_1];
        opcode: [ADD, SUB, AND, XOR];
        operand1 : byte;
        operand2 : byte;

        event stimulus;
        cover stimulus is {
            item alu using per_instance;
            item opcode;
            item operand1 using
                 ranges = { range([0..15]); range([16..0xff]); };
        };
    };
```

## Output

The following is coverage data for Example 5 on page 391.

```
    Coverage report
    ===============

    Command:               show cover -kind = full instruction.*.*
    Grading_formula:       linear
    At least multiplier:   1
    Show mode:             both
    Number of tests:       1
    Note:                  %t is a percent from total, %p is a percent from parent


    Cover group: instruction.stimulus
    =================================

    Grade: 0.83  Weight: 1

    ** alu **
    Samples: 26  Tests: 1  Grade: 1.00  Weight: 1

    grade     goal samples tests %t    alu

     1.00     1     12      1     46    ALU_1


    ** opcode **
    Samples: 26  Tests: 1  Grade: 1.00  Weight: 1

    grade       goal    samples     tests   %t    opcode

     1.00        1          4         1      15    ADD
     1.00        1          7         1      27    SUB
     1.00        1          6         1      23    AND
     1.00        1          9         1      35    XOR


    ** operand1 **
    Samples: 26  Tests: 1  Grade: 0.50  Weight: 1

    grade     goal      samples     tests    %t   operand1
```

```
 0.00        1          0        0       0    [0..15]
 1.00        1         26        1     100    [16..255]


Cover group: instruction.stimulus(alu==ALU_0)
=============================================

Grade: 0.75  Weight: 1

** opcode **
Samples: 12  Tests: 1  Grade: 1.00  Weight: 1

grade        goal     samples    tests   %t   opcode

 1.00        1          1        1       8    ADD
 1.00        1          3        1      25    SUB
 1.00        1          3        1      25    AND
 1.00        1          5        1      42    XOR


** operand1 **
Samples: 12  Tests: 1  Grade: 0.50  Weight: 1

grade     goal    samples     tests      %t    operand1

 0.00        1          0        0       0    [0..15]
 1.00        1         12        1     100    [16..255]


Cover group: instruction.stimulus(alu==ALU_1)
=============================================

Grade: 0.75  Weight: 1

** opcode **
Samples: 14  Tests: 1  Grade: 1.00  Weight: 1

grade       goal        samples   tests  %t    opcode

 1.00        1          3        1      21    ADD
 1.00        1          4        1      29    SUB
 1.00        1          3        1      21    AND
 1.00        1          4        1      29    XOR


** operand1 **
Samples: 14  Tests: 1  Grade: 0.50  Weight: 1

grade       goal      samples    tests   %t   operand1

 0.00        1          0        0       0    [0..15]
 1.00        1         14        1     100    [16..255]
```

### Example 6

The example below shows the ignore option used to ignore particular instances, that is, alu==ALU_0 of the alu item. The coverage report shows grades for alu overall and all ALU_1 instances. The output, shown following the code, does not contain any data for instances where alu is ALU_0.

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    operand1 : byte;
    operand2 : byte;

    event stimulus;
    cover stimulus is {
        item alu using per_instance, ignore = (alu==ALU_0);
        item opcode;
        item operand1 using
             ranges = { range([0..15]); range([16..0xff]); };
    };
};
```

### Output

The following is coverage data for Example 6 on page 394.

```
Coverage report
===============

Command:               show cover -kind = full instruction.*.*
Grading_formula:       linear
At least multiplier:   1
Show mode:             both
Number of tests:       1
Note:                  %t is a percent from total, %p is a percent from parent


Cover group: instruction.stimulus
=================================

Grade: 0.83  Weight: 1

** alu **
Samples: 14  Tests: 1  Grade: 1.00  Weight: 1

grade     goal    samples     tests      %t    alu

 1.00        1         14         1     100    ALU_1


** opcode **
Samples: 26  Tests: 1  Grade: 1.00  Weight: 1

grade         goal       samples   tests  %t     opcode

 1.00         1           4        1      15     ADD
 1.00         1           7        1      27     SUB
 1.00         1           6        1      23     AND
```

```
     1.00        1          9        1      35    XOR


    ** operand1 **
    Samples: 26  Tests: 1  Grade: 0.50  Weight: 1

    grade        goal      samples   tests   %t   operand1

     0.00        1          0        0       0    [0..15]
     1.00        1         26        1     100    [16..255]



    Cover group: instruction.stimulus(alu==ALU_1)
    =============================================

    Grade: 0.75  Weight: 1

    ** opcode **
    Samples: 14  Tests: 1  Grade: 1.00  Weight: 1

    grade        goal       samples  tests  %t    opcode

     1.00        1          3        1      21    ADD
     1.00        1          4        1      29    SUB
     1.00        1          3        1      21    AND
     1.00        1          4        1      29    XOR


    ** operand1 **
    Samples: 14  Tests: 1  Grade: 0.50  Weight: 1

    grade        goal       samples  tests   %t   operand1

     0.00        1          0        0       0    [0..15]
     1.00        1         14        1     100    [16..255]
```

## Example 7

The example below shows the illegal option used with a **per_instance** item.

```
define MAX_PORTS 4;
type port_id_kind: [PORT_0, PORT_1, PORT_2, PORT_3];
struct port {
    port_id: port_id_kind;
    status: uint(bits:2);
    event pkt_ended;
    cover pkt_ended is {
        item port_id using per_instance,
            illegal = (port_id.as_a(int) < MAX_PORTS);
        item status;
    };
};
```

### Example 8

The example below shows an extension of the stimulus cover group in Example 6 on page 394, to add the **at_least** option to the opcode item. For additional information about extending coverage items, see. "Extending Coverage Items" on page 416.

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    operand1 : byte;
    operand2 : byte;

    event stimulus;
    cover stimulus is {
        item alu using per_instance, ignore = (alu==ALU_0);
        item opcode;
        item operand1 using
             ranges = { range([0..15]); range([16..0xff]); };
    };
};

extend instruction {
    cover stimulus is also {
        item opcode using also at_least = 4;
    };
};
```

### See Also

— "Defining Coverage Groups: cover" on page 373
— "Defining Cross Coverage Items" on page 396
— "Defining Transition Coverage Items" on page 403
— "Extending Coverage Items" on page 416
— "Defining Structs: struct" on page 118

## 12.3 Defining Cross Coverage Items

### 12.3.1 Overview

Cross items are combinations of items from the same coverage group. The **cross** coverage construct is used to define cross items.

### 12.3.2 cross

#### Purpose

Define a cross coverage item

#### Category

Coverage group item

### Syntax

**cross *item-name-1*, *item-name-2*,** ... [**using *coverage-item-option*,** ...]

Syntax example:

```
cover inst_driven is {
    item opcode;
    item op1;
    cross opcode, op1;
};
```

### Parameters

| | |
|---|---|
| ***item-name-1*, *item-name-2*, ...** | Each item name must be one of the following.<br>• the name of an item defined previously in the current coverage group.<br>• the name of a transition item defined previously in the current coverage group<br>• the name of a cross item defined previously in the current coverage group |
| *coverage-item-option* | An option for the cross item. The options are listed in Table 12-4. |

**Table 12-4—Cross Coverage Item Options**

| Option | Description |
|---|---|
| **name**=*label* | Specifies a name for a cross coverage item. No white spaces are allowed in the label. The default is cross__***item-a***__***item-b***. |
| **text**=*string* | A text description for this coverage item. This can only be a quoted string, not a variable or expression. The text is shown along with the item name at the top of the coverage information for the item. |
| **when**=*bool-exp* | The item is sampled only when ***bool-exp*** is TRUE. The ***bool-exp*** is evaluated in the context of the parent struct. |
| **at_least**=*num* | The minimum number of samples for each bucket of the item. Anything less than ***num*** is considered a hole.<br><br>This option cannot be used with string items or for unconstrained integer items (items that do not have specified ranges).<br><br>You cannot specify a number less than 1. The default is 1. |

**Table 12-4—Cross Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **ignore**=*item-bool-exp* | Define values that are to be completely ignored. They do not appear in the statistics at all. The expression is a boolean expression that can contain a coverage item name and constants.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a cross, that means any of the participating items. In a transition, that means the item or **prev__*item***.<br><br>For example, if "i" is a coverage item and "j" is a reference to a struct field, the expression "i > 5" is a valid expression, but "i > me.j" is not legal.<br><br>If the **ignore** expression is TRUE when the data is sampled, the sampled value is ignored (that is, not added to the bucket count).<br><br>If you want to achieve the first effect (ignore specific samples), but you do not want to hide buckets containing holes and you want the grade to reflect all generated values, use the **when** option instead. |
| **illegal**=*item-bool-exp* | Define values that are illegal. An illegal value causes a DUT error. If the **check_illegal_immediately** coverage configuration option is FALSE, the DUT error occurs during the **check_test** phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on the fly). Note that checking illegal values immediately has a significant negative impact on ***e*** program performance.<br><br>See "illegal Example" on page 388 for an example of how to set the error effect of this check to WARNING instead of ERROR.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a cross, that means any of the participating items.<br><br>For example, if "i" is a coverage item and "j" is a reference to a struct field, the expression "i > 5" is a valid expression, but "i > me.j" is not legal.<br><br>If you want the coverage grades to reflect all bucket contents, use the **when** option instead to specify the circumstances under which a given value is counted. |
| **weight**=***uint*** | Specifies the weight of the current cross item relative to other items in the same coverage group. It is a non-negative integer with a default of 1. |

## Description

Defines cross coverage between items in the same coverage group. Creates a new item with a name specified using a **name** option, or with a default name of "**cross__*item-name-1*__*item-name-2***..." (with two underscores separating the parts of the name). This shows every combination of values of the first and second items, and every combination of the third item and the first item, the third item and the second item, and so on.

You can cross any combination of basic coverage items, cross items and transitions defined in the same coverage group.

When there is a hole in one of the items of a cross, the whole branch of samples that is spawned under the hole is, by default, omitted from the coverage report. To see the full report, including all the holes, set the coverage configuration **show_sub_holes** to TRUE.

## Example 1

In the following example, cross coverage is collected for the three coverage items “op1”, “op2”, and “opcode”. Constraints are applied to limit “opcode” values to either ADD or SUB, to limit “op1” values to reg0 or reg1, and to limit “op2” values to the range 1 to 24.

The coverage group is named “inst_driven”. The “inst_driven” event is emitted elsewhere in the code whenever an instruction is generated.

The “op2” coverage item definition uses the **ranges** option with a range of 1 to 16 and “every-count” equal to 4. This creates a bucket for values 1 to 16, which is divided into four smaller buckets for values from 1 to 4, 5 to 8, 9 to 12, and 13 to 16. Values from 17 to 24 go into the default “others” bucket, since they are not in the specified range.

The coverage report will show holes for all instances of “opcode” that are not “ADD” or “SUB”, and for all instances of “op1” that are not “reg0” or “reg1”.

```
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
type cpu_reg: [reg0, reg1, reg2, reg3];
struct inst {
    opcode: cpu_opcode;
    keep opcode in [ADD, SUB];
    op1: cpu_reg;
    keep op1 in [reg0, reg1];
    op2: byte;
    keep op2 in [1..24];
    event inst_driven;
    cover inst_driven is {
        item opcode;
        item op1;
        item op2 using ranges = {range([1..16], "", 4)};
        cross opcode, op1, op2 using name = opcode_op1_op2;
    };
};
```

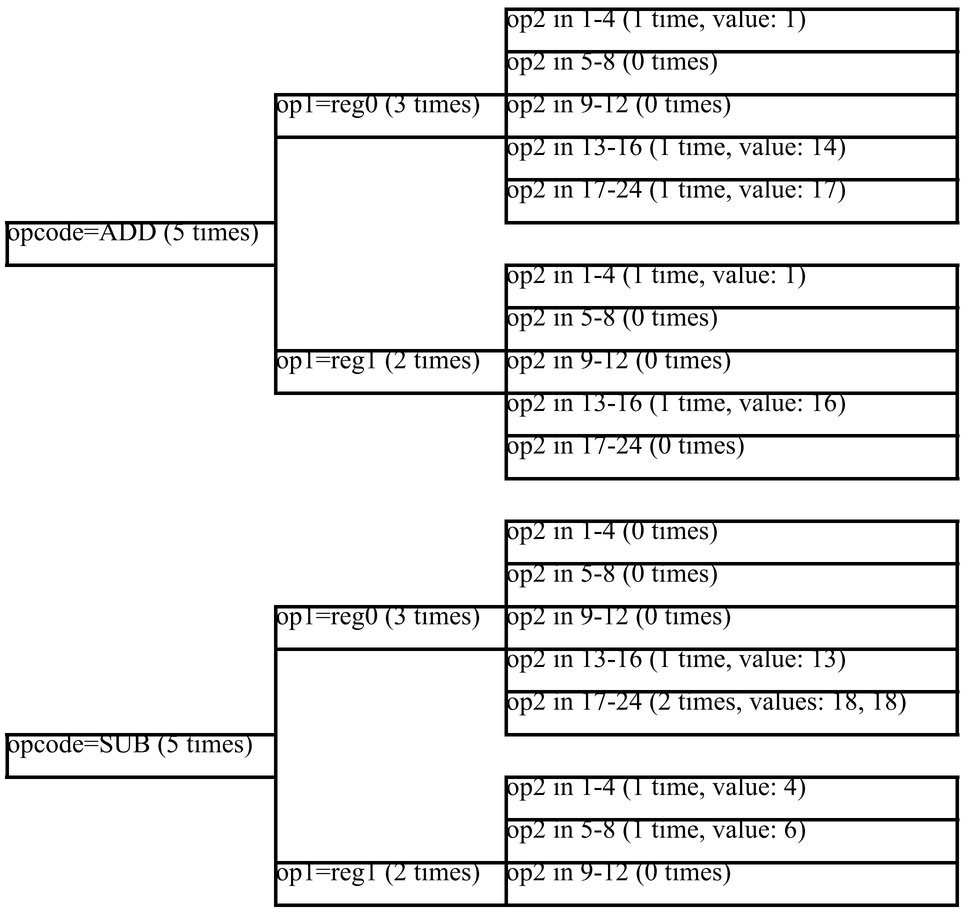
The cross of opcode, op1, and op2 shows all the combinations of values for those three items, sorted first by opcode value, by op1 under each opcode value, and by op2 under each op1 value.

For a sample test in which 10 instances of the “inst” struct were generated, the item values are shown in Figure 12-1 on page 400, and a chart of the cross coverage information is shown in Figure 12-2 on page 400.

**Figure 12-1—Description of Generated Instances of “inst”**

| instance | opcode | op1 | op2 |
|---|---|---|---|
| 1 | SUB | reg0 | 18 |
| 2 | SUB | reg0 | 18 |
| 3 | ADD | reg0 | 17 |
| 4 | ADD | reg0 | 1 |
| 5 | SUB | reg1 | 6 |
| 6 | ADD | reg0 | 14 |
| 7 | ADD | reg1 | 16 |
| 8 | SUB | reg0 | 13 |
| 9 | SUB | reg1 | 4 |
| 10 | ADD | reg1 | 1 |

**Figure 12-2—Cross Coverage Sample Results**



## Example 2

The example below shows the **name** option. For all occurrences of the “opcode” item together with the “op1” item, the coverage report shows the cross item name and its definition, “code_and_reg (cross opcode, op1)".

```
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
type cpu_reg: [reg0, reg1, reg2, reg3];
struct inst {
    opcode: cpu_opcode;
    op1: cpu_reg;
    event inst_driven;
    cover inst_driven is {
        item opcode;
        item op1;
        cross opcode, op1 using name = code_and_reg;
    };
};
```

### Example 3

The example below shows how to define cross coverage for two items sampled at different events. The events named request and grant are driven by HDL signals of the same names. The event named cov_req_ack occurs whenever a grant event follows a request event by any number of cycles in which request does not occur. Upon a request event, the request_type field is assigned the value of the HDL req_type signal, by the **exec** expression. Thus, the request_type field gets its value upon occurrence of the request event. This can be thought of as sampling request_type on the request event.

Since the cover_req_ack event occurs when grant occurs (when the sequence in the cov_req_ack definition succeeds), the items in the cov_req_ack group are sampled upon the occurrence of grant. Thus the request_type item depends on both the request event and the grant event, and the burst_mode item depends on the grant event. This has the effect of sampling request_type on the request event, and sampling burst_mode on the grant event.

```
struct monitor {
    request_type: uint;
    event request is rise('top.request')@sim;
    event grant is rise('top.grant')@sim;
    event cov_req_ack is {
            @request exec {request_type = 'top.req_type';};
            [..]* not @request;
            @grant;
        } @sys.clk;
    cover cov_req_ack is {
        item request_type using
            ranges = {
                range([0..3], "", 1);
                range(others, "invalid")
            };
        item burst_mode: bool = 'top.burst_mode';
    cross request_type, burst_mode;
    };
};
```

### Example 4

The example below shows cross coverage per instance. The alu item is defined as a **per_instance** item, so coverage is collected for the cross of opcode and operand1 when alu==ALU_0 and when alu==ALU_1.

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    operand1 : byte;
    operand2 : byte;

    event stimulus;
    cover stimulus is {
        item alu using per_instance;
        item opcode;
        item operand1;
        cross opcode, operand1;
    };
};
```

### Example 5

The example below shows how to extend a cross coverage item. In this case, the cross item, “cross opcode, operand1”, is initially defined without the **name** option, so the default name, “cross__opcode_operand1”, is required in the extension. If the cross item is initially defined using the **name** option, the given name is used in place of the default name. For additional information about extending coverage items, see “Extending Coverage Items” on page 416.

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    operand1 : byte;
    operand2 : byte;

    event stimulus;
    cover stimulus is {
        item opcode;
        item operand1;
        cross opcode, operand1;
    };
};

extend instruction {
    cover stimulus is also {
        item cross__opcode__operand1 using also text =
            "Cross of opcode and operand1";
    };
};
```

### Example 6

The example below shows a cross of two cross items. The items alu and operand1 are crossed, and the items opcode and operand2 are crossed, and then those two cross items are crossed.

```
struct instruction {
    alu: [ALU_0, ALU_1];
    opcode: [ADD, SUB, AND, XOR];
    operand1 : byte;
    operand2 : byte;
```

```
        event stimulus;
        cover stimulus is {
            item alu;
            item opcode;
            item operand1;
            item operand2;
            cross alu, operand1 using name=x_alu_op1;
            cross opcode, operand2 using name=x_opc_op2;
            cross x_alu_op1, x_opc_op2;
        };
    };
```

### See Also

- "Defining Coverage Groups: cover" on page 373
- "Defining Basic Coverage Items" on page 378
- "Coverage Item Options Examples" on page 386
- "Extending Coverage Items" on page 416
- The **coverage** option of "set_config()" on page 766

## 12.4 Defining Transition Coverage Items

### 12.4.1 Overview

Transition items are items for which value changes are collected in the coverage data. The **transition** coverage group item is used to define transition items.

### 12.4.2 transition

#### Purpose

Define a coverage transition item

#### Category

Coverage group item

#### Syntax

**transition *item-name*** [**using *coverage-item-option***, ...]

Syntax example:

```
    cover state_change is {
        item st: cpu_state = 'top.cpu.main_cur_state';
        transition st;
    };
```

### Parameters

| | |
|---|---|
| *item-name* | A coverage item defined previously in the current coverage group. |
| *coverage-item-option* | The coverage item options are listed in Table 12-5. |

**Table 12-5—Transition Coverage Item Options**

| Option | Description |
|---|---|
| **name**=***string*** | Specifies a name for a transition coverage item. The default name is **transition__*item-name*** (where two underscores separate "transition" and "item-name"). |
| **text=***string* | A text description for this coverage item. This can only be a quoted string, not a variable or expression. The text is shown along with the item name at the top of the coverage information for the item. |
| **when=***bool-exp* | The item is sampled only when ***bool-exp*** is TRUE. The *bool-exp* is evaluated in the context of the parent struct. |
| **at_least=***num* | The minimum number of samples for each bucket of each of the transition items. Anything less than ***num*** is considered a hole.<br><br>This option cannot be used with string items or for unconstrained integers (items that have no specified ranges).<br><br>You cannot specify a negative number. The default is 1. |
| **ignore=***item-bool-exp* | Define values that are to be completely ignored. They do not appear in the statistics at all. The expression is a boolean expression that can contain a coverage item name and constants.<br><br>The previous value can be accessed as **prev_*item-name***. The **prev** prefix is predefined for this purpose.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a cross, that means any of the participating items. In a transition, that means the item or **prev__*item***.<br><br>For example, if "i" is a coverage item and "j" is a reference to a struct field, the expression "i > 5" is a valid expression, but "i > me.j" is not legal.<br><br>If the **ignore** expression is TRUE when the data is sampled, the sampled value is ignored (that is, not added to the bucket count).<br><br>If you want to achieve the first effect (ignore specific samples), but you do not want to hide buckets containing holes and you want the grade to reflect all generated values, use the **when** option instead. |

**Table 12-5—Transition Coverage Item Options *(continued)***

| Option | Description |
|---|---|
| **illegal=***item-bool-exp* | Define values that are illegal. An illegal value causes a DUT error. If the **check_illegal_immediately** coverage configuration option is FALSE, the DUT error occurs during the **check_test** phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on the fly). Note that checking illegal values immediately has a significant negative impact on ***e*** program performance.<br><br>See "illegal Example" on page 388 for an example of how to set the error effect of this check to WARNING instead of ERROR.<br><br>The boolean expression is evaluated in a global context, not in instances of the struct. In other words, the expression must be valid at all times, even before generation. Therefore, you can only use constants and the item itself in the expression. In a transition, that means the item or **prev__*item***.<br><br>For example, if "i" is a coverage item and "j" is a reference to a struct field, the expression "i > 5" is a valid expression, but "i > me.j" is not legal.<br><br>If you want the coverage grades to reflect all bucket contents, use the **when** option instead to specify the circumstances under which a given value is counted. |
| **weight=*uint*** | Specifies the weight of the current transition item relative to other items in the same coverage group. It is a non-negative integer with a default of 1. |

## Description

Defines coverage for changes from one value to another of a coverage item. If no name is specified for the transition item with the **name** option, it gets a default name of "**transition__*item-name***" (with two underscores between "transition" and "***item-name***"). If ***item-name*** had ***n*** samples during the test, then the transition item has ***n***-1 samples, where each sample has the format ***previous-value, value***.

## Example 1

Any change from the previous value of "st" to the current value of "st" that is not one of the listed changes is illegal and causes a DUT error during **check_test**. This example uses the **prev_*item-name*** syntax, which refers to the previous value of the item ("st" in this case).

```
type cpu_state: [START, FETCH1, FETCH2, EXEC];
struct cpu {
    st: cpu_state;
    event state_change is
        change('top.cpu.main_cur_state') @sim;
    cover state_change is {
        item st;
        transition st using illegal =
            not ((prev_st == START and st == FETCH1)
            or (prev_st == FETCH1 and st == FETCH2)
            or (prev_st == FETCH1 and st == EXEC)
            or (prev_st == FETCH2 and st == EXEC)
            or (prev_st == EXEC and st == START));
        };
    };
```

### Example 2

The example below shows the **name** option. For all transitions of the “st” item, the coverage report shows the item name and its definition, “st_change (transition st)”.

```
    type cpu_state: [START, FETCH1, FETCH2, EXEC];
    struct cpu {
        st: cpu_state;
        event state_change is
            change('top.cpu.main_cur_state') @sim;
        cover state_change is {
            item st;
            transition st using name = st_change;
        };
    };
```

### Example 3

The example below shows transition coverage of a cross coverage item (see “Defining Cross Coverage Items” on page 396).

```
    struct instruction {
        alu: [ALU_0, ALU_1];
        opcode: [ADD, SUB, AND, XOR];
        operand1 : byte;
        operand2 : byte;

        event stimulus;
        cover stimulus is {
            item alu;
            item opcode;
            item operand1;
            item operand2;
            cross opcode, operand1 using name=x_opc_op1;
            transition x_opc_op1;
        };
    };
```

### Example 4

The example below shows transition coverage per instance. The alu item is defined as a **per_instance** item, so coverage is collected for the opcode transitions when alu==ALU_0 and when alu==ALU_1.

```
    struct instruction {
        alu: [ALU_0, ALU_1];
        opcode: [ADD, SUB, AND, XOR];
        operand1 : byte;
        operand2 : byte;

        event stimulus;
        cover stimulus is {
            item alu using per_instance;
            item opcode;
            transition opcode;
        };
    };
```

### Example 5

The example below shows how to extend a transition coverage item. In this case, the transition item, “transition opcode”, is initially defined without the **name** option, so the default name, “transition__opcode”, is required in the extension. If the transition item is initially defined using the **name** option, the given name is used in place of the default name. For additional information about extending coverage items, see “Extending Coverage Items” on page 416.

```
    struct instruction {
        alu: [ALU_0, ALU_1];
        opcode: [ADD, SUB, AND, XOR];
        operand1 : byte;
        operand2 : byte;

        event stimulus;
        cover stimulus is {
            item opcode;
            item operand1;
            transition opcode;
        };
    };

    extend instruction {
        cover stimulus is also {
            item transition__opcode using also at_least = 2;
        };
    };
```

### See Also

— “Defining Coverage Groups: cover” on page 373
— “Defining Basic Coverage Items” on page 378
— “Coverage Item Options Examples” on page 386
— “Extending Coverage Items” on page 416

## 12.5 Defining External Coverage Groups

### 12.5.1 Overview

You can import code coverage data from Verisity’s SureCov product into ***e*** and view the integrated data in ASCII reports. The code-coverage grades for each SureCov cover group and cover item are imported, and these grades are factored into the overall test grade.

To import SureCov data into an ***e*** program, you have to create a SureCov coverage group in ***e***. For information on how to do this, see “cover ... using external=surecov” on page 408.

For information on how to enable and disable the import and display of SureCov coverage groups, see "set_external_cover()" on page 707.

### 12.5.2 cover ... using external=surecov

#### Purpose

Create a customized SureCov cover group

#### Category

Struct member

#### Syntax

**cover *group-name* using external=surecov** [**,agent_options=*SureCov-options***]
  [, ***e-options***] **is {**
    **item *item-name* using** [**,agent_options=*SureCov-options***] [, ***e-options***] ;
  ...
**};**

#### Parameters

| | |
|---|---|
| *group-name* | The ***group-name*** is informational and can be any name you want—except that it cannot be an event name. |
| **using external = surecov** | Identifies SureCov as the external coverage tool to integrate with ***e*** . |
| *item-name* | The ***item-name*** is informational and can be any name you want |

*SureCov-options* The ***SureCov-options*** are the same for both the cover group definition and the item definitions. SureCov concatenates the cover group ***SureCov-option***s with each cover item ***SureCov-options*** to define the coverage parameters.

In general:

- The cover group ***SureCov-options*** are intended to define whether you want module or instance coverage
- The cover item ***SureCov-options*** are intended to define which type of code coverage to import.

However, you can mix the two, as described below and as shown in the examples:

- Options specified for the cover group apply to all items in the group.
- Options specified for a cover item apply for that item only.

You must include one of the following either in the group definition or in each item definition:

**module**[=***Verilog_module_name***]
**instance**[=***absolute_Verilog_instance_path***]

If you specify a particular instance (with a path), you can also enter the following option to ask for coverage statistics for the entire subdesign rooted at that instance (rather than for coverage statistics for the instance alone):

**hier**

If you specify **module** or **instance** without a name, data is collected for all Verilog modules or all Verilog instances. (The data for a given module consists of the cumulative data for all the instances of that module.) If you specify a module name or instance path without **hier**, data is collected for that particular module or instance only.

You must also include at least one of the following either in the group definition or in each item definition to define the type of code coverage you want to import:

**block**
**arc**
**state**
**trans**
**expr**
**event**
**toggle**

| | |
|---|---|
| *e- options* | The ***e-options*** are also the same for both the cover group definition and the cover item definitions.<br><br>The legal ***e-options*** are the regular cover **text** option and **weight** option:<br><br>**text=***string*<br>**weight=***uint*<br><br>Set the weight for a given cover group or item to reflect the risk associated with low coverage for that group or item compared to other cover groups or items. If the risk is higher, set the weight higher. |

## Description

Defines a SureCov coverage group.

The mechanism for integrating SureCov with ***e*** is ***e*** code that defines SureCov coverage groups. If you want to import all types of code coverage for all Verilog modules and instances, you can create this ***e*** code automatically when you invoke SureCov to instrument the Verilog HDL design description.

You should manually create the SureCov coverage groups if you want to

— Limit the import of SureCov data to particular Verilog modules or instances or to particular kinds of code coverage
— Associate the SureCov coverage group definition with a particular struct or unit.
— Supply a descriptive text string or set the weight for a SureCov coverage group or item.

## Notes

— You can define as many SureCov coverage groups as you want under any struct or unit that you want.
— You cannot extend a SureCov coverage group using the **is also** syntax.
— To disable or enable the importing of SureCov data, use the **set_external_cover()** routine.

## Cover Group Examples

Importing data for all modules:

```
cover sv_data using external=surecov, agent_options="module" is...
```

Importing data for all instances:

```
cover sv_data using external=surecov, agent_options="instance" is ...
```

Importing data for the ALU module only:

```
cover sv_data using external=surecov, agent_options="module=ALU" is ...
```

Importing data for the ALU_0 instance only:

cover sv_data using external=surecov, agent_options="instance=top.ALU_0" is ...

Importing data for the ALU module only and specifying a grading weight of 4 for the current group relative to other groups:

cover sv_data using external=surecov, agent_options="module=ALU", weight=4 is ...

Importing block and arc data for all modules. The following example shows the entire cover group definition. Note that this example gives you one combined grade for block and arc data. To get a separate grade for each type of coverage, you must define the types of coverage individually in item definitions:

```
cover sv_data using external=surecov, agent_options="module, block, arc" is
    {
    item sv_block_arc;
};
```

### Cover Item Examples

Importing block and arc data for all modules:

```
cover sv_data using external=surecov, agent_options="module" is
{
    item sv_block using agent_options="block";
    item sv_arc using agent_options="arc";
};
```

Importing block and arc data for all modules—this example is exactly the same as the previous example:

```
cover sv_data using external=surecov is      //Note that no module or
                                           //instance is specified.
{
    item block using agent_options="block, module";
    item arc using agent_options="arc, module";
};
```

Importing block and arc data for the ALU_0 instance and importing state and trans data for the ALU_1 instance:

```
cover sv_data using external=surecov is
{
  item block using agent_options="block,instance=top.ALU_0";
  item arc using agent_options="arc, instance=top.ALU_0";
  item state using agent_options="state, instance=top.ALU_1";
  item arc using agent_options="trans, instance=top.ALU_1";
};
```

Importing block and arc data for the ALU_0 instance and importing state and trans data for the ALU module:

```
cover sv_data using external=surecov is
{
    item block using agent_options="block, instance=top.ALU_0";
    item arc using agent_options="arc, instance=top.ALU_0";
    item state using agent_options="state, module=ALU";
    item arc using agent_options="trans, module=ALU";
};
```

### See Also

- “Defining Coverage Groups: cover” on page 373
- “Defining Basic Coverage Items” on page 378

## 12.6 Extending Coverage Groups

### 12.6.1 Overview

The **using also** and **is also** clauses are used to extend existing coverage groups.

### 12.6.2 cover ... using also ... is also

#### Purpose

Extend a coverage group

#### Category

Struct member

#### Syntax

**cover *event-type* using also *cover-option*, ...**[ **is also** {***coverage-item-definition***; ... } ]

Syntax examples:

```
cover rclk is also {
    item rflag;
};

cover rclk using also text = "RX clock";

cover rclk using also no_collect is also {
    item rvalue;
};
```

#### Parameters

| | |
|---|---|
| *event-type* | The name of the coverage group. This must be an event defined previously in the struct. The event is the sampling event for the coverage group. |
| *coverage-item-definition* | The definition of a coverage item. |

#### Description

The **using also** clause changes, overrides, or extends options previously defined for the coverage group. The **is also** clause adds new items to a previously defined coverage group, or can be used to change the options for previously defined items. See "Extending Coverage Items" on page 416.

Options for coverage-item-definition are listed in Table 12-1.

If a coverage group is defined under a when subtype, it can only be extended under that subtype.

If you have defined **per_instance** coverage (see "Coverage Per Instance" on page 384), you can extend a particular cover group instance to complement or override options set in the base type cover group. To change an item's options in particular instance, enter the instance name in the **cover is also** construct. For example, if a cover group named done contains a cover item named packet which has buckets named Ethernet and ATM, use “cover done(packet==ATM) is also {...}” to extend the cover group in ATM instances.

If you extend an instance that never gets created (due to an **ignore** or **illegal** option), a warning is issued and no information for the extension is put in the coverage data.

If you change the coverage options of an instance, the coverage data for the per_type item might no longer reflect or agree with the per-instance coverage data.

If, in an extension of a cover group, you override a cover group **when** option, then the overriding condition is only considered after the condition in the base group is satisfied. That is, sampling of the item is only performed when the logical AND of the cover group **when** options is true.

When you use **using also** to extend or change a **when**, **illegal**, or **ignore** option, a special variable named **prev** is automatically created. The **prev** variable holds the results of all previous **when**, **illegal**, or **ignore** options, so you can use it as a shorthand to assert those previous options combined with a new option value. For example, if a base struct cover group definition has “when = size == 5”, and an extension has “using also when = (prev and size <= 10)”, the result is the same as “when = (size == 5 and size <= 10)”.

### Example 1

The following example extends a coverage group named “info” by adding two cover items, “op2” and “cross op1, op2”.

```
struct op_st {
    op1: byte;
    op2: byte;
    event info;
    cover info is {
        item op1;
    };
};
extend op_st {
    cover info is also {
        item op2;
        cross op1, op2;
    };
};
```

### Example 2

In the following example, the cover done extension in the inst struct extension adds text to the cover group named done, and adds a new item named interrupt to the group for instances of cpu_id==CPU_1:

```
type cpu_type: [CPU_1, CPU_2];
struct inst {
    cpu_id: cpu_type;
    cpu_on: bool;
    interrupt: uint(bits:2);
    event done;
    cover done is {
        item cpu_id using per_instance;
        item cpu_on;
    };
    run() is also {
        emit done;
    };
};
extend inst {
    cover done(cpu_id==CPU_1) using also text="CPU #1" is also {
        item interrupt;
    };
};
```

### Example 3

In the following example, the **using also** is used to cancel the **global** option:

```
    struct packet{
        len: uint (bits: 4);
        kind: bool;
        a: int (bits: 4);
        event packet_gen;
        cover packet_gen using global is {
            item len;
            item kind;
            cross len, kind using text = "cross_l_k";
        };
    };
    extend packet {
        cover packet_gen using also global = FALSE;
    };
```

### Example 4

The following example show how to use **using also** to override and to narrow a **when** option:

```
    // Original definition:
    struct packet{
        len: uint (bits: 4);
        kind: bool;
        event packet_gen;
        cover packet_gen using when = len >= 3 is {
            item len;
        };
    };

    // Overriding the when definition:
    extend packet {
        cover packet_gen using also when = len == 4;
    };

    // Narrowing the when definition:
    extend packet {
        cover packet_gen using also when = (prev and len <= 7);
    };
```

### Example 5

This example uses **using also** to extend the original definition of the cover done group by adding “when = id > 64”, and then uses **is also** to add new coverage item, port, to the coverage group. Since the coverage group is initially defined under a when subtype, the coverage group extensions can only be done in an extension of that subtype.

```
<'
struct packet {
    len: uint (bits: 4);
    ptype: [ATM, Eth];
    good: bool;
    id: byte;
    port: uint (bits: 2);
    event done;
    when good packet {
        cover done is {
            item len;
            item ptype;
        };
    };
};
'>
<'
extend good packet {
    cover done using also when = id > 64 is also {
        item port;
    };
};
'>
```

## Example 6

The following example uses uses **using also** to set the weight of the per_type cover instance to zero so it will not affect the overall grade.

```
type cpu_type: [CPU_1, CPU_2];
struct inst {
    cpu_id: cpu_type;
    cpu_on: bool;
    event done;
    cover done is {
        item cpu_id using per_instance;
        item cpu_on;
    };
    run() is also {
        emit done;
    };
};
extend inst {
    cover done(per_type) using also weight = 0;
};
```

## See Also

— “Defining Coverage Groups: cover” on page 373
— “Defining Basic Coverage Items” on page 378

## 12.7 Extending Coverage Items

### 12.7.1 Overview

The **using also** clause is used to extend existing coverage items.

### 12.7.2 item ... using also

#### Purpose

Change or extend the options on a cover item.

#### Category

Coverage group item

#### Syntax

**item *item-name* using also *coverage-item-option*, ...**

Syntax example:

```
item len using also radix = HEX;
```

#### Parameters

| | |
|---|---|
| *item-name* | The name you assign to the coverage item.<br><br>If you do not specify the optional ***type***=***exp***, the value of the field named ***item-name*** is used as the coverage sample value. |
| *type* | The type of the item. The type expression must evaluate to a scalar not larger than 32 bits, or a string. |
| *exp* | The expression is evaluated at the time the whole coverage group is sampled. This value is used for the item. |
| *coverage-item-option* | Coverage item options are listed in Table 12-2. The options can appear in any order after the **using** keyword. |

#### Description

Cover item extensibility allows you to extend, change, or override a previously defined coverage item. Coverage item options are listed in Table 12-2.

To extend a coverage item, you must also use “is also” for its coverage group: “cover ***event-type*** is also { item ***item-name*** using also ...};”. See “Extending Coverage Groups” on page 412.

If a coverage item is originally defined under a when subtype, it can only be extended in the same subtype of the base type.

When you extend an item, you must refer to the item by its full name. If an item with that name does not exist, an error is issued.

For example, if an item "cross a, b" was defined previously without the **name** option, then you extend it by creating a new item with the cross item's default name, cross__a__b:

```
item cross__a__b using also ...    // Cross item was defined with no name
```

If a cross item was defined using the **name** option, such as "cross a, b using name = c_a_b", then you extend it by creating a new item using the name "c_a_b":

```
item c_a_b using also ...     // Cross item was defined with a name
```

Similarly, for a transition item that was defined without the **name** option, such as "transition b", you extend it by creating a new item with the transition item's default name, transition__b:

```
item transition__b using also ...// Transition item was defined with no name
```

If a transition item was defined using the **name** option, such as "transition b using name = t_b", then you extend it by creating a new item using the name "t_b":

```
item t_b using also ...     // Transition item was defined with a name
```

If an item is defined with an expression assigned to it, do not include the expression when you extend the item:

```
item b: bool = f(a,c) ...    // Item was defined with expression f(a,c)

item b using also ...    // Omit the type and expression in the extension
```

When you use **using also** to extend or change a **when**, **illegal**, or **ignore** option, a special variable named **prev** is automatically created. The **prev** variable holds the results of all previous **when**, **illegal**, or **ignore** options, so you can use it as a shorthand to assert those previous options combined with a new option value. For example, if an original coverage item definition has "when = size == 5", and an extension has "using also when = (prev and size <= 10)", the result is the same as "when = (size == 5 and size <= 10)".

## Example 1

In this example, an item named len is defined in the base type, then a new option, radix = HEX, is added, and finally the option is redefined to radix = BIN.

```
<'
struct packet {
    len: uint (bits: 4);
    event done;
    cover done is {
        item len;
    };
};
'>
<'
extend packet {
    cover done is also {
        item len using also radix = HEX;
    };
};
'>
<'
extend packet {
    cover done is also {
       item len using also radix = BIN;
    };
};
'>
```

### Example 2

In this example, an item named good_short is defined using an expression involving two other fields in the struct. The good_short coverage item is then extended by adding an **at_least** option. Note that the ": bool = (good == TRUE and len < 4)" part of the original item definition is left out of the extension.

```
<'
struct packet {
    len: uint (bits: 4);
    good: bool;
    event done;
    cover done is {
        item good_short: bool = (good == TRUE and len < 4);
    };
};
'>
<'
extend packet {
    cover done is also {
        item good_short using also at_least = 4;
    };
};
'>
```

### Example 3

In this example, the automatic **prev** variable is used to combine the coverage item option in the original coverage group definition with a narrower definition in the coverage item extension. The combined result is that coverage is collected for the logical AND of (len < 8 and len > 4).

```
<'
struct packet {
    len: uint (bits: 4);
    good: bool;
    event done;
    cover done is {
        item good using when = (len < 8), at_least = 10;
    };
};
'>
<'
extend packet {
    cover done is also {
        item good using also when = (prev and len > 4);
    };
};
'>
```

**Example 4**

This example extends the done coverage group in the good packet subtype, to restrict the sampling of the len item to when the port item is either 0 or 1.

```
<'
struct packet {
    len: uint (bits: 4);
    good: bool;
    port: uint (bits: 2);
    event done;
    when good packet {
        cover done is {
            item len;
        };
    };
};
'>
<'
extend good packet {
    cover done is also {
        item len using also when = port in [0,1];
    };
};
'>
```

**Example 5**

The following shows several examples of coverage group and coverage item extensions, with comments explaining what each one does.

```
<'
struct packet {
    len: uint (bits: 6);
    kind: bool;
    port: int (bits: 4);
    event packet_gen;
    // Original group and item definition:
    cover packet_gen using global, text = "Packet info" is {
        item len using ranges = {range([0..63], "", 16)}, at_least = 5;
        item kind;
        cross len, kind using name = l_k;
    };
    run() is also {emit packet_gen;};
};
'>
<'
extend packet {
    // Cancel the global option:
    cover packet_gen using also global = FALSE;
};
'>
<'
extend packet {
    // Add a new illegal option to the cross item:
    cover packet_gen is also {
        item l_k using also illegal = len == 3;
    };
};
'>
<'
extend packet {
    // Override the previous illegal option of the cross item:
    cover packet_gen is also {
        item l_k using also illegal = len == 4;
    };
};
'>
<'
extend packet {
    // Extend the illegal option, combining the previous definition
    // (prev) with a new condition (len > 10):
    cover packet_gen is also {
        item l_k using also illegal = prev or (len > 10);
    };
};
'>
<'
extend packet {
    // Override the text option of the group, and add a weight option
    // to the kind item:
    cover packet_gen using also text = "Packet Information " is also {
        item kind using also weight = 10;
    };
};
'>
<'
extend packet {
    // Add a new per_instance item to the packet_gen cover group:
    cover packet_gen is also {
        item port using per_instance;
    };
};
'>
<'
extend packet {
    // Exclude instances where the port number is 0:
    cover packet_gen is also {
        item port using also ignore = (port == 0);
    };
};
'>
<'
extend sys {
    packet_list[10]: list of packet;
};
extend global {
    setup_test() is also {
        set_config(cover, mode, on_interactive);
    };
};
'>
```

## See Also

— “Defining Coverage Groups: cover” on page 373

— “Defining Basic Coverage Items” on page 378

## 12.8 Coverage API Methods

This section contains descriptions of the following predefined methods:

- — “scan_cover()” on page 421
- — “start_group()” on page 422
- — “start_instance()” on page 423
- — “start_item()” on page 424
- — “scan_bucket()” on page 424
- — “end_item()” on page 425
- — “end_instance()” on page 426
- — “end_group()” on page 427

### 12.8.1 scan_cover()

#### Purpose

Activate the Coverage API and specify items to cover

#### Category

Method

#### Syntax

**scan_cover(*item-names***: string**)**: int**;**

Syntax example:

```
num_items = cover_info.scan_cover("cpu.inst_driven.*");
```

#### Parameters

| | |
|---|---|
| *item-names* | The names of the coverage items that should be scanned by **scan_cover()**. This is a string of the form ***struct-name.group-name.item-name*** (for example, “inst.start.opcode”). Wild cards are allowed. |

#### Description

The **scan_cover()** method initiates the coverage data-scanning process. It goes through all the items in all the groups specified in the ***item-names*** parameter in the order that groups and items have been defined.

For each group, **scan_cover()** calls **start_group()**. For each instance in the group, **scan_cover()** calls **start_instance()** . For each item in the current instance, **scan_cover()** calls **start_item()**. Then for each bucket of the item, it calls **scan_bucket()**. After all of the buckets of the item have been processed, it calls **end_item()**. After all items of the instance have been processed, it calls **end_instance().** After all instances in the group have been processed, it calls **end_group().**

Before each call to any of the above methods, the relevant fields in the **user_cover_struct** are updated to reflect the current item (and also the current bucket for **scan_bucket()**).

The **scan_cover()** method returns the number of coverage items actually scanned.

NOTE— The **scan_cover()** method cannot be extended. The methods called by **scan_cover()** — **start_group())**, **start_instance()**, **start_item()**, **scan_bucket()**, **end_item()**, **end_instance()** and **end_group()** — are initially empty and are meant to be extended.

#### Example

```
<'
struct simple_cover_struct like user_cover_struct {
};
extend sys {
    !cover_info: simple_cover_struct;
    simple_cover_report() is {
var num_items: int;
num_items = cover_info.scan_cover("cpu.inst_driven.*");
    };
};
'>
```

### 12.8.2 start_group()

#### Purpose

Process coverage group information according to user preferences

#### Category

Method

#### Syntax

**start_group();**

Syntax example:

```
start_group() is {
    if group_text != NULL {out("Description: ", group_text)};
};
```

#### Description

When the **scan_cover()** method initiates the coverage data scanning process for a group, it updates the group-related fields within the containing **user_cover_struct** and then calls the **start_group()** method. The **start_group()** method is called for every group to be processed by **scan_cover()**. For every instance within a group, **scan_cover()** calls the **start_instance()** method.

The **start_group()** method is originally empty. It is meant to be extended to process group data according to user preferences.

NOTE— **start_group()**, **start_instance()** and **scan_cover()** are all methods of the **user_cover_struct**.

### Example

```
<'
struct simple_cover_struct like user_cover_struct {
    start_group() is {
       if group_text != NULL {out("Description: ", group_text)};
    };
};
'>
```

## 12.8.3 start_instance()

### Purpose

Process coverage instance information according to user preferences.

### Category

Method

### Syntax

**start_instance();**

Syntax example:

```
start_instance() is {
    if instance_text != NULL {out("Description: ", instance_text)};
};
```

### Description

When the **scan_cover()** method initiates the coverage data scanning process for an instance, it updates the instance-related fields within the containing **user_cover_struct** and then calls the **start_instance()** method. The **start_instance()** method is called for every instance to be processed by **scan_cover()**.

The **start_instance()** method is originally empty. It is meant to be extended to process instance data according to user preferences.

NOTE— **start_instance()** and **scan_cover()** are methods of the **user_cover_struct**.

### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    start_instance() is {
        out("instance ", struct_name, ".", group_name, ".",
            instance_name, ":");
        if instance_text != NULL {out("Description: ", instance_text)};
    };
};
'>
```

### 12.8.4 start_item()

#### Purpose

Process coverage item information according to user preferences

#### Category

Method

#### Syntax

**start_item();**

Syntax example:

```
start_item() is {
    if item_text != NULL {out("Description: ", item_text)};
};
```

#### Description

When the **scan_cover()** method initiates the coverage data scanning process for an item, it updates the item-related fields within the containing **user_cover_struct** and then calls the **start_item()** method. The **start_item()** method is called for every item to be processed by **scan_cover()**.

The **start_item()** method is originally empty. It is meant to be extended to process item data according to user preferences.

NOTE— **start_item()** and **scan_cover()** are methods of the **user_cover_struct**.

#### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    start_item() is {
        out("item ", struct_name, ".", group_name, ".",
            item_name, ":");
        if item_text != NULL {out("Description: ", item_text)};
    };
};
'>
```

### 12.8.5 scan_bucket()

#### Purpose

Process coverage item information according to user preferences

#### Category

Method

### Syntax

**scan_bucket();**

Syntax example:

```
scan_bucket() is {
    out(count, " ", percent, "% ", bucket_name);
};
```

### Description

When the **scan_cover()** method processes coverage data, then for every bucket of the item it updates the bucket-related fields within the containing **user_cover_struct** and calls **scan_bucket()**.

The **scan_bucket()** method is originally empty. It is meant to be extended to process bucket data according to user preferences.

NOTE— **scan_bucket()** and **scan_cover()** are methods of the **user_cover_struct**.

### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    scan_bucket() is {
        out(str_repeat("   ", cross_level), count, " - ",
            percent, "% - ", status, " - ", bucket_name);
    };
};
'>
```

## 12.8.6 end_item()

### Purpose

Report end of item coverage information according to user preferences

### Category

Method

### Syntax

**end_item();**

Syntax example:

```
end_item() is {
    out("finished item ", item_name, "\n");
};
```

### Description

When the **scan_cover()** method completes the processing of coverage data for an item, it calls the **end_item()** method to report the end of item information according to user preferences.

When all items in the current group have been processed, **scan_cover()** calls the **start_instance()** method for the next instance.

The **end_item()** method is originally empty. It is meant to be extended so as to process item data according to user preferences.

NOTE— **end_item()**, **start_instance()** and **scan_cover()** are all methods of the **user_cover_struct**.

### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    end_item() is {
        out("end of item ", item_name, "\n");
    };
};
'>
```

## 12.8.7 end_instance()

### Purpose

Process end of instance coverage information according to user preferences.

### Category

Method

### Syntax

**end_instance();**

Syntax example:

```
end_instance() is {
    out("finished instance ", instance_name, "\n");
};
```

### Description

When the **scan_cover()** method completes the processing of coverage data for an instance, it calls the **end_instance()** method to report the end of instance information according to user preferences.

When all instances in the current group have been processed, **scan_cover()** calls the **start_group()** method for the next group.

The **end_instance()** method is originally empty. It is meant to be extended so as to process instance data according to user preferences.

NOTE— **end_instance()**, **start_group()** and **scan_cover()** are all methods of the **user_cover_struct**.

### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    end_instance() is {
        out("end of instance ", instance_name, "\n");
    };
};
'>
```

### 12.8.8 end_group()

#### Purpose

Report end of group coverage information according to user preferences

#### Category

Method

#### Syntax

**end_group();**

Syntax example:

```
end_group() is {
    out("finished group", group_name, "\n");
};
```

#### Description

When the **scan_cover()** method completes the processing of coverage data for a group, it calls the **end_group()** method to report the end of group information according to user preferences.

The **end_group()** method is originally empty. It is meant to be extended so as to process item data according to user preferences.

NOTE— **end_group()** and **scan_cover()** are both methods of the **user_cover_struct**.

#### Example

```
<'
struct simple_show_cover_struct like user_cover_struct {
    end_group() is {
        out("end of group", group_name, "\n");
    };
};
'>
```

# 13 Macros

This chapter describes the constructs used to create and debug ***e*** macros. Macro definitions specify a name or a pattern that is to be replaced by ***e*** code text. The constructs for defining and debugging macros are:

— "define as" on page 429
— "define as computed" on page 436

## See Also

— "#define" on page 630
— Chapter 20, "Preprocessor Directives"
— Chapter 21, "Importing e Files"

## 13.1 define as

### Purpose

Define a simple replacement macro

### Category

Statement

### Syntax

**define <*macro-name*'*nonterminal-type*> *match-string* as {"*replacement*"}**

Syntax example:

```
define <largest'action> "largest <exp> <num>" as {
    if <num> > <exp> then {<exp> = <num>};
};
```

## Parameters

*macro-name, nonterminal-type*
: A name you give to the macro and the syntactic type for the macro. The macro can be used wherever it is legal to use the ***nonterminal-type***. The ***macro-name*** and ***nonterminal-type*** together form a unique internal macro name. They must be separated by an apostrophe (').

The ***e*** nonterminal types are shown in Table 13-1.

The combination ***macro-name'nonterminal-type*** must be unique over all ***e*** modules. For example, it is possible to have both a <do_it'statement> and a <do_it'action>, but there cannot be two <do_it'action> macros.

*match-string*
: A quoted string containing text and parsing elements. It may be an expression. Items represented by parsing elements in the ***match-string*** are passed to corresponding parsing elements in the ***replacement***.

Parsing elements for both ***match-string*** and ***replacement*** are shown in Table 13-2. Parsing elements must be used exactly as shown in the table, including the angle brackets (<>). They may be preceded by an identifier and apostrophe (for example, <num> or <big'num>).

*replacement*
: A string containing text and parsing elements, all of which must be legal types in a construct of the ***nonterminal-type***. Each parsing element in the ***replacement*** corresponds to a parsing element of the same name in the ***match-string***.

When the macro is used in the ***e*** code, the ***match-string*** parsing elements are passed to the ***replacement***.

The ***replacement*** can contain replacement terms in angle brackets. If there are any replacement terms in the ***replacement*** string, they represent items for which corresponding items exist in the ***match-string***. When the macro is used, the items provided with the macro are substituted for the replacement terms in the ***replacement***.

Forms for replacement terms are shown in Table 13-3. You can combine the forms in replacement terms.

**Table 13-1—*e* Language Nonterminal Types**

| Nonterminal Type | Description |
|---|---|
| statement | The basic element type of ***e***. |
| action | A procedural element. |
| struct_member | A part of a struct definition. |
| exp | A construct that has a value. |
| type | A type. |
| block | A series of actions enclosed in curly braces ({}). |
| num | A number. |

**Table 13-1—*e* Language Nonterminal Types *(continued)***

| Nonterminal Type | Description |
|---|---|
| file | A file name. |

**Table 13-2—*e* Language Parsing Elements**

| Parsing Element | Description |
|---|---|
| <statement> | Any legal statement. |
| <action> | An action. |
| <command> | A command. |
| <struct_member> | A struct member (event, field, method, coverage group, when struct member, or constraint definition). |
| <exp> | An expression. |
| <name> | A legal name: it must start with a letter, and consist of letters, digits, underscores (_) and single quotes ('). |
| <file> | A file name. |
| <num> | A number. |
| <block> | A series of actions enclosed by {}. |
| <Type> | A type name: this can be used any place in the ***replacement*** where a type is expected. |
| <any> | Anything: any text can be entered for this item when the macro is used; <any> items in the ***match-string*** must have corresponding <any> items in the ***replacement***. |
| [ ] | Items enclosed in square brackets are optional. |
| \| | Items separated by \| (OR bar) are alternatives. |
| ( ) | Parentheses group items for associativity or for readability. |

**Table 13-3—Replacement Term Syntax for define as**

| Replacement Notation | What it Represents | Examples |
|---|---|---|
| <***x_string***> | A string matching <***x_string***> in the input. | <exp> <b'exp> |
| <***x_string***\|***y_string***> | The ***y_string*** is the default value if no match is found for the ***x_string*** in the input. This notation can be used only in the ***replacement***, not in the ***match-string***. | <num\|0> <big'num\|100> |

**Table 13-3—Replacement Term Syntax for define as**

| Replacement Notation | What it Represents | Examples |
|---|---|---|
| <***n***> | The number of the ***n***th substring in the input string. Each <***x_string***> is a substring, and thus can be represented by a number using this syntax. Similarly, each term enclosed in square brackets ([]), parentheses (()), or curly braces ({}) is a substring that can be represented by a number using this syntax. | <1><br><2> |
| <?> | A character sequence of the form __***n***__, where ***n*** is a number that is unique over all expansions of this macro. This is useful for creating unique variable names that will not collide in the various places this macro will be used. | var a<?>: int; |

## Description

You can use replacement macros to extend the ***e*** language, by defining new syntactic elements (actions, commands, and so on). The **define as** statement creates a new syntactic element for the ***e*** parser.

You assign a macro name and an ***e*** nonterminal type (see Table 13-1, "e Language Nonterminal Types", on page 430) to a string pattern. When the ***e*** parser finds a string that matches the pattern, it replaces that string with the replacement string. The replacement specifies what is to be done each time the parser finds the string pattern.

To find a definition of a macro it currently is parsing, the ***e*** parser searches definitions having the desired nonterminal type, starting with the most recent definitions.

## Notes

— A **define as computed** macro takes precedence over a **define as** macro when the match string satisfies either one, regardless of the order in which the macro definitions appear in the code. For example, the command "find least" matches either of the following macros:

```
define <my_first'command> "find least" as {
    action; ...
};


define <my_second'command> "find <any>" as computed {
    action; ...
};
```

However, the second macro, the **define as computed** macro, will always be the one that is applied when "find least", or "find ***anything***", is executed.

— The maximum number of replacement terms you can us in a replacement string is 14. (For some replacement term types, this number may be exceeded, but the results of exceeding 14 replacement terms are unpredictable, and doing so is not recommended.)

### Example 1

Define a new command with the internal name “dir_lis” that executes the UNIX ls command with any of its flags. The new command is invoked by “lis” with a list of flags. It executes the UNIX ls command with those flags:

```
define <dir_lis'command> "lis[<any>]" as {
    print output_from("ls <any>") using items=UNDEF;
};
```

You can enter the following commands, for example, which list files in the current directory. The “-lt” flags and the “*.e” syntax are the same as for the UNIX ls command:

```
> lis
> lis -lt *.e
```

### Example 2

Define a macro that creates a new action with the internal name “simple_frame” to generate a frame with specified field values, and call a method named “send()”:

```
define <simple_frame’action> "send simple frame \
    <dest_addr’num> <source_addr’num> <size’num>" as {
        var f: frame;
        gen f keeping {
            .kind not in [SRAM,DUT];
            .size == <size’num>;
            .dest_address == <dest_addr’num>;
            .source_address == <source_addr’num>;
        };
        start f.send();
};
```

Example of using the “send simple frame” macro:

```
extend sys {
    run() is also { send simple frame 0x00fe 0x0010 0xff };
};
```

### Example 3

Define a new action with the internal name “configure_frame” to generate a frame of a specified “kind” and with an optional “dest_address” value, and call the “send()” method. The “name” element in the match string will be replaced by “.kind” when the macro is used, and the “dest_addr'num” element will be replaced by “.dest_address” if one is entered. If no “dest_address” is given with the macro, a generated value is used:

```
<’
define <configure_frame'action>
    "<name> config frame[ <dest_addr’num>]" as {
    var f: frame;
    if str_empty("<dest_addr’num>") {
        gen f keeping {
            .kind == <name>;
            .size == 64;
        };
    } else {
        gen f keeping {
            .kind == <name>;
            .dest_address == <dest_addr'num|0>;
            .size == 64;
        };
    };
    f.send();
};
type frame_kind: [SRAM, DUT];
struct frame {
    kind: frame_kind;
    size: uint (bits: 16);
    data: byte;
    dest_address: uint;
    send() is {
    'top.frame.addr' = me.dest_address;
    'top.frame.data' = me.data;
    };
};
extend sys {
      run() is also {
        SRAM config frame;
        DUT config frame 0x1001;
    };
};
'>
```

In the “else” block above, the “| 0” following “dest_addr'num” is required, even though this is the condition where a “dest_address” is provided when the macro is used. This is because, even though it is in the else block that is only used when a “dest_address” is provided, the line parses to the following for the case where the optional “dest_addr'num” string is empty:

```
    .dest_address == ;
```

Because that syntax is not allowed by the *e* parser, writing just “<dest_addr'num>” in this line of the macro definition results in an illegal action error at load time. A default number must be included. It can be any value that is legal for “dest_address”.

### Example 4

Define a new action named “issue_struct”, which generates a struct and calls a method named “send()”. Constraints may optionally be entered for struct members. In the **sys** extension the macro is used four times, to generate four “transaction” structs, with constraints specified for the last two.

The Type nonterminal represents “transaction” structs. The <?> element attaches “**__*n*__**” to the “x” variable, where ***n*** is a number that is unique for every usage of the macro. The <2> parsing element, which is replaced by the second element of the match string, is determined by counting "<", "[" and "(" characters, starting from the left-most element. Thus, the <2> parsing element in this example is “keeping <block>” and the second element of the match string must have the form “**keeping** {***constraint***;…};”.

```
    <'
    define <issue_struct'action>
        "issue <Type>[ keeping <block>]" as {
            var x<?>: <Type>;
            gen x<?> <2>;
            x<?>.send();
```

```
};
type command_type: [opA, opB, opC];
struct transaction {
    command: command_type;
    value: int;
    send() is {
       out("\t command: ", command, ", value: ", value, "\n");
    };
};
extend sys {
    !transaction;
    post_generate() is also {
        issue transaction;
        issue opA transaction;
        issue transaction keeping {.command==opB; .value==5};
        issue opC transaction keeping {.value == 6};
    };
};
'>
```

~~To see how “x<?>” is expanded when the macro is parsed, the **trace reparse** command is entered before the example file is loaded. The following is a printout of the results. For each call to the macro, the <?> element is replaced by a unique number.~~

```
~~D> <issue_struct'action> 'issue transaction'~~
~~D> reparsed as: '{var x__14841__: transaction;gen x__14841__~~
    ~~;x__14841__.send()}'~~
~~D> <issue_struct'action> 'issue opA transaction'~~
~~D> reparsed as: '{var x__14842__: opA transaction;gen x__14842__~~
    ~~;x__14842__.send()}'~~
~~D> <issue_struct'action> 'issue transaction keeping {...}'~~
~~D> reparsed as: '{var x__14843__: transaction;gen x__14843__  keeping~~
    ~~{...};x__14843__.send()}'~~
~~D> <issue_struct'action> 'issue opC transaction keeping {...}'~~
~~D> reparsed as: '{var x__14844__: opC transaction;gen x__14844__  keeping~~
    ~~{...};x__14844__.send()}'~~
~~D> esb_reparse:  <outf'exp> reparsed as 'out(...)'~~
~~D> esb_reparse:  <out'exp> reparsed as 'append(...)'~~
```

### Example 5

The following is a definition of an action with the internal name “swap_var”. The match string contains two parsing element items, “<var1'exp>” and “<var2'exp>”, so the “<1>” in the third line corresponds to “<var1'exp>”, the first parsing element in the match string. The notation “<2>” could likewise be used for “<var2'exp>”. Thus, the third line could be written as “<1> = <2|z>”.

```
<'
define <swap_var'action> "swap <var1'exp>[ <var2'exp>]" as {
    var tmp<?> := <var1'exp>;
    <1> = <var2'exp|z>;
    <var2'exp|z> = tmp<?>;
};
extend sys {
    run() is also {
        var a:= 5;
        var b:= 9;
        var z:= 13;
```

```
            swap a b;               // a becomes 9, b becomes 5
            print a, b, z;
            swap a;                 // a becomes 13, z becomes 9
            print a, b, z;
        };
    };
    '>
```

### See Also

— “#define” on page 630
— “define as computed” on page 436

## 13.2 define as computed

### Purpose

Define an advanced replacement macro

### Category

Statement

### Syntax

**define <*macro-name*`*nonterminal-type*> *match-string* as computed {*action*; …}**

Syntax example:

```
    <'
    define <time_command'action> "time <string>" as computed {
        if <1> == "on" {
            return( "{tprint = TRUE; print sys.time;}" );
        } else if <1> == "off" {
            return("tprint = FALSE");
        } else {
            out("Usage: time [on|off]");
        };
    };
    '>
```

## Parameters

| | |
|---|---|
| *macro-name, non-terminal-type* | A name you give to the macro, and the syntactic type for the macro. The ***macro-name*** and ***nonterminal-type*** together form a unique internal name for the macro. They must be separated by an apostrophe ('). <br><br> The macro can be used wherever it is legal to use the ***nonterminal-type***. The ***e*** nonterminal types are shown in Table 13-1, "e Language Nonterminal Types", on page 430. <br><br> The combination ***macro-name'nonterminal-type*** must be unique over all ***e*** modules. For example, it is possible to have both a <do_it'statement> and a <do_it'action>, but there cannot be two <do_it'action> macros. |
| *match-string* | A double-quoted string consisting of text and parsing elements. It may be an expression. Items represented by parsing elements in the ***match-string*** are passed to corresponding parsing elements in the ***replacement***. <br><br> Parsing elements for both ***match-string*** and ***replacement*** are shown in Table 13-2, "e Language Parsing Elements", on page 431, with the exception that <***string***> and <***num***> cannot be used in the **define as computed** replacement string. Parsing elements must be used exactly as shown in the table, including the angle brackets (<>). They may be preceded by an identifier and apostrophe (for example, <exp> or <big'exp>). |
| ***action***; ... | A block of actions that are executed each time ***match-string*** is found. The action block is treated by the parser as the body of a method. Thus you can use the **result** variable or the **return** action to return a result from the action block. <br><br> The action block can contain replacement terms in angle brackets which represent items that will actually be input to the macro when it is used. When the macro is used, the items provided with the macro are substituted for the replacement terms in the action block. <br><br> Legal forms for the replacement terms are shown in Table 13-4. <br><br> You cannot use explicit syntactic parameters such as <name> or <string> in the action block or the computed replacement text. |

### Table 13-4—Replacement Term Syntax for define as computed

| Replacement Notation | What it Represents | Examples |
|---|---|---|
| <***n***> | The number of the ***n***th substring in the input string. Each <***x_string***> is a substring, and thus can be represented by a number using this syntax. Similarly, each term enclosed in square brackets ([]), parentheses (()), or curly braces ({}) is a substring that can be represented by a number using this syntax. | <1> <2> |
| <?> | A character sequence of the form **__*n*__**, where ***n*** is a number that is unique over all expansions of this macro. This is useful for creating unique variable names that will not collide in the various places this macro will be used. | var a<?>: int; |

## Description

This statement creates a new syntactic element that constructs a block of actions based on the inputs to the macro. These actions are executed like a method when the macro is encountered. The actions produce an output string that replaces the match string. The output string is a set of actions that are executed as ***e*** code.

For simple text replacement, use the **define as** statement rather than **define as computed**.

### Space Removal in String Patterns

The preprocessor first compresses all sequences of blanks and tabs into a single blank, before initial matching is done, except that blanks and tabs inside double quotes are not compressed.

### Finding Submatches

The same text may be part of more than one match. For example, " exp" matches both " exp" and "exp". Thus, " +sim" matches " +sim" and "+sim". If you are not interested in the outer match, you can ignore it.

### Expanding Matches

Anything that is in "{}", "()", "[]", or inside double quotes is replaced during preprocessing by a notation of the form "***_number_***". If you want to further parse the match, use **str_expand_dots()** to expand it back to its actual string form.

## Notes

— The returned string cannot contain any newline (\n) characters.

— If the returned string contains more than one action, the actions must be grouped into an action block by enclosing it in curly braces ({}). That is also true of statements and other kinds of struct members: Any semicolon-separated list of constructs in the return string must be enclosed in curly braces.

— The **define as computed** construct actually creates a method that is called during parsing. This method extension does not understand types because the parsing is done before types are identified. For this reason, notations such as <***string***> and <***num***> cannot appear in the ***replacement-string***.

— **define as computed** macros cannot be used in event declarations that use **@sim.**

— **define as computed** macros cannot be used in the same file they are defined in. To use a computed macro in a loaded file, you must also load the file that contains the **define as computed** statement. See the next paragraph for the procedure for using computed macros in compiled files.

— When a file needs to be compiled that uses a **define as computed** macro, first compile the file that defines the macro. The result of the compilation (the executable that includes the macro definition) can then be used to compile any file that uses this newly defined syntactic element.

— When preprocessing **define as computed** macros, the preprocessor might return syntactic blocks in a compressed notation, with dots as placeholders for substrings. If you need to expand these placeholders, use the **str_expand_dots()** routine described in the section on String Routines.

— A **define as computed** macro takes precedence over a **define as** macro when the match string satisfies either one, regardless of the order in which the macro definitions appear in the code. For example, the command "find least" matches either of the following macros:

```
define <my_first'command> "find least" as {
   action; ...
};


define <my_second'command> "find <any>" as computed {
    action ; ...
  };
```

However, the second macro, the **define as computed** macro, will always be the one that is applied when "find least", or "find ***anything***", is executed.

— The maximum number of replacement terms you can us in a replacement string is 14. (For some replacement term types, this number may be exceeded, but the results of exceeding 14 replacement terms are unpredictable, so doing so is not recommended.)

## Example

The macro in this example defines a new "add to list" statement. The statement adds a given number to the list named "num_list" in **sys**, if the number is not already in the list. (A duplicate keyed list is used to check whether the number is already in num_list.) If the given number is already in the list, the macro returns without error.

```
// def_add_to_list.e module:
<'

extend sys {
    !keyed_list: list (key: it) of string;
    adder() is empty;
    !num_list: list of int (bits: 5);
    run() is also {
        adder();
        print num_list;
    };
};

define <num_adder'statement> "add <exp> to list" as computed {
    if sys.keyed_list.key_exists(<1>) { return("{}");}
    else {
        sys.keyed_list.add(<1>);
        result = append( "extend sys { adder() is also \
          {num_list.add(",<1>,")};}");
    };
};

'>
```

The following is an example of using this macro:

1) Load the "def_add_to_list.e" module.
2) Load a module that contains one or more "add ***n*** to list" statements, such as the following:

```
// use_add_to_list.e module:

<'

    add 3 to list;

    add 5 to list;

    add 7 to list;

    add 3 to list;

    add 11 to list;
```

```
'>
```

3) Execute the program.
   The list will contain four items.

```
num_list =  (4 items, dec):
                                                 11   7   5   3           .0
```

### See Also

— “#define” on page 630
— “define as” on page 429

# 14 Checks and Error Handling

The ***e*** language has many constructs that check for errors in the DUT or add exception handling and diagnostics to an ***e*** program. This chapter covers these topics:

- "Handling DUT Errors" on page 441
- "Handling User Errors" on page 450
- "Handling Programming Errors" on page 456

## 14.1 Handling DUT Errors

There are several constructs you can use to perform data or protocol checks on the DUT and to specify how you want to handle any errors that occur:

- "check that" on page 441
- "dut_error()" on page 443
- "dut_error_struct" on page 444
- "set_check()" on page 448

### 14.1.1 check that

#### Purpose

Perform a data comparison and, depending on the results, print a message

#### Category

Action

#### Syntax

**check** [**that**] ***bool-exp*** [**else dut_error(***message*: exp**,** ...)]

Syntax example:

```
check_count(i:int) is {
    check that i == expected_count else
        dut_error("Bad i: ", i);
};
```

NOTE— Keep in mind that **check that**, as with all actions, must be associated with a method. Checks are also created implicitly from **expect** struct members.

### Parameters

| | |
|---|---|
| *bool-exp* | Boolean expression that performs a data comparison. |
| *message* | String or an expression that can be converted to a string. If the ***bool-exp*** is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file if it is open). |

### Description

Performs a data comparison and, depending on the results, prints a message. The following example.

```
check_hard_error() is {
    check that 'top.hard_error'== 1 else
        dut_error("Error-5 -- Hard error not asserted");
};
```

displays an error message like this one:

```
*** Dut error at time 0
        Checked at line 4 in check2.e
        In sys-@0.check_hard_error():

Error-5 -- Hard error not asserted

Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred
```

Using **check that** allows you to:

— ~~Manipulate the response to failed checks with the **set checks** command.~~
— ~~Show which checks have failed with the **show checks** command.~~
— Track the number of failed checks with predefined **session** fields.

### Omitting the else Clause

If you omit the **else dut_error** clause, the ***e*** program uses the **check that** clause as the error message. For example,

```
check_a() is {
    check that a < b;
};
```

displays an error message like this one:

```
*** Dut error at time 0
        Checked at line 6 in check3.e
        In sys-@0.check_a():

check that a < b

Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred
```

### Example

```
check_count(i:int) is {
    check that i == expected_count else
        dut_error("Bad i: ", i);
};
```

### Result

```
*** Dut error at time 0
        Checked at line 6 in check4.e
        In sys-@0.check_count():

Bad i: 4

Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred
```

## 14.1.2 dut_error()

### Purpose

Specify a DUT error message string

### Category

Action

### Syntax

**dut_error(*message***: exp**, ...)**

Syntax example:

```
if 'data_out' != 'data_in' then
  {dut_error("DATA MISMATCH: Expected ", 'data_in')};
```

### Parameters

| | |
|---|---|
| *message* | String or an expression that can be converted to a string. The message expressions are converted to strings, concatenated, and printed to the screen (and to the log file if it is open). |

### Description

Specifies a DUT error message string. This action is usually associated with an **if** action, a **check that** action, or an **expect** struct member. If the boolean expression in the associated action or struct member evaluates to TRUE, then the error message string is displayed.

Calling **dut_error()** directly is exactly equivalent to:

```
check that FALSE else dut_error();
```

NOTE— When you call **dut_error()** directly (not within a **check that** or an **expect**), there is no way to see that a check was successfully performed. **session.check_ok** is always FALSE after a direct call to **dut_error()**.

### Example

```
<'
extend sys {
    m() is {
        if 'data_out' != 'data_in' then
          {dut_error("DATA MISMATCH: Expected ", 'data_in')};
    };
};
'>
```

### Result

```
*** Dut error at time 0
        Checked at line 4 in /tests/check6.e
        In sys-@0.m():

DATA MISMATCH: Expected 1

Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred
```

## 14.1.3 dut_error_struct

### Purpose

Define DUT error response

### Category

Predefined struct

### Syntax

**struct dut_error_struct {**
  ***message***: string**;**
  **source_struct()**: any_struct**;**
  **source_location()**: string;
  **source_struct_name()**: string;
  **source_method_name()**: string;
  **check_effect()**: check_effect**;**
  **set_check_effect(*effect***:check_effect**);**
  **write();**
  **pre_error() is empty;**
**};**

Syntax example:

```
extend dut_error_struct {
    write() is also {
        if source_struct() is a XYZ_packet (p) then {
            print p.parity_calc();
        };
    };
};
```

### Struct Members

| | |
|---|---|
| *message* | The message that was defined by the temporal or data DUT check and is printed by **dut_error_struct.write()**. |
| **source_struct()** | Returns a reference to the struct where the temporal or data DUT check is defined. |
| **source_location()** | Returns a string giving the line number and source module name, for example, “At line 13 in @checker”. |
| source_struct_name() | Returns a string giving the name of the source struct, for example, “packet”. |
| **source_method_name()** | Returns a string giving the name of the method containing the DUT data check, for example, “done()”. |
| **check_effect()** | Returns the check effect of that DUT check, for example, ERROR_AUTOMATIC. |
| **set_check_effect()** | Sets the check effect in this instance of the **dut_error_struct**. You can call this method from **pre_error()** to change the check effect of selected checks. |
| **pre_error()** | The first method that is called when a DUT error occurs, unless the check effect is IGNORE. This method is defined as empty, unless extended by the user. Extending this method lets you modify error handling for a particular instance or set of instances of a DUT error. |
| **write()** | The method that is called after **dut_error_struct.pre_error()** is called when a DUT error happens. This method causes the DUT message to be displayed, unless the check effect is IGNORE. You can extend this method to perform additional actions. |

### Description

The predefined struct **dut_error_struct** defines the DUT error response. To modify the error response, extend either **write()** or **pre_error()**.

Only the **write()** and **pre_error()** methods are called directly by ***e*** program**s,** but you can use the other fields and predefined methods of **dut_error_struct** when you extend **write()** or **pre_error()**.

NOTE— Do not use **dut_error_struct.write()** to change the value of the check effect. Use **pre_error()** instead.

### Example 1

The following code implements a parity checker using DUT error checks. **dut_error_struct.write()** has been extended to print additional information.

```
<'
type XYZ_kind_type : [good, bad] ;

struct XYZ_packet {
    kind : XYZ_kind_type ;
```

```
        %addr : uint (bits : 2) ;
        %len : uint (bits : 6) ;
        %data [len] : list of byte ;
        %parity : byte ;

        parity_calc() : byte is {
            result = addr | (len << 2) ;
            for each (item) in data do {
                result ^= item ;
            };
        };

        parity_check(p: XYZ_packet) is {
            p.kind = ('data' == p.parity_calc()) ? good : bad;
            if (p.kind == good) then {
                check that 'err' == 0 else
                  dut_error ("Err != 0 for good pkt");
            }
            else {
                check that 'err' == 1 else
                  dut_error ("Err != 1 for bad pkt");
            };
        };
    };

    extend dut_error_struct {
        write() is also {
            if source_struct() is a XYZ_packet (p) then {
                print p.parity_calc();
            };
        };
    };

    extend sys {
        packets[10]: list of XYZ_packet;

    check_packets() is {
            for each XYZ_packet (p) in packets {
                p.parity_check(p);
            };
        };

    setup() is also {
            set_check("...Err...pkt...", ERROR_CONTINUE);
        };

        run() is also {
            check_packets();
        };

    };
    '>
```

**Result**

In this test, 9 DUT errors occur.

```
    -------------------------------------------------------
```

```
   *** Dut error at time 0
        Checked at line 24 in check8.e
        In XYZ_packet-@0.parity_check():

Err != 1 for bad pkt
  p.parity_calc() = 228
-------------------------------------------------------
Will continue execution (check effect is ERROR_CONTINUE)

...

No actual running requested.
Checking the test ...
Checking is complete - 9 DUT errors, 0 DUT warnings.
```

### Example 2

This example extends Example 1, extending **pre_error()** so that no more than 3 parity errors will be displayed.

```
<'
extend sys {
    num_parity_errors: uint;
    keep num_parity_errors == 0;
};
extend dut_error_struct {
  pre_error() is also {
    if source_struct() is a XYZ_packet then {
      sys.num_parity_errors = sys.num_parity_errors +1;
      if sys.num_parity_errors > 3 then {
        set_check_effect(IGNORE);
      };
    };
  };
};
'>
```

### Result

```
-------------------------------------------------------
   *** Dut error at time 0
        Checked at line 25 in @checking8
        In XYZ_packet-@0.parity_check():

Err != 1 for bad pkt
  p.parity_calc() = 228
-------------------------------------------------------
```

### Example 3

This example shows how error messages can be handled within units. (See “Units Overview” on page 157 for a description of units and modular verification.)

To print some unit status information upon any error happening within a unit, you could extend **dut_error_struct.write()** as shown below. The call to **try_enclosing_unit()** returns NULL if not called from within a MIPS unit. If called from within a MIPS unit, the status of that MIPS unit is printed.

```
<'
extend dut_error_struct {
    write() is also {
        var MIPS:= source_struct().try_enclosing_unit(MIPS);

        if MIPS != NULL then {
            out("-- Status of ", MIPS.e_path(),
              " at time of error: --");
            MIPS.show_status();
        };
    };
};
'>
```

### 14.1.4 set_check()

#### Purpose

Set check severity

#### Category

Predefined routine

#### Syntax

**set_check(*static-match*:** string**, *check-effect*:** keyword**)**

Syntax example:

```
<'
extend sys {
    setup() is also {
        set_check("...", WARNING);
    };
};
'>
```

## Parameters

| | |
|---|---|
| ***static-match*** | A regular expression enclosed in double quotes. Only checks whose message string matches this regular expression are modified. The match string must use either the native ***e*** syntax or an AWK-like syntax. See "String Matching" on page 51.<br><br>NOTE— You must enclose AWK-like syntax in forward slashes, for example, "/Vio/". Also, the * character in native ***e*** syntax matches only non-white characters. Use ... to match white or non-white characters. |

***check-effect*** is one of the following:

| | |
|---|---|
| ERROR | Issues an error message, increases **num_of_dut_errors**, breaks the run immediately and returns to the simulator prompt. |
| ERROR_BREAK_RUN | Issues an error message, increases **num_of_dut_errors**, breaks the run at the next cycle boundary. |
| ERROR_AUTOMATIC | Issues an error message, increases **num_of_dut_errors**, breaks the run at the next cycle boundary, and performs the end of test checking and finalization of test data that is normally performed when **stop_run()** is called. |
| ERROR_CONTINUE | Issues an error message, increases **num_of_dut_errors**, and continues execution. |
| WARNING | Issues a warning, increases **num_of_dut_warnings** and continues execution. |
| IGNORE | Issues no messages, does not increase **num_of_dut_errors** or **num_of_dut_warnings,** and continues execution. |

## Description

Sets the severity or the check effect of specific DUT checks, so that failing checks will produce errors or warnings.

NOTE— This routine affects only checks that are currently loaded.

— ~~If a DUT check's check effect is ERROR and a **configure run -error_command** option is specified, the error command actions are also executed.~~

## Example

Loading the following extension changes the check effect of all currently defined checks to WARNING during the **setup** phase.

```
<'
extend sys {
    setup() is also {
        set_check("...", WARNING);
    };
};
'>
```

## Result

```
Defined checks =

0. check  @sn_cover in simulator.error_from_simulator (WARNING)
    count=0 Error from simulator: ...
1. check  @sn_cover_misc in covers.cover_dut_error (WARNING)
    count=0 Illegal value for cover item ...
2. check  @sn_coverage in scheduler.handle_event (WARNING)
    count=0 event ... which has a global cover group, occurred
    more than once
3.  check  @check8 in XYZ_packet.parity_check (WARNING) count=0
    Err != 0 for good pkt
4.  check  @check8 in XYZ_packet.parity_check (WARNING) count=0
    Err != 1 for bad pkt
```

## 14.2 Handling User Errors

The ***e*** language has several constructs that help you handle user errors, such as file I/O errors or semantic errors. This section describes the constructs used for handling these kinds of errors:

— "warning()" on page 450, which issues a warning message when a given error occurs.
— "error()" on page 451, which issues an error message and exits when a given error is detected.
— "fatal()" on page 452, which issues an error message and exits to the OS prompt when a given error is detected.
— "try" on page 454, which defines an alternative response for fixing or bypassing an error.

NOTE— Errors handled by these constructs do not increase the **session.num_of_dut_errors** and **session.num_of_dut_warnings** fields that are used to track DUT errors. In addition, the error responses defined with these constructs are not influenced by modifications to the **dut_error_struct.write()** method.

### 14.2.1 warning()

#### Purpose

Issue a warning message

#### Category

Action

#### Syntax

**warning(*message*: string, ...)**

Syntax example:

```
warning("len exceeds 50");
```

### Parameter

| | |
|---|---|
| *message* | String or an expression that can be converted to a string. When the **warning** action is executed, the message expressions are converted to strings, concatenated, and printed to the screen. |

### Description

Issues the specified warning error message. Does not halt the methods being currently run.

### Example

```
check_size() is {
    if (pkt.size != LARGE) {
        warning("packet size is ", pkt.size);
    };
};
```

### Result

```
*** Warning: packet size is SMALL
```

### See Also

- "error()" on page 451
- "fatal()" on page 452
- "try" on page 454
- The **run** option of "set_config()" on page 766

## 14.2.2 error()

### Purpose

Issue an error message

### Category

Action

### Syntax

**error(*message*:** string, ...**)**

Syntax example:

```
check_size() is {
    if (pkt.size != LARGE) {
        error("packet size is ", pkt.size);
    };
};
```

### Parameter

| | |
|---|---|
| *message* | String or an expression that can be converted to a string. When the **error** action is executed, the message expressions are converted to strings, concatenated, and printed to the screen. |

### Description

Issues the specified error message, halts all methods being currently run. The only exception to this is if the **error** action appears inside the first action block given in a **try** action. In that case, the ***e*** program jumps to the **else** action block within the **try** action and continues running.

### Example

```
<'
type size : [SMALL, LARGE];

struct packet {
    size;
};

extend sys {
    pkt: packet;

    check_size() is {
        if (pkt.size != LARGE) {
            error("packet size is ", pkt.size);
        };
    };
};
'>
```

### Result

```
*** Error: packet size is SMALL
```

### See Also

— "warning()" on page 450
— "fatal()" on page 452
— "try" on page 454
— The **run** option of "set_config()" on page 766

## 14.2.3 fatal()

### Purpose

Issue error message and exit to the OS prompt

### Category

Action

### Syntax

**fatal(*message*:** string, ...**)**

Syntax example:

```
fatal("Run-time error - exiting");
```

### Parameter

| | |
|---|---|
| *message* | String or an expression that can be converted to a string. When the **fatal()** action is executed, the message expressions are converted to strings, concatenated, and printed to the screen. |

### Description

Issues the specified error message, halts all activity, exits immediately, and returns to the OS prompt.

**fatal()** returns a non-zero status to the OS shell.

### ~~Using fatal with config run error_command~~

~~You can use **fatal()** with the **-error_command** option of the **config run** command to automatically stop simulation completely when an error occurs. For example, the following code creates the "error_actions()" method, which is called when any error occurs:~~

```
~~extend sys {~~
~~    setup() is also {~~
~~        set_config(run, error_command, "sys.error_actions()");~~
~~    };~~
~~};~~
```

~~And the following code defines "sys.error-actions()" to exit with the **fatal()** action when an error occurs:~~

```
~~extend sys {~~
~~    error_actions() is {~~
~~        -- .. Maybe perform other error actions here~~
~~        fatal("Run-time error - exiting");~~
~~    };~~
~~};~~
```

### Example

The following code shows the use of **warning()**, **error(), fatal()**, and **try**. The code is intended to open a log file. If the log file cannot be opened, the simulation issues a warning and tries to open a temporary log file. If the temporary log file cannot be opened and if the simulation is in batch, it issues an error message and exits to the OS prompt. If the simulation is interactive, it issues an error message only.

```
open_checking_log_file(file_name:string) is {
    try {
        var my_file :file = files.open(file_name, "w",
            "Log file");
    } else {
        warning("Could not open ", file_name,
            "; opening temporary log file sim.log");
        try {
            var my_file :file = files.open("sim.log", "w",
                "Temp Log file");
        } else {
            close_stimulus_log_files();
```

```
                if interactive == FALSE {
                   fatal("Could not open temp file sim.log.\n\n",
                     "Please check write permissions on current",
                     " directory, sim.log, and ", file_name,
                     ".\n\nError level is ", error_level, ".");
                } else {
                   error("Could not open temp file sim.log.\n\n",
                     "Please check write permissions on current",
                     " directory, sim.log, and ", file_name,
                     ".\n\nError level is ", error_level, ".");
                };
            };
        };
    };
```

### See Also

— “warning()” on page 450
— “error()” on page 451
— “try” on page 454
— The **run** of “set_config()” on page 766

## 14.2.4 try

### Purpose

Define an alternative response for fixing or bypassing an error

### Category

Action

### Syntax

**try {*action*; ...} [else {*action*; ...}]**

Syntax example:

```
    try {
        var my_file :file = files.open(file_name, "w",
            "Log file");
        } else {
            warning("Could not open ", file_name,
                "; opening temporary log file sim.log");
        };
```

## Parameters

| | |
|---|---|
| ***action***; ... | A series of zero or more actions enclosed in curly braces and separated by semicolons.<br><br>The first action block (following **try**) cannot include the **fatal()** action. Subsequent action blocks (following **else**) can. |

## Description

Executes the action block following **try**. If an error occurs, executes the action block specified in the **else** clause, in which the error can be fixed or handled. If no error occurs, the **else** clause is skipped.

If you do not specify an **else** clause, execution after errors continues normally from the first action following the **try** block.

## Example

The following code example shows the use of **warning(), error(), fatal()**, and **try.** The code is intended to open a log file. If the log file cannot be opened, the simulation issues a warning and tries to open a temporary log file. If the temporary log file cannot be opened and if the simulation is in batch, it issues an error message and exits to the OS prompt. If the simulation is interactive, it issues an error message only.

```
open_checking_log_file(file_name:string) is {
    try {
        var my_file :file = files.open(file_name, "w",
            "Log file");
    } else {
        warning("Could not open ", file_name,
            "; opening temporary log file sim.log");
        try {
            var my_file :file = files.open("sim.log", "w",
                "Temp Log file");
        } else {
            close_stimulus_log_files();
            if interactive == FALSE {
               fatal("Could not open temp file sim.log.\n\n",
                "Please check write permissions on current",
                " directory, sim.log, and ", file_name,
                ".\n\nError level is ", error_level, ".");
            } else {
               error("Could not open temp file sim.log.\n\n",
                "Please check write permissions on current",
                " directory, sim.log, and ", file_name,
                ".\n\nError level is ", error_level, ".");
            };
        };
    };
};
```

## See Also

— “warning()” on page 450
— “error()” on page 451
— “fatal()” on page 452
— The **run** option of “set_config()” on page 766

## 14.3 Handling Programming Errors

### 14.3.1 Overview

The ***e*** language has a special construct, the **assert** action, to help you handle programming errors, such as internal contradictions or invalid parameters.

### 14.3.2 assert

#### Purpose

Check the ***e*** code for correct behavior

#### Category

Action

#### Syntax

**assert *bool-exp*** [**else error(*message***: string**,** ...)]

Syntax example:

```
assert a < 20;
```

#### Parameters

| | |
|---|---|
| *bool-exp* | Boolean expression that checks the behavior of the code. |
| *message* | String or an expression that can be converted to a string. If the ***bool-exp*** is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file if it is open). |

#### Description

Checks the ***e*** code for correct behavior. Use this action to catch coding errors. When an assert fails, it prints the specified error message plus the line number and name of the file in which the error occurred. If you omit the **else error** clause, **assert** prints a global error message.

NOTE— When an error is encountered, **assert** stops the method being executed.

#### Example

```
<'
extend sys {
    a: uint;

    m() is {
        assert a < 20 else error("The value of a is ", a);
        out("Should never get here if a is 20 or more");
    };
};
'>
```

### Result*** Error: Assertion failed (a programming error):

```
    The value of a is 1840385568
    In 'sys.m()' at line 6 in check22.e
```

### See Also

- "warning()" on page 450
- "error()" on page 451
- "fatal()" on page 452
- "try" on page 454
- The **run** option of "set_config()" on page 766

# 15 Methods

***e*** methods are similar to C functions, Verilog tasks, and VHDL processes. An ***e*** method is an operational procedure containing actions that define its behavior. A method can have parameters, local variables, and a return value. You can define a method only within a struct and you must create an instance of the struct before you can execute the method. When a method is executed, it can manipulate the fields of that struct instance.

You can define methods that execute within a single point of simulation time (within zero time) or methods that execute over multiple cycles. The first type of method is referred to as a regular method. The second type is called a time consuming method or TCM. TCMs are used to synchronize processes in an ***e*** program with processes or events in the DUT. Within a single ***e*** program, multiple TCMs can execute either in sequence or concurrently, along parallel but separate threads. A TCM can also have internal branches, which are multiple action blocks executing concurrently.

Methods defined in one module can later be overwritten, modified or enhanced in subsequent modules using the **extend** mechanism. See "Rules for Defining and Extending Methods" on page 459 for information on how to define and extend methods.

Implementing an ***e*** method is usually done in ***e***. However, you might want to write a C routine and convert it into an ***e*** method. You can do this by declaring an ***e*** method that is implemented in C.

### See Also

— "Rules for Defining and Extending Methods" on page 459
— "Invoking Methods" on page 474
— "Parameter Passing" on page 484

## 15.1 Rules for Defining and Extending Methods

There are two phases in the declaration of regular methods and time-consuming methods (TCMs):

— Introduction
— Extension

You must introduce a method before you extend it. The introduction can be in the same struct as the extension or in any struct that this struct inherits from, but it must precede the extension during file loading.

To introduce a method, you can use:

— **is** [ **C routine** ]
— **is undefined** | **empty**

To extend a method, you can use:

— **is also** | **first** | **only**
— is only C routine

You can also use **is** to extend a method in the following cases:

— The method was previously introduced with **is undefined** or **is empty** and has not been previously extended in this struct or in any struct that inherits from this struct.

— The method was previously introduced (and perhaps extended) in a struct that this struct inherits from, as long as the method has not already been extended in this struct or in any struct that inherits from this struct using **like**.

In these cases, using **is** after **is** or after **is also** | **first** | **only** in a when or like child is similar to using **is only** in this context except that an error message is generated if child is extended more than once or if a child is extended after any of its like grandchild are extended. The advantage of using **is** instead of **is only** is that you will see an error if the method extensions do not occur in the order you expect.

NOTE— As you might expect, if you use **is** after **is** or after **is also** | **first** | **only** in a when or like child or in one of their descendents, you cannot subsequently use **is** to redefine the method in the parent.

Table 15-1, "Rules for Method Extension", on page 460 summarizes the rules for introducing and extending methods. Please keep in mind the following:

— The "none" heading in the table indicates that the method has not been introduced yet.
— The **only** heading represents **is also**, **is first**, and **is only**.
— The "+" character indicates "is allowed".
— The "-" character indicates "is not allowed".
— The "C" character indicates "is allowed" only in a like child, a when child, or one of its descendents.

**Table 15-1—Rules for Method Extension**

| | **Previous Declaration** | | | | |
|---|---|---|---|---|---|
| | none | undefined | empty | is | only |
| **Extending by** | | | | | |
| undefined | + | - | - | - | - |
| empty | + | - | - | - | - |
| is | + | + | + | C | C |
| only | - | + | + | + | + |

## Notes

The following restrictions apply to all methods:

— The maximum number of parameters you can declare for a method is 14.
You can work around this restriction by passing a compound parameter such as a struct or a list.
— You cannot define methods with variable argument lists.
You can work around this restriction by passing a list, which can have variable lengths, or a struct, which can have conditional fields. For example, the following method accepts a list of structs, and performs appropriate operations on each struct in the list, depending on its type:

```
m(l: list of any_struct) is {
    for each (s) in l do {
        if s is a packet (p) then {};
        if s is a cell (c) then {};
```

```
            };
        };
```

This method can then be called as follows:

```
        m({my_cell; my_packet});
```

### Example 1

The following example shows that you can use **is** to extend a method in a child after the method has been introduced with **is** in the parent. However, extending a child more than once with **is** generates an error. Extending a child (B) with **is** after extending its descendant (C) also generates an error.

```
struct A {my_type() is {out("I am type A")}};

struct B like A {};

struct C like B {my_type() is {out("I am type C, grandchild of A")}};

--extending a child more than once with 'is' gives an error
--extend C {my_type() is {out("This extension is not allowed!")}};

--extending a child with 'is' after extending a descendant gives an error
--extend B {my_type() is {out("I am type B, child of A")}};
```

### Example 2

This example shows that extending a method in a child (FALSE'bye A) is allowed, even though the method has been extended in a sibling's descendant (stop bye A).

```
struct A {
    bye:bool;
    greetings() is empty;
};

extend bye A {
    stop:bool;

    when stop bye A{
         greetings() is {out("when bye and stop are TRUE, A says bye")};
    };
};

--this extension of a method in a child is allowed, even
--though the method has been extended in a sibling
extend FALSE'bye A {greetings() is {out("when bye is FALSE, A says hi")}};
```

### Example 3

This example shows that if you use **is** after **is** or after **is also** | **first** | **only** in a when or like child or in one of their descendents, you cannot subsequently use **is** to redefine the method in the parent. The last line in the following example generates an error:

```
struct A {
    bye:bool;
    greetings() is empty;
};

extend bye A {
    stop:bool;

    when stop bye A{
         greetings() is {out("when bye and stop are TRUE, A says bye")};
    };
};

--this extension of a method in the parent generates an error
--extend A {greetings() is {out("A says hello")}};
```

Changing the last line to

```
extend A {greetings() is only {out("A says hello")}};
```

removes the error, but overrides the method definition in all A's subtypes as well.

The following sections describe the syntax for defining and extending methods:

— "method is [inline]" on page 462
— "method @event is" on page 464
— "method [@event] is also | first | only | inline only" on page 467
— "method [@event] is undefined | empty" on page 472

**See Also**

— "Extending Methods in When Subtypes" on page 136
— "Restrictions on Like Inheritance" on page 149

## 15.1.1 method is [inline]

**Purpose**

Declare a regular method

**Category**

Struct member

**Syntax**

***method-name*** **(**[***parameter-list***]**)** [**:** ***return-type***] **is** [**inline**] **{*****action;***...**}**

Syntax example:

```
struct print {
    print_length() is { out("The length is: ", length); };
};
```

## Parameters

| | |
|---|---|
| *method-name* | A legal ***e*** name. See Chapter 2, “e Basics” for more information on names. |
| *parameter-list* | A list composed of zero or more parameter declarations of the form ***param-name*: [*]*param-type*** separated by commas. The parentheses around the parameter list are required even if the parameter list is empty. |
| | param-name: A legal ***e*** name. See Chapter 2, “e Basics” for more information on names. |
| | *: When an asterisk is prefixed to a scalar parameter type, the location of the parameter, not its value, is passed. When an asterisk is prefixed to a list or struct type, the method can completely replace the struct or list. See “Parameter Passing” on page 484 for more information. |
| | param-type: Specifies the parameter type. |
| *return-type* | For methods that return values, specifies the data type of the return value. See Chapter 3, “Data Types” for more information. |
| inline | Defines a new inline method and allows the compiler to optimize the inline method code for the best performance. |
| ***action***;... | A list of zero or more actions. Actions that consume time are illegal in the action block of a regular method. For information on actions, see “Actions” on page 14. |

## Description

An ***e*** method is an operational procedure containing actions that define its behavior. A method can have parameters, local variables, and a return value. You can define a method only within a struct, and you must create an instance of the struct before you can execute the method. When a method is executed, it can manipulate the fields of that struct instance.

Defining a method as **inline** requires the ***e*** program to generate code that enables the C compiler to inline the method. The C compiler to place all the code for the method at each point in the code where the method is called. This inline expansion allows the compiler to optimize the inline method code for the best performance. Methods that are frequently called and involve computation, such as the one shown below, are good candidates for inline definition. This method takes two integers as arguments and returns an integer:

```
struct meth {
    get_free_area_size(size: int, taken: int): int is inline {
        result = size - taken;
    };
};
```

## Notes

In addition to the restrictions on all regular methods (see “Notes” in “Rules for Defining and Extending Methods” on page 459), the following restrictions apply to inline methods:

- — The Gnu C compiler can inline most methods declared as inline without any additional flags. For the Sun native C compiler, you might need to use the -xO4 flag; for the HP native compiler, the +O3 flag.
- — A method originally defined as inline cannot be redefined using **is only**, **is first**, or **is also**.
- — Methods defined in **when** conditional struct members cannot be inline.
- — Time-consuming methods (TCMs) cannot be inline.

— You cannot set a breakpoint on an inline method when there are compiled files that call the method.

### Example

This example shows a method that adds two parameters and returns the result. **result** is an implicit variable of the declared return type **int**.

```
sum(a: int, b: int): int is {
    result = a + b;
};
```

It is legal to assign to the **result** variable implicitly by using the following alternate syntax:

```
sum(a: int, b: int): int is {
    return a + b;
};
```

### See Also

— "method @event is" on page 464
— "method [@event] is also | first | only | inline only" on page 467
— "method [@event] is undefined | empty" on page 472
— "Rules for Defining and Extending Methods" on page 459
— "Invoking Methods" on page 474
— "Parameter Passing" on page 484
— Chapter 23, "Predefined Methods Library"
— Chapter 2, "e Basics" (in particular, "Actions" on page 14)

## 15.1.2 method @event is

### Purpose

Declare a time-consuming method

### Category

Struct member

### Syntax

***method-name*** **(**[***parameter-list***]**)** [**:** ***return-type***]**@*event*** **is {*action;*...}**

Syntax example:

```
struct meth {
    main() @pclk is {
        wait @ready;
        wait [2];
        init_dut();
        emit init_complete;
    };
};
```

## Parameters

| | |
|---|---|
| *method-name* | A legal ***e*** name. See "Chapter 2, "e Basics" for more information on names. |
| *parameter-list* | A list composed of zero or more parameter declarations of the form ***param-name*: [*]*param-type*** separated by commas. The parentheses around the parameter list are required even if the parameter list is empty. |
| | param-name: A legal ***e*** name. See "Chapter 2, "e Basics" for more information on names. |
| | *: When an asterisk is prefixed to a scalar parameter type, the location of the parameter, not its value, is passed. When an asterisk is prefixed to a list or struct type, the method can completely replace the struct or list. See "Parameter Passing" on page 484 for more information. |
| | param-type: Specifies the parameter type. |
| *return-type* | For methods that return values, specifies the data type of the return value. See Chapter 3, "Data Types" for more information. |
| *@event* | Specifies a default sampling event that determines the sampling points of the TCM. This event must be a defined event in ***e*** and serves as the default sampling event for the time consuming method itself as well as for time consuming actions, such as **wait**, within the TCM body. Other sampling points can also be added within the TCM. See Chapter 9, "Temporal Expressions" for information on defining default sampling events. |
| ***action;...*** | A list of zero or more actions, either time-consuming actions or regular actions. For information on actions, see "Actions" on page 14. |

## Description

Defines a new time consuming method (TCM). ***e*** methods are similar to C functions, Verilog tasks, and VHDL processes. An ***e*** method is an operational procedure containing actions that define its behavior. A method can have parameters, local variables, and a return value. You can define a method only within a struct and you must create an instance of the struct before you can execute the method. When a method is executed, it can manipulate the fields of that struct instance.

TCMs can execute over multiple cycles and are used to synchronize processes in an ***e*** program with processes or events in the DUT. TCMs can contain actions that consume time, such as **wait**, **sync**, and **state machine**, and can call other TCMs. Within a single ***e*** program, multiple TCMs can execute either in sequence or in parallel, along separate threads. A TCM can also have internal branches, which are multiple action blocks executing concurrently.

The "main()" TCM shown here waits two "pclk" cycles after the event "ready" occurs. It calls a method to initialize the DUT and emits an event when the initialization is complete.

```
struct meth {
    event pclk is rise('top.pclk')@sim;
    event ready is rise('top.ready')@sim;
    event init_complete;
    init_dut() is empty;
    main() @pclk is {
        wait @ready;
        wait [2];
        init_dut();
        emit init_complete;
        };
    };
```

A TCM can specify events other than the default event as sampling points for actions. For example, adding the "@ready" sampling event to the "wait [2]" causes the TCM to wait two "ready" cycles rather than two pclk cycles:

```
    main() @pclk is {
        wait @ready;
        wait [2] @ready;
        init_dut();
        emit init_complete;
    };
```

For more information on sampling events, see Chapter 8, "Events". For more information on temporal expressions, see Chapter 9, "Temporal Expressions".

## Notes

The following restrictions apply to all TCMs:

- The maximum number of parameters you can declare for a TCM is 14.
  You can work around this restriction by passing a compound parameter such as a struct or a list.
- You cannot define methods with variable argument lists.
  You can work around this restriction by passing a list, which can have variable lengths, or a struct, which can have conditional fields. For example, the following TCM accepts a list of structs, and performs appropriate operations on each struct in the list, depending on its type:

  ```
  m(l: list of any_struct)@send_data is {

      for each (s) in l do {

          if s is a packet (p) then {};

          if s is a cell (c) then {};

      };

  };
  ```

  This method can then be called as follows:

  ```
  start m({my_cell; my_packet});
  ```

## Example

The "init_dut" TCM shown has two branches running in parallel. The first branch is waiting for a "reset" event. The second branch waits for two cycles of "cclk" and then launches three methods that check the state of the DUT. If all three checking methods complete before a reset occurs, then the "Checking is complete..." message is displayed, and the branch that is waiting for the reset event terminates. If the reset occurs before the checking methods have completed, they are terminated and the message "A reset has happened..." is displayed. The "cclk" event and the "reset" event are defined as events in the DUT.

```
    event mready;
    event reset is rise('top.reset')@sim;
    event cclk is rise('top.cclk')@sim;

    init_dut() @cclk is {
```

```
        first of {
            {
                wait @reset;
                out("A reset has happened...");
             };
            {
                wait [2];
                all of {
                    {check_bus_controller()};
                    {check_memory_controller()};
                    {check_alu()};
                };
                out("Checking is complete...");
                emit mready;
             };
        };
    };
```

### See Also

— “method is [inline]” on page 462
— “method [@event] is also | first | only | inline only” on page 467
— “method [@event] is undefined | empty” on page 472
— “Rules for Defining and Extending Methods” on page 459
— “Invoking Methods” on page 474
— “Parameter Passing” on page 484
— Chapter 9, “Temporal Expressions”
— Chapter 10, “Temporal Struct Members”
— Chapter 11, “Time-Consuming Actions”
— “start tcm()” on page 477
— “Semaphore Methods” on page 680
— “event” on page 305

## 15.1.3 method [@event] is also | first | only | inline only

### Purpose

Extend a regular method or a TCM

### Category

Struct member

### Syntax

***method-name*** **([*parameter-list*])** [**:** ***return-type***] [**@*event-type***] **is**
[**also**|**first**|**only**|**inline only**] **{*action;*...}**

Syntax example:

```
    struct meth {
        run() is also {
            out("Starting main...");
            start main();
        };
```

```
};
```

### Parameters

| | |
|---|---|
| *method-name* | The name of the original method. |
| *parameter-list* | Specifies the same parameter list as defined in the original method, or a compile-time error is issued. |
| *return-type* | Specifies the same return value as defined in the original method, or a compile-time error is issued. |
| *@event-type* | Specifies the same sampling event as defined in the original method or a compile-time error is issued. |
| also | The new action block is appended to the end of the original action block. |
| first | The new action block is inserted before the original action block. |
| only | The new action block overrides the original action block. |
| inline only | Replaces the original method definition with an inline definition. The original method must be a regular method, not a TCM. |
| ***action;****...* | A list of zero or more actions. Actions that consume time are illegal in the action block of a regular method. For information on actions, see "Actions" on page 14. |

### Description

Replaces or extends the action block in the original method declaration with the specified action block. The following example extends the struct's predefined method "run()" to start a user-defined TCM. (A TCM is a time-consuming method that is distinguished from a regular method by the presence of ***@event*** and can use time-consuming actions such as sync and wait.)

```
run() is also {
    out("Starting main...");
    start main();
};
```

This example extends the "init_dut()" TCM to start another user-defined TCM, "load_mem". This TCM is called after all the checking methods in the original method have completed successfully.

```
extend check_all {
    load_mem() @cclk is {out("Loading memory...");};

    init_dut() @cclk is also {
        wait @mready;
        start load_mem();
    };
};
```

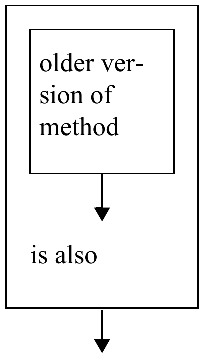
### Notes

— Methods that were originally defined as inline cannot be extended or redefined.
— The original method and its extensions share the **me** and **result** variables. No other local variables can be shared across extensions.
— The following rules apply for **return** actions in extended methods, as illustrated in Figure 15-1 on page 469, Figure 15-2 on page 470 and Figure 15-3 on page 470:
— When an extension issues a return, any actions following that return within the extension itself are not executed.

— When a method is extended with **is also**, the extension starts executing right after the older version of the method completes execution.
— **is also** extensions are executed regardless of whether the older version of the method issues a return or not.
— When a method is extended with **is first**, the older version of the method is never executed if the extension issues a **return**.
— When a method is extended with **is only**, the older version of the method is never executed, whether the extension issues a **return** or not.

Figure 15-1 on page 469 shows how a method with an **is also** extension is executed. The older version executes first and then the **is also** extension. Notice that the "This is also2..." statement is not executed because it follows a return.

**Figure 15-1—Execution of is also Method Extension**



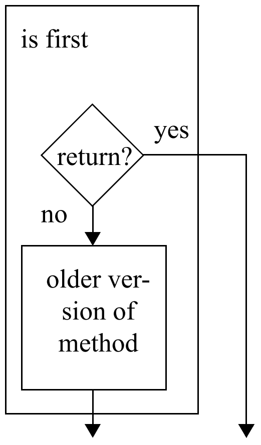
```
module methods1.e
<'
struct meth {
    m() is {
        out("This is...");
    };
};

extend sys {
    mi:meth;
};

extend meth {
    m() is also {
        out("This is also...");
        return;
        out("This is also2...");
    };
};
```

Result

```
This is...
This is also...
```

Figure 15-2 on page 470 shows the same method extended again, this time with **is first**. If a **return** statement is included in the **is first** extension, the older version of the method (the original method definition and the **is also** extension) do not execute. If the **return** statement is deleted, the **is first** extension executes and then the older version of the method executes.

**Figure 15-2—Execution of is first Method Extension**



```
module methods2.e
<'
import methods1.e;
extend meth {
    m() is first {
        out("This is first...");
        return;
    };
};
'>
```

Result with a **return** in **is first**

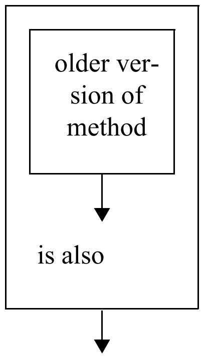
```
   This is first...
```

Result with no **return** in **is first**

```
   This is first...
   This is...
   This is also...
```

Figure 15-3 on page 470 shows another extension with **is also**. Notice that this new extension executes, regardless of whether there is a return in the older version of the method or not.

**Figure 15-3—Execution of is first Method Extension**



```
module methods3.e
<'
import methods2.e;
extend meth {
    m() is also {
        out("This is also3...");
    };
};
'>
```

Result with a **return** in **is first**

```
   This is first...
   This is also3...
```

Result with no **return** in **is first**

```
   This is first...
   This is...
   This is also...
   This is also3...
```

## Example 1

This example redefines the "increment_cnt()" method as an **inline** method.

```
extend meth {
    increment_cnt (cnt: *int) is inline only {
        cnt = cnt + 1;
    };
};
```

## Example 2

In this example, the “show()” method is defined originally to identify the kind of packet. The “show()” method extension displays a different version of the message when the packet has an error.

```
type packet_kind: [ETH, ATM] (bits:8);

struct packet {
    kind: packet_kind;
    has_error: bool;

    show() is {
        out("Packet kind is...", kind);
    };

    when has_error packet{
        show() is only {
            out("This packet has an error...");
        };
    };
};
```

## Example 3

This example extends the “execute()” method to return immediately if the “top.interrupt” signal is active, without executing any of the actions in the original method. Note that the parameter list in the extension is the same as the parameter list in the original method definition, and the sampling event must also be repeated for the extension. Similarly, when the original method definition has a return type, it must be repeated in the method extension.

```
struct ctrl_stub {
    execute(cmd: ctrl_cmd) @cclk is {
        out(appendf("Executing a %s (addr %s) control command",
            cmd.kind, cmd.addr));
        case cmd.kind {
            RD: {
                wait [2];
            };
            WR: {
                wait [2];
            };
        };
    };
};

extend ctrl_stub {
    execute(cmd: ctrl_cmd) @cclk is first {
        if ('top.interrupt' == 1) then {
            return;
            };
        };
    };
```

### See Also

- "method is [inline]" on page 462
- "method @event is" on page 464
- "method [@event] is undefined | empty" on page 472
- "Rules for Defining and Extending Methods" on page 459
- "Extending Methods in When Subtypes" on page 136

## 15.1.4 method [@event] is undefined | empty

### Purpose

Declare an abstract method

### Category

Struct member

### Syntax

***method-name*** **(**[***parameter-list***]**)** [**:** ***return-type***] [**@*event-type***] **is** [**undefined**|**empty**]

Syntax example:

```
    struct packet {
        show() is undefined;
    };
```

### Parameters

| | |
|---|---|
| *method-name* | A legal ***e*** name. See "Chapter 2, "e Basics" for more information on names. |
| *parameter-list* | A list composed of zero or more parameter declarations of the form ***param-name*: [*]*param-type*** separated by commas. The parentheses around the parameter list are required even if the parameter list is empty. |
| | param-name — A legal ***e*** name. See Chapter 2, "e Basics" for more information on names. |
| | * — When an asterisk is prefixed to a scalar parameter type, the location of the parameter, not its value, is passed. When an asterisk is prefixed to a list or struct type, the method can completely replace the struct or list. See "Parameter Passing" on page 484 for more information. |
| | param-type — Specifies the parameter type. |
| *return-type* | For methods that return values, specifies the data type of the return value. See Chapter 3, "Data Types" for more information. |
| *@event-type* | Specifies a default sampling event that determines the sampling points of the TCM. This event must be a defined event in ***e*** and serves as the default sampling event for the TCM itself as well as for time consuming actions, such as **wait**, within the TCM body. Other sampling points can also be added within the TCM. |
| undefined | No action block is defined for the method yet; an action block must be defined in a subsequent module before this method is called. A runtime error is issued if it is called before it is defined. |
| empty | The action block is empty, but no error is issued if it is called. Empty value-returning methods return the default value for the type. |

### Description

Declares an abstract regular method or an abstract TCM with no defined functionality. Abstract methods are place holders that you can extend at a later point. A TCM is a time-consuming method that is distinguished from a regular method by the presence of ***@event*** and can use time-consuming actions such as sync and wait.

### Notes

The following restrictions apply to all abstract methods:

— The maximum number of parameters you can declare for a TCM is 14.
  You can work around this restriction by passing a compound parameter such as a struct or a list.

— You cannot define methods with variable argument lists.
  You can work around this restriction by passing a list, which can have variable lengths, or a struct, which can have conditional fields.

### Example

Undefined or empty methods are often used in base types. This example declares an abstract method "show()" in the base struct "packet" and defines the appropriate functionality in the "Ethernet packet" and "IEEE packet" subtypes.

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

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

    show() is undefined;
};

extend Ethernet packet {
    e_field: int;

    show() is {out("This is an Ethernet packet")};
};

extend IEEE packet {
    i_field: int;

    show() is {out("This is an IEEE packet")};
};

extend sys {
    a: foreign packet;
    b: Ethernet packet;
    c: IEEE packet;

    run() is also {
    a.show();
    b.show();
    c.show();
    };
};
```

### Result

Notice that no message is printed by the foreign packet.

```
This is an Ethernet packet
This is an IEEE packet
No actual running requested.
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

### See Also

- "method is [inline]" on page 462
- "method @event is" on page 464
- "method [@event] is also | first | only | inline only" on page 467
- "Rules for Defining and Extending Methods" on page 459

## 15.2 Invoking Methods

Before invoking a method, you must create an instance of the struct that contains it. The call must conform to the proper syntax and must be made from an appropriate context, as described below.

The following sections describe the two ways to invoke a TCM:

— “tcm()” on page 475
— “start tcm()” on page 477

The following sections describe how you can call regular methods:

— “method()” on page 478
— “compute method()” on page 480

The last section describes the **return** action:

— “return” on page 481

### See Also

— “Rules for Defining and Extending Methods” on page 459
— “Parameter Passing” on page 484

## 15.2.1 tcm()

### Purpose

Call a TCM

### Category

Action or expression

### Syntax

[[***struct-exp***].]***method-name***([***parameter-list***])

Syntax example:

```
init_dut();
```

## Parameters

| | |
|---|---|
| *struct-exp* | The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable **it** is assumed. If both struct expression and the period (.) are missing, the method name is resolved according to the scoping rules. In other words,<br>• .init_dut() means it.init_dut()<br>• init_dut() means me.init_dut(), or if that does not exist, global.init_dut()<br>See "Chapter 2, "e Basics" for more information on naming resolution. |
| *method-name* | The method name as specified in the method definition. |
| *parameter-list* | A list of zero or more parameters separated by commas, one parameter for each parameter in the parameter list of the method definition. Parameters are passed by their relative position in the list, so the name of the parameter being passed does not have to match the name of the parameter in the method definition. The parentheses around the parameter list are required even if the parameter list is empty. |

## Description

You can call a TCM only from another TCM.

A TCM that does not return a value can be started (see "start tcm()" on page 477) or called. A call of a TCM that does not return a value is syntactically an action.

A call of a TCM that does return a value is an expression, and the return type of the TCM must conform to the type of the variable or field it is assigned to.

A called TCM begins execution either when its sampling event occurs or immediately, if the sampling event has already occurred for the current tick.

The calling TCM waits until the called TCM returns before continuing execution. For this reason, a called TCM is considered a subthread of the calling TCM and shares the same thread handle (thread ID) with the calling TCM. In contrast, a started TCM runs in parallel with the TCM that started it, on a separate thread.

NOTE— You cannot call a TCM from a regular method. To invoke a TCM from within a regular method, use **start**.

## Example

This example shows how to call a TCM from another TCM.

```
struct meth {
    event pclk is rise('top.pclk')@sim;
    event ready is rise('top.ready')@sim;
    event init_complete;
    init_dut() @pclk is empty;
    main() @pclk is {
        wait @ready;
        wait [2];
        init_dut();
        emit init_complete;
        };
    };
```

### See Also

- "start tcm()" on page 477
- Chapter 11, "Time-Consuming Actions"
- "Struct Hierarchy and Name Resolution" on page 19
- "Invoking Methods" on page 474
- "Rules for Defining and Extending Methods" on page 459
- "Parameter Passing" on page 484

## 15.2.2 start tcm()

### Purpose

Start a TCM

### Category

Action

### Syntax

**start** [[***struct-exp***].]***method-name***(**[*parameter-list*]**)

Syntax example:

```
start main();
```

### Parameters

| | |
|---|---|
| *struct-exp* | The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable **it** is assumed. If both struct expression and the period (.) are missing, the method name is resolved according to the scoping rules. See "Chapter 2, "e Basics" for more information on naming resolution. |
| *method-name* | The method name as specified in the method definition. |
| *parameter-list* | A list of zero or more parameters separated by commas, one parameter for each parameter in the parameter list of the method definition. Parameters are passed by their relative position in the list, so the name of the parameter being passed does not have to match the name of the parameter in the method definition. The parentheses around the parameter list are required even if the parameter list is empty. |

### Description

You can use a **start** action within another method, either a TCM or a regular method. A started TCM begins execution either when its sampling event occurs or immediately, if the sampling event has already occurred for the current tick.

A started TCM runs in parallel with the TCM that started it on a separate thread. A started TCM has a unique thread handle (thread ID) that is assigned to it automatically by the **scheduler**. You can retrieve this handle using one of the predefined methods of the scheduler.

The recommended way to start an initial TCM, which can then invoke other TCMs, is to extend the related struct's predefined **run()** method.

NOTE— A TCM that has a return value cannot be started with a **start** action.

### Example

This example shows how to extend a struct's **run()** method to start a TCM. Note that the **start** syntax omits the default sampling event.

```
struct meth {
    event clk is rise('top.clk');
    run() is also {
        out("Starting main...");
        start main();
    };
};
```

### See Also

- — "The run() Method of sys" on page 646
- — Chapter 11, "Time-Consuming Actions"
- — "Struct Hierarchy and Name Resolution" on page 19
- — "Invoking Methods" on page 474
- — "Rules for Defining and Extending Methods" on page 459
- — "Parameter Passing" on page 484

## 15.2.3 method()

### Purpose

Call a regular method

### Category

Action or expression

### Syntax

[[***struct-exp***].]***method-name*****(**[***parameter-list***]**)**

Syntax example:

```
tmp1 = get_free_area_size(size, taken);
```

## Parameters

| | |
|---|---|
| *struct-exp* | The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable **it** is assumed. If both struct expression and the period (.) are missing, the method name is resolved according to the scoping rules. See Chapter 2, "e Basics", for more information about naming resolution. |
| *method-name* | The method name as specified in the method definition. |
| *parameter-list* | A list of zero or more parameters separated by commas, one parameter for each parameter in the parameter list of the method definition. Parameters are passed by their relative position in the list, so the name of the parameter being passed does not have to match the name of the parameter in the method definition. The parentheses around the parameter list are required even if the parameter list is empty. |

## Description

The proper context for calling a regular method depends on whether the method returns a value or not.

— If the method returns a value, it is an expression and can be called from any context where an expression is valid.
— If the method does not return a value, it is an action and can be called from any context where an action is valid.

## Example 1

Two common contexts for calling value-returning methods are shown below.

```
m() is {
    var tmp1: int;
    tmp1 = get_free_area_size(size, taken);
    print tmp1;
};
keep length <= get_free_area_size(size, taken);
```

When placed on the right-hand side of an assignment operator, the method's return value type must conform to the type of the variable or field it is assigned to.

## Example 2

In some cases you may want to call a value-returning method without using the value that is returned. To do this, you can use the **compute** action. In the example shown below, the "m()" method increments the "counter" variable, but does not use the value returned.

```
inc_counter() : int is {
    counter += 1;
    result = counter;
};
m() is {
    if 'top.b' > 15 {compute inc_counter();};
};
```

### Example 3

You can call regular methods that do not return values either from other methods, including TCMs, from action blocks associated with other constructs, as shown below.

```
event alu_watcher is {rise('inst_start') exec {
    var i: inst;
    i = new;
    instructions.add(i);
    };
};
```

### See Also

- “compute method()” on page 480
- “Struct Hierarchy and Name Resolution” on page 19
- “Rules for Defining and Extending Methods” on page 459
- “Parameter Passing” on page 484

## 15.2.4 compute method()

### Purpose

Compute a regular method

### Category

Action

### Syntax

**compute** [[***struct-exp***].]***method-name***([***parameter-list***])

Syntax example:

```
if 'top.b' > 15 {compute inc_counter();};
```

### Parameters

| | |
|---|---|
| *struct-exp* | The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable **it** is assumed. If both struct expression and the period (.) are missing, the method name is resolved according to the scoping rules. See Chapter 2, “e Basics”, for more information about naming resolution. |
| *method-name* | The method name as specified in the method definition. |
| *parameter-list* | A list of zero or more parameters separated by commas, one parameter for each parameter in the parameter list of the method definition. Parameters are passed by their relative position in the list, so the name of the parameter being passed does not have to match the name of the parameter in the method definition. The parentheses around the parameter list are required even if the parameter list is empty. |

### Description

In some cases you may want to call a value-returning method without using the value that is returned. To do this, you can use the **compute** action.

### Example

In the example shown below, the “m()” method increments the “counter” variable, but does not use the value returned.

```
inc_counter() : int is {
    counter += 1;
    result = counter;
};
m() is {
    if 'top.b' > 15 {compute inc_counter();};
};
```

### See Also

— “method()” on page 478
— “Struct Hierarchy and Name Resolution” on page 19
— “Rules for Defining and Extending Methods” on page 459
— “Parameter Passing” on page 484

## 15.2.5 return

### Purpose

Return from regular method or a TCM

### Category

Action

### Syntax

**return** [*exp*]

Syntax example:

```
return i*i;
```

## Parameters

*exp* In value-returning methods, an expression specifying the return value is required in each **return** action. In non-value-returning methods, expressions are not allowed in **return** actions.

## Description

Returns immediately from the current method to the method that called it. The execution of the calling method then continues.

It is not always necessary to provide a **return** action. When a value returning method ends without a **return** action, the value of **result** is returned.

## Notes

- The **return** action must be used carefully in method extensions. See "method [@event] is also | first | only | inline only" on page 467 for more information.
- Any actions that follow a **return** action in the method definition are ignored.
- Actions placed in a method extension are performed before the **return** is executed.

## Example 1

This example shows **return** in a value-returning expression.

```
<'
extend sys {
    sqr(i: uint): uint is {
        return i*i;
    };
};
'>
```

## Example 2

This example shows **return** in a non-value-returning method named "start_eng()".

```
<'
struct eng_str {
    on: bool;
};

struct st_eng {
    engine: eng_str;
    start_eng(e: eng_str) is {
        if e.on then {
            out("engine is already on");
            return;
        };
        e.on = TRUE;
        out("engine has been turned on");
    };
};
'>
```

### Example 3

For value-returning methods, instead of a **return**, the special variable **result** can be assigned and its value is returned. In the example below, if “t” is less than 100, the “sqr_1()” method exits, returning “t”. Otherwise, it returns 101.

```
<'
extend sys {
    sqr_1(i: uint): uint is {
        var t := i*i;
        if t < 100 then {
            return t;
        };
        result = 101;
    };
};
'>
```

### Example 4

This example illustrates that any actions following a **return** action in a method definition or in a method extension are ignored:

```
<'
extend sys {
    m1(): int is {
        return 5;
        result = 0;
        out ("FAIL");
    };
    m1():int is also {
        return result + 7;
        out ("FAIL");
    };
};
'>
```

### Result

**print sys.m1() using dec**

```
sys.m1() = 12
```

### Example 5

The following example shows a method that has a compound type as a return value. In the get_alpha_num() method, the **return** action calls another method, select_list(), which has an index, slctr, which can have a value from 0 to 3.

The select_list() method returns a list of strings (“a0”, “a1”, “a2”, “a3”, for example), which is determined by the value (A, B, C, or D) of the ALPHA field.

In the call to select_list(), the slctr value is used as an index into the list of strings returned from the **case** action by select_list(). Thus, the get_alpha_num() method, called in **run()** in **sys**, returns the string with index = slctr from the list for case = ALPHA.

```
<'
```

```
type a_type: [A, B, C, D, E];
struct top {
    ALPHA: a_type;
    slctr: uint (bits: 2);
    select_list() :list of string is {
        case ALPHA {
            A : { return {"a0"; "a1"; "a2"; "a3"}; };
            B : { return {"b0"; "b1"; "b2"; "b3"}; };
            C : { return {"c0"; "c1"; "c2"; "c3"}; };
            D : { return {"d0"; "d1"; "d2"; "d3"}; };
            E : { return {"e0"; "e1"; "e2"; "e3"}; };
        };
    };

    get_alpha_num(slctr: uint): string is {
        return select_list()[slctr];
    };
};

extend sys {
    top;
    run() is also {
        print top.ALPHA;
        print top.slctr;
        var an_strng: string;
        an_strng = top.get_alpha_num(top.slctr);
        print an_strng;
    };
};
'>
```

### Result

```
Running the test ...
  top.ALPHA = A
  top.slctr = 2
  an_strng = "a2" // "a2" is the string with index 2 in list A
```

### See Also

— "method [@event] is also | first | only | inline only" on page 467

## 15.3 Parameter Passing

How a parameter is passed depends on whether the parameter is scalar or compound, as described in these sections:

— "Scalar Parameter Passing" on page 484
— "Compound Parameter Passing" on page 485
— "Notes on Passing by Reference" on page 486

### 15.3.1 Scalar Parameter Passing

Scalar parameters include numeric, boolean, and enumerated types. When you pass a scalar parameter to a method, by default the value of the parameter is passed. This is called "passing by value". Any change to the value of that parameter by the method applies only within that method instance and is lost when the method returns. For example, the “increment()” method defined below increments the value of the parameter passed to it.

```
increment (cnt: int) is {
    cnt = cnt + 1;
};
m() is {
    var tmp: int = 7;
    increment(tmp);
    print tmp;
};
```

However, since “cnt” is passed by value, the variable “tmp” retains its original value, and the print statement displays:

```
tmp = 7
```

To allow a method to modify the parameter, prefix the parameter type with an asterisk. This is called “passing by reference”. If you modify the “increment() “method as follows:

```
increment (cnt: *int) is {
    cnt = cnt + 1;
};
m() is {
    var tmp: int = 7;
    increment(tmp);
    print tmp;
};
```

“tmp” has the value 8 after the method returns. Note that the asterisk is used only in the method definition, not in the method call.

## 15.3.2 Compound Parameter Passing

Compound parameters are either structs or lists. Passing a struct or a list to a method allows the method to modify the struct fields and the list items. This is called “passing by reference”. Thus, if you modify the “increment()” method to accept a list:

```
increment_list (cnt: list of int) is {
    for each in cnt {
        cnt[index] = cnt[index] + 1;
    };
};
```

and pass a list of integers to it, each item in the list reflects its incremented value after the method returns.

Placing an asterisk in front of the list or struct type allows the method to completely replace the struct or list. For example, the “create_if_illegal()” method accepts a struct instance of type packet. If it determines that the “legal” field of struct instance is FALSE, it allocates a new struct instance of type “packet”.

```
create_if_illegal(pkt: *packet) is {
    if pkt.legal == FALSE then {
        pkt = new;
    };
};
```

### 15.3.3 Notes on Passing by Reference

There are several restrictions that apply when you pass parameters by reference:

- There is no automatic casting to a reference parameter. Thus, if you try to pass a variable that is declared as a type other than a 32-bit **int** to the “increment()” method, you get a compile-time error.
- You cannot pass a list element by reference. Thus, if a variable “tmp” is declared as a **list of int**, it is illegal to pass “tmp[***index***]” to “increment()”, as defined above.
- An expression that cannot be placed on the left-hand-side of an assignment cannot be passed by reference.
- Called TCMs can accept reference parameters, but started TCMs cannot.

# 16 Creating and Modifying *e* Variables

The following sections describe how to create and assign values to ***e*** variables:

## 16.1 About *e* Variables

An ***e*** variable is a named data object of a declared type. ***e*** variables are declared and manipulated in methods. They are dynamic; they do not retain their values across subsequent calls to the same method.

The scope of an ***e*** variable is the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes. Variable scoping is described in more detail in “Struct Hierarchy and Name Resolution” on page 19.

Some ***e*** actions create implicit variables. They are described in more detail in “Implicit Variables” on page 24.

The following sections describe the actions that create and modify ***e*** variables explicitly:

— “var” on page 487
— “=” on page 489
— “op=” on page 491

## 16.2 var

### Title

Variable declaration

### Category

Action

### Syntax

**var** ***name*** [**:** [***type***] [= ***exp***]]

Syntax example:

```
var a: int;
```

## Parameters

| | |
|---|---|
| *name* | A legal ***e*** name. |
| *type* | A declared ***e*** type. The type can be omitted if the variable name is the same as the name of a struct type or if the variable is assigned a typed expression. |
| *exp* | The initial value of the variable. If no initial value is specified, the variables are initialized to 0 for integer types, NULL for structs, FALSE for boolean types, and lists as empty. |

## Description

Declares a new variable with the specified name as an element or list of elements of the specified type, and having an optional initial value.

The **var** action is legal in any place that an action is legal, and the variable is recognized from that point on. ***e*** variables are dynamic; they do not retain their values across subsequent calls to the same method.

The scope of an ***e*** variable is the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes. Variable scoping is described in more detail in "Struct Hierarchy and Name Resolution" on page 19.

## Example 1

This example shows the declaration of two variables, one with an assigned initial value:

```
<'
extend sys {
    m() is {
        var a: int;
        var m: int = 2 + a;
    };
};
'>
```

## Example 2

This example shows the keywords list of used to create a list.

```
<'
struct packet {
    protocol: [atm, eth, other];
    len: uint [0..10];
    data[len]: list of byte;
};

extend sys {
    m() is {
        var packets: list of packet;
    };
};
'>
```

### Example 3

This example shows a variable declarations with the type omitted. The variable “packet” is assumed to be of type “packet”.

```
<'
struct packet {
    protocol: [atm, eth, other];
    len: uint [0..10];
    data[len]: list of byte;
};

extend sys {
    m() is {
        var packet;
    };
};
'>
```

### Example 4

In this example, “p” gets type packet, because that is the type of “my_packets[3]”, and “z” gets type **int**, because 5 has type **int**.

```
<'
extend sys {
    my_packets: list of packet;
    m() is {
        var p := my_packets[3];
        var z := 5;
    };
};
'>
```

## 16.3 =

### Purpose

Simple assignment

### Category

Action

### Syntax

*lhs-exp=exp*

Syntax example:

```
sys.u = 0x2345;
```

## Parameters

| | |
|---|---|
| *lhs-exp* | A legal ***e*** expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object. The expression can contain the list index operator [n], the bit access operator [i:j], or the bit concatenation operator %{}. |
| *exp* | A legal ***e*** expression, either an untyped expression (such as an HDL object) or an expression of the same type as the left-hand-side expression. |

## Description

Assigns the value of the right-hand-side expression to the left-hand-side expression.

NOTE— There are two other places within the ***e*** language which make use of the equal sign. These are a double equal sign (==) for specifying equality in boolean expression, and a triple equal sign (===) for the Verilog-like identity operator. These two operators should not be confused with the single equal sign (=) assignment operator.

## Example 1

This example shows the operators that are allowed in the left-hand-side expression.

```
<'
struct n {
    m() is {
        var i: int (bits:16);
        var j: int (bits:16);
        var lint: list of int = {15;31;63;127};

        sys.u = 0x2345;
        print sys.u;
        lint[0] = 0x98765432;
        print lint[0];
        i[1:0] = 3;
        print i;
        %{i,j} = lint[0];
        print i, j;
    };
};

extend sys {
u:uint;
ni:n;
};
'>
```

## Result

**sys.ni.m()**

```
  sys.u = 0x2345
  lint[0] = 0x98765432
  i = 0x0003
  i = 0x9876
  j = 0x5432
```

### Example 2

This example shows the assignment operator used in initialization.

```
<'
struct packet {
   good : bool;
   size : [small, medium, large];
    length : int;
};
extend sys {
   post_generate() is also {
        var p : packet = new;
        print p;
        var q : packet = new good large packet;
        print q;
        var x := new packet (p) with {
               p.length = 5;
            print p;
        };
    };
};
'>
```

### See Also

— “Untyped Expressions” on page 87
— “Assignment Rules” on page 89
— “Precision Rules for Numeric Operations” on page 93
— “Automatic Type Casting” on page 96

## 16.4 op=

### Purpose

Compound assignment

### Category

Action

### Syntax

***lhs-exp op=exp***

Syntax example:

```
sys.c.count1 += 5;
```

### Parameters

| | |
|---|---|
| *lhs-exp* | A legal ***e*** expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object. |
| *exp* | A legal ***e*** expression of the same type as the left-hand-side expression. |
| *op* | A binary operator, including binary bitwise operators (except ~), the boolean operators **and** and **or**, and the binary arithmetic operators. |

### Description

Performs the specified operation on the two expressions and assigns the result to the left-hand-side expression.

### Example 1

This example shows the compound assignment operator used with arithmetic operators.

```
<'
struct counter {
    count1: uint;
    mult: uint;

    keep count1 == 0;
    keep mult == 2;
};

extend sys {
    c: counter;
    m() is {
        var i: int = 2;
        sys.c.count1 += 5;
        sys.c.mult *= i;
        print sys.c.count1, sys.c.mult;
    };
};
'>
```

### Result

**sys.m()**

```
  sys.c.count1 = 0x5
  sys.c.mult = 0x4
```

### Example 2

This example shows the compound assignment operator used with the shift operator.

```
<'
extend sys {
    m() is {
        print 'top.address';
        'top.address' <<= 4;
        print 'top.address';
    };
};
'>
```

### Result

**sys.m()**
```
  'top.address' = 0xff
  'top.address' = 0xff0
```

### Example 3

This example shows the compound assignment operator used with a boolean operator.

```
<'
extend sys {
    m() is {
        var is_ok: bool = TRUE;
        var is_legal: bool;
        is_ok or= is_legal;
        print is_ok;
    };
};
'>
```

### Result

**sys.m()**
```
  is_ok = TRUE
```

## 16.5 <=

### Purpose

Delayed assignment

### Category

Action

### Syntax

[***struct-exp***.]***field-name*** <= ***exp***

Syntax examples:

```
da <= da+1;
```

## Parameters

| | |
|---|---|
| struct-exp | A legal ***e*** expression that evaluates to a struct. The default is **me**. |
| field-name | A field of the struct referenced by ***struct-exp.*** |
| exp | A legal ***e*** expression, either an untyped expression (such as an HDL object) or an expression of the same type as the left-hand-side expression. |

## Description

The delayed assignment action assigns a struct field just before the next **@sys.new_time** after the action. The purpose is to support raceless coding in ***e*** by providing the same results regardless of the evaluation order of TCMs and temporal expressions. (See "Simulation Time and Ticks" on page 312 for a description of **@sys.new_time**.)

Both expressions are evaluated immediately (not delayed) in the current context. The assignment is not considered a time-consuming action, so you can use it in both TCMs and in regular methods, in **on** action blocks and in **exec** action blocks.

If a field has multiple delayed assignments in the same cycle, they are performed in the specified order. The final result is taken from the last delayed assignment action.

Unlike in HDL languages, the delayed assignment in ***e*** does not emit any events; thus, zero delay iterations are not supported.

NOTE— The left-hand-side expression in the delayed assignment action can only be a field. Unlike the assignment action, the delayed assignment action does not accept assignment to any of the following:

— A variable of a method
— A list item
— A bit
— A bit slice
— A bit concatenation expression

## Example

The following example shows how delayed assignment provides raceless coding. In this example there is one incrementing() TCM, which repeatedly increments the sys.a and sys.da fields, and one observer() TCM, which observes their value.

```
<'
extend sys {
    !a   : int;
    !da  : int;

    incrementing()@any is {
        for i from 1 to 5 {
            a = a+1;
            da <= da+1;
            wait cycle;
        };
        stop_run();
    };

    observer()@any is {
        while (TRUE) {
            out( "observing 'a' as ", a, " observing 'da' as ", da );
            wait cycle;
        };
    };

    run() is also {
        start observer();
        start incrementing();
    };
};
'>
```

## Result

From the results you can see that the value of sys.a observed by the observer() TCM is order-dependent, depending whether observer() is executed before or after incrementing(). The observed value of sys.da, however, is independent of the execution order. Even if incrementing() runs first, sys.da gets its incremented value just before the next **new_time** event and thus is not be seen by observer().

If observer() runs before incrementing():

```
observing 'a' as 0 observing 'da' as 0
-----------------------------
observing 'a' as 1 observing 'da' as 1
-----------------------------
observing 'a' as 2 observing 'da' as 2
-----------------------------
observing 'a' as 3 observing 'da' as 3
-----------------------------
observing 'a' as 4 observing 'da' as 4
-----------------------------
```

If incrementing() runs before observer():

```
observing 'a' as 1 observing 'da' as 0
-----------------------------
observing 'a' as 2 observing 'da' as 1
-----------------------------
observing 'a' as 3 observing 'da' as 2
-----------------------------
observing 'a' as 4 observing 'da' as 3
-----------------------------
observing 'a' as 5 observing 'da' as 4
-----------------------------
```

# 17 Packing and Unpacking

Packing performs concatenation of scalars, strings, list elements, or struct fields in the order that you specify. Unpacking performs the reverse operation, splitting a single expression into multiple expressions.

As part of the concatenation or splitting process, packing and unpacking also perform type conversion between any of the following:

— scalars
— strings
— lists and list subtypes (same type but different width)

For type conversion, ***e*** provides additional techniques. Here are some general recommendations on when to use each technique:

| | |
|---|---|
| as_a() | Recommended for converting a single scalar to another scalar type, for example, from a 32-bit integer to an 8-bit integer. It is also recommended for conversion between strings and lists of ASCII bytes. For more information, see "as_a()" on page 104. |
| sublisting with [..] | Recommended for converting a single scalar to a list of bit. For more information, see Chapter 2, "e Basics". |
| bit extraction with [:] | Recommended for converting a list of bit into a single scalar. For more information, see Chapter 2, "e Basics". |
| unpacking | Recommended for converting from a list of bit to strings, lists, structs, or multiple scalars. |
| packing | Recommended for all other purposes. |

This chapter contains the following sections

— "Basic Packing" on page 497
— "Advanced Packing" on page 505
— "Constructs for Packing and Unpacking" on page 516

## 17.1 Basic Packing

Packing and unpacking operate on scalars, strings, lists and structs. The following sections show how to perform basic packing and unpacking of these data types using two of the***e*** basic packing tools, the **pack()** and **unpack()** methods.

— "A Simple Example of Packing" on page 498
— "A Simple Example of Unpacking" on page 500
— "Packing and Unpacking Scalar Expressions" on page 501
— "Packing and Unpacking Strings" on page 501
— "Packing and Unpacking Structs" on page 502
— "Packing and Unpacking Lists" on page 503

For information on two other basic tools for packing, see:

— "%{... , ...}" on page 62
— "swap()" on page 524

### See Also

— "Advanced Packing" on page 505
— "Constructs for Packing and Unpacking" on page 516

### 17.1.1 A Simple Example of Packing

This example shows how packing converts data from a struct into a stream of bits. An "instruction" struct is defined as:

```
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);

    !data_packed_high : list of bit;
    !data_packed_low  : list of bit;

    keep opcode ==  0b100;
    keep operand == 0b11001;
    keep address == 0b00001111;
};
```

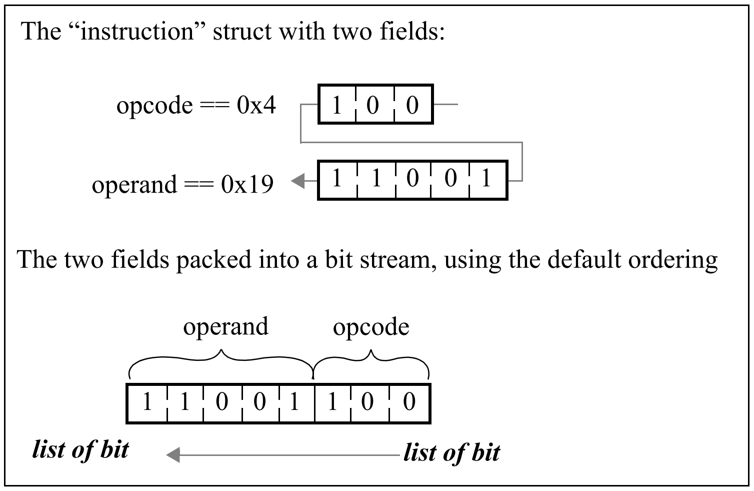
The **post_generate()** method of this struct is extended to pack the "opcode" and the "operand" fields into two variables. The order in which the fields are packed is controlled with the **packing.low** and **packing.high** options:

```
data_packed_low = pack(packing.low, opcode, operand);
data_packed_high = pack(packing.high, opcode, operand);
```

With the **packing.low** option, the least significant bit of the first expression in **pack()**, "opcode", is placed at index [0] in the resulting list of bit. The most significant bit of the last expression, "operand", is placed at the highest index in the resulting list of bit. Figure 17-1 on page 499 shows this packing order.

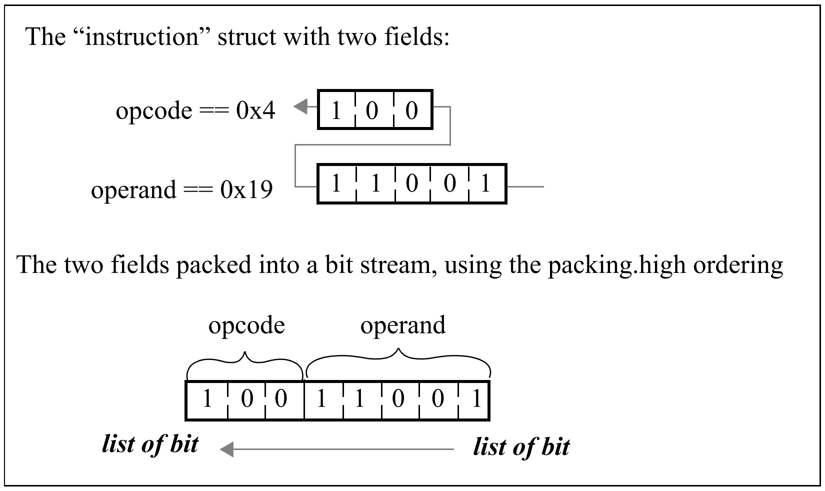
**packing.low** is the default packing order.

**Figure 17-1—Simple Packing Example Showing packing.low**


The "instruction" struct with two fields:
opcode == 0x4
1 0 0
operand == 0x19
1 1 0 0 1
The two fields packed into a bit stream, using the default ordering
operand
opcode
1 1 0 0 1 1 0 0
list of bit
list of bit

With **packing.high**, the least significant bit of the last expression, "operand", is placed at index [0] in the resulting list of bit. The most significant bit of the first expression, "opcode", is placed at the highest index in the resulting list of bit. Figure 17-2 shows this packing order.

**Figure 17-2—Simple Packing Example Showing packing.high**


The "instruction" struct with two fields:
opcode == 0x4
1 0 0
operand == 0x19
1 1 0 0 1
The two fields packed into a bit stream, using the packing.high ordering
opcode
operand
1 0 0 1 1 0 0 1
list of bit
list of bit

Pack expressions, like the ones shown in the example above, are untyped expressions. In many cases, as in this example, the ***e*** program can deduce the required type from the context of the pack expression. See "Untyped Expressions" on page 87 for more information.

## See Also

— "A Simple Example of Unpacking" on page 500
— "Packing and Unpacking Scalar Expressions" on page 501
— "Packing and Unpacking Strings" on page 501
— "Packing and Unpacking Structs" on page 502
— "Packing and Unpacking Lists" on page 503

### 17.1.2 A Simple Example of Unpacking

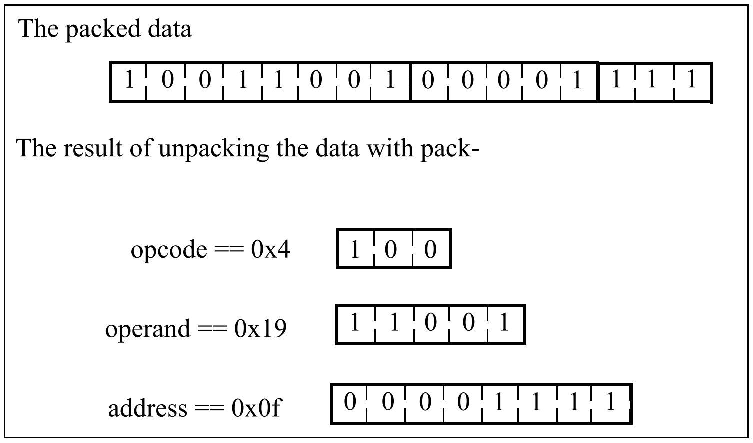
This example shows how packing fills the fields of a struct instance with data from a bit stream. An “instruction” struct is defined as:

```
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);
};
```

The extension to **post_generate()** shown below unpacks a list of bits, “packed_data”, into a variable “inst” of type “instruction” using the **packing.high** option. The results are shown in Figure 17-3.

```
extend sys {
    post_generate() is also {
        var inst : instruction;
        var packed_data: list of bit;
        packed_data = {1;1;1;1;0;0;0;0;1;0;0;1;1;0;0;1};

        unpack(packing.high, packed_data, inst);
    };
};
```

**Figure 17-3—Simple Unpacking Example Showing packing.high**



In this case, the expression that provides the value, “packed_data”, is a list of bits. When a value expression is not a list of bit, the ***e*** program uses implicit packing to store the data in the target expression. See “Implicit Packing and Unpacking” on page 515 for more details.

#### See Also

- “A Simple Example of Packing” on page 498
- “Packing and Unpacking Scalar Expressions” on page 501
- “Packing and Unpacking Strings” on page 501
- “Packing and Unpacking Structs” on page 502
- “Packing and Unpacking Lists” on page 503

### 17.1.3 Packing and Unpacking Scalar Expressions

Packing a scalar expression creates an ordered bit stream by concatenating the bits of the expression together. Unpacking a bit stream into a scalar expression fills the scalar expression, starting by default by putting the lowest bit of the bit stream into the lowest bit of the scalar expression.

Packing and unpacking of a scalar expression is performed using the expression's inherent size, except when the expression contains a bit slice operator. Missing bits are assumed to be zero, and extra bits are allowed and ignored. See "Overview of e Data Types" on page 75 for information on how the size of a scalar expression is determined. See also "Bit Slice Operator and Packing" on page 514.

#### Example

The example below packs two integers, "int_5" and "int_2" and then unpacks a new list of bit "lob" into the same two integers. In the unpack action, the first five bits of "lob" are assigned to "int_5" and the next two to "int_2". The remaining bits in "lob" are not used.

```
var int_5: int(bits:5) = 3;
var int_2: int(bits:2);
print pack(packing.low, int_5, int_2) using bin;
var lob:list of bit = {1;1;1;1;1;1;0;0;0;0;1;0;0;};
unpack(packing.low,lob,int_5,int_2);
print int_5, int_2;
```

#### Result

```
pack(packing.low, int_5, int_2) = (7 items, bin):
                                                 0 0 0  0 0 1 1    .0

int_5 = 31
int_2 = 1
```

NOTE—

If you unpack a list into one or more scalar expressions and there are not enough bits in the list to put a value into each scalar, a runtime error is issued.

#### See Also

- "A Simple Example of Packing" on page 498
- "A Simple Example of Unpacking" on page 500
- "Packing and Unpacking Strings" on page 501
- "Packing and Unpacking Structs" on page 502
- "Packing and Unpacking Lists" on page 503

### 17.1.4 Packing and Unpacking Strings

Packing a string creates an ordered bit stream by concatenating each ASCII byte of the string together from left to right ending with a byte with the value zero (the final NULL byte). Unpacking a string places the bytes of the string into the target expression, starting with the first ASCII byte in the string up to and including the first byte with the value zero.

To obtain different results, you can use the **as_a()** method, which converts directly between the **string** and **list of byte** types. See "as_a()" on page 104 for more information.

### Example

In this example, the packed string is implicitly unpacked into a list of byte.

The last byte is zero since it is the final NULL byte.

```
var my_string: string = "ABC";
print pack(packing.low, my_string);

var my_list_of_byte: list of byte = pack(packing.low, my_string);
print my_list_of_byte using hex;
```

### Result

```
pack(packing.low,my_string) = (32 items, hex):
                                                0 0 4 3  4 2 4 1    .0

my_list_of_byte = (4 items, hex):
                                                   00 43 42 41      .0
```

### See Also

— "A Simple Example of Packing" on page 498
— "A Simple Example of Unpacking" on page 500
— "Packing and Unpacking Scalar Expressions" on page 501
— "Packing and Unpacking Structs" on page 502
— "Packing and Unpacking Lists" on page 503

## 17.1.5 Packing and Unpacking Structs

Packing a struct creates an ordered bit stream from all the physical fields (marked with %) in the struct, starting by default, with the first physical field declared. Other fields (called virtual fields) are ignored by the packing process. If a physical field is of a compound type (struct or list) the packing process descends recursively into the struct or list.

Unpacking a bit stream into a struct fills the physical fields of the struct, starting by default with the first field declared and proceeding recursively through all the physical fields of the struct. Unpacking a bit stream into a field that is a list follows some additional rules described in "Packing and Unpacking Lists" on page 503.

Unpacking a struct that has not yet been allocated (with **new**) causes the ***e*** program to allocate the struct and run the struct's **init()** method. Unlike **new**, the struct's **run()** method is not called.

A struct is packed or unpacked using its predefined methods **do_pack()** and **do_unpack()**. It is possible to modify these predefined methods for a particular struct. See "do_pack()" on page 526 and "do_unpack()" on page 529 for more information.

### Example

This example packs the two physical fields in "my_struct" into the variable "ms". The resulting bit stream is 14 bits, which is exactly the combination of both the physical fields. The virtual field "int_15" does not participate in the pack process at all.

```
struct my_struct{
    %int_4:  int(bits:4);
```

```
        %int_10: int(bits:10);
        int_15:  int(bits:15);
        init() is also {
            int_4 = 3;
            int_10 = 15;
        };
    };

    struct pac {
        m() is {
            var ms: my_struct = new;
            print pack(packing.low, ms) using hex;
        };
    };
```

### Result

```
    pack(packing.low,ms) = (14 items, bin):
                                  0 0  0 0 0 0  1 1 1 1  0 0 1 1    .0
```

### See Also

- "A Simple Example of Packing" on page 498
- "A Simple Example of Unpacking" on page 500
- "Packing and Unpacking Scalar Expressions" on page 501
- "Packing and Unpacking Strings" on page 501
- "Packing and Unpacking Lists" on page 503

## 17.1.6 Packing and Unpacking Lists

Packing a list creates a bit stream by concatenating the list items together, starting by default with the item at index 0.

Unpacking a bit stream into a list fills the list item by item, starting by default with the item at index zero. The size of the list that is unpacked into is determined by whether the list is sized and whether it is empty:

- Unpacking into an empty list expands the list as needed to contain all the available bits.
- Unpacking into a non-empty list unpacks only until the existing list size is reached.
- Unpacking to a struct fills the sized lists only to their defined size, regardless of their actual size at the time.

NOTE— When a struct is allocated, the lists within it are empty. If the lists are sized, unpacking is performed until the defined size is reached.

See Chapter 2, "e Basics", for more information on sizing lists.

### Example 1

This first example shows the effect of packing a list of integers.

```
    var my_list: list of int(bits:4) = {3;5;7;9;11;13;15};
    print pack(packing.low,my_list);
```

### Result

```
    pack(packing.low,my_list) = (28 items, hex):
```

```
                                                   f d b  9 7 5 3    .0
```

### Example 2

The list “my_list” is empty, so it is expanded to contain all 32 bits of the integer.

```
var my_list: list of int(bits:4);
unpack(packing.low,5,my_list);
print my_list using hex;
```

### Result

```
my_list = (8 items, hex):
                                  0  0  0  0   0  0  0  5      .0
```

### Example 3

The list was not empty because it was initialized to have four items. Thus it is not expanded and the resulting list has four items.

```
var my_list: list of int(bits:4) = {0;0;0;0};
unpack(packing.low,5,my_list);
print my_list using hex;
```

### Result

```
my_list = (4 items, dec):
                                                 0  0  0  5      .0
```

### Example 4

The “my_list” field is cleared in order to demonstrate that although the procedural code has corrupted the list’s initial size, it is restored when unpack is performed.

```
struct my_struct{
    %my_list[4]: list of int(bits:4);
};

struct pac {
    m() is {
        var t:my_struct = new;
        t.my_list.clear();
        print t.my_list;
        unpack(packing.low,5,t);
        print t.my_list;
    };
};
```

### Result

```
t.my_list = (empty)
t.my_list = (4 items, hex):
                                                      0 0 0 5    .0
```

### Example 5

Unpacking into an unsized, uninitialized list causes a runtime error message because the list is expanded as needed to consume all the given bits. The field “int_1” remains bit-less.

```
struct my_struct{
    %my_list: list of int(bits:4);
    %int_1: bit;
};

struct pac {
    m() is {
        var t:my_struct = new;
        unpack(packing.low,5,t);
    };
};
```

### Example 6

This example shows the recommended way to get a variable number of list items. The specification order is important because the “len1” and “len2” values must be set before initializing “data1” and “data2”. Declaring “len1” and “len2” before “data1” and “data2” ensures that the list length is generated first. When unpacking into a list with a variable number of items, you will have to calculate the number of items in the list before unpacking. See “do_unpack()” on page 529 for an example of how to do this.

```
struct packet{
    %len1: int;
    %len2: int;
    %data1[len1]: list of byte;
    %data2[len1 + len2]: list of byte;
};
```

### See Also

- “A Simple Example of Packing” on page 498
- “A Simple Example of Unpacking” on page 500
- “Packing and Unpacking Scalar Expressions” on page 501
- “Packing and Unpacking Strings” on page 501
- “Packing and Unpacking Structs” on page 502

## 17.2 Advanced Packing

The following sections describe how to use the ***e*** advanced packing features and provide you with the concepts you need to use them efficiently:

- “Using the Predefined pack_options Instances” on page 506
- “Customizing Pack Options” on page 510
- “Customizing Packing for a Particular Struct” on page 514
- “Bit Slice Operator and Packing” on page 514
- “Implicit Packing and Unpacking” on page 515
- “Untyped Expressions” on page 87

### See Also

- "Basic Packing" on page 497
- "Constructs for Packing and Unpacking" on page 516

## 17.2.1 Using the Predefined pack_options Instances

Packing and unpacking are controlled using a struct under **global** named **packing**. There are five predefined instances of the **pack_options** struct that you can use with or without modification to control the way packing or unpacking is performed. You are probably familiar with two of these **pack_options** instances, **packing.low** and **packing.high**.

When you call **pack()**, **unpack()**, **do_pack()** or **do_unpack()** you pass one of the predefined **pack_options** instances as the first parameter. In the example below, **packing.high** is passed as the **pack_options** instance:

```
data_packed_high = pack(packing.high, opcode, operand);
```

The following sections describe the **pack_options** instances:

- "packing.low" on page 506
- "packing.low_big_endian" on page 507
- "packing.high" on page 508
- "packing.high_big_endian" on page 508
- "packing.network" on page 509
- "packing.global_default" on page 510

### See Also

- "Customizing Pack Options" on page 510
- "Customizing Packing for a Particular Struct" on page 514
- "Bit Slice Operator and Packing" on page 514
- "Implicit Packing and Unpacking" on page 515
- "Untyped Expressions" on page 87

## 17.2.2 packing.low

This **pack_options** instance traverses the source fields or variables in the order they appear in code, placing the least significant bit of the first field or list item at index [0] in the resulting list of bit. The most significant bit of the last field or list item is placed at the highest index in the resulting list of bit.

```
<'
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);

    !data_packed_low : list of bit;

    keep opcode ==  0b100;
    keep operand == 0b11001;
    keep address == 0b00001111;

    post_generate() is also {
        data_packed_low = pack(packing.low, me);
        print me using bin;
```

```
            print data_packed_low using bin;
        };
    };
    '>
```

### Result

```
    me = instruction-@0: instruction
                                                            @packing20
    0       %opcode:                        0b100
    1       %operand:                       0b11001
    2       %address:                       0b00001111
    3       !data_packed_low:               (16 items)
      data_packed_low = (16 items, bin):
                                 0 0 0 0  1 1 1 1  1 1 0 0  1 1 0 0     .0
```

### See Also

— "Using the Predefined pack_options Instances" on page 506

## 17.2.3 packing.low_big_endian

This **pack_options** instance, like **packing.low**, traverses the source fields or variables in the order they appear in code. In addition, for every scalar field or variable, it

— Swaps every byte in each pair of bytes
— Swaps every two bytes in each 32-bit word

NOTE— If the scalar's width is not a multiple of 16, no swapping is performed

The example below shows the difference between **packing.low** and **packing.low_big_endian**.

```
    struct pac {
    %opcode: uint (bits:4);
    %operand1: uint (bytes:2);
    %operand2: uint (bytes:2);

    keep opcode == 0xf;
    keep operand1 == 0xcc55;
    keep operand2 == 0xff00;

        m() is {
            var i_stream: list of bit;
            i_stream = pack(packing.low, opcode,
                operand1, operand2);
            print i_stream using bin;
            i_stream = pack(packing.low_big_endian, opcode,
                operand1, operand2);
            print i_stream using bin;
        };
    };
```

### Result

```
    i_stream = (36 items, bin):
            0 0 0 0  1 1 0 0  1 1 0 0  0 1 0 1  0 1 0 1  1 1 1 1     .0
                                       1 1 1 1  1 1 1 1  0 0 0 0     .24
```

```
i_stream = (36 items, bin):
      1 1 1 1  0 1 0 1  0 1 0 1  1 1 0 0  1 1 0 0  1 1 1 1    .0
                                 0 0 0 0  0 0 0 0  1 1 1 1    .24
```

### See Also

— "Using the Predefined pack_options Instances" on page 506

## 17.2.4 packing.high

This **pack_options** instance traverses the source fields or variables in the reverse order from the order in which they appear in code, placing the least significant bit of the last field or list item at index [0] in the resulting list of bit. The most significant bit of the first field or list item is placed at the highest index in the resulting list of bit.

```
<'
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);

    !data_packed_high : list of bit;

    keep opcode ==  0b100;
    keep operand == 0b11001;
    keep address == 0b00001111;

    post_generate() is also {
        data_packed_high = pack(packing.high, opcode, operand);
        print me using bin;
        print data_packed_high using bin;
    };
};
'>
```

### Result

```
me = instruction-@0: instruction
                                                       @packing18
0       %opcode:                    0b100
1       %operand:                   0b11001
2       %address:                   0b00001111
3       !data_packed_high:          (8 items)
  data_packed_high = (8 items, bin):
                                        1 0 0 1  1 0 0 1    .0
```

### See Also

— "Using the Predefined pack_options Instances" on page 506

## 17.2.5 packing.high_big_endian

This **pack_options** instance, like **packing.high**, traverses the source fields or variables in the reverse order from the order in which they appear in code. In addition, for every scalar field or variable, it

— Swaps every byte in each pair of bytes
— Swaps every two bytes in each 32-bit word

NOTE— If the scalar's width is not a multiple of 16, no swapping is performed.

The example below shows the difference between **packing.high** and **packing.high_big_endian**.

```
struct pac {
    %opcode: uint (bits:4);
    %operand1: uint (bytes:2);
    %operand2: uint (bytes:2);

    keep opcode == 0xf;
    keep operand1 == 0xcc55;
    keep operand2 == 0xff00;

    m() is {
        var i_stream: list of bit;
        i_stream = pack(packing.high, opcode,
            operand1,operand2);
        print i_stream using bin;
        i_stream = pack(packing.high_big_endian, opcode,
            operand1,operand2);
        print i_stream using bin;
    };
};
```

### Result

```
i_stream = (36 items, bin):
        0 1 0 1  0 1 0 1  1 1 1 1  1 1 1 1  0 0 0 0  0 0 0 0    .0
                                   1 1 1 1  1 1 0 0  1 1 0 0    .24

  i_stream =  (36 items, bin):
        1 1 0 0  1 1 0 0  0 0 0 0  0 0 0 0  1 1 1 1  1 1 1 1    .0
                                   1 1 1 1  0 1 0 1  0 1 0 1    .24
```

### See Also

— "Using the Predefined pack_options Instances" on page 506

## 17.2.6 packing.network

This packing option is the same as **packing.high** if the total number of bits that will comprise the target is not a multiple of 8. When it is a multiple of 8, then the target bits of the entire bit stream are byte-order reversed.

```
<'
struct instruction {
    %opcode     : uint (bits : 3);
    %operand    : uint (bits : 5);
    %address    : uint (bits : 8);

    !data_packed_high : list of bit;
    !data_packed_network : list of bit;

    keep opcode ==  0b100;
    keep operand == 0b11001;
    keep address == 0b00001111;
```

```
        post_generate() is also {
            data_packed_high = pack(packing.high, opcode, operand,
              address);
            data_packed_network = pack(packing.network, opcode,
              operand, address);

            print me using bin;
            print data_packed_high using bin;
            print data_packed_network using bin;
        };
    };
    '>
```

### Results

```
    me = instruction-@0: instruction
            ----------------------------------------------  @packing13
    0       %opcode:                        0b100
    1       %operand:                       0b11001
    2       %address:                       0b00001111
    3       !data_packed_high:              (16 items)
    4       !data_packed_network:           (16 items)
      data_packed_high =  (16 items, bin):
                                1 0 0 1  1 0 0 1  0 0 0 0  1 1 1 1    .0

      data_packed_network =  (16 items, bin):
                                0 0 0 0  1 1 1 1  1 0 0 1  1 0 0 1    .0
```

## 17.2.7 packing.global_default

This **pack_options** instance is used when the first parameter of **pack()**, **unpack()**, **do_pack()**, or **do_unpack()** is NULL. It has the same flags as **packing.low**.

### See Also

— "Using the Predefined pack_options Instances" on page 506

## 17.2.8 Customizing Pack Options

Each of the predefined instances described in "Using the Predefined pack_options Instances" on page 506 is an instance of the **pack_options** struct. The **pack_options** declaration is as follows:

```
    struct pack_options {
        reverse_fields: bool;
        reverse_list_items: bool;
        final_reorder: list of int;
        scalar_reorder: list of int;
    };
```

To customize packing options, you can create an instance of the **pack_options** struct, modify one or more of its fields, and pass the struct instance as the first parameter to **pack()**, **unpack()**, **do_pack()**, or **do_unpack()**.

The following sections describe each field of the **pack_options** struct:

— "reverse_fields" on page 511

— “reverse_list_items” on page 512
— “scalar_reorder” on page 512
— “final_reorder” on page 513

### See Also

— “Using the Predefined pack_options Instances” on page 506
— “Customizing Packing for a Particular Struct” on page 514
— “Bit Slice Operator and Packing” on page 514
— “Implicit Packing and Unpacking” on page 515
— “Untyped Expressions” on page 87

## 17.2.9 reverse_fields

If this flag is set to be FALSE, the fields in a struct are packed in the order they appear in the struct declaration; if TRUE, they are packed in reverse order. The default is FALSE.

### Example

When **reverse_fields** is FALSE, “first” is packed and then “second”. When the flag is set to TRUE, “second” is packed before “first”. Thus, the first result is 0b01; the second result is 0b10.

```
struct my_struct{
    %first  :int(bits:1); -- value 1
    %second :int(bits:1); -- value 0;
    init() is also {
        first = 1;
    };
};

extend sys{
    my_struct;
    foo() is {
        var p1:pack_options = new;
        var p2:pack_options = new;
        p2.reverse_fields = TRUE;
        my_struct = new;
        print pack(p1,my_struct);
        print pack(p2,my_struct);
    };
};
```

### Result

```
pack(p1,my_struct) = (2 items, hex):
                                                  1   .0

pack(p2,my_struct) = (2 items, hex):
                                                  2   .0
```

### See Also

— “Customizing Pack Options” on page 510

### 17.2.10 reverse_list_items

If this flag is set to be FALSE, the items in a list are packed in ascending order; if TRUE, they are packed in descending order. The default is FALSE.

#### Example

The second print statement shows that “my_list” is packed in reverse order.

```
var my_list:list of int(bits:1) = {1;0;0};
var p1:pack_options = new;
var p2:pack_options = new;
p2.reverse_list_items = TRUE;
print pack(p1,my_list) using bin;
print pack(p2,my_list) using bin;
```

#### Result

```
pack(p1,my_list) = (3 items, bin):
                                        0 0 1      .0

pack(p2,my_list) = (3 items, bin):
                                        1 0 0      .0
```

#### See Also

— “Customizing Pack Options” on page 510

### 17.2.11 scalar_reorder

You can perform one or more **swap()** operations on each scalar before packing using the **scalar_reorder** field.

The list in the **scalar_reorder** field must include an even number of items. Each pair of items in the list is the parameter list of a **swap()** operation (see “swap()” on page 524). By entering multiple pairs of parameters, you can perform multiple swaps, using each pair of parameters as a pair of swap parameters.

Unlike **swap()**, if the large parameter is not a factor of the number of bits in the list, **scalar_reorder** ignores it, while **swap()** gives an error.

#### Example 1

The bits in “mid” and “sma” are swapped using 4 and 2 as parameters. See “swap()” on page 524. Thus, within every four bits the first two bits and the last two are swapped.

```
var p1:pack_options = new;
var mid:int(bits:8) = 0xcc;
var sma:int(bits:4) = 0x6;
print pack(p1,mid) using bin;
print pack(p1,sma) using bin;
p1.scalar_reorder = {2;4};
print pack(p1,mid) using bin;
print pack(p1,sma) using bin;
```

### Result

```
pack(p1,mid) = (8 items, bin):
                               1 1 0 0  1 1 0 0    .0

pack(p1,sma) = (4 items, bin):
                                        0 1 1 0    .0

pack(p1,mid) = (8 items, bin):
                               0 0 1 1  0 0 1 1    .0

pack(p1,sma) = (4 items, bin):
                                        1 0 0 1    .0
```

### Example 2

The bits in “midb” are swapped, first using 8 and 4 as parameters, and then using 16 and 8 as parameters. See “swap()” on page 524. Thus, within every eight bits the first four bits and the last four are swapped, and then within the entire 16-bit number, the first eight bits and the last eight bits are swapped.

```
var p1:pack_options = new;
var midb:int(bits:16) = 0xc5fd;
print pack(p1,midb) using bin;
p1.scalar_reorder = {4;8;8;16};
print pack(p1,midb) using bin;
```

### Result

```
pack(p1,midb) =  (16 items, bin):
                               1 1 0 0  0 1 0 1  1 1 1 1  1 1 0 1    .0

pack(p1,midb) =  (16 items, bin):
                               1 1 0 1  1 1 1 1  0 1 0 1  1 1 0 0    .0
```

### See Also

— “Customizing Pack Options” on page 510

## 17.2.12 final_reorder

After packing each element in the packing expression you can perform final swapping on the resulting bit stream, using the **final_reorder** field.

The list in the **final_reorder** field must include an even number of items. Each pair of items in the list is the parameter list of a **swap()** operation (see “swap()” on page 524). By entering multiple pairs of parameters, you can perform multiple swaps, using each pair of parameters as a pair of swap parameters.

Unlike **swap()**, if the large parameter is not a factor of the number of bits in the list, **final_reorder** ignores it, while **swap()** gives an error.

### Example 1

After performing the second pack, a swap is performed using 4 and 8 as parameters, thus reversing the order of nibbles in every byte. See “swap()” on page 524.

```
var p1:pack_options = new;
var int__4: int(bits:4) = 0x7;
```

```
    var int__12: int(bits:12) = 0x070;
    print pack(p1,int__4,int__12) using bin;
    p1.final_reorder = {4;8};
    print pack(p1,int__4,int__12) using bin;
```

### Result

```
    pack(p1,int__4,int__12) = (16 items, bin):
                                0 0 0 0  0 1 1 1  0 0 0 0  0 1 1 1    .0

    pack(p1,int__4,int__12) = (16 items, bin):
                                0 1 1 1  0 0 0 0  0 1 1 1  0 0 0 0    .0
```

### Example 2

The bits in “midb” are swapped, first using 8 and 4 as parameters, and then using 16 and 8 as parameters. See “swap()” on page 524. Thus, within every eight bits the first four bits and the last four are swapped, and then within the entire 16-bit number, the first eight bits and the last eight bits are swapped.

```
    var p1:pack_options = new;
    var midb:int(bits:16) = 0xc5fd;
    print pack(p1,midb) using bin;
    p1.scalar_reorder = {4;8;8;16};
    print pack(p1,midb) using bin;
```

### Result

```
    pack(p1,midb) =  (16 items, bin):
                                   1 1 0 0  0 1 0 1  1 1 1 1  1 1 0 1    .0

    pack(p1,midb) =  (16 items, bin):
                                   1 1 0 1  1 1 1 1  0 1 0 1  1 1 0 0    .0
```

### See Also

— “Customizing Pack Options” on page 510

## 17.2.13 Customizing Packing for a Particular Struct

You can customize packing for a particular struct by modifying the **do_pack()** or **do_unpack()** methods of the struct. These methods are called automatically whenever data is packed from or unpacked into the struct. See “do_pack()” on page 526 and “do_unpack()” on page 529 for more information.

### See Also

— “Using the Predefined pack_options Instances” on page 506
— “Customizing Pack Options” on page 510
— “Bit Slice Operator and Packing” on page 514
— “Implicit Packing and Unpacking” on page 515
— “Untyped Expressions” on page 87

## 17.2.14 Bit Slice Operator and Packing

You can use the bit slice operator **[ : ]** to select a subrange of an expression to be packed or unpacked. The bit slice operator does not change the type of the pack or unpack expression.

### Example 1

In the following example, the result of the first print statement is 20 bits long. However, the pack expression, which extracts only 2 bits of "int_20", is only 2 bits long.

```
var int_20: int(bits:20) = 7;
print int_20[1:0] using bin;
print pack(packing.low,int_20[1:0]) using bin;
```

### Result

```
int_20[1:0] = 0b00000000000000000011
pack(packing.low,int_20[1:0]) = (2 items, bin):
                                                       1 1   .0
```

### Example 2

"int_5" did not consume five bits as its type suggests. Because of the bit slice operator, it consumed only two bits. Thus "int_1" gets the third bit from "lob" and remains 0.

```
var int_5: int(bits:5);
var int_1: bit;
var lob: list of bit = {1;1;0;1;1;1;1};
unpack(packing.low, lob, int_5[1:0], int_1);
print int_5, int_1 using bin;
```

### Result

```
int_5 = 0b00011
int_1 = 0b0
```

### See Also

— "Using the Predefined pack_options Instances" on page 506
— "Customizing Pack Options" on page 510
— "Customizing Packing for a Particular Struct" on page 514
— "Implicit Packing and Unpacking" on page 515
— "Untyped Expressions" on page 87

## 17.2.15 Implicit Packing and Unpacking

Implicit packing and unpacking is always performed using the parameters of **packing.low** and takes place in the following cases:

— When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked before it is assigned.

```
var my_list: list of int = {1;2;3};

var int_10: int(bits:10);

my_list = 'top.foo';

int_10 = pack(NULL, 5);
```

Untyped expressions include HDL signals, pack expressions and bit concatenations. See "Untyped Expressions" on page 87 for more information.

NOTE— Implicit packing and unpacking is not supported for strings, structs, or lists of non-scalar types. As a result, the following causes a load-time error if "i" is a string, a struct, or a list of a non-scalar type:

```
i = pack(packing.low, 5);    // Load-time error
```

— When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned:

```
'top.foo' = {1;2;3};
```

— When the value expression of an unpack action is other than a list of bit, it is implicitly packed before it is unpacked:

```
unpack(packing.low,5,my_list);
```

### See Also

— "Using the Predefined pack_options Instances" on page 506
— "Customizing Pack Options" on page 510
— "Customizing Packing for a Particular Struct" on page 514
— "Bit Slice Operator and Packing" on page 514
— "Untyped Expressions" on page 87

## 17.3 Constructs for Packing and Unpacking

The following sections describe the constructs used in packing and unpacking:

— "pack()" on page 516
— "unpack()" on page 521
— "%{... , ...}" on page 62
— "swap()" on page 524
— "do_pack()" on page 526
— "do_unpack()" on page 529

### See Also

— "Basic Packing" on page 497
— "Advanced Packing" on page 505

### 17.3.1 pack()

#### Purpose

Perform concatenation and type conversion

#### Category

Pseudo-method

#### Syntax

**pack(*option***:pack option**, *item***: exp, ...**)**: list of bit

Syntax example:

```
i_stream = pack(packing.high, opcode, operand1, operand2);
```

## Parameters

| | |
|---|---|
| *option* | For basic packing, this parameter is one of the following. See "Using the Pre-defined pack_options Instances" on page 506 for information on other pack options. |
| packing.high | Places the least significant bit of the last physical field declared or the highest list item at index [0] in the resulting list of bit. The most significant bit of the first physical field or lowest list item is placed at the highest index in the resulting list of bit. |
| packing.low | Places the least significant bit of the first physical field declared or lowest list item at index [0] in the resulting list of bit. The most significant bit of the last physical field or highest list item is placed at the highest index in the resulting list of bit. |
| NULL | If NULL is specified, the global default is used. This global default is set initially to **packing.low**. |
| item | A legal ***e*** expression that is a path to a scalar or a compound data item, such as a struct, field, list, or variable. |

## Description

Performs concatenation of items, including items in a list or fields in a struct, in the order specified by the pack options parameter and returns a list of bits. This method also performs type conversion between any of the following:

- scalars
- strings
- lists and list subtypes (same type but different width)

Packing is commonly used to prepare high-level ***e*** data into a form that can be applied to a DUT. For other uses, see "Packing and Unpacking" on page 497.

Packing operates on scalar or compound (struct, list) data items. For more information and examples of how packing operates on different data types, see "Basic Packing" on page 497.

Pack expressions are untyped expressions. In many cases, the ***e*** program can deduce the required type from the context of the pack expression. See "Untyped Expressions" on page 87 for more information.

NOTE— You cannot pack an unbounded integer.

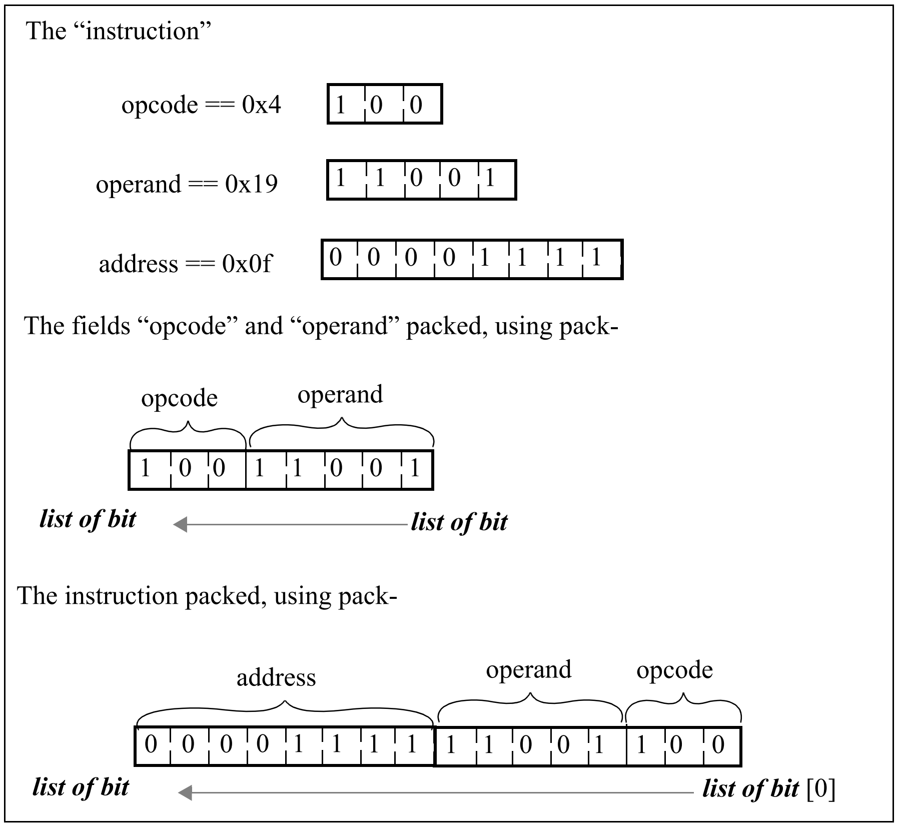
## Example 1

The extension to **post_generate()** shown below packs the "opcode" and the "operand" fields of the "instruction" struct from the low bit of the last field defined ("operand") to the high bit of the first field defined ("opcode") into the "data_packed_high" field. It also packs all the physical fields into "data_packed_low" using the **packing.low** option. The results are shown in Figure 17-4 on page 519.

```
<'
struct instruction {
    %opcode      : uint (bits : 3);
```

```
        %operand    : uint (bits : 5);
        %address    : uint (bits : 8);

        !data_packed_high : list of bit;
        !data_packed_low  : list of bit;

        keep opcode ==  0b100;
        keep operand == 0b11001;
        keep address == 0b00001111;

        post_generate() is also {
            data_packed_high = pack(packing.high,
              opcode,operand);
            data_packed_low = pack(packing.low, me);
            print me using bin;
            print data_packed_low using bin;
            print data_packed_high using bin;
        };
    };
    '>
```

**Figure 17-4—Packed Instruction Data**



## Example 2

In this example, **post_generate()** is extended to pack the packet data. The “header” field of the “packet” struct is a struct itself, so this is a recursive pack. The results are shown in Figure 17-5 on page 520.

```
<'
struct packet {
    %header : header;
    %payload : list of byte;

    !data_packed_low : list of byte;

    keep payload.size() == 6;
    keep for each in payload {
        it == index;
        };

    post_generate() is also {
        data_packed_low = pack(packing.low, me);
        out("payload:         ", payload);
        out("data packed low: ", data_packed_low);
    };
};

struct header {
    %dest      : int (bits : 8);
```

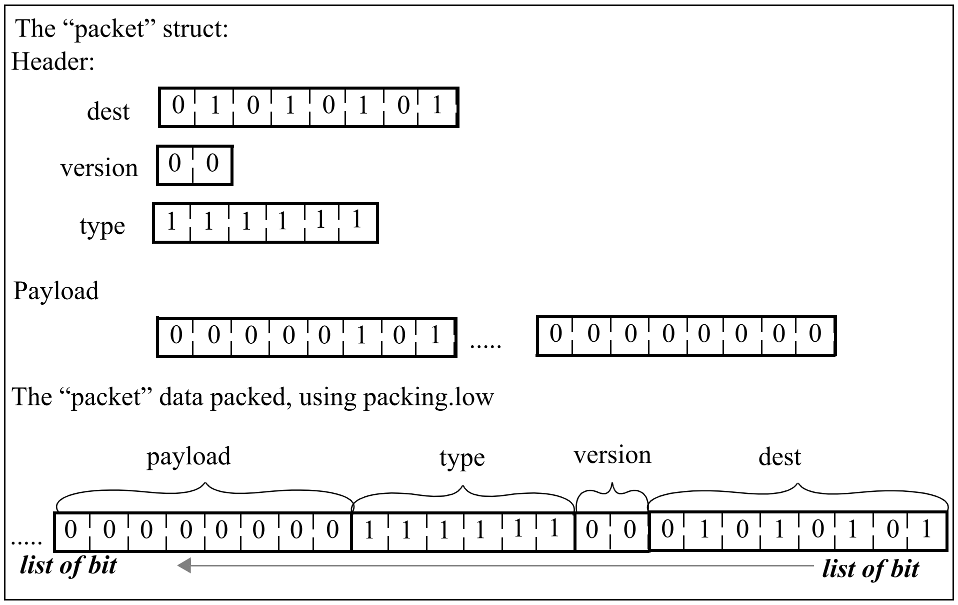
```
        %version    : int (bits : 2);
        %type       : uint (bits : 6);


        keep dest == 0x55;
        keep version == 0x0;
        keep type == 0x3f;
        post_generate() is also {
            print me;
        };
    };
    '>
```

## Result

Note that the “**out()**” action displays the bytes from least significant to most significant (from left to right), whereas the **print** action displays the bytes from most significant to least significant (from left to right).

```
        me = header-@0: header
           --------------------------------------------- @pack23
    0        %dest:                    0x55
    1        %version:                 0x0
    2        %type:                    0x3f
    payload          : 0x00 0x01 0x02 0x03 0x04 0x05
    packet packed low: 0x55 0xfc 0x00 0x01 0x02 0x03 0x04 0x05
```

**Figure 17-5—Packed Packet Data**



## See Also

- — “unpack()” on page 521
- — “%{... , ...}” on page 62
- — “swap()” on page 524
- — “do_pack()” on page 526

— "do_unpack()" on page 529

### 17.3.2 unpack()

#### Purpose

Unpack a bit stream into one or more expressions

#### Category

Pseudo-method

#### Syntax

**unpack(*option*:** pack option**, *value*:** exp**, *target1*:** exp [**, *target2*:** exp, ...]**)**

Syntax example:

```
unpack(packing.high, lob, s1, s2);
```

#### Parameters

| | |
|---|---|
| option | For basic packing, this parameter is one of the following. See "Using the Predefined pack_options Instances" on page 506 for information on other pack options. |
| packing.high | Places the most significant bit of the list of bit at the most significant bit of the first field or lowest list item. The least significant bit of the list of bit is placed into the least significant bit of the last field or highest list item. |
| packing.low | Places the least significant bit of the list of bit into the least significant bit of the first field or lowest list item. The most significant bit of the list of bit is placed at the most significant bit of the last field or highest list item. |
| NULL | If NULL is specified, the global default is used. This global default is set initially to **packing.low**. |
| value | A scalar expression or list of scalars that provides a value that is to be unpacked. |
| target1, target2 | One or more expressions separated by commas. Each expression is a path to a scalar or a compound data item, such as a struct, field, list, or variable. |

#### Description

Converts a raw bit stream into high level data by storing the bits of the value expression into the target expressions.

If the value expression is not a list of bit, it is first converted into a list of bit by calling **pack()** using **packing.low**. (See "Implicit Packing and Unpacking" on page 515 for more information.) Then the list of bit is unpacked into the target expressions.

The value expression is allowed to have more bits than are consumed by the target expressions. In that case, if **packing.low** is used, the extra high-order bits are ignored; if **packing.high** is used, the extra low-order bits are ignored.

Unpacking is commonly used to convert raw bit stream output from the DUT into high-level *e* data.

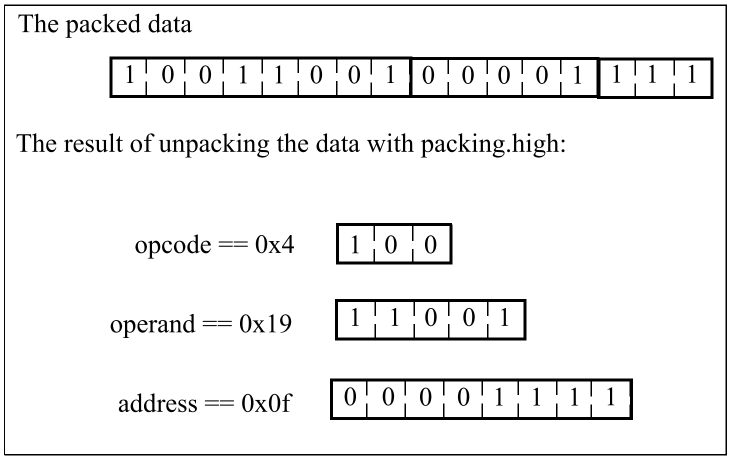
Unpacking operates on scalar or compound (struct, list) data items. For more information and examples of how packing operates on different data types, see "Basic Packing" on page 497.

### Example 1

The extension to **post_generate()** shown below unpacks a list of bits into a variable "inst". The results are shown in Figure 17-6 on page 522.

```
extend sys {
    post_generate() is also {
        var inst : instruction;
        var packed_data: list of bit;
        packed_data = {1;1;1;1;0;0;0;0;1;0;0;1;1;0;0;1};

        unpack(packing.high, packed_data, inst);

        print packed_data using bin;
        out("the result of unpacking it: ");
        print inst using bin;
    };
};

struct instruction {
    %opcode    : uint (bits : 3);
    %operand   : uint (bits : 5);
    %address   : uint (bits : 8);
};
```

**Figure 17-6—Unpacked Instruction Data**

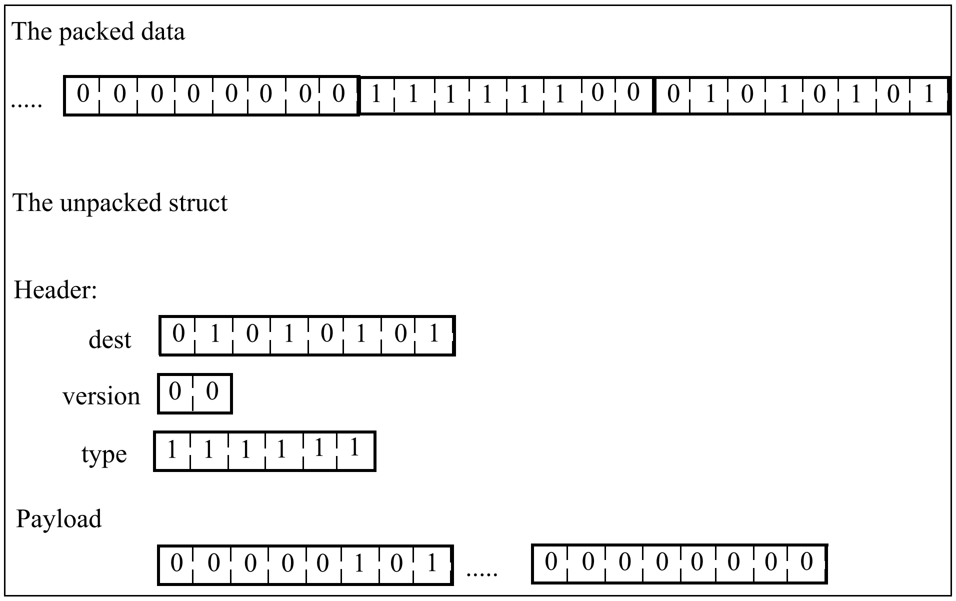


### Example 2

The extension to **post_generate()** shown below unpacks a list of bytes into a variable "pkt" using **packing.low**. This is a recursive unpack because the "header" field of "packet" is a struct itself. The results are shown in Figure 17-7 on page 523.

```
extend sys {
    post_generate() is also {
        var pkt : packet;
        var packed_data : list of byte;
        packed_data =
            {0x55;0xfc;0x00;0x01;0x02;0x03;0x04;0x05};

        unpack(packing.low, packed_data, pkt);

        print packed_data;
        out("the unpacked struct:");
        print pkt.header, pkt.payload;
    };
};

struct packet {
    %header : header;
    %payload : list of byte;
};

struct header {
    %dest      : int (bits : 8);
    %version   : int (bits : 2);
    %type      : int (bits : 6);
};
```

**Figure 17-7—Unpacked Packet Data**



## Example 3

This example uses **unpack()** sequentially to set up virtual fields that are required for the full unpack.

```
    struct packet {
        %header: header;
        len : uint;
        %data[len] : list of byte;
```

```
};
struct header {
    %code : uint;
};
extend sys {
    m() is {
        var DUT_bytes: list of byte = {0x11;0xff;0x22;
        0xee;0x33;0xdd;0x44;0xcc;0x55;0xbb;0x66};
        var p : packet = new;
        unpack(packing.low, DUT_bytes, p.header);
        if p.header.code > 1500 {
            p.len = 10;
        } else {
            p.len = 20;
        };
        unpack(packing.low, DUT_bytes,p);
        print p;
        print p.data;
    };
};
```

### See Also

- "pack()" on page 516
- "%{... , ...}" on page 62
- "swap()" on page 524
- "do_pack()" on page 526
- "do_unpack()" on page 529

## 17.3.3 swap()

### Purpose

Swap small bit chunks within larger chunks

### Category

Pseudo-method

### Syntax

***list-of-bit*.swap(*small*:** int**, *large*:** int**)**: list of bit

Syntax example:

```
s2 = s1.swap(2, 4);
```

### Parameters

| | |
|---|---|
| small | An integer that is a factor of ***large***. |
| large | An integer that is either UNDEF or a factor of the number of bits in the entire list. If UNDEF, the method reverses the order of small chunks within the entire list. Thus, "lob.swap(1, UNDEF)" is the same as "lob.reverse()". |

### Description

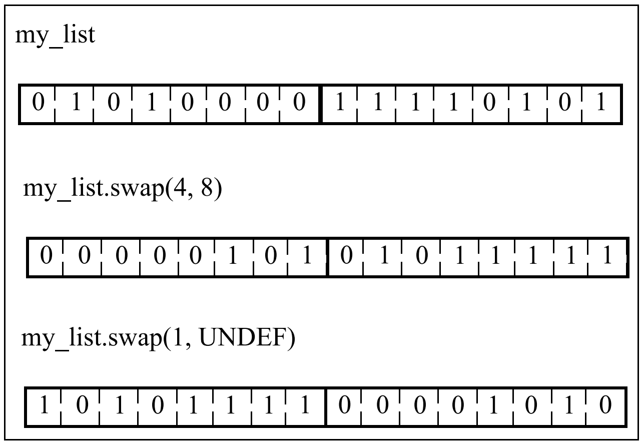
This predefined list method accepts a list of bits, changes the order of the bits, and then returns the reordered list of bits. This method is often used in conjunction with **pack()** or **unpack()** to reorder the bits in a bit stream going to or coming from the DUT.

### Notes

— If ***large*** is not a factor of the number of bits in the entire list, an error message results.
— If ***small*** is not a factor of ***large***, you will see an error message. The only exception is if ***large*** is UNDEF and ***small*** is not a factor, no swap is performed and no error is issued.

### Example 1

This example shows two swaps. The first swap reverses the order of nibbles in every byte. The second swap reverses the whole list.



### Example 2

This example shows **swap()** used with **unpack()** to reorder the bits before unpacking them.

```
extend sys {
    post_generate() is also {
        var num1 : uint (bits : 32);
        var num2 : uint (bits : 32);
        num1 = 0x12345678;

        unpack(NULL, pack(NULL, num1).swap(16, -1), num2);
        print num2;
        unpack(NULL, pack(NULL, num1).swap(8, -1), num2);
        print num2;
    };
};
```

### Results

```
num2 = 0x56781234
num2 = 0x78563412
```

### See Also

— "pack()" on page 516
— "unpack()" on page 521
— "%{... , ...}" on page 62
— "do_pack()" on page 526
— "do_unpack()" on page 529

## 17.3.4 do_pack()

### Purpose

Pack the physical fields of the struct

### Category

Predefined method of any struct

### Syntax

**do_pack(*options*:**pack options, ***l***: *list of bit**)**

Syntax example:

```
do_pack(options:pack_options, l: *list of bit) is only {
    var L : list of bit = pack(packing.low, operand2,
        operand1,operand3);
    l.add(L);
};
```

### Parameters

| | |
|---|---|
| options | This parameter is an instance of the pack options struct. See "Using the Pre-defined pack_options Instances" on page 506 for information on this struct. |
| l | An empty list of bits that is extended as necessary to hold the data from the struct fields. |

### Description

The **do_pack()** method of a struct is called automatically whenever the struct is packed. This method appends data from the physical fields (the fields marked with %) of the struct into a list of bits according to flags determined by the pack options parameter. The virtual fields of the struct are skipped. The method issues a runtime error message if this struct has no physical fields.

For example, the following assignment to "lob"

```
lob = pack(packing.high, i_struct, p_struct);
```

makes the following calls to the **do_pack** method of each struct, where ***tmp*** is an empty list of bits:

```
i_struct.do_pack(packing.high, *tmp)
p_struct.do_pack(packing.high, *tmp)
```

You can extend the **do_pack()** method for a struct in order to create a unique packing scenario for that struct. You should handle variations in packing that apply to many structs by creating a custom **pack_options** instance. See "Customizing Pack Options" on page 510 for information on how to do this.

## Notes

— Do not call the **do_pack()** method of any struct directly, for example "my_struct.do_pack()". Instead use **pack(),** for example "pack(packing.high, my_struct)".

— Do not call **pack(me)** in the **do_pack()** method. This causes infinite recursion. Call **packing.pack_struct(me)** instead. You can call **pack()** within the **do_pack()** method to pack objects other than **me**.

— Do not forget to append the results of any pack operation within **do_pack()** to the empty list of bits referenced in the **do_pack()** parameter list.

— If you modify the **do_pack()** method and then later add physical fields in an extension to the struct, you may have to make adjustments in the modifications to **do_pack()**.

## Example 1

This example shows how to override the **do_pack()** method for a struct called "cell". The extension to **do_pack()** overrides any packing option passed in and always uses **packing.low**. It packs "operand2" first, then "operand1" and "operand3".

```
<'
struct cell {
    %operand1: uint(bytes:2);
    %operand2: uint(bytes:2);
    %operand3: uint(bytes:2);
};

extend cell {
    do_pack(options:pack_options, l: *list of bit) is only {
        var L : list of bit = pack(packing.low, operand2,
            operand1,operand3);
        l.add(L);
    };
};
```

## Result

```
sys.pi = cell-@0: cell
    ---------------------------------------------- @pack33
0    %operand1:                      0b0010001000111001
1    %operand2:                      0b0001101001110101
2    %operand3:                      0b0001001010110010

var L : list of bit = pack(packing.high, sys.pi)

    L = (48 items, bin):
    0 0 1 1  1 0 0 1  0 0 0 1  1 0 1 0  0 1 1 1  0 1 0 1     .0
    0 0 0 1  0 0 1 0  1 0 1 1  0 0 1 0  0 0 1 0  0 0 1 0     .24
```

### Example 2

In the following example, the **do_pack()** method for “cell” is overwritten to use the **low_big_endian** packing option by default.

```
struct cell {
    %operand1: uint(bytes: 2);
    %operand2: uint(bytes: 2);
    %operand3: uint(bytes: 2);
};

extend cell {
    do_pack(options: pack_options, l: *list of bit) is only {
        if (options == NULL) then {
            packing.pack_struct(me,
              packing.low_big_endian,l);
        } else {
            packing.pack_struct(me, options, l);
        };
    };
};

extend sys {
    pi: cell;
};
```

### Result

```
    sys.pi = cell-@0: cell
    ---------------------------------------------- @pack34
0    %operand1:                      0b0010001000111001
1    %operand2:                      0b0001101001110101
2    %operand3:                      0b0001001010110010

var M : list of bit = pack(NULL, sys.pi)

    M = (48 items, bin):
    0 0 0 1  1 0 1 0  0 0 1 1  1 0 0 1  0 0 1 0  0 0 1 0    .0
    1 0 1 1  0 0 1 0  0 0 0 1  0 0 1 0  0 1 1 1  0 1 0 1    .24
```

### Example 3

This example swaps every pair of bits within each 4-bit chunk after packing with the packing options specified in the **pack()** call.

```
struct cell {
    %operand1: uint(bytes: 2);
    %operand2: uint(bytes: 2);
    %operand3: uint(bytes: 2);
};

extend cell {
    do_pack(options:pack_options, l: *list of bit) is only {
        var L1 : list of bit;
        packing.pack_struct(me, options, L1);
        var L2 : list of bit = L1.swap(2,4);
        l.add(L2);
    };
    };
```

### Result

```
        sys.pi = cell-@0: cell
        ---------------------------------------------- @pack35
    0    %operand1:                    0b0010001000111001
    1    %operand2:                    0b0001101001110101
    2    %operand3:                    0b0001001010110010

    var M : list of bit = pack(NULL, sys.pi)

        M = (48 items, bin):
        1 1 0 1  0 1 0 1  1 0 0 0  1 0 0 0  1 1 0 0  0 1 1 0    .0
        0 1 0 0  1 0 0 0  1 1 1 0  1 0 0 0  0 1 0 0  1 0 1 0    .24
```

### See Also

— "pack()" on page 516
— "unpack()" on page 521
— "%{... , ...}" on page 62
— "swap()" on page 524
— "do_unpack()" on page 529

## 17.3.5 do_unpack()

### Purpose

Unpack a packed list of bit into a struct

### Category

Predefined method of any struct

### Syntax

**do_unpack(*options*:**pack options, ***l***: list of bit, ***from***: int**):** int

Syntax example:

```
    do_unpack(options:pack_options, l: list of bit, from: int):int is only {
        var L : list of bit = l[from..];
        unpack(packing.low, L, op2, op1, op3);
        return from + 8 + 5 + 3;
    };
```

### Parameters

| | |
|---|---|
| *options* | This parameter is an instance of the pack options struct. See "Using the Predefined pack_options Instances" on page 506 for information on this struct. |
| *l* | A list of bits containing data to be stored in the struct fields. |
| *from* | An integer that specifies the index of the bit to start unpacking. |
| int (return value) | An integer that specifies the index of the last bit in the list of bits that was unpacked. |

### Description

The **do_unpack()** method is called automatically whenever data is unpacked into the current struct. This method unpacks bits from a list of bits into the physical fields of the struct. It starts at the bit with the specified index, unpacks in the order defined by the pack options, and fills the current struct's physical fields in the order they are defined.

For example, the following call to unpack()

```
unpack(packing.low, lob, c1, c2);
```

makes the following calls to the **do_unpack** method of each struct:

```
c1.do_unpack(packing.low, lob, index);
c2.do_unpack(packing.low, lob, index);
```

The method returns an integer, which is the index of the last bit unpacked into the list of bits.

The method issues a runtime error message if the struct has no physical fields. If at the end of packing there are leftover bits, it is not an error. If more bits are needed than currently exist in the list of bits, a runtime error is issued ("Ran out of bits while trying to unpack into ***struct_name***").

You can extend the **do_unpack()** method for a struct in order to create a unique unpacking scenario for that struct. You should handle variations in unpacking that apply to many structs by creating a custom **pack_options** instance. See "Customizing Pack Options" on page 510 for information on how to do this.

### Notes

— Do not call the **do_unpack()** method of any struct directly, for example "my_struct.do_unpack()". Instead use **unpack(),** for example "unpack(packing.high, lob, my_struct)".

— When you modify the **do_unpack()** method, you need to calculate and return the index of the last bit in the list of bits that was unpacked. In most cases, you simply add the bit width of each physical field in the struct to the starting ***index*** parameter. If you are unpacking into a struct that has conditional physical fields (physical fields defined under a **when**, **extend**, or **like** construct), this calculation is a bit tricky. See the Verification Advisor's patterns on packing for an example of how to do this.

### Example 1

This first example shows how to modify **do_unpack()** to change the order in which the fields of a struct are filled. In this case, the order is changed from "op1", "op2", "op3" to "op2", "op1", "op3". You can see also that **do_unpack()** returns the bit widths of the three physical fields, "op1", "op2", and "op3", to the starting index, "from".

```
struct cell {
    %op1: uint(bytes:1);
    %op2: uint(bits:5);
    %op3: uint(bits:3);
};

extend cell {
    do_unpack(options:pack_options, l: list of bit,
            from: int)  :int is only {
        var L : list of bit = l[from..];
        unpack(packing.low, L, op2, op1, op3);
        return from + 8 + 5 + 3;
    };
};
```

### Result

```
var P : list of bit
    {0;0;0;0;1;1;0;1;1;1;0;0;0;0;1;0;};
unpack(NULL, P, sys.pi)

    sys.pi = cell-@0: cell
----------------------------------------------- @pack36
0      %op1:                            0b00011101
1      %op2:                            0b10000
2      %op3:                            0b010
```

### Example 2

This example modifies the **do_unpack** method of the “frame” struct to first calculate the length of the “data” field. The calculation uses “from”, which indicates the last bit to be unpacked, to calculate the length of “data”.

```
extend sys {
    !packet1 : packet;
    !packet2 : packet;

    post_generate() is also {
        var raw_data : list of byte;
        for i from 0 to 39 {
            raw_data.add(i);
        };
        unpack(packing.low, raw_data, packet1);
        print packet1.header, packet1.frame.data,
            packet1.frame.crc;
        unpack(packing.high, raw_data, packet2);
        print packet2.header, packet2.frame.data,
            packet2.frame.crc;
    };
};

struct packet {
    %header    : int (bits : 16);
    %frame     : frame;
};

struct frame {
    %data[len] : list of byte;
```

```
        %crc       : int (bits : 32);
        len : int;

        do_unpack(options :pack_options, l :list of bit,
            from :int):int is first {

                if options.reverse_fields then {
                    len = (from - 32 + 1) / 8;
                } else {
                    len = (l.size() - from - 32) / 8;
                };
        };
    };
```

## Results

```
packet1.header = 256
packet1.frame.data = (34 items, dec):
         13  12  11  10    9   8   7   6    5   4   3   2        .0
         25  24  23  22   21  20  19  18   17  16  15  14        .12
                 35  34   33  32  31  30   29  28  27  26        .24

packet1.frame.crc = 656811300
packet2.header = 10022
packet2.frame.data = (34 items, dec):
         26  27  28  29   30  31  32  33   34  35  36  37        .0
         14  15  16  17   18  19  20  21   22  23  24  25        .12
                  4   5    6   7   8   9   10  11  12  13        .24

packet2.frame.crc = 50462976
```

## See Also

— “pack()” on page 516
— “unpack()” on page 521
— “%{... , ...}” on page 62
— “swap()” on page 524
— “do_pack()” on page 526

# 18 Control Flow Actions

The following sections describe the control flow actions:

- "Conditional Actions" on page 533
- "Iterative Actions" on page 537
- "File Iteration Actions" on page 545
- "Actions for Controlling the Program Flow" on page 547

## 18.1 Conditional Actions

The following conditional actions are used to specify code segments that will be executed only if a certain condition is met:

- "if then else" on page 533
- "case labeled-case-item" on page 534
- "case bool-case-item" on page 536

### 18.1.1 if then else

#### Purpose

Perform an action block based on whether a given boolean expression is TRUE

#### Category

Action

#### Syntax

**if** ***bool-exp*** [**then**] **{*action*;** ...**}** [**else if** ***bool-exp*** [**then**] **{*action*;** ...**}**] [**else** **{*action*;** ...**}**]

Syntax example:

```
if a > b {print a, b;} else {print b, a};
```

#### Notes

- Because the **if then else** clause comprises one action, the semicolon comes at the end of the clause and not after each action block within the clause. (Do not put a semicolon between the closing curly bracket for the action block and the **else** keyword.)
- You can repeat the **else if** clause multiple times.

### Parameters

| | |
|---|---|
| *bool-exp* | A boolean expression. |
| ***action***; ... | A series of zero or more actions separated by semicolons and enclosed in curly braces. |

### Description

If the first ***bool-exp*** is TRUE, the **then *action*** block is executed. If the first ***bool-exp*** is FALSE, the **else if** clauses are executed sequentially: if an **else if *bool-exp*** is found that is TRUE, its **then *action*** block is executed; otherwise the final **else *action*** block is executed.

The **else if then** clauses are used for multiple boolean checks (comparisons). If you require many **else if then** clauses, you might prefer to use a **case *bool-case-item*** action.

### Example 1

Following is the syntax example expressed as a multi-line example rather than a single-line example.

```
if a > b then {
    print a, b;
}
else {
    print b, a;
};
```

### Example 2

The following example includes an **else if** clause:

```
if a_ok {
    print x;
}
else if b_ok {
    print y;
}
else {
    print z;
};
```

### See Also

— “case labeled-case-item” on page 534
— “case labeled-case-item” on page 534
— “Iterative Actions” on page 537
— “File Iteration Actions” on page 545
— “Actions for Controlling the Program Flow” on page 547

## 18.1.2 case labeled-case-item

### Purpose

Execute an action block based on whether a given comparison is true

### Category

Action

### Syntax

**case *case-exp* {*labeled-case-item*;** ... **[default: {*default-action*;** ...**}]}**

Syntax example:

```
case packet.length {
    64:          {out("minimal packet")};
    [65..256]:   {out("short packet")};
    [257..512]:  {out("long packet")};
    default:    {out("illegal packet length")};
};
```

### Parameters

| | |
|---|---|
| *case-exp* | A legal ***e*** expression. |
| *labeled-case-item* | ***label-exp***[:] ***action-block***<br><br>Where<br><br>• ***label-exp*** is a value or a range<br>• ***action-block*** is a list of zero or more actions separated by semicolons and enclosed in curly braces. Syntax: {***action***;...}<br><br>Note that the entire ***labeled-case-item*** is repeatable, not just the ***action-block*** related to the ***label-exp***. |
| ***default-action***; ... | A sequence of zero or more default actions separated by semicolons and enclosed in curly braces. |

### Description

Evaluates the ***case-exp*** and executes the first ***action-block*** for which *label-exp* matches the ***case-exp***. If no ***label-exp*** equals the ***case-exp***, executes the ***default-action*** block, if specified.

After an ***action-block*** is executed, the ***e*** program proceeds to the line that immediately follows the entire **case** statement.

### Example

```
struct m {
    counter: uint;
    kind: [small, medium, large, xlarge, xxlarge];
    c_meth() is {
        case me.kind {
            small: {print "SMALL"};
            [large, medium]:  { print "LARGE or MEDIUM";
                                me.counter += 1;
                              };
            default: {print "OTHER"};
        };
        };
    };
```

### See Also

- “if then else” on page 533
- “case bool-case-item” on page 536
- “Iterative Actions” on page 537
- “File Iteration Actions” on page 545
- “Actions for Controlling the Program Flow” on page 547

## 18.1.3 case bool-case-item

### Purpose

Execute an action block based on whether a given boolean comparison is true

### Category

Action

### Syntax

**case** {***bool-case-item***; ... [**default** {***default-action***; ...}]}

Syntax example:

```
    case {
        packet.length == 64         {out("minimal packet"); };
        packet.length in [65..255]  {out("short packet"); };
        default                     {out("illegal packet"); };
    };
```

### Parameters

| | |
|---|---|
| *bool-case-item* | ***bool-exp***[:] ***action-block***<br><br>Where<br><br>• ***bool-exp*** is a boolean expression.<br><br>• ***action-block*** is a list of zero or more actions separated by semicolons and enclosed in curly braces. Syntax: {***action***;...}<br><br>Note that the entire ***bool-case-item*** is repeatable, not just the ***action-block*** related to the ***bool-exp***. |
| ***default-action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

### Description

Evaluates the ***bool-exp*** conditions one after the other; executes the ***action-block*** associated with the first TRUE ***bool-exp***. If no ***bool-exp*** is TRUE, executes the ***default-action-block***, if specified.

After an ***action-block*** is executed, the ***e*** program proceeds to the line that immediately follows the entire case statement.

Each of the ***bool-exp*** conditions is independent of the other ***bool-exp*** conditions, and there is no main ***case-exp*** to which all cases refer (unlike the "case labeled-case-item" on page 534).

This case action has the same functionality as a single **if then else** action in which you enter each ***bool-case-item*** as a separate **else if then** clause.

### Example

The ***bool-exp*** conditions are totally independent, and can refer to many arbitrary fields and attributes (not only to a single field as in the example above). For example, here is a set of independent boolean conditions:

```
case {
    kind == small {     // condition 1: relates to kind
        print "SMALL";
    };
    a > b {            // condition 2: relates to a and b
        print "a > b";
        var temp := a;
        a = b;
        b = temp;
    };
    a < b && kind == large {
                      // condition 3: relates to a,b,kind
        print "a < b && kind == large";
    };
    default {print "OTHER"};
                     // condition 4: default
};
```

### See Also

— "if then else" on page 533
— "case labeled-case-item" on page 534
— "Iterative Actions" on page 537
— "File Iteration Actions" on page 545
— "Actions for Controlling the Program Flow" on page 547

## 18.2 Iterative Actions

This section describes the following iterative actions, which are used to specify code segments that will be executed in a loop, multiple times, in a sequential order:

— "while" on page 538
— "repeat until" on page 539
— "for each in" on page 540
— "for from to" on page 543
— "for" on page 544

### 18.2.1 while

#### Purpose

Execute a **while** loop

#### Category

Action

#### Syntax

**while** ***bool-exp*** [**do**] **{*action*; ...}**

Syntax example:

```
while a < b {a += 1;};
```

#### Parameters

| | |
|---|---|
| *bool-exp* | A boolean expression. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

#### Example 1

The while loop in the following example adds 10 to “ctr” as many times as it takes it to get from 100 to the value of SMAX in steps of 1.

```
ctr_assn() is {
    var i: uint;
    i = 100;
    while (i <= SMAX) {
        ctr = ctr + 10;
        i+=1;
    };
};
```

#### Example 2

The while loop in the following example assigns “top.inc” to “ctr” every two cycles, as long as “done” remains FALSE.

```
lp_read()@clk is {
    while (!done) {
        wait [2]*cycle;
        ctr = top.inc;
    };
};
```

### Example 3

The while loop in the following example assigns “top.inc” to “ctr” every two cycles, in an endless loop. It loops until the test run is stopped.

```
lp_read()@clk is {
    while TRUE {
        wait [2]*cycle;
        ctr = top.inc;
    };
};
```

### Description

Executes the ***action*** block repeatedly in a loop while ***bool-exp*** is TRUE. You can use this construct to set up a perpetual loop as “while TRUE {}”.

### See Also

- “repeat until” on page 539
- “for each in” on page 540
- “for from to” on page 543
- “for” on page 544
- “Conditional Actions” on page 533
- “File Iteration Actions” on page 545
- “Actions for Controlling the Program Flow” on page 547

## 18.2.2 repeat until

### Purpose

Execute a **repeat until** loop

### Category

Action

### Syntax

**repeat {*action*; ...} until *bool-exp***

Syntax example:

```
repeat {i+=1;} until i==3;
```

### Parameters

| | |
|---|---|
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |
| *bool-exp* | A boolean expression. |

### Description

Execute the ***action*** block repeatedly in a loop until ***bool-exp*** is TRUE.

NOTE— A **repeat until** action performs the action block at least once. A **while** action might not perform the action block at all.

### Example

```
repeat {
    i+=1;
    print i;
} until i==3;
```

### See Also

- — "while" on page 538
- — "for each in" on page 540
- — "for from to" on page 543
- — "for" on page 544
- — "Conditional Actions" on page 533
- — "File Iteration Actions" on page 545
- — "Actions for Controlling the Program Flow" on page 547

## 18.2.3 for each in

### Purpose

Execute a **for each** loop

### Category

Action

### Syntax

**for each** [*type*] [(***item-name***)] [**using index (*index-name*)**]
**in** [**reverse**] ***list-exp*** [**do**] **{*action*;** ...**}**

Syntax example:

```
for each transmit packet (tp) in sys.pkts do {print tp};
                        // "transmit packet" is a type
```

## Parameters

| | |
|---|---|
| *type* | A type of the struct comprising the list specified by ***list-exp***. Elements in the list must match this type to be acted upon. |
| *item-name* | A name you give to specify the current item in ***list-exp***. <br><br> If you do not include this parameter, the item is referred to with the implicit variable "it". We recommend that you explicitly name the item to avoid confusion about the contents of "it". |
| *index-name* | A name you give to specify the index of the current list item. <br><br> If you do not include this parameter, the item is referred to with the implicit variable "index". We recommend that you explicitly name the item to avoid confusion about the contents of "index". |
| *list-exp* | An expression that results in a list. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

## Description

For each item in ***list-exp***, if its type matches ***type***, execute the ***action*** block. Inside the ***action*** block, the implicit variable **it** (or the optional ***item-name***) refers to the matched item, and the implicit variable **index** (or the optional ***index-name***) reflects the index of the current item. If **reverse** is specified, ***list-exp*** is traversed in reverse order, from last to first. The implicit variable **index** (or the optional ***index-name***) starts at zero for regular loops, and is calculated to start at "(list.size() - 1)" for reverse loops.

### The it and index Implicit Variables

Each **for each in** action defines two new local variables for the loop, named by default **it** and **index**. Keep the following in mind:

— If loops are nested one inside the other, the local variables of the internal loop hide those of the external loop. To overcome this hiding, specify the *item-name* and ***index-name*** with unique names.
— Within the action block, you cannot assign a value to **it** or **index**—or to ***item-name*** or ***index-name***.

## Example 1

```
<'
extend sys {
    do_it() is {
        var numbers := {1; 2; 3};
        for each in numbers {
            print it;
        };
        var sum: int;
        for each (n) in numbers {
            sum += n;
            print sum;
        };
    };
    run() is also {
        do_it();
    };
};
```

```
'>
```

### Example 2

```
for each in reverse pktList do {
                    // Traverse in reverse order
    print it;       // "it" can refer to the various subtypes
};
```

### Example 3

This example has two for each loops, each of which invokes a method and an **out()** routine for the particular subtype (ATM cell or IP cell).

```
<'
type cell_t: [ATM,IP];
struct cell{
    kind: cell_t;
    when ATM cell {
        meth() is  {
            outf("ATM cell: ");
        };
    };
    when IP cell {
        meth() is  {
            outf("IP cell: ");
        };
    };
};

extend sys {
    cell_l[20] : list of cell;
    run () is also {
        for each ATM cell (a) in cell_l {
            a.meth();
            out(index);
        };
        for each IP cell (a) in cell_l {
            a.meth();
            out(index);
        };
    }
};
'>
```

### See Also

— “while” on page 538
— “repeat until” on page 539
— “for from to” on page 543
— “for” on page 544
— “Implicit Variables” on page 24
— “Conditional Actions” on page 533
— “File Iteration Actions” on page 545
— “Actions for Controlling the Program Flow” on page 547

### 18.2.4 for from to

#### Purpose

Execute a **for** loop for the number of times specified by **from to**

#### Category

Action

#### Syntax

**for *var-name* from *from-exp* [down] to *to-exp* [step *step-exp*] [do] {*action*; ...}**

Syntax example:

```
for i from 5 down to 1 do {out(i);};    // Outputs 5,4,3,2,1
```

#### Parameters

| | |
|---|---|
| *var-name* | A temporary variable of type **int**. |
| *from-exp, to-exp, step-exp* | Valid ***e*** expressions that resolve to type **int**.<br>The default value for ***step-exp*** is one. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

#### Description

Creates a temporary variable ***var-name*** of type **int**, and repeatedly executes the ***action*** block while incrementing (or decrementing if **down** is specified) its value from ***from-exp*** to ***to-exp*** in interval values specified by ***step-exp*** (defaults to 1).

In other words, the loop is executed until the value of ***var-name*** is greater than the value of ***to-exp***. For example, the following line of code prints "in" one time:

```
for j from 1 to 1 {out("in");};
```

NOTE— The temporary variable ***var-name*** is visible only within the **for from to** loop in which it is created.

#### Example

```
for i from 2 to 2 * a do {
    out(i);
};
for i from 1 to 4 step 2 do {
    out(i);
};             // Outputs 1,3
for i from 4 down to 2 step 2 do {
    out(i);
};             // Outputs 4,2
```

### See Also

— “while” on page 538
— “repeat until” on page 539
— “for each in” on page 540
— “for” on page 544
— “Conditional Actions” on page 533
— “File Iteration Actions” on page 545
— “Actions for Controlling the Program Flow” on page 547

### 18.2.5 for

#### Purpose

Execute a C-style **for** loop

#### Category

Action

#### Syntax

**for** {***initial-action; bool-exp; step-action***} [**do**] {***action***; ...}

Syntax example:

```
for {i=0; i<=10; i+=1} do {out(i);};
```

#### Parameters

| | |
|---|---|
| *initial-action* | An action. |
| *bool-exp* | A boolean expression |
| *step-action* | An action. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

#### Description

The **for** loop works similarly to the for loop in the C language. This **for** loop executes the ***initial-action*** once, and then checks the bool-exp. If the ***bool-exp*** is TRUE, it executes the ***action*** block followed by the ***step-action***. It repeats this sequence in a loop for as long as ***bool-exp*** is TRUE.

#### Notes

— You must enter an ***initial-action***.
— If you use a loop variable within a **for** loop, you must declare it before the loop (unlike the temporary variable of type **int** automatically declared in a **for from to** loop).
— Although this action is similar to a C-style **for** loop, keep in mind that the ***initial-action*** and step-action must be ***e*** style actions. For example, the following syntax won’t run:

```
for {i=0,j=0; i < 10; i += 1} //incorrect syntax
```

While the following syntax will run:

```
for {{i=0;j=0}; i < 10; i += 1} //correct syntax
```

### Example

```
var i: int;
var j: int;
for {i = 0; i < 10; i += 1} do {
    if i % 3 == 0 then {
        continue;
    };
    j = j + i;
    if j > 100 then {
        break;
    };
};
```

### See Also

- "while" on page 538
- "repeat until" on page 539
- "for each in" on page 540
- "for from to" on page 543
- "Conditional Actions" on page 533
- "File Iteration Actions" on page 545
- "Actions for Controlling the Program Flow" on page 547

## 18.3 File Iteration Actions

This section describes the following two loop constructs, which are used to manipulate general ASCII files:

- "for each line in file" on page 545
- "for each file matching" on page 546

### 18.3.1 for each line in file

#### Purpose

Iterate a **for** loop over all lines in a text file

#### Category

Action

#### Syntax

**for each** [**line**] [(***name***)] **in file** ***file-name-exp*** [**do**] **{*action*;** ...**}**

Syntax example:

```
for each line in file "signals.dat" do {'(it)' = 1};
    // Reads a list of signal names and
    // assigns to each the value 1
```

### Parameters

| | |
|---|---|
| *name* | Variable referring to the current line in the file. |
| *file-name-exp* | A string expression that gives the name of a text file. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

### Description

Executes the ***action*** block for each line in the text file ***file-name***. Inside the block, **it** (or optional ***name***) refers to the current line (as string) without the final "\n" (the final new line character, CR).

### Example

This example reads each line of a file and prints the line if it is not blank. String matching is used to see if the line is blank: "l !~ "/^$/"" means "l does not match the beginning of a line, ^, followed immediately by the end of a line, $".

```
for each line (l) in file "test.dat" {
    // Print all the nonblank lines in the file
    if l !~ "/^$/" then { print l; };
    };
```

If the file cannot be opened, an error message similar to the following appears.

```
for each line in file "er_file" {print it};
*** Error: Cannot open input file 'er_file' for reading
```

### See Also

- "for each file matching" on page 546
- "Conditional Actions" on page 533
- "Iterative Actions" on page 537
- "Actions for Controlling the Program Flow" on page 547

## 18.3.2 for each file matching

### Purpose

Iterate a **for** loop over a group of files

### Category

Action

### Syntax

**for each file** [(***name***)] **matching** ***file-name-exp*** [**do**] {***action***; ...}

Syntax example:

```
for each file matching "*.e" {out(it);}
    //lists the e files in the current directory
```

### Parameters

| | |
|---|---|
| *name* | Variable referring to the current line in the file. |
| *file-name-exp* | A string expression giving a file name. |
| ***action***; ... | A sequence of zero or more actions separated by semicolons and enclosed in curly braces. |

### Description

For each file (in the file search path) whose name matches ***file-name-exp*** execute the ***action*** block. Inside the block, **it** (or optional *name*) refers to the matching file name.

### Example

```
for each file (f_name) matching "*.txt" do {
    for each line in file f_name{
        if it ~ "/error/" then {out(it);
        };
    };
};
```

### See Also

- "for each line in file" on page 545
- "Conditional Actions" on page 533
- "Iterative Actions" on page 537
- "Actions for Controlling the Program Flow" on page 547

## 18.4 Actions for Controlling the Program Flow

The actions described in this section are used to alter the flow of the program in places where the flow would otherwise continue differently. The ***e*** language provides the following actions for controlling the program flow:

- "break" on page 547
- "continue" on page 548

### 18.4.1 break

#### Purpose

Break the execution of a loop

#### Category

Action

#### Syntax

break

Syntax example:

```
break
```

### Description

Breaks the execution of the nearest enclosing iterative action (**for** or **while)**. When a **break** action is encountered within a loop, the execution of actions within the loop is terminated, and the next action to be executed is the first one following the loop.

You cannot place **break** actions outside the scope of a loop (the compiler will report an error).

### Example

```
for each (p) in packet_list do {
    // ... other code
    if p.len == 0 then {
    break; // Get out of this loop
    };
    // ... other code
};
```

### See Also

- "continue" on page 548
- "Conditional Actions" on page 533
- "Iterative Actions" on page 537
- "File Iteration Actions" on page 545

## 18.4.2 continue

### Purpose

Stop executing the current loop iteration and start executing the next loop iteration

### Category

Action

### Syntax

continue

Syntax example:

```
continue
```

### Description

Stops the execution of the nearest enclosing iteration of a **for** or a **while** loop, and continues with the next iteration of the same loop. When a **continue** action is encountered within a loop, the current iteration of the loop is aborted, and execution continues with the next iteration of the same loop.

You cannot place **continue** actions outside the scope of a loop (the compiler will report an error).

### Example

```
for each (p) in packet_list do {
    if p.len == 1 then {
        continue;     // Go to next iteration, skip "print p"
    };
    print p;
};
```

### See Also

- "break" on page 547
- "Conditional Actions" on page 533
- "Iterative Actions" on page 537
- "File Iteration Actions" on page 545

# 19 List Pseudo-Methods Library

This chapter describes pseudo-methods used to work with lists. It contains the following sections:

## 19.1 Pseudo-Methods Overview

A pseudo-method is a type of method unique to the ***e*** language. Pseudo-methods are ***e*** macros that look like methods. They have the following characteristics:

— Unlike methods, pseudo-methods are not restricted to structs.
— They can be applied to any expression, including literal values, scalars, and compound arithmetic expressions.
— You cannot extend pseudo-methods.
— You can define your own pseudo-methods using “define as” on page 429 or “define as computed” on page 436.
— List pseudo-methods are associated with list data types, as opposed to being within the scope of a struct.

### See Also

— “e Data Types” on page 75
— “list.method()” on page 66
— “in” on page 49
— Chapter 23, “Predefined Methods Library”

## 19.2 Using List Pseudo-Methods

Once a list field or variable has been declared, you can operate on it with a list pseudo-method by attaching the pseudo-method name, preceded by a period, to the list name. Any parameters required by the pseudo-method go in parentheses after the pseudo-method name.

Many of the list pseudo-methods take expressions as parameters, an operate on every item in the list.

For example, the following calls the **apply()** pseudo-method for the list named “p_list”, with the expression “.length + 2” as a parameter. The pseudo-method returns a list of numbers found by adding 2 to the “length” field value in each item in the list.

```
n_list = p_list.apply(.length + 2);
```

It is important to put a period (.) in front of field names being accessed by pseudo-methods, as in “.length +2”, above.

In pseudo-methods that take expressions as parameters, the **it** variable can be used in the expression to refer to the current list item, and the **index** variable can be used to refer to the current item's list index number.

Pseudo-methods that return values can only be used in expressions.

### See Also

— "Pseudo-Methods Overview" on page 551
— "e Data Types" on page 75
— "list.method()" on page 66

## 19.3 Pseudo-Methods to Modify Lists

This section describes the pseudo-methods that change one or more items in a list.

The pseudo-methods in this section are:

— "add(item)" on page 552
— "add(list)" on page 554
— "add0(item)" on page 555
— "add0(list)" on page 556
— "clear()" on page 557
— "delete()" on page 558
— "fast_delete()" on page 560
— "insert(index, item)" on page 561
— "insert(index, list)" on page 562
— "pop()" on page 563
— "pop0()" on page 564
— "push()" on page 565
— "push0()" on page 565
— "resize()" on page 566

### See Also

— "Pseudo-Methods Overview" on page 551
— "Using List Pseudo-Methods" on page 551

### 19.3.1 add(item)

#### Purpose

Add an item to the end of a list

#### Category

Pseudo-method

#### Syntax

***list*.add(*item*:** list-type**)**

Syntax example:

```
var i_list: list of int;
i_list.add(5);
```

## Parameters

| | |
|---|---|
| *list* | A list. |
| *item* | An item of the same type as the list type, which is to be added to the list. The item is added at index list.size(). That is, if the list contains five items, the last item is at index list.size() - 1, or 4. Adding an item to this list places it at index  5. |

## Description

Adds the ***item*** to the end of the ***list***.

If the item is a struct, no new struct instance is generated, a pointer to the existing instance of the struct is simply added to the list. For information about creating new structs, use the **new**, **gen**, **copy()**, **deep_copy()**, and **unpack()** links in the "See Also" list below.

## Example 1

The following adds 2 to the end of the list named "i_list" (that is, to index position 3).

```
var i_list: list of int = {3; 4; 6};
i_list.add(2);
```

## Result

```
print i_list
  i_list =
0.    3
1.    4
2.    6
3.    2
```

## Example 2

The following generates an instance of a "packet" struct and adds it to the list of packets named "p_lst".

```
struct p_l {
    !packet_i: packet;
    !p_lst: list of packet;
    mk_lst()@sys.clk is {
        gen packet_i;
        p_lst.add(packet_i);
        stop_run();
    };
};
```

## See Also

— "new" on page 69
— "gen" on page 296
— "The copy() Method of any_struct" on page 647
— "deep_copy()" on page 713
— "unpack()" on page 521

- — “add(list)” on page 554
- — “add0(item)” on page 555
- — “add0(list)” on page 556
- — “insert(index, item)” on page 561
- — “insert(index, list)” on page 562
- — “push()” on page 565
- — “push0()” on page 565
- — “resize()” on page 566

## 19.3.2 add(list)

### Purpose

Add a list to the end of another list

### Category

Pseudo-method

### Syntax

***list_1*.add(*list_2*:** list**)**

Syntax example:

```
i_list.add(l_list);
```

### Parameters

| | |
|---|---|
| *list_1* | A list. |
| *list_2* | A list of the same type as list_1, which is to be added to the end of list_1. The list is added at index list.size(). That is, if the list contains five lists, the last list is at index list.size() - 1, or 4. Adding a list to this list places it at index 5. |

### Description

Adds ***list_2*** to the end of ***list_1***.

### Example 1

The following adds “blue”, “green”, and “red” to the list named “colors_1”.

```
<'
type color: [blue, green, yellow, orange, red];
extend sys {
    run() is also {
        var colors_1: list of color = {red; red; blue};
        var colors_2: list of color = {blue; green; red};
        colors_1.add(colors_2);
        print colors_1;
    };
};
'>
```

### Example 2

The following example adds the literal list {“blue”; “green”; “red”} to the list named “colors_3”. The “colors_3” list then contains “red”, “red”, “blue”, “blue”, “green”, “red”.

```
var colors_3 := {"red"; "red"; "blue"};
colors_3.add({"blue"; "green"; "red"});
```

### See Also

— “add(item)” on page 552
— “add0(item)” on page 555
— “add0(list)” on page 556
— “insert(index, item)” on page 561
— “insert(index, list)” on page 562
— “push()” on page 565
— “push0()” on page 565
— “resize()” on page 566

## 19.3.3 add0(item)

### Purpose

Add an item to the head of a list

### Category

Pseudo-method

### Syntax

***list*.add0(*item*:** list-type**)**

Syntax example:

```
var l_list: list of int = {4; 6; 8};
l_list.add0(2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *item* | An item of the same type as the list items, which is to be added to the head of the list. |

### Description

Adds a new item to an existing list. The item is placed at the head of the existing list, as the first position (that is, at index 0). All subsequent items are then reindexed by incrementing their old index by one.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list. For information about generating new struct instances, use the **new**, **gen**, and **unpack()** links in the "See Also" list below.

### Example

The following example adds 1 to the beginning of the list named “i_list”. The “i_list” then contains 1, 1, 2, 3, 4, 5.

```
var i_list: list of int = {1;2;3;4;5};
i_list.add0(1);
```

### See Also

- “new” on page 69
- “gen” on page 296
- “The copy() Method of any_struct” on page 647
- “deep_copy()” on page 713
- “unpack()” on page 521
- “add(item)” on page 552
- “add(list)” on page 554
- “add0(list)” on page 556
- “insert(index, item)” on page 561
- “insert(index, list)” on page 562
- “push()” on page 565
- “push0()” on page 565
- “resize()” on page 566

## 19.3.4 add0(list)

### Purpose

Add a list to the head of another list

### Category

Pseudo-method

### Syntax

***list_1*.add0(*list_2*: list)**

Syntax example:

```
var i_list: list of int = {1; 3; 5};
var l_list: list of int = {2; 4; 6};
i_list.add0(l_list);
```

### Parameters

| | |
|---|---|
| *list_1* | A list. |
| *list_2* | A list of the same type as list_1, which is to be added to the beginning of list_1 (at list_1 index 0) |

### Description

Adds a new list to an existing list. The ***list_2*** list is placed at the head of the existing ***list_1*** list, starting at the first ***list_1*** index. All subsequent items are then reindexed by incrementing their old index by the size of the new list being added.

### Example

The following adds 1, 2, 3, and 4 to the beginning of the list named “b_list”. The “b_list” then contains 1, 2, 3, 4, 5, 6.

```
var a_list: list of int = {1;2;3;4};
var b_list: list of int = {5;6};
b_list.add0(a_list);
```

NOTE— b_list**.add0(**a**)** returns the same result as a_list**.add(**b**)** in the above example, except that in the example, “b_list” is added into “a_list”, while b_list**.add0(**a**)** adds “a_list” into “b_list”.

### See Also

- — “add(item)” on page 552
- — “add(list)” on page 554
- — “add0(item)” on page 555
- — “insert(index, item)” on page 561
- — “insert(index, list)” on page 562
- — “push()” on page 565
- — “push0()” on page 565
- — “resize()” on page 566

## 19.3.5 clear()

### Purpose

Delete all items from a list

### Category

Pseudo-method

### Syntax

*list*.clear()

Syntax example:

```
a_list.clear();
```

#### Parameters

*list* A list.

#### Return Value

None

#### Description

Deletes all items in the list.

#### Example

The following removes all items from the list named “l_list”.

```
l_list.clear();
```

#### See Also

— “delete()” on page 558
— “fast_delete()” on page 560
— “resize()” on page 566
— “pop()” on page 563
— “pop0()” on page 564

### 19.3.6 delete()

#### Purpose

Delete an item from a list

#### Category

Pseudo-method

#### Syntax

***list*.delete(*index*:** int**)**

Syntax example:

```
var l_list: list of int = {2; 4; 6; 8};
l_list.delete(2);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *index* | The index of the item that is to be deleted from the list. |

#### Description

Removes item number ***index*** from ***list*** (indexes start counting from 0). The indexes of the remaining items are adjusted to keep the numbering correct.

If the index does not exist in the list, an error is issued.

### Example 1

The following deletes 7 from index position 1 in the list named “y_list”. The list then consists of 5 (index 0) and 9 (index 1).

```
var y_list := {5; 7; 9};
y_list.delete(1);
```

### Example 2

Since ***list*.delete()** only accepts a single item as its argument, you cannot use it to delete a range of items in one call. This example shows a way to do that.

The following shows a user-defined method named del_range() which, given a list, a from value, and a to value, produces a new list (with the same name) of the items in the previous list, minus the items with indexes in the given range. If the range of values is not legal for the given list, the method fails with an error message.

```
<'
extend sys {
    my_list: list of byte;
    keep my_list.size() == 20;

    del_range(in_l:list of byte,from:int,to:int):list of byte
        is {
            result.add(in_l[..from-1]);
            result.add(in_l[to+1..]);
       };

    post_generate() is also {
        my_list = del_range(my_list,5,15);
    };
};
'>
```

### See Also

— “clear()” on page 557
— “fast_delete()” on page 560
— “pop()” on page 563
— “pop0()” on page 564
— “resize()” on page 566
— “first_index()” on page 577
— “get_indices()” on page 579
— “last_index()” on page 584
— “max_index()” on page 587
— “min_index()” on page 590

### 19.3.7 fast_delete()

#### Purpose

Delete an item without adjusting all indexes

#### Category

Pseudo-method

#### Syntax

***list*.fast_delete(*index*:** int**)**

Syntax example:

```
var l_list: list of int = {2; 4; 6; 8};
l_list.fast_delete(2);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *index* | The index that is to be deleted from the list. |

#### Description

Removes item number ***index*** from ***list*** (indexes start counting from 0). The index of the last item in the list is changed to the index of the item that was deleted, so all items following the deleted item keep their original indexes except that the original last index is removed.

If the index does not exist in the list, an error is issued.

#### Example

The following deletes “C” from index position 2 in the list named “y_list”, and changes the index of the last item from 4 to 2. The new “y_list” is “A”, “B”, “E”, “D”.

```
<'
extend sys {
    run() is also {
        var y_list := {"A"; "B"; "C"; "D"; "E"};
        y_list.fast_delete(2);
        for i from 0 to 3 do {
            print y_list[i];
        };
    };
};
'>
```

#### See Also

— “clear()” on page 557
— “delete()” on page 558
— “pop()” on page 563
— “pop0()” on page 564

- “resize()” on page 566
- “first_index()” on page 577
- “get_indices()” on page 579
- “last_index()” on page 584
- “max_index()” on page 587
- “min_index()” on page 590

### 19.3.8 insert(index, item)

#### Purpose

Insert an item in a list at a specified index

#### Category

Pseudo-method

#### Syntax

***list*****.insert(*****index*****:** int, ***item*****:** list-type**)**

Syntax example:

```
var l_list := {10; 20; 30; 40; 50};
l_list.insert(3, 99);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *index* | The index in the list where the item is to be inserted. |
| *item* | An item of the same type as the list. |

#### Description

Inserts the ***item*** at the ***index*** location in the ***list***. If ***index*** is the size of the list, then the ***item*** is simply added at the end of the list. All indexes in the list are adjusted to keep the numbering correct.

If the number of items in the list is smaller than ***index***, an error is issued.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list. For information about generating new struct instances, use the **new**, **gen**, and **unpack()** links in the “See Also” list below.

#### Example

In the following example, 10 is first inserted into position 2 in “s_list”, and then 77 is inserted into position 1. The resulting list contains 5, 77, 1, 10.

```
var s_list := {5; 1};
s_list.insert(2,10);
s_list.insert(1,77);
```

### See Also

- "new" on page 69
- "gen" on page 296
- "The copy() Method of any_struct" on page 647
- "deep_copy()" on page 713
- "unpack()" on page 521
- "add(item)" on page 552
- "add(list)" on page 554
- "add0(item)" on page 555
- "add0(list)" on page 556
- "insert(index, list)" on page 562
- "push()" on page 565
- "push0()" on page 565
- "resize()" on page 566

## 19.3.9 insert(index, list)

### Purpose

Insert a list in another list starting at a specified index

### Category

Pseudo-method

### Syntax

***list_1*.insert(*index*:** int, ***list_2*:** list**)**

Syntax example:

```
var l_list := {10; 20; 30; 40; 50};
var m_list := {11; 12; 13};
l_list.insert(1, m_list);
```

### Parameters

| | |
|---|---|
| *list_1* | A list. |
| *index* | The index of the position in list_1 where list_2 is to be inserted. |
| *list_2* | A list that is to be inserted into list_1. |

### Description

Inserts all items of ***list_2*** into ***list_1*** starting at ***index***. The ***index*** must be a positive integer. The size of the new list size is equal to the sum of the sizes of ***list_1*** and ***list_2***.

If the number of items in ***list_1*** is smaller than ***index***, an error is issued.

### Example

In the following example, "blue", "green", and "red" are inserted after "red" in the "colors_1" list. The "colors_l" list is then "red", "blue", "green", "red", "green", "blue".

```
var colors_14 := {"red"; "green"; "blue"};
var colors_15 := {"blue"; "green"; "red"};
colors_14.insert(1, colors_15);
```

### See Also

- "add(item)" on page 552
- "add(list)" on page 554
- "add0(item)" on page 555
- "add0(list)" on page 556
- "insert(index, item)" on page 561
- "push()" on page 565
- "push0()" on page 565
- "resize()" on page 566

## 19.3.10 pop()

### Purpose

Remove and return the last list item

### Category

Pseudo-method

### Syntax

***list*.pop()**: list-type

Syntax example:

```
var i_list:= {10; 20; 30};
var i_item: int;
i_item = i_list.pop();
```

### Parameters

| | |
|---|---|
| *list* | A list. |

### Description

Removes the last item (the item at index list.size() - 1) in the ***list*** and returns it. If the list is empty, an error is issued.

NOTE— Use ***list*.top()** to return the last item in ***list*** without removing it from the list.

### Example

In the following example, the "s_item" variable gets "d", and the "s_list" becomes "a", "b", "c".

```
var s_item: string;
var s_list := {"a"; "b"; "c"; "d"};
s_item = s_list.pop();
```

### See Also

— "pop0()" on page 564
— "delete()" on page 558
— "top()" on page 600

## 19.3.11 pop0()

### Purpose

Remove and return the first list item

### Category

Pseudo-method

### Syntax

***list*.pop0()**: list-type

Syntax example:

```
var i_list:= {10; 20; 30};
var i_item: int;
i_item = i_list.pop0();
```

### Parameters

| | |
|---|---|
| *list* | A list. |

### Description

Removes the first item (the item at index 0) from the ***list*** and returns it. Subtracts 1 from the index of each item remaining in the list. If the list is empty, an error is issued.

NOTE— Use *list*.**top0()** to return the first item in *list* without removing it from the list.

### Example

In the following example, the "s_item" variable gets "a" and "s_list" becomes "b", "c", "d".

```
var s_item: string;
var s_list := {"a"; "b"; "c"; "d"};
s_item = s_list.pop0();
```

### See Also

— "delete()" on page 558
— "pop()" on page 563
— "top0()" on page 601

### 19.3.12 push()

#### Purpose

Add an item to the end of a list (same as “add(item)” on page 552)

#### Category

Pseudo-method

#### Syntax

***list*.push(*item*:** list-type**)**

Syntax example:

```
var i_list: list of int;
i_list.push(5);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *item* | An item of the same type as the list type, which is to be added to the list. The item is added at index list.size(). That is, if the list contains five items, the last item is at index list.size() - 1, or 4. Adding an item to this list places it at index 5. |

#### Description

This pseudo-method performs the same function as “add(item)” on page 552.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list. For information about generating new struct instances, use the **new**, **gen**, and **unpack()** links in the "See Also" list below.

#### See Also

— “new” on page 69
— “gen” on page 296
— “The copy() Method of any_struct” on page 647
— “deep_copy()” on page 713
— “unpack()” on page 521
— “add(item)” on page 552
— “add(list)” on page 554
— “add0(item)” on page 555
— “add0(list)” on page 556
— “insert(index, item)” on page 561
— “insert(index, list)” on page 562
— “push0()” on page 565

### 19.3.13 push0()

#### Purpose

Add an item to the head of a list (same as “add0(item)” on page 555)

### Category

Pseudo-method

### Syntax

***list*.push0(*item*:** list-type**)**

Syntax example:

```
var l_list: list of int = {4; 6; 8};
l_list.push0(2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *item* | An item of the same type as the list items, which is to be added to the head of the list. |

### Description

This pseudo-method performs the same function as “add0(item)” on page 555.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list. For information about generating new struct instances, use the **new**, **gen**, and **unpack()** links in the "See Also" list below.

### See Also

— “new” on page 69
— “gen” on page 296
— “The copy() Method of any_struct” on page 647
— “deep_copy()” on page 713
— “unpack()” on page 521
— “add(item)” on page 552
— “add(list)” on page 554
— “add0(item)” on page 555
— “add0(list)” on page 556
— “insert(index, item)” on page 561
— “insert(index, list)” on page 562
— “push()” on page 565

## 19.3.14 resize()

### Purpose

Change the size of a list

### Category

Pseudo-method

### Syntax

***list*.resize(*size*:** int [**, *full*:** bool**, *filler*:** exp**, *keep_old*:** bool]**)**

Syntax example:

```
var r_list := {2; 3; 5; 6; 8; 9};
r_list.resize(10, TRUE, 1, TRUE);
```

## Parameters

| | |
|---|---|
| *list* | A list. |
| *size* | A positive integer specifying the desired size. |
| *full* | A boolean value specifying all items will be filled with filler. Default: TRUE. |
| *filler* | An item of the same type of the list items, used as a filler when FULL is TRUE. |
| *keep_old* | A boolean value specifying whether to keep existing items already in the list. Default: FALSE. |

## Description

Allocates a new list of declared ***size***, or resizes an old list if ***keep_old*** is TRUE. If ***full*** is TRUE, sets all new items to have ***filler*** as their value.

If only the first parameter, ***size***, is used, this method allocates a new list of the given size and all items are initialized to the default value for the list type.

If any of the three parameters after ***size*** are used, all three of them must be used.

To resize and list and keep its old values, set both ***full*** and ***keep_old*** to TRUE. If the list is made longer, additional items with the value of ***filler*** are appended to the list.

Following is the behavior of this method for all combinations of ***full*** and ***keep_old***.

— ***full*** is FALSE, ***keep_old*** is FALSE:
An empty list (that is, a list of zero size) is created, and memory is allocated for a list of the given ***size***.

— ***full*** is TRUE, ***keep_old*** is FALSE:
The list is resized to ***size***, and filled completely with ***filler***.

— ***full*** is FALSE, ***keep_old*** is TRUE:

- If ***size*** is greater than the size of the existing list, the list is enlarged to the new ***size***, and the new positions are filled with the default value of the list type.
- If ***size*** is less than or equal to the size of the existing list, the list is shortened to the new ***size***, and all of the existing values up to that size are retained.

— ***full*** is TRUE, ***keep_old*** is TRUE:

- If ***size*** is greater than the size of the existing list, the list is enlarged to the new ***size***, and the new positions are filled with ***filler***.
- If ***size*** is less than or equal to the size of the existing list, the list is shortened to the new ***size***, and all of the existing values up to that size are retained.

## Example 1

The following example puts 200 NULL “packet” instances into “q_list”. The initial size of the list is 0 when it is created by the **var** action. The packets are NULL because that is the default value for a struct instance.

```
<'
struct packet {
    len: uint;
    addr: byte;
};
extend sys {
    run() is also {
        var q_list: list of packet;
        print q_list.size();
        q_list.resize(200);
        print q_list.size();
    };
};
'>
```

### Result

```
q_list.size() = 0
q_list.size() = 200
```

### Example 2

The following example puts 20 NULL strings in “r_list”. The initial size of the list is 0 when it is created by the **var** action.

```
var r_list: list of string;
r_list.resize(20, TRUE, NULL, FALSE);
print r_list.size();
print r_list;
```

### Result

```
  r_list.size() = 20
  r_list =
0.      ""
1.      ""
2.      ""
3.      ""
4.      ""
5.      ""
6.      ""
7.      ""
8.      ""
9.      ""
10.     ""
11.     ""
12.     ""
13.     ""
14.     ""
15.     ""
16.     ""
17.     ""
18.     ""
19.     ""
```

### Example 3

The following example makes “s_list” an empty list, but allocates space for it to hold 20 integers. The initial size of the list is 0 when it is created by the **var** action, since “full” is FALSE.

```
var s_list: list of int;
s_list.resize(20, FALSE, 0, FALSE);
print s_list.size();
print s_list;
```

### Result

```
s_list.size() = 0
s_list = (empty)
```

### Example 4

The following example adds four items to an existing list.

```
var r_list := {2; 3; 5; 6; 8; 9};
r_list.resize(10, TRUE, 1, TRUE);
print r_list.size();
print r_list;
```

### Result

```
  r_list.size() = 10
  r_list =
0.      2
1.      3
2.      5
3.      6
4.      8
5.      9
6.      1
7.      1
8.      1
9.      1
```

### Example 5

This example shortens an existing list.

```
var r_list := {2; 3; 5; 6; 8; 9};
r_list.resize(4, TRUE, 7, TRUE);
print r_list.size();
print r_list;
```

### Result

```
  r_list.size() = 4
  r_list =
0.      2
1.      3
2.      5
3.      6
```

**See Also**

— “add(item)” on page 552
— “add(list)” on page 554
— “add0(item)” on page 555
— “add0(list)” on page 556
— “insert(index, item)” on page 561
— “insert(index, list)” on page 562
— “push()” on page 565
— “push0()” on page 565
— “max_index()” on page 587
— “size()” on page 594

## 19.4 General List Pseudo-Methods

This section describes the syntax for pseudo-methods that perform various operations on lists.

The pseudo-methods in this section are:

— “apply()” on page 571
— “copy()” on page 572
— “count()” on page 573
— “exists()” on page 574
— “field” on page 575
— “first()” on page 576
— “first_index()” on page 577
— “get_indices()” on page 579
— “has()” on page 579
— “is_a_permutation()” on page 581
— “is_empty()” on page 582
— “last()” on page 583
— “last_index()” on page 584
— “max()” on page 585
— “max_index()” on page 587
— “max_value()” on page 588
— “min()” on page 589
— “min_index()” on page 590
— “min_value()” on page 591
— “reverse()” on page 593
— “size()” on page 594
— “sort()” on page 595
— “sort_by_field()” on page 596
— “split()” on page 597
— “top()” on page 600
— “top0()” on page 601
— “unique()” on page 602

**See Also**

— “Pseudo-Methods Overview” on page 551
— “Using List Pseudo-Methods” on page 551

### 19.4.1 apply()

#### Purpose

Perform a computation on each item in a list

#### Category

Pseudo-method

#### Syntax

***list*.apply(*item***: exp**)**: list

Syntax example:

```
var p_list:= {1; 3; 5};
var n_list: list of int;;
n_list = p_list.apply(it*2);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *item* | Any expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

#### Description

Applies the ***exp*** to each item in the ***list*** and returns the changed list.

NOTE— The expression "*list***.apply(it.***field***)**" is the same as "*list.field*" when *field* is a scalar type. For example, the following expressions both return a concatenated list of the "addr" field in each packet item:

```
packets.apply(it.addr)
sys.packets.addr
```

The two expressions are different, however, if the field not scalar. For example, assuming that "data" is a list of byte, the first expression returns a list containing the first byte of "data" of each packet item. The second expression is a single item, which is the first item in the concatenated list of all "data" fields in all packet items.

```
packets.apply(it.data[0])
packets.data[0]
```

#### Example 1

In the following example, the "n_list" in the **sys** struct gets a list of integers resulting from adding 1 to each "len" value in the list of packets.

```
<'
struct packet {
    len: uint;
    addr: byte;
```

```
        data: list of bit;
        !%crc: uint;
    };
    extend sys {
        p_list: list of packet;
        !n_list: list of uint;
        run() is also {
            n_list = p_list.apply(it.len + 1);
        };
    };
    '>
```

### Example 2

In the following example, the “packet” struct contains a “get_crc()” method that calls the predefined “crc_32()” on page 615. The “crc_plus” list gets the values returned by applying the “get_crc()” method to every packet in the “p_list” list.

```
    <'
    struct packet {
        %len: uint;
        %addr: byte;
        %data: list of byte;
        !%crc: int;
        get_crc(): int is {
            return data.crc_32(0, data.size());
        };
    };
    extend sys {
        p_list: list of packet;
        !crc_plus: list of int;
        post_generate() is also {
            crc_plus = p_list.apply(it.get_crc());
        };
    };
    '>
```

### Example 3

In the following example, the “ind_list” gets the indexes (0, 1, 2) of the items in the “l_list”.

```
    var l_list: list of int = {5; 7; 9};
    var ind_list: list of int = l_list.apply(index);
    print ind_list;
```

### See Also

— “field” on page 575
— “all()” on page 604

## 19.4.2 copy()

### Purpose

Make a shallow copy of a list

### Category

Predefined method of any struct or unit

### Syntax

***list*.copy()**: list

Syntax example:

```
var strlist_1: list of string = {"A"; "B"; "C"};
var strlist_2: list of string;
strlist_2 = strlist_1.copy();
```

### Description

This is a specific case of ***exp*.copy()**, where ***exp*** is the name of a list. See “The copy() Method of any_struct” on page 647 for additional information and examples.

## 19.4.3 count()

### Purpose

Return the number of items that satisfy a given condition

### Category

Pseudo-method

### Syntax

***list*.count(*exp***: bool**)**: int

Syntax example:

```
var ct: int;
ct = instr_list.count(it.op1 > 200);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the number of items for which the ***exp*** is TRUE.

NOTE— The syntax ***list*.all(*exp*).size()** returns the same result as the ***list*.count**(***exp***) pseudo-method, but ***list*.all(*exp*).size()** creates a new list and is faster.

### Example 1

The following example prints 3, since there are three items in “l_list” with values greater than 3.

```
var l_list: list of int = {2; 3; 4; 5; 6};
print l_list.count(it > 3)
```

### Example 2

The following example prints the number of “packet” struct instances in the “packets” list that have a “length” field value smaller than 5.

```
<'
struct packet {
    length: uint (bits: 4);
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var pl: int;
        pl = packets.count(.length < 5);
        print pl;
    };
};
'>
```

### See Also

— “has()” on page 579
— “all()” on page 604
— “all_indices()” on page 605
— “key()” on page 619
— “key_exists()” on page 623

## 19.4.4 exists()

### Purpose

Check if an index exists in a list

### Category

Pseudo-method

### Syntax

***list*.exists(*index*:** int**)**: bool

Syntax example:

```
var i_chk: bool;
i_chk = packets.exists(5);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *index* | An integer expression representing an index to the list. |

### Description

Returns TRUE if an item with the ***index*** number exists in the ***list***, or returns FALSE if the index does not exist.

### Example

The first **print** action in the following prints TRUE, because the “int_list” contains an item with an index of 1. The second **print** action prints FALSE, because there is no item with index 7 in the list.

```
<'
extend sys {
    run() is also {
        var int_lst: list of int = {1; 2; 3; 4; 5};
        var ind_ex: bool;
        ind_ex = int_lst.exists(1);
        print ind_ex;
        ind_ex = int_lst.exists(7);
        print ind_ex;
    };
};
'>
```

### See Also

— “first_index()” on page 577
— “all_indices()” on page 605

## 19.4.5 field

### Purpose

Specifying a field from all items in a list

### Category

Pseudo-method

### Syntax

*list.field-name*

Syntax example:

```
s_list.fld_nm
```

### Parameters

| | |
|---|---|
| *list* | A list of structs. |
| *field-name* | A name of a field or list in the struct type. |

### Description

Returns a list containing the contents of the specified ***field-name*** for each item in the ***list***. If the ***list*** is empty, it returns an empty list. This syntax is the same as ***list*.apply(*field*)**.

An error is issued if the field name is not the name of a struct or if the struct type does not have the specified field

### Example

The following prints the values of the "length" fields in all the items in the "packets" list.

```
<'
struct packet {
    length: uint;
};
extend sys {
    packets: list of packet;
    run() is also {
        print packets.length;
    };
};
'>
```

### See Also

— "apply()" on page 571

## 19.4.6 first()

### Purpose

Get the first item that satisfies a given condition

### Category

Pseudo-method

### Syntax

***list*.first(*exp***: bool**)**: list-type

Syntax example:

```
var i_item: instr;
i_item = instr_list.first(it.op1 > 15);
```

### Parameters

*list* A list.

*exp* A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number.

### Description

Returns the first item for which ***exp*** is TRUE. If there is no such item, the default for the item's type is returned (see "e Data Types" on page 75).

For a list of scalars, a value of zero is returned if there is no such item. Since zero might be confused with a value found, it is safer to use ***list*.first_index()** for lists of scalars.

### Example 1

The first line below creates a list of five integers. The second line prints the first item in the list smaller than 5 (that is, it prints 3).

```
var i_list :list of int = {8;3;7;3;4};
print i_list.first(it < 5 );
```

### Example 2

In the following example, the ***list*.first.()** pseudo-method is used to make sure all items in the "packets" list contain non-empty "cells" lists.

```
<'
struct cell {
    data: list of byte;
};
struct packet {
    cells: list of cell;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
       check that sys.packets.first(.cells is empty) == NULL;
    };
};
'>
```

### See Also

— "first_index()" on page 577
— "has()" on page 579
— "last()" on page 583

## 19.4.7 first_index()

### Purpose

Get the index of the first item that satisfies a given condition

### Category

Pseudo-method

### Syntax

***list*.first_index(*exp*:** bool**):** int

Syntax example:

```
var i_item: int;
i_item = instr_list.first_index(it.op1 > 15);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the index of the first item for which ***exp*** is TRUE or return UNDEF if there is no such item.

### Example 1

The first line below creates a list of five integers. The second line prints 1, which is the index of the first item in the list smaller than 5.

```
var i_list :list of int = {8;3;7;3};
print i_list.first_index(it < 5);
```

### Example 2

In the following example, the ***list*.first_index.()** pseudo-method is used to make sure all items in the "pack-ets" list contain non-empty "cells" lists.

```
<'
struct cell {
    data: list of byte;
};
struct packet {
    cells: list of cell;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        check that
            packets.first_index(.cells is empty) == UNDEF;
    };
};
'>
```

### See Also

— "first()" on page 576
— "has()" on page 579

— “last_index()” on page 584

## 19.4.8 get_indices()

### Purpose

Return a sublist of another list

### Category

Pseudo-method

### Syntax

***list*.get_indices(*index-list*:** list of int**):** list-type

Syntax example:

```
var i_list: list of packet;
i_list = packets.get_indices({0; 1; 2});
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *index-list* | A list of indexes within the list. Each index must exist in the list. |

### Description

Copies the items in ***list*** that have the indexes specified in ***index-list*** and returns a new list containing those items. If the ***index-list*** is empty, an empty list is returned.

### Example

The following example puts “green” and “orange” in the list named “o_list”.

```
var c_list := {"red"; "green"; "blue"; "orange"};
var o_list := c_list.get_indices({1;3});
```

### See Also

— “first_index()” on page 577
— “has()” on page 579
— “last_index()” on page 584
— “all_indices()” on page 605

## 19.4.9 has()

### Purpose

Check that a list has at least one item that satisfies a given condition

### Category

Pseudo-method

### Syntax

***list*.has(*exp*:** bool**):** bool

Syntax example:

```
var i_ck: bool;
i_ck = sys.instr_list.has(it.op1 > 31);
```

### Parameters

*list* A list.

*exp* A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number.

### Description

Returns TRUE if the ***list*** contains at least one item for which the ***exp*** is TRUE, or returns FALSE if the expression is not TRUE for any item.

### Example 1

The first line below creates a list containing the integers 8, 3, 7, and 3. The second line checks that the list contains 7, and prints TRUE.

```
var l: list of int = {8;3;7;3};
print l.has(it == 7);
```

### Example 2

The line below checks that there is no packet in the “packets” list that contains an empty “cells” list.

```
<'
struct cell {
    data: list of byte;
};
struct packet {
    cells: list of cell;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        check that not sys.packets.has(.cells is empty);
    };
};
'>
```

### See Also

— “first()” on page 576
— “last()” on page 583
— “max()” on page 585

— "min()" on page 589
— "all()" on page 604

### 19.4.10 is_a_permutation()

#### Purpose

Check that two lists contain exactly the same items

#### Category

Pseudo-method

#### Syntax

***list_1*.is_a_permutation(*list_2***: list**)**: bool

Syntax example:

```
var lc: bool;
lc = packets_1a.is_a_permutation(packets_1b);
```

#### Parameters

| | |
|---|---|
| *list_1* | A list. |
| *list_2* | A list that is to be compared to list_1. Must be the same type as list_1. |

#### Description

Returns TRUE if ***list_2*** contains the same items as ***list_1***, or FALSE if any items in one list are not in the other list. The order of the items in the two lists does not need to be the same, but the number of items must be the same for both lists. That is, items that are repeated in one list must appear the same number of times in the other list.

#### Notes

— If the lists are lists of structs, ***list_1*.is_a_permutation(*list_2*)** compares the addresses of the struct items, not their contents.
— This pseudo-method can be used in a **keep** constraint to fill ***list_1*** with the same items contained in the ***list_2***, although not necessarily in the same order.

#### Example 1

In the following example, the "l_comp" variable is TRUE because the two lists contain the same items.

```
<'
extend sys {
    run() is also {
        var l_1 := {1;3;5;7;9};
        var l_2 := {1;9;7;3;5};
        var l_comp: bool;
        l_comp = l_1.is_a_permutation(l_2);
        print l_comp;
    };
};
'>
```

### Example 2

In the following example, the **keep** constraint causes the list named “l_2” have the same items the generator puts in the list named “l_1”. Since “l_1” will have the same number of items as “l_2”, there is an implicit constraint that “l_2” will be the same size at “l_1”. To constrain the size of the two lists, you can specify a **keep** constraint on the size of either “l_1” or “l_2”. Using a **keep soft** constraint to try to constrain the size of “l_2” is an error.

```
<'
extend sys {
    l_1: list of int;
    l_2: list of int;
    keep l_2.is_a_permutation(l_1);
};
'>
```

### See Also

— “keep” on page 270
— “has()” on page 579

## 19.4.11 is_empty()

### Purpose

Check if a list is empty

### Category

Pseudo-method

### Syntax

***list*.is_empty()**: bool

Syntax example:

```
var no_l: bool;
no_l = packets.is_empty();
```

### Parameters

| | |
|---|---|
| *list* | A list. |

### Description

Returns TRUE if ***list*** is empty, or FALSE if the list is not empty.

NOTE— You can use “***list*** is empty” as a synonym for “***list*.is_empty()**”.

Similarly, you can use “***list*** is not empty” to mean “**not(*list*.is_empty())**”.

### Example 1

In the following example, the first **print** action prints TRUE because the “int_lst” is initially empty. After an item is added, the second **print** action prints TRUE because the list is not empty.

```
<'
extend sys {
    int_list: list of int;
    run() is also {
        var emp: bool;
        emp = int_list.is_empty();
        print emp;
        int_list.add(3);
        emp = int_list is not empty;
        print emp;
    };
};
'>
```

### Example 2

The following gives the same result as the “ck_instr()” method in the previous example.

```
ck_instr() is {
    if int_list is not empty {
        print int_list;
    }
    else {
        outf("list is empty\n");
        return;
    };
};
```

### See Also

- “clear()” on page 557
- “exists()” on page 574
- “size()” on page 594

## 19.4.12 last()

### Purpose

Get the last item that satisfies a given condition

### Category

Pseudo-method

### Syntax

***list*.last(*exp*:** bool**)**: list-type

Syntax example:

```
var i_item: instr;
```

```
i_item = sys.instr_list.last(it.op1 > 15);
```

## Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

## Description

Returns the last item in the list that satisfies the boolean expression. If there is no such item, the default for the item's type is returned (see "e Data Types" on page 75).

For a list of scalars, a value of zero is returned if there is no such item. Since zero might be confused with a found value, it is safer to use ***list*.last_index()** for lists of scalars.

## Example 1

The first line below creates a list containing the integers 8, 3, 7, 3, and 4. The second line prints 4.

```
var l :list of int = {8;3;7;3;4};
print l.last(it < 5);
```

## Example 2

The **check that** line below checks that there is no packet in the "packets" list that contains an empty "cells" list.

```
<'
struct cell {
    data: list of byte;
};
struct packet {
    cells: list of cell;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        check that sys.packets.last(.cells is empty) == NULL;
    };
};
'>
```

## See Also

— "first()" on page 576
— "has()" on page 579
— "last_index()" on page 584
— "all()" on page 604

## 19.4.13 last_index()

### Purpose

Get the index of the last item that satisfies a given condition

### Category

Pseudo-method

### Syntax

***list*.last_index(*exp***: bool**)**: int

Syntax example:

```
var i_item: int;
i_item = instr_list.last_index(it.op1 > 15);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the index of the last item for which ***exp*** is TRUE, or returns UNDEF if there is no such item.

### Example 1

The first lilne below creates a list containing the integers 8, 3, 7, 3, and 4. The second line prints 3.

```
var l: list of int = {8;3;7;3;4};
print l.last_index(it == 3);
```

### Example 2

The code below checks that every packet in the "packets" list has a non-empty "cells" list: if the index of the last packet that has a non-empty "cells" list is one less than the size of the list, the check succeeds.

```
check that
    sys.packets.last_index(.cells is not empty) ==
    sys.packets.size() - 1;
```

### See Also

— "first_index()" on page 577
— "has()" on page 579
— "last()" on page 583
— "all_indices()" on page 605

## 19.4.14 max()

### Purpose

Get the item with the maximum value of a given expression

### Category

Pseudo-method

### Syntax

***list*.max(*exp***: int**)**: list-type

Syntax example:

```
var high_item: item_instance;
high_item = item_list.max(it.f_1 + it.f_2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the item for which the ***exp*** evaluates to the largest value. If more than one item results in the same maximum value, the item latest in the list is returned.

If the ***list*** is empty, an error is issued.

### Example

In the example below, the “high_item” variable gets the “instr” instance that has the largest value of the sum of “op1” and “op2”.

```
<'
struct instr {
    op1: int;
    op2: int;
};
extend sys {
    instr_list: list of instr;
    keep instr_list.size() > 5;
    post_generate() is also {
        var high_item: instr;
        high_item = instr_list.max(.op1 + .op2);
    };
};
'>
```

### See Also

— “has()” on page 579
— “max_index()” on page 587
— “max_value()” on page 588
— “min()” on page 589
— “all()” on page 604

### 19.4.15 max_index()

#### Purpose

Get the index of the item with the maximum value of a given expression

#### Category

Pseudo-method

#### Syntax

***list*.max_index(*exp***: int**)**: int

Syntax example:

```
var item_index: index;
item_index = sys.item_list.max_index(it.f_1 + it.f_2);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

#### Description

Returns the index of the item for which the ***exp*** evaluates to the largest value. If more than one item results in the same maximum value, the index of the item latest in the list is returned.

If the ***list*** is empty, and error is issued.

#### Example

In the example below, the “high_indx” variable gets the index of the “instr” instance that has the largest value of the sum of “op1” and “op2”.

```
<'
struct instr {
    op1: int;
    op2: int;
};
extend sys {
    instr_list: list of instr;
    keep instr_list.size() > 5;
    post_generate() is also {
        var high_indx: int;
        high_indx = instr_list.max_index(.op1 + .op2);
    };
};
'>
```

#### See Also

— “first_index()” on page 577

— "has()" on page 579
— "last_index()" on page 584
— "max()" on page 585
— "max_value()" on page 588
— "min_index()" on page 590
— "all_indices()" on page 605

## 19.4.16 max_value()

### Purpose

Return the maximum value found by evaluating a given expression for all items

### Category

Pseudo-method

### Syntax

***list*.max_value(*exp*:** int**):** (int | uint)

Syntax example:

```
var item_val: int;
item_val = sys.item_list.max_value(it.f_1 + it.f_2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the largest integer value found by evaluating the ***exp*** for every item in the list. If more than one item results in the same maximum value, the value of the expression for the item latest in the list is returned.

For lists of integer types, one of the following is returned if the ***list*** is empty:

| List Item Type | Value Returned by list.max_value() |
|---|---|
| signed integer | MIN_INT (see "Predefined Constants" on page 8) |
| unsigned integer | zero |
| long integer | error |

### Example 1

The example below prints the largest absolute value in the list of integers named "i_list".

```
<'
extend sys {
        i_list: list of int;
        post_generate() is also {
            print i_list.max_value(abs(it));
        };
    };
    '>
```

### Example 2

In the example below, the “high_val” variable gets the “instr” instance that has the largest value of the sum of “op1” and “op2”.

```
    <'
    struct instr {
        op1: int;
        op2: int;
    };
    extend sys {
        instr_list: list of instr;
        keep instr_list.size() < 10;
        post_generate() is also {
            var high_val: int;
            high_val = instr_list.max_value(.op1 + .op2);
            print high_val;
        };
    };
    '>
```

### See Also

- “has()” on page 579
- “max()” on page 585
- “max_index()” on page 587
- “min_value()” on page 591

## 19.4.17 min()

### Purpose

Get the item with the minimum value of a given expression

### Category

Pseudo-method

### Syntax

***list*.min(*exp*:** int**)**: list-type

Syntax example:

```
    var low_item: item_instance;
    low_item = sys.item_list.min(it.f_1 + it.f_2);
```

### Parameters

*list* A list.

*exp* An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number.

### Description

Returns the item for which the ***exp*** evaluates to the smallest value. If more than one item results in the same minimum value, the item latest in the list is returned.

### Example

In the example below, the "low_item" variable gets the "instr" instance that has the smallest value of the sum of "op1" and "op2".

```
<'
struct instr {
    op1: int;
    op2: int;
};
extend sys {
    instr_list: list of instr;
    keep instr_list.size() < 10;
    post_generate() is also {
        var low_item: instr;
        low_item = instr_list.min(.op1 + .op2);
    };
};
'>
```

### See Also

— "first()" on page 576
— "has()" on page 579
— "last()" on page 583
— "max()" on page 585
— "min_index()" on page 590
— "min_value()" on page 591
— "all()" on page 604

## 19.4.18 min_index()

### Purpose

Get the index of the item with the minimum value of a given expression

### Category

Pseudo-method

### Syntax

***list*.min_index(*exp*:** int**):** int

Syntax example:

```
var item_index: index;
item_index = sys.item_list.min_index(it.f_1 + it.f_2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Return the index of the item for which the specified ***exp*** gives the minimal value. If more than one item results in the same minimum value, the index of the item latest in the list is returned.

If the ***list*** is empty, an error is issued.

### Example

In the example below, the “low_indx” variable gets the index of the “instr” instance that has the smallest value of the sum of “op1” and “op2”.

```
<'
struct instr {
    op1: int;
    op2: int;
};
extend sys {
    instr_list: list of instr;
    keep instr_list.size() < 10;
    post_generate() is also {
        var low_indx: int;
        low_indx = instr_list.min_index(.op1 + .op2);
    };
};
'>
```

### See Also

— “first_index()” on page 577
— “has()” on page 579
— “last_index()” on page 584
— “max_index()” on page 587
— “min()” on page 589
— “min_value()” on page 591
— “all_indices()” on page 605

## 19.4.19 min_value()

### Purpose

Return the minimum value found by evaluating a given expression for all items

### Category

Pseudo-method

### Syntax

***list*.min_value(*exp*: int): (int | uint)**

Syntax example:

```
var item_val: int;
item_val = sys.item_list.min_value(it.f_1 + it.f_2);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the smallest integer value found by evaluating the ***exp*** for every item in the list. If more than one item results in the same minimum value, the value of the expression for the item latest in the list is returned.

For lists of integer types, one of the following is returned if the ***list*** is empty:

| List Item Type | Value Returned |
|---|---|
| signed integer | MAX_INT (see "Predefined Constants" on page 8) |
| unsigned integer | zero |
| long integer | error |

### Example

In the example below, the "low_val" variable gets the "instr" instance that has the smallest value of the sum of "op1" and "op2".

```
<'
struct instr {
    op1: int;
    op2: int;
};
extend sys {
    instr_list: list of instr;
    keep instr_list.size() < 10;
    post_generate() is also {
        var low_val: int;
        low_val = instr_list.min_value(.op1 + .op2);
        print low_val;
    };
};
'>
```

### See Also

- — "has()" on page 579
- — "max_value()" on page 588
- — "min()" on page 589
- — "min_index()" on page 590

## 19.4.20 reverse()

### Purpose

Reverse the order of a list

### Category

Pseudo-method

### Syntax

***list*****.reverse()**: list

Syntax example:

```
var s_list := {"A"; "B"; "C"; "D"};
var r_list := s_list.reverse();
```

### Parameters

*list* A list.

### Description

Returns a new list of all the items in ***list*** in reverse order.

### Example 1

In the following example the "r_packets" field gets a list that contains all the items in the "packets" list, but in reverse order.

```
<'
struct cell {
    data: list of byte;
};
struct packet {
    cells: list of cell;
};
extend sys {
    packets: list of packet;
    r_packets: list of packet;
    post_generate() is also {
        r_packets = sys.packets.reverse();
    };
};
'>
```

### Example 2

The following example prints 2, 1, 2, 4.

```
var i_list: list of int = {4; 2; 1; 2};
var r_list: list of int = i_list.reverse();
print r_list;
```

### See Also

— "sort()" on page 595
— "sort_by_field()" on page 596

## 19.4.21 size()

### Purpose

Return the size of a list

### Category

Pseudo-method

### Syntax

***list*.size()**: int

Syntax example:

```
print packets.size();
```

### Parameters

*list* A list.

### Description

Returns an integer equal to the number of items in the ***list***.

A common use for this method is in a **keep** constraint, to specify an exact size or a range of values for the list size. The default maximum list size for generated lists is 50, set by the **default_max_list_size** generation configuration option. Generated lists have a random size between 0 and that number. You can control the list size using a construct like "**keep *list*.size() == *n***", where ***n*** is an integer expression.

The ***list*[*n*]** index syntax is another way to specify an exact size of a list, when you use it in the list declaration, such as "p_list[***n***]: list of p".

See "List Size" on page 264 for more information about constraining the size of lists.

### Example 1

In the following example, the "lsz" variable gets the number of items in the list named "s_list".

```
<'
```

```
extend sys {
    s_list: list of string;
    keep s_list == {"Aa"; "Ba"; "Cc"};
    post_generate() is also {
        var lsz: int;
        lsz = sys.s_list.size();
        print lsz;
    };
};
'>
```

### Example 2

In the following example, a list of packets named p_list will be generated. A **keep** constraint is used to set the size of the list to exactly 10 packets.

```
<'
extend sys {
    p_list: list of packet;
    keep p_list.size() == 10;
};
'>
```

### See Also

— "resize()" on page 566
— "count()" on page 573
— "keep" on page 270
— "[ ]" on page 54

## 19.4.22 sort()

### Purpose

Sort a list

### Category

Pseudo-method

### Syntax

***list*.sort(*sort-exp*: exp)**: list

Syntax example:

```
var s_list: list of packet;
s_list = packets.sort(it.f_1 + it.f_2);
```

### Parameters

| | |
|---|---|
| *list* | A list of integers, strings, enumerated items, or boolean values to sort. |
| *sort-exp* | A scalar or nonscalar expression. The expression may contain references to fields or structs. The **it** variable can be used to refer to the current list item. |

### Description

Returns a new list of all the items in ***list***, sorted in increasing order of the values of the ***sort-exp***.

If the ***sort-exp*** is a scalar value, the list is sorted by value. If the ***sort-exp*** is a nonscalar, the list is sorted by address.

### Example 1

The following example prints 1, 2, 2, 4.

```
var sl: list of int = {4; 2; 1; 2};
print sl.sort(it);
```

### Example 2

In the following example, the “s_list” variable gets the items in the “packets” list, sorted in increasing value of the product of the “length” and “width” fields.

```
<'
struct packet {
    length: uint;
    width: uint;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var s_list: list of packet;
        s_list = packets.sort(.length * .width);
    };
};
'>
```

### See Also

— “reverse()” on page 593
— “sort_by_field()” on page 596

## 19.4.23 sort_by_field()

### Purpose

Sort a list of structs by a selected field

### Category

Pseudo-method

### Syntax

***struct-list*.sort_by_field(*field***: field-name**)**: list

Syntax example:

```
var s_list: list of packet;
s_list = sys.packets.sort_by_field(length);
```

### Parameters

| | |
|---|---|
| *struct-list* | A list of structs. |
| *field* | The name of a field of the list's struct type. Enter the name of the field only, with no preceding "." or "it.". |

### Description

Returns a new list of all the items in ***struct-list***, sorted in increasing order of their ***field*** values.

NOTE— The ***list*.sort()** pseudo-method returns the same value as the ***list*.sort_by_field()** pseudo-method, but ***list*.sort_by_field()** is more efficient.

### Example

In the following example, the "sf_list" variable gets the items in the "packets" list, sorted in increasing value of the "ptype" field (first "ATM", then "ETH", then "foreign").

```
<'
type pkt_type: [ATM, ETH, foreign];
struct packet {
    length: uint;
    width: uint;
    ptype: pkt_type;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var sf_list: list of packet;
        sf_list = packets.sort_by_field(ptype);
        print sf_list;
    };
};
'>
```

### See Also

- "reverse()" on page 593
- "sort()" on page 595

## 19.4.24 split()

### Purpose

Splits a list at each point where an expression is true

## Category

Pseudo-method

## Syntax

***list*.split(*split-exp*:** exp**)**: list**,** ...

Syntax example:

```
var sl_hold := s_list.split(it.f_1 == 16);
```

## Parameters

| | |
|---|---|
| *list* | A list of type struct-list-holder. |
| *split-exp* | An expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

## Description

Splits the items in ***list*** into separate lists, each containing consecutive items in ***list*** which evaluate to the same ***exp*** value.

Since ***e*** does not support lists of lists, this pseudo-method returns a list of type struct-list-holder. The struct-list-holder type is a struct with a single field, "value: list of ***any-struct***;". A struct-list-holder is a list of structs, with each struct containing a list of items of the original ***list*** type.

Each struct-list-holder in the returned list contains consecutive items from the ***list*** that have the same ***split-exp*** value.

NOTE— Fields used in the expression must be defined in the base type definition, not in **when** subtypes.

## Example 1

Suppose "packets" is a list that contains packet instances that have the following "length" values:

3; 5; 5; 7; 7; 7; 5;

The "packets**.split**(.length)" pseudo-method in the following creates a list of four lists by splitting the "packets" list at each point where the "length" value changes, that is, between 3 and 5, between 5 and 7, and between 7 and 5.

```
<'
struct packet {
    length: uint;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var sl_hold := packets.split(.length);
        print sl_hold[2].value;
        print sl_hold.value[4];
    };
};
```

```
'>
```

The struct-list-holder variable “sl_hold” then contains four lists:

3
5, 5
7, 7, 7
5

The “**print** sl_hold**[**2**].value**” action prints the third list, which is the one containing three items whose “length” values are 7.

The **print** sl_hold.**value[**4**]**” action prints the fifth item in the “sl_hold” list, which is the same as the fifth item in the “packets” list.

## Example 2

In the following example, the “length” field values used in Example 1 on page 598, are assigned to a list of seven “packet” structs. The ***list*.split()** pseudo-method is then called to split the list of packets on the “length” values. This creates the following four lists in the “sl_hold” struct list holder variable:

list sl_hold[1] : length value 3
list sl_hold[2] : length values 5, 5
list sl_hold[3] : length values 7, 7, 7
list sl_hold[4] : length value 5

The “sl_hold” list values are then printed in the **for** loop.

```
<'
struct packet {
    address: byte;
    length: uint;
};
extend sys {
    packets: list of packet;
    keep packets.size() == 7;
    post_generate() is also {
        packets[0].length = 3;
        packets[1].length = 5;
        packets[2].length = 5;
        packets[3].length = 7;
        packets[4].length = 7;
        packets[5].length = 7;
        packets[6].length = 5;
        var sl_hold: list of struct_list_holder;
        sl_hold = packets.split(.length);
        for i from 0 to sl_hold.size() - 1 do {
            print sl_hold[i].value;
        };
    };
};
'>
```

The output of the “**print** sl_hold**[**i**].value**” loop is shown below. The “length” field values for the “packet” items are in the column on the right.

```
  p[i].value =
item   type
0.     packet      112         3

  p[i].value =
item   type
0.     packet      32          5
1.     packet      78          5

  p[i].value =
item   type
0.     packet      172         7
1.     packet      172         7
2.     packet      25          7

  p[i].value =
item   type
0.     packet      70          5
```

### Example 3

The following splits the list in the preceding example at each point where the value of the expression ".length > 5" changes, that is, between 5 and 7 and between 7 and 5.

```
var sl_hold := sys.packets.split(.length > 5);
```

The struct-list-holder variable "sl_hold" then contains three lists:

3, 5, 5
7, 7, 7
5

### Example 4

To sort the list before you split it, you can use the following syntax.

```
var sl_hold := sys.packets.sort(.length).split(.length);
```

### See Also

— "get_indices()" on page 579
— "has()" on page 579
— "all()" on page 604

## 19.4.25 top()

### Purpose

Return the last item in a list

### Category

Pseudo-method

### Syntax

***list*.top()**: list-item

Syntax example:

```
var pk: packet;
pk = sys.packets.top();
```

### Parameters

*list* A list.

### Description

Returns the last item in the ***list*** without removing it from the list. If the list is empty, an error is issued.

### Example

The following example prints the contents of the last packet in the “packets” list.

```
print sys.packets.top();
```

### See Also

— “pop()” on page 563
— “top0()” on page 601

## 19.4.26 top0()

### Purpose

Return the first item of a list

### Category

Pseudo-method

### Syntax

***list*.top0()**: list-item

Syntax example:

```
var pk: packet;
pk = sys.packets.top0();
```

### Parameters

*list* A list.

### Description

Returns the first item in the ***list*** without removing it from the list. If the list is empty, an error is issued.

This pseudo-method can be used with **pop0()** to emulate queues.

### Example

The following example prints the contents of the first packet in the "packets" list.

```
print sys.packets.top0();
```

### See Also

— "pop0()" on page 564
— "top()" on page 600

## 19.4.27 unique()

### Purpose

Collapse consecutive items that have the same value into one item

### Category

Pseudo-method

### Syntax

***list*.unique(*select-exp*:** exp**)**: list

Syntax example:

```
var u_list: list of l_item;
u_list = sys.l_list.unique(it.f_1);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *select-exp* | An expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns a new list of all the distinct values in ***list***. In the new list, all consecutive occurrences of items for which the value of ***exp*** are the same are collapsed into one item.

### Example 1

In the following example, the ***list*.unique()** pseudo-method collapses the consecutive 5s and the consecutive 7s in "i_list" into a single 5 and a single seven. The example prints 3, 5, 7, 5.

```
var i_list := {3; 5; 5; 7; 7; 7; 5};
var pl: list of int;
pl = i_list.unique(it);
print pl;
```

### Example 2

Suppose the “packets” list contains seven packets with the following “length” field values: 3, 5, 5, 7, 7, 7, 5. The ***list*.unique()** pseudo-method collapses the consecutive packets with lengths of 5 into a single item, and collapses the consecutive items with lengths of 7 into a single item. The “pl” list gets four packets, with lengths of 3, 5, 7, and 5.

```
var pl: list of packet;
pl = packets.unique(.length);
```

### Example 3

In the following example, the ***list*.unique()** pseudo-method removes any packet items with repeated “length” values from the “packets” list before the list is sorted using the ***list*.sort()** pseudo-method.

```
<'
struct packet {
    length: uint (bits: 8);
    width: uint (bits: 8);
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var s_list: list of packet;
        s_list= packets.sort(.length).unique(.length);
        print s_list;
    };
};
'>
```

### See Also

— “count()” on page 573
— “size()” on page 594
— “sort()” on page 595
— “sort_by_field()” on page 596
— “all()” on page 604

## 19.5 Sublist Pseudo-Methods

This section describes the syntax for pseudo-methods that construct a new list from all the items in another list that satisfy specified conditions.

The pseudo-methods in this section are:

— “all()” on page 604
— “all_indices()” on page 605

### See Also

— “Pseudo-Methods Overview” on page 551
— “Using List Pseudo-Methods” on page 551
— “Type Conversion Between Scalars and Strings” on page 104

### 19.5.1 all()

#### Purpose

Get all items that satisfy a condition

#### Category

Pseudo-method

#### Syntax

***list*.all(*exp*: bool)**: list

Syntax example:

```
var l_2: list of packet;
l_2 = sys.packets.all(it.length > 64);
```

#### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current item, and the **index** variable can be used to refer to its index number. |

#### Description

Returns a list of all the items in ***list*** for which ***exp*** is TRUE. If no items satisfy the boolean expression, an empty list is returned.

#### Example 1

The following example prints 7, 9, 11, since those are the values in “l_list” that are greater than 5.

```
var l_list: list of int = {1; 3; 5; 7; 9; 11};
print l_list.all(it > 5);
```

#### Example 2

The following example creates a list named “pl” of all packets that have a “length” field value less than 5, and prints the “pl” list.

```
<'
type packet_type: [ETH, ATM];
struct packet {
    length: uint (bits: 4);
    ptype: packet_type;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var pl: list of packet;
        pl = packets.all(.length < 5);
        print pl;
    };
};
'>
```

### Example 3

The following creates a list named “pt” of all packets that have a “ptype” field value of “ETH”, and prints the “pt” list. This example uses the “**it is a *type***” syntax to specify which subtype of the packet struct to look for.

```
<'
type packet_type: [ETH, ATM];
struct packet {
    length: uint (bits: 4);
    ptype: packet_type;
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        var pt:= packets.all(it is a ETH packet);
        print pt;
    };
};
'>
```

### See Also

- “is [not] a” on page 67
- “first()” on page 576
- “has()” on page 579
- “last()” on page 583
- “max()” on page 585
- “min()” on page 589
- “all_indices()” on page 605

## 19.5.2 all_indices()

### Purpose

Get indexes of all items that satisfy a condition

### Category

Pseudo-method

### Syntax

***list*.all_indices(*exp*:** bool**)**: list of int

Syntax example:

```
var l_2: list of int;
l_2 = sys.packets.all_indices(it.length > 5);
```

### Parameters

*list* A list.

*exp* A boolean expression.

### Description

Returns a list of all indexes of items in ***list*** for which ***exp*** is TRUE. If no items satisfy the boolean expression, an empty list is returned.

### Example 1

The following example creates a list name “tl” that contains the index numbers of all the “instr” instances in the list named “i_list” which have “op1” field values greater than 63.

```
<'
type op_code: [ADD, SUB, MLT];
struct instr {
    op1: int (bits: 8);
    op2: int;
    opcode: op_code;
};
extend sys {
    i_list: list of instr;
    post_generate() is also {
        var tl: list of int;
        tl = i_list.all_indices(it.op1 > 63);
        print tl;
    };
};
'>
```

### Results

```
tl =
0.      0
1.      1
2.      2
3.      3
4.      4
5.      8
6.     12
7.     28
8.     29
9.     30
10.    31
11.    33
```

### Example 2

In the following example, the ***list*.all_indices()** pseudo-method is used to create a list named “pl” of the indexes of the “packets” list items that are of subtype “small packet”.

```
<'
type pkt_type: [small, medium, large];
struct packet {
    address: byte;
```

```
        ptype: pkt_type;
    };
    extend sys {
        packets: list of packet;
        keep packets.size() == 10;
        run() is also {
            print packets;
            var pl: list of int;
            pl = packets.all_indices(it is a small packet);
            print pl;
        };
    };
    '>
```

### Results

```
    packets =
    item   type        address     ptype

    0.     packet      97          medium
    1.     packet      66          large
    2.     packet      10          large
    3.     packet      180         medium
    4.     packet      235         small
    5.     packet      196         small
    6.     packet      85          large
    7.     packet      163         small
    8.     packet      62          large
    9.     packet      86          large
       pl =
    0.      4
    1.      5
    2.      7
```

### Example 3

Using **all_indices()** on an empty list produces another empty list. Trying to use this result in a **gen keeping** constraint can cause a generation contradiction error. To avoid this, you can use a check like “**if** !test_ix**.is_empty**()” in the following example.

```
    <'
    struct st_eng {
        v_list: list of uint (bits: 7);
        ameth(): list of int is {
            var test_ix: list of int;
            test_ix = v_list.all_indices(it > 100);
            var s_index: uint;
            if !test_ix.is_empty() {
                gen s_index keeping {it in test_ix};
                return test_ix;
            };
        };
    };
    extend sys {
        st_i: st_eng;
        s_list: list of int;
        run() is also {
            s_list = st_i.ameth();
            print s_list;
        };
    };
    '>
```

### Results

```
    s_list =
    0.      9
    1.     20
    2.     21
    3.     37
    4.     39
    5.     44
```

### See Also

— “field” on page 575
— “first_index()” on page 577
— “has()” on page 579
— “last_index()” on page 584
— “max_index()” on page 587
— “min_index()” on page 590
— “all()” on page 604

## 19.6 Math and Logic Pseudo-Methods

This section describes the syntax for pseudo-methods that perform arithmetic or logical operations to compute a value using all items in a list.

The pseudo-methods in this section are:

— “and_all()” on page 608
— “average()” on page 609
— “or_all()” on page 610
— “product()” on page 611
— “sum()” on page 612

### See Also

— “Pseudo-Methods Overview” on page 551
— “Using List Pseudo-Methods” on page 551

### 19.6.1 and_all()

#### Purpose

Compute the logical AND of all items

#### Category

Pseudo-method

### Syntax

***list*.and_all(*exp*:** bool**)**: bool

Syntax example:

```
var bool_val: bool;
bool_val = m_list.and_all(it >= 1);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns a TRUE if all values of the ***exp*** are true, or returns FALSE if the ***exp*** is false for any item in the ***list***.

### Example

The following line prints TRUE if the "length" field value of all items in the "packets" list is greater than 63. If any packet has a length less than or equal to 63, it prints FALSE.

```
print sys.packets.and_all(it.length > 63);
```

### See Also

— "average()" on page 609
— "or_all()" on page 610
— "product()" on page 611
— "sum()" on page 612

## 19.6.2 average()

### Purpose

Compute the average of an expression for all items

### Category

Pseudo-method

### Syntax

***list*.average(*exp*:** int**)**: int

Syntax example:

```
var list_ave: int;
list_ave = sys.item_list.average(it.f_1 * it.f_2);
```

## Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

## Description

Returns the integer average of the ***exp*** computed for all the items in the ***list***. Returns UNDEF if the list is empty.

## Example 1

The following example prints 6 ((3 + 5 + 10)/3).

```
var a_list := {3; 5; 10};
print a_list.average(it);
```

## Example 2

The following example prints the average value of the “length” fields for all the items in the “packets” list.

```
<'
struct packet {
    length: uint (bits: 4);
    width: uint (bits: 4);
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        print packets.average(it.length);
    };
};
'>
```

## See Also

— “and_all()” on page 608
— “or_all()” on page 610
— “product()” on page 611
— “sum()” on page 612

## 19.6.3 or_all()

## Purpose

Compute the logical OR of all items

## Category

Pseudo-method

### Syntax

***list*.or_all(*exp*:** bool**)**: bool

Syntax example:

```
var bool_val: bool;
bool_val = m_list.or_all(it >= 100);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | A boolean expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns a TRUE if any value of the ***exp*** is true, or returns FALSE if the ***exp*** is false for every item in the list. Returns FALSE if the list is empty.

### Example

The following code prints TRUE if the "length" field value of any item in the "packets" list is greater than 150. If no packet has a length greater than 150, it prints FALSE.

```
<'
struct packet {
    length: uint (bits: 4);
    width: uint (bits: 4);
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        print packets.or_all(it.length > 150);
    };
};
'>
```

### See Also

— "and_all()" on page 608
— "average()" on page 609
— "product()" on page 611
— "sum()" on page 612

## 19.6.4 product()

### Purpose

Compute the product of an expression for all items

### Category

Pseudo-method

### Syntax

***list*.product(*exp*:** int**)**: int

Syntax example:

```
var list_prod: int;
list_prod = sys.item_list.product(it.f_1);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the integer product of the ***exp*** computed over all the items in the ***list***. Returns 1 if the list is empty.

### Example 1

The following example prints 150 (3 * 5 * 10).

```
var p_list := {3; 5; 10};
print p_list.product(it);
```

### Example 2

The following example prints the product of the “mlt” fields in all the items in the “packets” list.

```
<'
struct packet {
    mlt: uint (bits: 3);
    keep mlt > 0;
};
extend sys {
    packets[5]: list of packet;
    post_generate() is also {
        print packets.product(it.mlt);
    };
};
'>
```

### See Also

— “and_all()” on page 608
— “average()” on page 609
— “or_all()” on page 610
— “sum()” on page 612

## 19.6.5 sum()

### Purpose

Compute the sum of all items

### Category

Pseudo-method

### Syntax

***list*.sum(*exp*:** int**)**: int

Syntax example:

```
var op_sum: int;
op_sum = sys.instr_list.sum(.op1);
```

### Parameters

| | |
|---|---|
| *list* | A list. |
| *exp* | An integer expression. The **it** variable can be used to refer to the current list item, and the **index** variable can be used to refer to its index number. |

### Description

Returns the integer sum of the ***exp*** computed over all the items in the ***list***. Returns 0 if the list is empty.

The following example prints 18 (3 + 5 + 10).

```
var s_list := {3; 5; 10};
print s_list.sum(it);
```

### Example

The following example prints the sum of the “length” field values for all the items in the “packets” list.

```
<'
struct packet {
    length: uint (bits: 4);
    keep length in [1..5];
    width: uint (bits: 4);
    keep width in [1..5];
};
extend sys {
    packets[5]: list of packet;
    post_generate() is also {
        print packets.sum(it.length);
    };
};
'>
```

### See Also

— “and_all()” on page 608
— “average()” on page 609
— “or_all()” on page 610
— “product()” on page 611

## 19.7 List CRC Pseudo-Methods

This section describes the syntax for pseudo-methods that perform CRC (cyclic redundancy check) functions on lists.

The pseudo-methods in this section are:

— "crc_8()" on page 614
— "crc_32()" on page 615
— "crc_32_flip()" on page 617

### See Also

— "Pseudo-Methods Overview" on page 551
— "Using List Pseudo-Methods" on page 551

### 19.7.1 crc_8()

#### Purpose

Compute the CRC8 of a list of bits or a list of bytes

#### Category

Pseudo-method

#### Syntax

***list*.crc_8(*from-byte***: int**,** ***num-bytes***: int**)**: int

Syntax example:

```
print b_data.crc_8(2,4);
```

#### Parameters

| | |
|---|---|
| *list* | A list of bits or bytes. |
| *from-byte* | The index number of the starting byte. |
| *num-bytes* | The number of bytes to use. |

#### Description

Reads the ***list*** byte-by-byte and returns the integer value of the CRC8 function of a list of bits or bytes. Only the least significant byte (LSB) is used in the result.

The CRC is computed starting with the ***from-byte***, for ***num-bytes***. If ***from-byte*** or ***from-byte+num-bytes*** is not in the range of the list, an error is issued.

NOTE— The algorithm for computing CRC8 is specific for the ATM HEC (Header Error Control) computation. The code used for HEC is a cyclic code with the following generating polynomial:

x**8 + x**2 + x + 1

### Example

In the example below, the “e_crc” variable gets the CRC8 of the bytes 2, 3, 4, and 5 in the list named “b_data”.

```
<'
extend sys {
     post_generate() is also {
     var b_data: list of byte =
        {0xff;0xaa;0xdd;0xee;0xbb;0xcc};
     print  b_data.crc_8(2,4);
    };
};
'>
```

### Results

```
b_data.crc_8(2,4)  = 0x63
```

### See Also

- “crc_32()” on page 615
- “crc_32_flip()” on page 617
- “product()” on page 611
- “sum()” on page 612
- “pack()” on page 516
- “unpack()” on page 521

## 19.7.2 crc_32()

### Purpose

Compute the CRC32 of a list of bits or a list of bytes

### Category

Pseudo-method

### Syntax

***list*.crc_32(*from-byte*:** int**, *num-bytes*:** int**):** int

Syntax example:

```
print b_data.crc_32(2,4);
```

## Parameters

| | |
|---|---|
| *list* | A list of bits or bytes. |
| *from-byte* | The index number of the starting byte. |
| *num-bytes* | The number of bytes to use. |

## Description

Reads the ***list*** byte-by-byte and returns the integer value of the CRC32 function of a list of bits or bytes. Only the least significant word is used in the result.

The CRC is computed starting with the ***from-byte***, for ***num-bytes***. If ***from-byte*** or ***from-byte+num-bytes*** is not in the range of the list, an error is issued.

NOTE— The algorithm for computing CRC32 generates a 32-bit CRC that is used for messages up to 64 kilobytes in length. Such a CRC can detect 99.999999977% of all errors. The generator polynomial for the 32-bit CRC used for both Ethernet and token ring is:

x**32 + x**26 + x**23 + x**22 + x**16 + x**12 + x**11 + x**10 + x**8 + x**7 + x**5 + x**4 +x**2 + x + 1

## Example 1

In the example below, the "b_data" variable gets the packed "packet" struct instance as a list of bytes, and the CRC32 of bytes 2, 3, 4, and 5 of "b_data" is printed.

```
<'
struct packet {
    %byte_1: byte;
    %byte_2: byte;
    %byte_3: byte;
    %byte_4: byte;
    %byte_5: byte;
    %byte_6: byte;
};
extend sys {
    packet;
    post_generate() is also {
        var b_data: list of byte = pack(NULL, me.packet);
        print b_data.crc_32(2,4);
    };
};
'>
```

## Example 2

In the example below, the CRC32 value is calculated for the data field value. The "is_good_crc()" method checks the value and returns TRUE if it is good, FALSE if it is bad.

```
<'
struct packet {
    %data: list of byte;
    !%crc: uint;
    packed: list of byte;
    post_generate() is also {
        crc = (data.crc_32(0, data.size()) ^ 0xffff_ffff);
        packed = pack(packing.low, me);
    };
    is_good_crc(): bool is {
         result = (packed.crc_32(0, packed.size()) ==
             0xffff_ffff);
    };
};
extend sys {
    packets: list of packet;
    post_generate() is also {
        for each in packets {
            outf("frame %d ", index);
            print it.is_good_crc();
        };
    };
};
'>
```

### See Also

— “crc_8()” on page 614
— “crc_32_flip()” on page 617
— “product()” on page 611
— “sum()” on page 612
— “pack()” on page 516
— “unpack()” on page 521

## 19.7.3 crc_32_flip()

### Purpose

Compute the CRC32 of a list of bits or a list of bytes, flipping the bits

### Category

Pseudo-method

### Syntax

***list*.crc_32_flip(*from-byte*:** int**, *num-bytes*:** int**):** int

Syntax example:

```
    print b_data.crc_32_flip(2,4);
```

### Parameters

| | |
|---|---|
| *list* | A list of bits or bytes. |
| *from-byte* | The index number of the starting byte. |
| *num-bytes* | The number of bytes to use. |

### Description

Reads the ***list*** byte-by-byte and returns the integer value of the CRC32 function of a list of bits or bytes, with the bits flipped. Only the least significant word is used in the result.

The CRC is computed starting with the ***from-byte***, for ***num-bytes***. If ***from-byte*** or ***from-byte+num-bytes*** is not in the range of the list, an error is issued.

The bits are flipped as follows:

1) The bits inside each byte of the input are flipped.
2) The bits in the result are flipped.

### Example

In the example below, the “tc_crc” variable gets the CRC32 of the bytes 2, 3, 4, and 5 in the list named “b_data”, with the bits flipped.

```
<'
struct packet {
    %byte_1: byte;
    %byte_2: byte;
    %byte_3: byte;
    %byte_4: byte;
    %byte_5: byte;
    %byte_6: byte;
};
extend sys {
    packet;
    post_generate() is also {
        var b_data: list of byte = pack(NULL, me.packet);
        print b_data.crc_32_flip(2,4);
    };
};
'>
```

### See Also

— “crc_8()” on page 614
— “crc_32()” on page 615
— “product()” on page 611
— “sum()” on page 612
— “pack()” on page 516
— “unpack()” on page 521

## 19.8 Keyed List Pseudo-Methods

This section describes the syntax for pseudo-methods that can be used only on keyed lists.

Keyed lists are list in which each item has a key associated with it. For a list of structs, the key typically is the name of a particular field in each struct. Each unique value for that field may be used as a key. For a list of scalars, the key can be the **it** variable referring to each item.

While creating a keyed list, you must ensure that the key has a unique value for each item.

Keyed lists can be searched quickly, by searching on a key value.

This section contains descriptions of pseudo-methods that can only be used for keyed lists. Using one of these methods on a regular list results in an error.

The pseudo-methods in this section are:

— "key()" on page 619
— "key_index()" on page 622
— "key_exists()" on page 623

### See Also

— "Pseudo-Methods Overview" on page 551
— "Using List Pseudo-Methods" on page 551

## 19.8.1 key()

### Purpose

Get the item that has a particular key

### Category

Pseudo-method

### Syntax

***list*.key(*key-exp*: exp): list-item**

Syntax example:

```
var loc_list_item: location;
var i_key: uint;
i_key = 5;
loc_list_item = locations.key(i_key);
```

### Parameters

| | |
|---|---|
| *list* | A keyed list. |
| *key-exp* | The key of the item that is to be returned. |

### Description

Returns the list item that has the specified key, or NULL if no item with that key exists in the list.

For a list of scalars, a value of zero is returned if there is no such item. Since zero might be confused with a found value, it is not advisable to use zero as a key for scalar lists.

### Example 1

The following example uses a list of integers for which the key is the item itself. This example prints 5.

```
var l_list: list(key: it) of int = {5; 4; 3; 2; 1};
print l_list.key(5);
```

### Example 2

In the following example, the “mklist()” method generates a list of 10 “location” instances with even numbered address from 2 to 20. The “locations” list is a list of “location” instances with “address” as its key. The “l_item” variable gets the “locations” list item that has an “address” value of 6. If there is no item in the list with an address of 6, the locations**.key()** pseudo-method returns NULL.

```
<'
struct location {
    address: int;
    value: int;
};
struct l_s {
    mklist() is {
        var l: location;
        for i from 1 to 10 do {
            gen l keeping {it.address == 2*i};
            sys.locations.add(l);
        };
    };
};
extend sys {
    l_s;
    !locations: list(key: address) of location;
    run() is also {
        l_s.mklist();
        print locations;
        var l_item: location;
        l_item = locations.key(6);
        print l_item;
    };
};
'>
```

### Results

```
locations =
item   type        address     value

0.     location    2           -396796955
1.     location    4           1796592623
2.     location    6           2081332301
3.     location    8           -15822625*
4.     location    10          116159091
5.     location    12          -15052943*
6.     location    14          1128419469
7.     location    16          -20275240*
8.     location    18          -508036604
9.     location    20          116597347
  l_item = location-@0: location
```

```
         ---------------------------------------------@kl_2
    0    address:                       6
    1    value:                         2081332301
```

### Example 3

The following example shows how to use a keyed list on the lefthand side of assignment. The “mklist()” method generates a list of 10 “location” instances with even-numbered address values from 2 to 20.

The “locations” list is a list of “location” instances with “address” as its key. The “l_item” variable is a location instance which is generated with a constraint to keep its value equal to 100000. That location instance’s value field is then assigned to the locations list item that has the address key value of 6.

```
<'
struct location {
    address: int;
    value: int;
};
struct l_s {
    mklist() is {
        var l: location;
        for i from 1 to 10 do {
            gen l keeping {it.address == 2*i};
            sys.locations.add(l);
        };
    };
};
extend sys {
    l_s;
    !locations: list(key: address) of location;
    run() is also {
        l_s.mklist();
        var l_item: location;
        gen l_item keeping {it.value == 100000};
        print l_item;
        locations.key(6).value = l_item.value;
        print locations.key(6);
    };
};
'>
```

### Results

```
l_item = location-@0: location
       ---------------------------------------------@kl_assign_3
0    address:                     -513087844
1    value:                       100000
  locations.key(6) = location-@1: location
       ---------------------------------------------@kl_assign_3
0    address:                       6
1    value:                         100000
```

### See Also

— “key_index()” on page 622
— “key_exists()” on page 623

### 19.8.2 key_index()

#### Purpose

Get the index of an item that has a particular key

#### Category

Pseudo-method

#### Syntax

***list*****.key_index(*****key-exp***: exp**)**: int

Syntax example:

```
var loc_list_ix: int;
loc_list_ix = locations.key_index(i);
```

#### Parameters

| | |
|---|---|
| *list* | A keyed list. |
| *key-exp* | The key of the item for which the index is to be returned. |

#### Description

Returns the integer index of the item that has the specified key, or returns UNDEF if no item with that key exists in the list.

#### Example 1

The following example uses a list of integers for which the key is the item itself. This example prints 1, since that is the index of the list item with a value of 2.

```
var l_list: list(key: it) of int = {1; 2; 3; 4; 5};
print l_list.key_index(2);
```

#### Example 2

The locations**.key_index()** pseudo-method in the following gets the index of the item in the “locations” list that has an address of 9, if any item in the list has that address.

```
<'
struct location {
    address: int;
    value: int;
};
extend sys {
    !locations: list(key: address) of location;
    post_generate() is also {
        var l_ix: int;
        l_ix = locations.key_index(9);
        if l_ix != UNDEF {print locations[l_ix].value}
            else {outf("key_index %d does not exist\n", 9)};
        };
    };
    '>
```

### See Also

— "key()" on page 619
— "key_exists()" on page 623

## 19.8.3 key_exists()

### Purpose

Check that a particular key is in a list

### Category

Pseudo-method

### Syntax

***list*.key_exists(*key-exp*:** exp**)**: bool

Syntax example:

```
    var loc_list_k: bool;
    var i:= 5;
    loc_list_k = locations.key_exists(i);
```

### Parameters

| | |
|---|---|
| *list* | A keyed list. |
| *key* | The key that is to be searched for. |

### Description

Returns TRUE if the key exists in the list, or FALSE if it does not.

### Example 1

The following example uses a list of integers for which the key is the item itself. The first **print** action prints TRUE, since 2 exists in the list. The second **print** action prints FALSE, since 7 does not exist in the list.

```
    var l_list: list(key: it) of int = {1; 2; 3; 4; 5};
    print l_list.key_exists(2);
    print l_list.key_exists(7);
```

### Example 2

The locations**.key_exists()** pseudo-method in the following example returns TRUE to "k" if there is an item in the "locations" list that has a key value of 30, or FALSE if there is no such item.

```
    <'
```

```
struct location {
    address: int;
    value: int;
};
extend sys {
    !locations: list(key: address) of location;
    post_generate() is also {
        var k: bool;
        k = locations.key_exists(30);
        if k {outf("key %d exists\n", 30)}
            else {outf("key %d does not exist\n", 30)};
    };
};
'>
```

### See Also

— "key()" on page 619
— "key_index()" on page 622

## 19.9 Restrictions on Keyed Lists

— The following pseudo-methods cannot be used on keyed lists:

- *list*.resize()
- *list*.apply()
- *list.field*

— Keyed lists and regular (unkeyed) lists are different types. Assignment is not allowed between a keyed list and a regular list.
— Keyed lists cannot be generated. Trying to generate a keyed list results in an error. Therefore, keyed lists must be defined with the do-not-generate sign, an exclamation mark, as in the example below.
— Some operations are less efficient for keyed lists than for unkeyed lists, because after they change the list they must also update the keys. The following operations are not recommended on keyed lists:

- *list*.insert()
- *list*.delete()
- slice assignment
- *list*.reverse()

— Prior to using ***list*.insert()** or ***list*.delete**, you can make the operation more efficient by using one of the following pseudo-methods to find the desired index or item in a keyed list:

- *list*.first()
- *list*.first_index()
- *list*.has()

For example, the recommended way to delete an item from a keyed list is to check for the existence of the key first as in the following:

```
!locations: list (key: address) of location;
```

```
if (locations.key_exists(38)) then {
    locations.delete(location.key_index(38));
};
```

The above example searches for the key in the fastest way, and it updates the keyed list only if the key exists.

## See Also

— “Keyed List Pseudo-Methods” on page 618
— “Pseudo-Methods Overview” on page 551
— “Using List Pseudo-Methods” on page 551

# 20 Preprocessor Directives

This chapter contains the following sections:

— "#ifdef, #ifndef" on page 627. You use these preprocessor directives used to control ***e*** processing. The preprocessor directives check for the existence of a #**define** for a given name:

- **#ifdef** directive: if a given name is defined, use the attached code, otherwise, use different code
- **#ifndef**: if a given name is not defined, use the attached code, otherwise, use different code

The **#ifdef** and **#ifndef** directives can be used as statements, struct members, or actions.

— "#define" on page 630, which defines a name macro, also called a replacement macro.

— "#undef" on page 632, which removes the definition of a name macro.

### See Also

— Chapter 13, "Macros"

— Chapter 21, "Importing e Files"

## 20.1 #ifdef, #ifndef

### Purpose

Define a preprocessor directive

### Category

Statement, struct member, action

### Syntax

**#if[n]def** [\`]***name*** **then** {***e-code***}
  [**#else** {***e-code***}]

Syntax example:

```
#ifdef MEM_LG then {
    import mml.e;
};
```

NOTE— The import statement in the syntax example above must be on a line by itself. The syntax "**#ifdef** MEM_LG **then {import** mml.e**};**", where the **import** statement is part of the **#ifdef** statement line, will not work.

## Parameters

| | |
|---|---|
| *name* | Without a backtick, a name defined in a **define** statement. For information about **define**, see Chapter 13, "Macros".<br><br>With a backtick (`name), a name defined with a Verilog `define directive, or in a **define** statement where the macro is defined in Verilog style. |
| *e-code* | ***e*** code to be included based on whether the name macro has been defined.<br><br>• For an **#ifdef** or **#ifndef** statement, only ***e*** statements can be used in ***e-code***.<br>• For an **#ifdef** or **#ifndef** struct member, only struct members can be used in ***e-code***.<br>• For an **#ifdef** or **#ifndef** action, only actions can be used in ***e-code***. |

## Description

The **#ifdef** and **#ifndef** preprocessor directives are used together with **define** name macros to cause the ***e*** parser to process particular code or ignore it, depending on whether a given name macro has been defined.

— The **#ifdef** syntax checks whether the name macro has been defined and, if it has, includes the code following **then**.
— The **#ifndef** syntax checks whether the name macro has been defined and if it has not, includes the code following **then**.

The optional **#else** syntax provides an alternative statement when the **#ifdef** or **#ifndef** is not true. For **#ifdef**, if the name macro has not been defined, the **#else** code is included. For **#ifndef**, if the name macro has been defined, the **#else** text is included.

NOTE— Except when it is within an **#else** block, the **#ifdef** or **#ifndef** keyword must be the first keyword on the line.

## Example 1

In this example, **#ifdef** is used as statements. The module named "t_1.e" contains the statement "define test_C". Neither "test_A" nor "test_B" is defined anywhere. Thus, only the "t_4.e" module is imported by the **#ifdef** statements.

```
<'
import t_1.e;// defines test_C;
#ifdef test_A then {
    import t_2;
}
#else {
    #ifdef test_B then {
        import t_3;
    };
    #ifdef test_C then {
        import t_4;
    };
};
'>
```

### Example 2

In this example, **#ifdef** is used as a struct member. The module contains the statement “define test_C”. Neither “test_A” nor “test_B” is defined anywhere. Thus, only the “keep t_data in [300..399]” constraint is applied to the generator after the **#ifdef** statements have been processed.

```
<'
define test_C;
struct exa_str {
    t_data: uint;
    #ifdef test_A then {
        keep t_data in [100..199];
    }
    #else {
        #ifdef test_B then {
            keep t_data in [200..299];
        };
        #ifdef test_C then {
            keep t_data in [300..399];
        };
    };
};
'>
```

### Example 3

In this example, **#ifdef** is used as an action. The module contains the statement “define test_C”. Neither “test_A” nor “test_B” is defined anywhere. Thus, only the “gen t_data keeping it in [300..399]” action is applied by the **#ifdef** statements.

```
<'
define test_C;
struct t_str {
    !t_data: int;
    t_meth() is {
        #ifdef test_A then {
            gen t_data keeping {it in [100..199]};
        }
        #else {
            #ifdef test_B then {
                gen t_data keeping {it in [200..299]};
            };
            #ifdef test_C then {
                gen t_data keeping {it in [300..399]};
            };
        };
    };
};
'>
```

### See Also

— Chapter 13, “Macros”
— Chapter 21, “Importing e Files”

## 20.2 #define

### Purpose

Define a name macro

### Category

Statement

### Syntax

[#]**define** [`]***name*** [***replacement***]

Syntax example:

```
define PLIST_SIZE 100;
```

### Parameters

| | |
|---|---|
| *name* | Any ***e*** name.<br><br>This is used with no replacement parameter for conditional code processing. An #ifdef preprocessor directive later in the ***e*** code that has the name as its argument evaluates to TRUE. See "#ifdef, #ifndef" on page 627 for more information.<br><br>When a replacement is given, the parser substitutes the replacement for the macro name everywhere name appears, except inside strings.<br><br>The name can be preceded with a backtick, `. This makes the name look like a Verilog `define name, but it is treated the same as a name without a backtick.<br><br>The name is case sensitive: "LEN" is not the same as "len". |
| *replacement* | Any syntactic element, for example, an expression or an HDL variable. This replaces the name wherever the name appears in the ***e*** code that follows the define statement. |

### Notes

- Be sure to use parentheses around the ***replacement*** when they are needed to ensure proper associativity. For example, the effect of:

  ```
  define LX 2*len+m;
  ```

  Is different from the effect of:

  ```
  define LX 2*(len+m);
  ```
- In an expression like "lenx = LX", the first case becomes "lenx = 2*len + m", while the second case becomes "lenx = 2*len + 2*m".
- The leading "#" is shown as optional in the syntax, in order to support the **define** statement syntax from previous ***e*** releases, in which the # does not appear.

### Description

With a ***replacement***, defines a macro that replaces the ***name*** wherever it occurs in the ***e*** code. With no ***replacement***, specifies a name that can be used in **#ifdef** preprocessor directives for conditional code. A subsequent evaluation of an **#ifdef** that has the ***name*** as its argument returns TRUE.

### Notes

— A **#define** statement must be on a line by itself. The following is illegal:

```
#define BIGMEM; import mod1.e;    // Illegal syntax
```

The correct way to write the above is:

```
#define BIGMEM;

import mod1.e;
```

— A **define** statement only applies to ***e*** code that is loaded after the **define**.

— The ***replacement*** must not contain the ***name***. A statement like the following causes a runtime error:

```
define bus_width top.bus_width;    // Run-time error
```

This causes the parser to recursively replace “bus_width” with “top.bus_width”. That is, “bus_width” would become “top.bus_width”, as desired, but then “top.bus_width” would become “top.top.bus_width”, and so on.

### Example 1

The following are name macro definitions:

```
#define     OFFSET 5;
#define     FIRST (OFFSET + 1);
#define     SECOND (FIRST + 1);
#define     MULTIPLY_I_J i * j;
#define     LG_CASE;
#define     `bus_width 64;
#define     bus_width_1 'top.bus_wire';
#define     bus_width_2 top.bus_wire;
```

### Example 2

To use a **#define** macro, refer to the ***name***. Given the definitions above, you could use them as in the following:

```
struct example {
    test_defines() is {
        var i: int;
        var j: int;
        print OFFSET, FIRST, SECOND;  // Prints 5, 6, 7
        i = 5;
        j = 6;
        print MULTIPLY_I_J + 3;       // Prints 33 (5 * 6 + 3)
        #ifdef LG_CASE then {
            i = j * 2;
            print i;                  // Prints 12 (6 * 2)
        }
        #else {
            out("LG_CASE is not defined");
        };
```

```
            j = `bus_width * 2;
            print j;                    // Prints 128 (64 * 2)
        };
    };
```

### See Also

- "#undef" on page 632
- "#define" on page 630
- "define as computed" on page 436

## 20.3 #undef

### Purpose

Undefine a name macro

### Category

Statement

### Syntax

**undef** [`]***name***

Syntax example:

```
    #undef PLIST_SIZE;
```

### Parameters

| | |
|---|---|
| *name* | Any ***e*** name.<br><br>This is used with no replacement parameter for conditional code processing. An #ifdef preprocessor directive later in the ***e*** code that has the name as its argument evaluates to TRUE. See "#ifdef, #ifndef" on page 627 for more information.<br><br>When a replacement is given, the parser substitutes the replacement for the macro name everywhere name appears, except inside strings.<br><br>The name can be preceded with a backtick, `. This makes the name look like a Verilog `define name, but it is treated the same as a name without a backtick.<br><br>The name is case sensitive: "LEN" is not the same as "len". |

### Description

Removes a name macro that was defined using the **#define** statement. The **#undef** statement can appear anywhere in the ***e*** code. The name macro is not recognized from the point where the **#undef** statement appears onward. The effect is propagated to all files that are loaded after the **#undef** statement is encountered.

## Notes

- If the undefined name macro was not previously defined, **#undef** has no effect – it is not an error.
- A name macro that is undefined in a compiled ***e*** module is not accessible to the C interface at any time.
- A name macro that has been undefined can be redefined later, with any value. The last value is accessible to the C interface.

## Example

Say you have two ***e*** files, my_design.e and external_code.e, and the following appears in the my_design.e module:

```
<'
struct semaphore {
    // Contents of the user-defined semaphore struct
};
'>
```

The following appears in the external_code.e module:

```
<'
extend sys{
    event clk is rise('~/top/clk')@sim;
    sem: semaphore; // Uses the built-in e semaphore struct
    increment()@clk is {
        sem.up()
    };
};
'>
```

In the external_code.e file, the built-in semaphore struct is used. In order to be able to use the built-in semaphore struct, you can put the following in a top file, which imports both my_design.e and external_code.e. This first defines a name macro that replaces "semaphore" with "my_semaphore, and then, after the my_design.e module is loaded, undefines semaphore so that the built-in semaphore struct is used from that point on:

```
<'
#define semaphore my_semaphore;
import my_design.e;
#undef semaphore;
import external_code.e;
'>
```

## See Also

- "#define" on page 630
- "define as" on page 429
- "define as computed" on page 436

# 21 Importing *e* Files

## 21.1 Overview

Imports (**import** statement, **verilog import** statement) load a given ***e*** file or Verilog file This chapter describes the **import** statement.

### See Also

— "File Structure" on page 3
— "verilog import" on page 799

## 21.2 import

### Purpose

Load other ***e*** modules

### Category

Statement

### Syntax

**import** ***file-name***, ... | **(** ***file-name*, ... )**

Syntax example:

```
import test_drv.e;
```

### Parameters

| | |
|---|---|
| ***file-name***, ... | The names of files, separated by commas, that contain ***e*** modules to be imported. If no extension is given for a file name, an ".e" extension is assumed.<br><br>The (***file-name***, ...) syntax is for cyclic importing, in which one module references a field in a second module, and the second module references a field in the first module.<br><br>File names can contain references to environment variables using the UNIX notation "$***name***" or "${***name***}".<br><br>Relative path indicators "./" and "../" can be used in filenames. |

### Description

Loads additional ***e*** modules before continuing to load the current file.

If a specified module has already been loaded or compiled, the statement is ignored. For modules not already loaded or compiled, the search sequence is:

1) Directories specified by the PATH environment variable.

2) The current directory.
3) The directory in which the importing module resides.

If you need to refer in struct A to a struct member of struct B, and you also need to refer in struct B to a struct member of struct A, that is called a cyclic reference. The **import** statement can handle cyclic references if you do the following:

1) Before the definition of struct A in module A, add an **import** of module B in which struct B is defined.
2) Before the definition of struct B in module B, add an **import** of module A.

This is called implicit cyclic importing.

Another way to do a cyclic import is to use the **import (** ***file-name***, ... **)** syntax, which resolves cycles by loading the two or more modules as one.

When multiple modules are loaded together, the behavior is as if the files are concatenated.

No module is imported more than once. If an **import** statement includes a module that has already been loaded, that module is not imported.

## Notes

— Within a given ***e*** module, **import** statements must appear before any other statements except preprocessor directives (**#ifdef**, etc), **define** statements, **verilog import** statements and **package** statements. (**package** statements must always precede any other statements.) Any other type of statement preceding an **import** statement causes a load-time or compile-time error. See Example 6 on page 639 for a special case where this restriction also applies to **import** statements in different ***e*** modules.
— You cannot import modules that reside in different directories but have the same base names, even if they have different extensions. This is because the ***e*** program internally uses only the base name, without its extension.
— If you do not enter at least one file name after the **import** keyword, a load or compile-time error is issued.
— Cyclic importing requires more memory to load multiple modules together than it takes to import modules singly, and it takes longer, which delays the automatic consistency checking of the ***e*** code.
— Cyclic importing can also mask problems with the ability of a module to be used standalone, in future applications, due to one module relying on another module being loaded.
— Attention must be given to the import order in implicit cyclic importing just as it is in non-cyclic importing. You cannot reference or extend a struct before it is defined. If module A imports module B, and if you load A from the command line or **import** A from another module, the body of module B is parsed before module A. However, if you use explicit cyclic importing, **import (**A, B**)**, then the body of module A is parsed before module B is imported.

## Example 1

The following UNIX commands are executed prior to loading the ***e*** module shown below:

```
setenv S_PATH /top
cd /cad/test
```

All of the **import** statements in the following ***e*** module are legal:

```
import t_fil_1.e;
    // Load /cad/test/t_fil_1.e if it exists, otherwise load
```

```
        // $PATH/t_fil_1.e
    import ./t_fil_2.e;
        // Load /cad/test/t_fil_2.e (relative path)
    import ../t_fil_3.e;
        // Load /cad/t_fil_3.e (relative path)
    import /cad/test/t_fil_4.e;
        // Load /cad/test/t_fil_4.e (absolute path)
    import $S_PATH/t_fil_5.e;
        // Load /top/t_fil_5.e
```

### Example 2

In the following example, a struct named “pci_transaction” is defined in one module, which is then imported into another module where additional fields and constraints are added in an extension to the struct definition.

```
<'
// module pci_transaction_definition.e
type PCICommandType: [ IO_READ=0x2, IO_WRITE=0x3,
                        MEM_READ=0x6, MEM_WRITE=0x7 ];
struct pci_transaction {
    address: uint;
    command: PCICommandType;
    bus_id: uint;
};
'>
------------------------------------------------------------
<'
// module pci_transaction_extension.e
import pci_transaction_definition;
extend pci_transaction {
    data: list of uint;
    num_data_phases: uint;
    keep num_data_phases in [0..7];
    keep data.size() == num_data_phases;
};
'>
```

### Example 3

In the following example, three modules are involved in cyclic referencing:

— the switch.e module references the packet struct definition in the packet.e module
— the packet.e module references the cell struct definition in the cell.e module
— the cell.e module references the switch struct definition in the switch.e module.

You only need to load the switch.e module. The switch.e module imports the packet.e module, which imports the cell.e module. Then the cell.e module imports the switch.e module, completing the cycle. This is implicit cyclic importing, since each **import** statement imports only one of the other modules.

```
<’
// module switch.e - needs packet.e for the list of packet
import packet;
struct switch {
    packets: list of packet;
};
’>
```

```
------------------------------------------------------------
<'
// module packet.e - needs cell.e for the list of cell
import cell;
struct packet {
    !cells: list of cell;
    len: uint;
};
'>
------------------------------------------------------------
<'
// module cell.e - needs switch.e for the switch definition
import switch;
struct cell {
    parent: switch;
    data[20]: list of byte;
};
'>
```

### Example 4

This example shows the explicit cyclic import syntax, **import (*file-name,*** ...**)**, using the same modules as Example 3 on page 637. All three of the modules involved in the cyclic referencing are imported by one **import** statement in a fourth module named top_imp.e. You only need to load the top_imp.e module.

```
<'
// module top_imp.e
import (switch, packet, cell);
'>
------------------------------------------------------------
<'
// module switch.e - needs packet.e for the list of packet
struct switch {
    packets: list of packet;
};
'>
------------------------------------------------------------
<'
// module packet.e - needs cell.e for the list of cell
struct packet {
    !cells: list of cell;
    len: uint;
};
'>
------------------------------------------------------------
<'
// module cell.e - needs switch.e for the switch definition
struct cell {
    parent: switch;
    data[20]: list of byte;
};
'>
```

### Example 5

This example shows how to load the files in Example 3 on page 637 while avoiding the loss of modularity that results from cyclic importing.

In Example 3, the cell.e module relies on a type (switch) that is defined in the switch.e module. This means that a cell struct cannot be used without also using a switch struct, so that cell.e cannot stand alone.

In this example, the switch struct instance has been moved from the cell.e module to an extension of cell in the switch.e module, so that the cell.e module does not rely on the presence of the switch.e module.

```
<'
// module switch.e - needs packet.e for the list of packet
import packet;
struct switch {
    packets: list of packet;
};
extend cell {
    parent: switch;
};
'>
------------------------------------------------------------
<'
// module packet.e - needs cell.e for the list of cell
import cell;
struct packet {
    !cells: list of cell;
    len: uint;
};
'>
------------------------------------------------------------
<'
// module cell.e
struct cell {
    data[20]: list of byte;
};
'>
```

## Example 6

The case of an **import** followed by an **#ifdef** which, in turn, imports another module causes a load error if the second imported module has a statement other than a macro, preprocessor directive, or another **import** preceding one of those types of statements. This is a special case which causes a violation of the statement order rule given in "Statements" on page 12. The three ***e*** modules below illustrate this.

```
===============
<'
import defs.e;
#ifdef VENUS then {
    import venus.e;
};
'>

================
<'
type d_type: [A_DR, X_DR]; // At this location, this causes an error
define VENUS;
'>

=================
<'
extend sys {
    event venus_init_done;
};
'>
```

Trying to load top.e results in this load error. The **type** statement preceding the **#ifdef** statement in the defs.e module is seen as out of order with respect to the **import** venus.e statement. The error is shown below:

```
*** Error: Import Statements should be placed at the top of the file - please
   change the statements order, pay attention to the imported module
   'venus.e'.
                at line 6 in top.e
       import venus.e;
```

This error can be avoided by simply moving the **define** VENUS statement above the **type** statement in the defs.e module:

```
// module defs.e
================
<'
define VENUS;
type d_type: [A_DR, X_DR]; // At this location, no error occurs
'>
```

### See Also

- "#define" on page 630
- "#ifdef, #ifndef" on page 627
- "verilog import" on page 799

# 22 Encapsulation Constructs

This chapter contains syntax and descriptions of the ***e*** statements that are used to create packages and modify access control. The constructs are:

— "package package-name" on page 641
— "package type-declaration" on page 642
— "package | protected | private struct-member" on page 643

## 22.1 package package-name

### Purpose

Associates a module with a package.

### Category

Statement

### Syntax

package *package-name*

Syntax example:

```
package vr_xb;
```

### Parameters

| | |
|---|---|
| *package-name* | A standard ***e*** identifier which assigns a unique name to the package. It is legal for a package name to be the same as a module or type name. |

### Description

Only one **package** statement can appear in a file, and it must be the first statement in the file.

A file with no **package** statement is equivalent to a file beginning with the statement, **package main**.

### Example

```
<'
// module vr_xb_top
package vr_xb;
'>
```

### See Also

— "package type-declaration" on page 642
— "package | protected | private struct-member" on page 643

## 22.2 package type-declaration

### Purpose

Modifies access to a type or a struct.

### Category

Statement

### Syntax

[**package**] *type-declaration*

Syntax example:

```
package type t: int(bits: 16);
```

### Parameters

| | |
|---|---|
| *type-declaration* | An ***e*** type declaration (for a struct, unit, enumerated list, or other type). |

### Description

The **package** modifier means that code outside the package files cannot access the defined struct member. This includes declaring a variable of the type, extending, inheriting, casting using the **as_a** operator, and all other contexts in which the name of a type is used. It is equivalent to the default (package) access level for classes in Java.

NOTE— The package type does not determine the visibility of a package, but only its access control.

Without the **package** modifier, the type or struct has no access restriction.

A derived struct (using like inheritance) must be explicitly declared **package** if its base struct is declared **package**. It can be declared **package** even if its base struct is not.

Definition of a when subtype (using a **when** or **extend** clause) does not allow for an access modifier. A when subtype is public unless either its base struct or one of its determinant fields is declared **package**.

A **when** subtype cannot have a **private** or **protected** determinant field. Any reference to a **when** subtype, even in a context in which the **when** determinant field is accessible, results in a compilation error.

### Example

```
<'
// module vr_xb_top
package vr_xb;;
package type width: uint(bits: 8);
'>
```

### See Also

— "package package-name" on page 641

— “package | protected | private struct-member” on page 643

## 22.3 package | protected | private struct-member

### Purpose

Modifies access to a struct field, method, or event.

### Category

Keyword

### Syntax

package *struct-member-definition*

protected *struct-member-definition*

private *struct-member-definition*

Syntax examples:

```
private f: int;
protected m() is {};
package event e;
```

### Parameters

| | |
|---|---|
| *struct-member-definition* | A struct or unit field, method, or event definition. SeeChapter 4, “Structs, Fields, and Subtypes” for the syntax of struct and unit member definitions. |

### Description

A struct member declaration may include a **package**, **protected**, or **private** keyword to modify access to the struct member.

If no access modifier exists in the declaration of a struct member, the struct member has no access restriction (the default is public).

The **package** modifier means that code outside the package files cannot access the struct member. It is equivalent to the default (package) access level for fields and methods in Java.

The **protected** modifier means that code outside the struct family scope cannot access the struct member. It is similar (although not equivalent) to the *protected* semantics in other object-oriented languages.

The **private** modifier means that only code within both the package and the struct family scope can access the struct member. This means that code within the extension of the same struct in a different package is outside its accessibility scope. It is less restrictive than *private* attribute of other object-oriented languages in the sense that methods of derived structs or units within the same package can access a private struct member.

An extension of a struct member can restate the same access modifier as the declaration has, or omit the modifier altogether. If a different modifier appears, the compiler issues an error.

All references to a struct member outside its accessibility scope result in an error at compile time. Using an enumerated field's value as a **when** determinant is considered such a reference, even if the field name is not explicitly mentioned.

A field must be declared **package** or **private** if its type is **package**, unless it is a member of struct which is declared **package**.

A method must be declared **package** or **private** if its return type or any of its parameter types are **package**, unless it is a method of a struct which is declared **package**.

Only fields, methods and events can have access restrictions. There are other named struct members in ***e***, namely cover groups and named expects, to which access control does not apply - they are completely public. However, cover groups and expects are defined in terms of fields, methods and events, and can refer to other entities in their definitions according to the accessibility rules.

## Example

```
<'
package P1;

struct s1 {
    private f: int;
    protected m() is {};
    package event e;
};
'>
```

## See Also

— "package package-name" on page 641
— "package type-declaration" on page 642

# 23 Predefined Methods Library

A significant part of ***e*** functionality is implemented as set of predefined methods defined directly under the **global** and **sys** structs.

Furthermore, every struct that is already available to you or is defined by you inherits a set of predefined methods. Some of these methods can be extended to add functionality, and some of them are empty, allowing you to define their function.

Three other predefined structs, **semaphore**, **locker**, and **scheduler**, provide predefined methods that are useful in controlling TCMs and in controlling resource sharing between TCMs. A third predefined struct, the **simulator** struct, has a predefined method that allows access to Verilog macros during a run.

Finally, there are pseudo-methods. Calls to pseudo-methods look like method calls. However, they are associated not with struct expressions, but with other kinds of expressions.

The following sections describe the predefined methods:

— "Predefined Methods of sys" on page 645
— "Predefined Methods of Any Struct" on page 647
— "Predefined Methods of Any Unit" on page 662
— "Unit-Related Predefined Methods of Any Struct" on page 670
— "Pseudo-Methods" on page 676
— "Semaphore Methods" on page 680
— "TCM Related Methods" on page 692
— "Coverage Methods" on page 700

### See Also

— Chapter 19, "List Pseudo-Methods Library"
— Chapter 24, "Predefined Routines Library"
— Chapter 26, "Predefined File Routines Library"

## 23.1 Predefined Methods of sys

This section contains descriptions of the extendable methods of **sys**:

— "The init() Method of sys" on page 645
— "The run() Method of sys" on page 646

It is not recommended to extend **sys.generate()**. Instead, you should extend the related **pre_generate()** or **post_generate()** method of a particular struct or unit. See "Predefined Methods of Any Struct" on page 647 for more information on these methods.

### 23.1.1 The init() Method of sys

#### Purpose

Perform general preparations for the test

### Category

Predefined method for **sys**

### Syntax

**[sys.]init()**

Syntax example:

```
extend sys {
    init() is also {
        out("Performing initialization of sys...");
    };
};
```

### Description

This method is called when you load an ***e*** file or when you invoke an extended executable that contains compiled ***e*** code.

It is not invoked when you restore an environment from a save file.

You can extend this method to perform general preparations for the test.

### See Also

— "The init() Method of any_struct" on page 656

## 23.1.2 The run() Method of sys

### Purpose

Recommended place for starting TCMs

### Category

Predefined method for **sys**

### Syntax

**[sys.]run()**

Syntax example:

```
run() is also {
    start monitor();
};
```

### Description

Can be extended to start user-defined TCMs. The method is initially empty.

NOTE— Starting a TCM before the end of **start_test()** causes a runtime error.

### Example

```
<'
extend sys {
    id : int;
    monitor() @sys.any is {
        while (TRUE) {
            wait [2] * cycle;
            out("cycle ", sys.time,
                " Packet id ", id, " is still running");
        };
    };

    run() is also {
        start monitor();
    };
};
'>
```

### See Also

— "The run()  Method of any_struct" on page 661
— "start tcm()" on page 477

## 23.2 Predefined Methods of Any Struct

The following methods are available for any instantiated user-defined struct or unit.

— "The copy() Method of any_struct" on page 647
— "do_pack()" on page 649
— "do_unpack()" on page 652
— "The do_print() Method of any_struct" on page 655
— "The init() Method of any_struct" on page 656
— "The print_line() Method of any_struct" on page 658
— "The quit()  Method of any_struct" on page 659
— "The run()  Method of any_struct" on page 661

### 23.2.1 The copy() Method of any_struct

#### Purpose

Make a shallow copy

#### Category

Predefined method of any struct or unit

#### Syntax

***exp*.copy()**: exp

Syntax example:

```
var pmv: packet = sys.pmi.copy();
```

## Parameters

| | |
|---|---|
| *exp* | Any legal ***e*** expression. |

## Description

Returns a shallow, non-recursive copy of the expression. This means that if the expression is a list or a struct that contains other lists or structs, the second-level items are not duplicated. Instead, they are copied by reference.

The following list details how the copy is made, depending on the type of the expression:

| | |
|---|---|
| scalar | The scalar value is simply assigned as in a normal assignment. |
| string | The whole string is copied. |
| scalar list | If ***exp*** is a scalar list, a new list with the same size as the original list is allocated. The contents of the original list is duplicated. |
| list of structs | A new list with the same size as the original list is allocated. The contents of the list is copied by reference, meaning that each item in the new list points to the corresponding item in the original list. |
| struct | If ***exp*** is a struct instance, a new struct instance with the same type as the original struct is allocated. All scalar fields are duplicated. All compound fields (lists or structs) in the new struct instance point to the corresponding fields in the original struct. |

## Notes

- Do not use the assignment operator (=) to copy structs or lists into other data objects. The assignment operator simply manipulates pointers to the data being assigned and does not create new struct instances or lists.
- Use the **deep_copy()** method if you want a recursive copy of a struct or list that contains compound fields or items.

## Example

```
<'
struct packet {
    header: header;
    data[10] :list of byte;
    type: [ATM, ETH, IEEE];
};

struct header {
    code: uint;
};

extend sys {
    pmi: packet;

    m() is {
        var pmv: packet = sys.pmi.copy();
        pmv.data[0] = 0xff;
        pmv.header.code = 0xaa;
        pmv.type = IEEE;
        print pmi.data[0], pmi.header.code, pmi.type;
        print pmv.data[0], pmv.header.code, pmv.type;
        };
    };
    '>
```

### Result

This example shows that any changes in value to lists and structs contained in the copied struct instance ("pmv") are reflected in the original struct instance ("pmi") because these items are copied by reference.

```
pmi.data[0] = 0xff
pmi.header.code = 0xaa
pmi.type = ATM
pmv.data[0] = 0xff
pmv.header.code = 0xaa
pmv.type = IEEE
```

### See Also

— "deep_copy()" on page 713

## 23.2.2 do_pack()

### Purpose

Pack the physical fields of the struct

### Category

Predefined method of any struct

### Syntax

**do_pack(*options***:pack options, ***l***: *list of bit**)**

Syntax example:

```
do_pack(options:pack_options, l: *list of bit) is only {
    var L : list of bit = pack(packing.low, operand2,
        operand1,operand3);
    l.add(L);
};
```

### Parameters

| | |
|---|---|
| options | This parameter is an instance of the pack options struct. See "Using the Predefined pack_options Instances" on page 506 for information on this struct. |
| l | An empty list of bits that is extended as necessary to hold the data from the struct fields. |

### Description

The **do_pack()** method of a struct is called automatically whenever the struct is packed. This method appends data from the physical fields (the fields marked with %) of the struct into a list of bits according to flags determined by the pack options parameter. The virtual fields of the struct are skipped. The method issues a runtime error message if this struct has no physical fields.

For example, the following assignment to “lob”

```
lob = pack(packing.high, i_struct, p_struct);
```

makes the following calls to the **do_pack** method of each struct, where ***tmp*** is an empty list of bits:

```
i_struct.do_pack(packing.high, *tmp)
p_struct.do_pack(packing.high, *tmp)
```

You can extend the **do_pack()** method for a struct in order to create a unique packing scenario for that struct. You should handle variations in packing that apply to many structs by creating a custom **pack_options** instance. See “Customizing Pack Options” on page 510 for information on how to do this.

### Notes

- — Do not call the **do_pack()** method of any struct directly, for example “my_struct.do_pack()”. Instead use **pack(),** for example “pack(packing.high, my_struct)”.
- — Do not call **pack(me)** in the **do_pack()** method. This causes infinite recursion. Call **packing.pack_struct(me)** instead. You can call **pack()** within the **do_pack()** method to pack objects other than **me**.
- — Do not forget to append the results of any pack operation within **do_pack()** to the empty list of bits referenced in the **do_pack()** parameter list.
- — If you modify the **do_pack()** method and then later add physical fields in an extension to the struct, you may have to make adjustments in the modifications to **do_pack()**.

### Example 1

This example shows how to override the **do_pack()** method for a struct called “cell”. The extension to **do_pack()** overrides any packing option passed in and always uses **packing.low**. It packs “operand2” first, then “operand1” and “operand3”.

```
<'
struct cell {
    %operand1: uint(bytes:2);
    %operand2: uint(bytes:2);
    %operand3: uint(bytes:2);
};

extend cell {
    do_pack(options:pack_options, l: *list of bit) is only {
        var L : list of bit = pack(packing.low, operand2,
            operand1,operand3);
        l.add(L);
    };
};
```

### Result

```
sys.pi = cell-@0: cell
    ---------------------------------------------- @pack33
0    %operand1:                     0b0010001000111001
1    %operand2:                     0b0001101001110101
2    %operand3:                     0b0001001010110010

var L : list of bit = pack(packing.high, sys.pi)

    L = (48 items, bin):
```

```
    0 0 1 1  1 0 0 1  0 0 0 1  1 0 1 0  0 1 1 1  0 1 0 1    .0
    0 0 0 1  0 0 1 0  1 0 1 1  0 0 1 0  0 0 1 0  0 0 1 0    .24
```

### Example 2

In the following example, the **do_pack()** method for "cell" is overwritten to use the **low_big_endian** packing option by default.

```
struct cell {
    %operand1: uint(bytes: 2);
    %operand2: uint(bytes: 2);
    %operand3: uint(bytes: 2);
};

extend cell {
    do_pack(options: pack_options, l: *list of bit) is only {
        if (options == NULL) then {
            packing.pack_struct(me,
              packing.low_big_endian,l);
        } else {
            packing.pack_struct(me, options, l);
        };
    };
};

extend sys {
    pi: cell;
};
```

### Result

```
    sys.pi = cell-@0: cell
    ---------------------------------------------- @pack34
0    %operand1:                    0b0010001000111001
1    %operand2:                    0b0001101001110101
2    %operand3:                    0b0001001010110010

var M : list of bit = pack(NULL, sys.pi)

    M = (48 items, bin):
    0 0 0 1  1 0 1 0  0 0 1 1  1 0 0 1  0 0 1 0  0 0 1 0    .0
    1 0 1 1  0 0 1 0  0 0 0 1  0 0 1 0  0 1 1 1  0 1 0 1    .24
```

### Example 3

This example swaps every pair of bits within each 4-bit chunk after packing with the packing options specified in the **pack()** call.

```
struct cell {
    %operand1: uint(bytes: 2);
    %operand2: uint(bytes: 2);
    %operand3: uint(bytes: 2);
};

extend cell {
    do_pack(options:pack_options, l: *list of bit) is only {
        var L1 : list of bit;
        packing.pack_struct(me, options, L1);
```

```
            var L2 : list of bit = L1.swap(2,4);
            l.add(L2);
        };
    };
```

### Result

```
        sys.pi = cell-@0: cell
        ---------------------------------------------- @pack35
    0    %operand1:                      0b0010001000111001
    1    %operand2:                      0b0001101001110101
    2    %operand3:                      0b0001001010110010

    var M : list of bit = pack(NULL, sys.pi)

        M = (48 items, bin):
        1 1 0 1  0 1 0 1  1 0 0 0  1 0 0 0  1 1 0 0  0 1 1 0    .0
        0 1 0 0  1 0 0 0  1 1 1 0  1 0 0 0  0 1 0 0  1 0 1 0    .24
```

### See Also

- "pack()" on page 516
- "unpack()" on page 521
- "%{... , ...}" on page 62
- "swap()" on page 524
- "do_unpack()" on page 529

## 23.2.3 do_unpack()

### Purpose

Unpack a packed list of bit into a struct

### Category

Predefined method of any struct

### Syntax

**do_unpack(*options*:**pack options, ***l***: list of bit, ***from***: int**)**: int

Syntax example:

```
    do_unpack(options:pack_options, l: list of bit, from: int):int is only {
        var L : list of bit = l[from..];
        unpack(packing.low, L, op2, op1, op3);
        return from + 8 + 5 + 3;
    };
```

### Parameters

| | |
|---|---|
| *options* | This parameter is an instance of the pack options struct. See "Using the Predefined pack_options Instances" on page 506 for information on this struct. |
| *l* | A list of bits containing data to be stored in the struct fields. |
| *from* | An integer that specifies the index of the bit to start unpacking. |
| int (return value) | An integer that specifies the index of the last bit in the list of bits that was unpacked. |

### Description

The **do_unpack()** method is called automatically whenever data is unpacked into the current struct. This method unpacks bits from a list of bits into the physical fields of the struct. It starts at the bit with the specified index, unpacks in the order defined by the pack options, and fills the current struct's physical fields in the order they are defined.

For example, the following call to unpack()

```
unpack(packing.low, lob, c1, c2);
```

makes the following calls to the **do_unpack** method of each struct:

```
c1.do_unpack(packing.low, lob, index);
c2.do_unpack(packing.low, lob, index);
```

The method returns an integer, which is the index of the last bit unpacked into the list of bits.

The method issues a runtime error message if the struct has no physical fields. If at the end of packing there are leftover bits, it is not an error. If more bits are needed than currently exist in the list of bits, a runtime error is issued ("Ran out of bits while trying to unpack into ***struct_name***").

You can extend the **do_unpack()** method for a struct in order to create a unique unpacking scenario for that struct. You should handle variations in unpacking that apply to many structs by creating a custom **pack_options** instance. See "Customizing Pack Options" on page 510 for information on how to do this.

### Notes

— Do not call the **do_unpack()** method of any struct directly, for example "my_struct.do_unpack()". Instead use **unpack(),** for example "unpack(packing.high, lob, my_struct)".

— When you modify the **do_unpack()** method, you need to calculate and return the index of the last bit in the list of bits that was unpacked. In most cases, you simply add the bit width of each physical field in the struct to the starting ***index*** parameter. If you are unpacking into a struct that has conditional physical fields (physical fields defined under a **when**, **extend**, or **like** construct), this calculation is a bit tricky. See the Verification Advisor's patterns on packing for an example of how to do this.

### Example 1

This first example shows how to modify **do_unpack()** to change the order in which the fields of a struct are filled. In this case, the order is changed from "op1", "op2", "op3" to "op2", "op1", "op3". You can see also that **do_unpack()** returns the bit widths of the three physical fields, "op1", "op2", and "op3", to the starting index, "from".

```
struct cell {
    %op1: uint(bytes:1);
    %op2: uint(bits:5);
    %op3: uint(bits:3);
};

extend cell {
    do_unpack(options:pack_options, l: list of bit,
            from: int)  :int is only {
        var L : list of bit = l[from..];
        unpack(packing.low, L, op2, op1, op3);
        return from + 8 + 5 + 3;
    };
};
```

### Result

```
var P : list of bit = {0;0;0;0;1;1;0;1;1;1;0;0;0;0;1;0;};
unpack(NULL, P, sys.pi)
print sys.pi using bin
    sys.pi = cell-@0: cell
---------------------------------------------- @pack36
0      %op1:                                  0b00011101
1      %op2:                                  0b10000
2      %op3:                                  0b010
```

### Example 2

This example modifies the **do_unpack** method of the “frame” struct to first calculate the length of the “data” field. The calculation uses “from”, which indicates the last bit to be unpacked, to calculate the length of “data”.

```
extend sys {
    !packet1 : packet;
    !packet2 : packet;

    post_generate() is also {
        var raw_data : list of byte;
        for i from 0 to 39 {
            raw_data.add(i);
        };
        unpack(packing.low, raw_data, packet1);
        print packet1.header, packet1.frame.data,
            packet1.frame.crc;
        unpack(packing.high, raw_data, packet2);
        print packet2.header, packet2.frame.data,
            packet2.frame.crc;
    };
};

struct packet {
    %header    : int (bits : 16);
    %frame     : frame;
};

struct frame {
    %data[len] : list of byte;
    %crc      : int (bits : 32);
```

```
        len : int;

        do_unpack(options :pack_options, l :list of bit,
            from :int):int is first {

                if options.reverse_fields then {
                    len = (from - 32 + 1) / 8;
                } else {
                    len = (l.size() - from - 32) / 8;
                };
        };
    };
```

### Results

```
    packet1.header = 256
    packet1.frame.data = (34 items, dec):
             13  12  11  10    9   8   7   6    5   4   3   2        .0
             25  24  23  22   21  20  19  18   17  16  15  14        .12
                     35  34   33  32  31  30   29  28  27  26        .24

    packet1.frame.crc = 656811300
    packet2.header = 10022
    packet2.frame.data = (34 items, dec):
             26  27  28  29   30  31  32  33   34  35  36  37        .0
             14  15  16  17   18  19  20  21   22  23  24  25        .12
                      4   5    6   7   8   9   10  11  12  13        .24

    packet2.frame.crc = 50462976
```

### See Also

— “pack()” on page 516
— “unpack()” on page 521
— “%{... , ...}” on page 62
— “swap()” on page 524
— “do_unpack()” on page 529

## 23.2.4 The do_print() Method of any_struct

### Purpose

Print struct info

### Category

Predefined method of any struct or unit

### Syntax

[*exp*.]**do_print()**

Syntax example:

```
    do_print() is first {
        outf("Struct %s :", me.s);
    };
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. |

### Description

Controls the printing of information about a particular struct. You can extend this method to customize the way information is displayed.

This method is called by the **print** action whenever you print the struct.

### Example

```
<'
struct a {
    i: int;
    s: string;
    do_print() is first {
        outf("Struct %s :", me.s);
    };
};
extend sys {
    m() is {
        var aa := a new a with {
            .i = 1;
            .s = "AA";
        };
        print aa;
    };
};
'>
```

### Result

**sys.m()**

```
  aa = Struct AA :a-@0: a
        ----------------------------------------------  @predefined_methods5
0       i:                                0x1
1       s:                                "AA"
```

## 23.2.5 The init() Method of any_struct

### Purpose

Initialize struct

### Category

Predefined method of any struct or unit

### Syntax

[***exp***.]**init()**

Syntax example:

```
init() is also {
    is_ok = TRUE;
    list_subs = {320; 330; 340; 350; 360};
    list_color = {black; red; green; blue; yellow; white};
};
```

## Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. |

## Description

The **init()** method of a struct is called when a new instance of the struct is created.

You can extend the **init()** method of a struct to set values for fields that you want to have a different value than the default value. By default, all fields of scalar type are initialized to zero. The initial value of a struct or list is NULL, unless the list is a sized list of scalars, in which case it is initialized to the proper size with each item set to the default value.

You should consider initializing the non-generated fields of a struct, especially fields of an enumerated scalar type or unsized lists. Enumerated scalar types are initialized to zero, even if that is not a legal value for that type. If the field is sampled before it is assigned, you should initialize it. As for lists, if you intend to fill a list with data from the DUT, you should either size the list or initialize it. Unpacking data from the DUT into an unsized, uninitialized list causes a runtime error.

If a field is initialized but not marked as non-generated, the initialization is overwritten during generation. To mark a field as non-generated, place a ! character in front of the field name.

## Example

```
<'
type color: [black, red, green, blue, yellow, white];
type sub_rang1: int [300..500];
struct pm {
    !list_color: list of color;
    !list_subs: list of sub_rang1;
    !is_ok:bool;

    init() is also {
        is_ok = TRUE;
        list_subs = {320; 330; 340; 350; 360};
        list_color = {black; red; green; blue; yellow; white};
    };
};

extend sys {
    pmi:pm;
};
'>
```

## Result

```
print sys.pmi.list_color, sys.pmi.list_subs,
    sys.pmi.is_ok
  sys.pmi.list_color =
0.      black
1.      red
```

```
2.      green
3.      blue
4.      yellow
5.      white
  sys.pmi.list_subs =
0.      320
1.      330
2.      340
3.      350
4.      360
  sys.pmi.is_ok = TRUE
```

### See Also

- "The init() Method of sys" on page 645
- "Packing and Unpacking Lists" on page 503
- "e Data Types" on page 75
- "{... ; ...}" on page 60

## 23.2.6 The print_line() Method of any_struct

### Purpose

Print a struct or a unit in a single line

### Category

Predefined method of any struct or unit

### Syntax

[***exp***.]**print_line(NULL |** ***struct-type*****.type())**

Syntax example:

```
sys.pmi[0].print_line(sys.pmi[0].type());
sys.pmi[0].print_line(NULL);
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a struct or a unit. |
| NULL \| ***struct-type*****.type()** | To print a row representation of the struct or unit, the parameter is NULL.To print the header for the list, the parameter is of the form: `struct-type.type()` |

### Description

You can call this method to print lists of structs of a common struct type in a tabulated table format. Each struct in the list is printed in a single line of the table.

When printing the structs, there is a limit on the number of fields printed in each line. The first fields that fit into a single line are printed; the rest are not printed at all. Each field is printed in a separate column, and there is a limitation on the column width. When a field exceeds this width, it is truncated and an asterisk is placed as the last character of that field's value.

### Example

```
<'
struct packet {
    protocol: [eth, ieee, atm];
    size: int [0..1k];
    data[size]: list of byte;
};

extend eth packet {
    e: int;
};

extend ieee packet {
    i: int;
};
extend atm packet {
    a: int;
};

extend sys {
    pmi[5]: list of packet;

    m() is {
        sys.pmi[0].print_line(sys.pmi[0].type());
        sys.pmi[0].print_line(NULL);
        sys.pmi[1].print_line(NULL);
        sys.pmi[2].print_line(NULL);
        sys.pmi[3].print_line(NULL);
        sys.pmi[4].print_line(NULL);
    };
};
'>
```

### Result

```
item type    protocol    size    data        eth'e      ieee'i
packet       eth         872     (872 item*  -481087693
packet       eth         830     (830 item*  2019716495
packet       eth         834     (834 item*  -20064418*
packet       ieee        663     (663 item*          1557645288
packet       eth         213     (213 item*  1797949675
```

## 23.2.7 The quit()  Method of any_struct

### Purpose

Kill all threads of a struct or unit instance

### Category

Predefined method of any struct or unit

### Syntax

[***exp***.]**quit()**

Syntax example:

```
packet.quit();
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. |

### Description

Deactivates a struct instance, killing all threads associated with the struct and enabling garbage collection. The **quit()** method emits a quit event for that struct instance at the end of the current tick. At the end of the current tick, the **quit()** method kills any TCM threads that were started within the struct in which the **quit()** method is called. All attached events and **expect** members of the struct that are still running are also killed.

A thread is any started TCM. If a started TCM calls other TCMs, those TCMs are considered subthreads of the started TCM thread, and the **quit()** method kills those subthreads, too.

If a struct has more than one started TCM, each TCM runs on a separate, parallel thread. Each thread shares a unique identifier, or thread handle, with its subthreads. The thread handle is automatically assigned by the scheduler. You can access threads using the predefined methods of the scheduler.

The **quit()** method is called by the **global.stop_run()** method. You can also call it explicitly.

### Example

This example shows the **quit()** method used in conjunction with the **stop_run()** routine to stop a run cleanly. When a "packet" struct is generated by the "inject_packets()" method, its TCM "monitor()" is also started. The TCM monitor checks the status of the "inject_packets()" method. Four cycles after the "packet" is generated, it is killed by the **quit()** method. After all packet have been generated and killed, the **stop_run()** method is called to stop the simulator.

```
<'
extend sys {
    inject_packets() @sys.any is {
        var packet : packet;
        for i from 0 to 5 {
            wait [1] * cycle;
            gen packet;
            out("\nInject packet id ", packet.id);
            wait [4] * cycle;
            packet.quit();
        };
        stop_run();
    };

    run() is also {
        start inject_packets();
    };
};

struct packet {
    id : int;
    monitor() @sys.any is {
        while (TRUE) {
            wait [2] * cycle;
```

```
                out("cycle ", sys.time, " Packet id ", id,
                    " is still running");
            };
        };

        run() is also {
            start monitor();
        };
    };
    '>
```

### Result

```
    Inject packet id 134576642
    cycle 3 Packet id 134576642 is still running
    cycle 5 Packet id 134576642 is still running

    Inject packet id -1767441690
    cycle 8 Packet id -1767441690 is still running
    cycle 10 Packet id -1767441690 is still running

    Inject packet id 1480193010
    cycle 13 Packet id 1480193010 is still running
    cycle 15 Packet id 1480193010 is still running

    Inject packet id 370225594
    cycle 18 Packet id 370225594 is still running
    cycle 20 Packet id 370225594 is still running

    Inject packet id 595104854
    cycle 23 Packet id 595104854 is still running
    cycle 25 Packet id 595104854 is still running

    Inject packet id 631871925
    cycle 28 Packet id 631871925 is still running
```

### See Also

— “stop_run()” on page 841
— “kill()” on page 696
— “terminate_branch()” on page 698
— “terminate_thread()” on page 699

## 23.2.8 The run() Method of any_struct

### Purpose

Recommended place for starting TCMs

### Category

Method of any struct or unit

### Syntax

[*exp*.]**run()**

Syntax example:

```
run() is also {
    start monitor();
};
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. |

### Description

Can be extended to start user-defined TCMs. The method is initially empty.

The **run()** methods of all structs under **sys** are called, starting from **sys** in depth-first search order, by the **global.run_test()** method, when you execute a test.

After this initial pass, when any struct is generated (with the **gen** action) or allocated (with **new**), its **run()** method is also invoked. This ensures that:

— The **run()** method of each struct instance is called exactly once, thus avoiding multiple instances of the same started TCM;
— TCMs do not start and events do not occur before the ***e*** program is ready to accept them;
— The **run()** method is called after generation and uses the generated values.

If you run multiple tests in the same session, the **run()** method is called once for each test in the session. The **init()** method is called only once before the first test.

### Example

```
<'
struct packet {
    id : int;
    monitor() @sys.any is {
        while (TRUE) {
            wait [2] * cycle;
            out("cycle ", sys.time,
                " Packet id ", id, " is still running");
        };
    };

    run() is also {
        start monitor();
    };
};
'>
```

### See Also

— "The run() Method of sys" on page 646
— "start tcm()" on page 477

## 23.3 Predefined Methods of Any Unit

The following methods are available for any instantiated user-defined unit.

### 23.3.1 hdl_path()

#### Purpose

Return a relative HDL path for a unit instance

#### Category

Predefined pseudo-method for any unit

#### Syntax

[***unit-exp***.]**hdl_path()**: string

Syntax example:

```
extend dut_error_struct {
    write() is first {
        var channel: XYZ_channel =
          source_struct().try_enclosing_unit(XYZ_channel);
        if (channel != NULL) {
            out("Error in XYZ channel: instance ",
              channel.hdl_path());
        };
    };
};
```

#### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

#### Description

Returns the HDL path of a unit instance. The most important role of this method is to bind a unit instance to a particular component in the DUT hierarchy. Binding an ***e*** unit or unit instance to a DUT component allows you to reference signals within that component using relative HDL path names. Regardless of where the DUT component is instantiated in the final integration, the HDL path names are still valid. The binding of unit instances to HDL components is a part of the pre-run generation process and must be done in **keep** constraints.

Although absolute HDL paths are allowed, relative HDL paths are recommended if you intend to follow a modular verification strategy.

This method always returns an HDL path exactly as it was specified in constraints. If, for example, you use a macro in a constraint string, then **hdl_path()** returns the original and not substituted string.

### Notes

- All instances of the same unit must be bound to the same kind of HDL components.
- You cannot constrain the HDL path for **sys**.

### Example 1

This example shows how you can use relative paths in lower-level instances in the unit instance tree. To create the full HDL path of each unit instance, its HDL path is prefixed with the HDL path of its parent instance. In this example, because the HDL path of **sys** is “”, the full HDL path of “unit_core” is “top.router_i”. The full HDL path of “extra_channel” is “top.router_i.chan3”.

```
extend sys {
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() == "top.router_i";
};

extend XYZ_router {
    extra_channel: XYZ_channel is instance;
    keep extra_channel.hdl_path() == "chan3";
};
```

### Example 2

This example shows how **hdl_path()** returns the HDL path exactly as specified in the constraint. Thus the first **print** action prints “`TOP.router_i”. The second **print** action, in contrast, accesses “top.router_i.clk”.

```
verilog import macros.v;
extend sys {
    unit_core: XYZ_router is instance;
    keep unit_core.hdl_path() == "`TOP.router_i";
    run() is also {
        print unit_core.hdl_path();
        print '(unit_core).clk';
    };
};
```

### Result

```
unit_core.hdl_path() = "`TOP.router_i"
'top.router_i.clk' = 0
```

### See Also

- “HDL Paths and Units” on page 159
- “full_hdl_path()” on page 664
- “e_path()” on page 666

## 23.3.2 full_hdl_path()

### Purpose

Return an absolute HDL path for a unit instance

### Category

Predefined method for any unit

### Syntax

[***unit-exp***.]**full_hdl_path()**: string

Syntax example:

```
out ("Mutex violation in ", get_unit().full_hdl_path());};
```

### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

### Description

Returns the absolute HDL path for the specified unit instance. This method is used mainly in informational messages. Like the **hdl_path()** method, this method returns the path as originally specified in the **keep** constraint, without making any macro substitutions.

### Example

This example uses **full_hdl_path()** to display information about where a mutex violation has occurred.

```
extend XYZ_router {
!current_chan: XYZ_channel;
  mutex_checker() @pclk is {
    while ('packet_valid') {
      var active_channel: int = UNDEF;
      for each XYZ_channel(current_chan) in channels {
        if '(current_chan).valid_out' {
          if active_channel != UNDEF then {
              out ("Mutex violation in ",
                get_unit().full_hdl_path());};
          active_channel = index;
        };
      };
      wait cycle;
    };
  };
};
```

### Result

```
Mutual exclusion violation in top.router_i
```

### See Also

— "hdl_path()" on page 663
— "e_path()" on page 666

### 23.3.3 e_path()

#### Purpose

Returns the location of a unit instance in the unit tree

#### Category

Predefined method for any unit

#### Syntax

[***unit-exp***.]**e_path()**: string

Syntax example:

```
out("Started checking ", get_unit().e_path());
```

#### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

#### Description

Returns the location of a unit instance in the unit tree. This method is used mainly in informational messages.

#### Example

```
<'
unit ex_u {
    run() is also {
        inst = get_unit().e_path();
        var inst: string;
        inst = get_unit().e_path();
        out("ex instance: ", inst);
    };
};

unit top_u {
    exlist[10]: list of ex_u is instance;
};

extend sys {
    top: top_u is instance;
};
'>
```

#### Result

```
    ex instance: sys.top.exlist[0]
    ex instance: sys.top.exlist[1]
    ex instance: sys.top.exlist[2]
    ex instance: sys.top.exlist[3]
    ex instance: sys.top.exlist[4]
    ex instance: sys.top.exlist[5]
    ex instance: sys.top.exlist[6]
    ex instance: sys.top.exlist[7]
    ex instance: sys.top.exlist[8]
    ex instance: sys.top.exlist[9]
```

### See Also

— "full_hdl_path()" on page 664
— "hdl_path()" on page 663

## 23.3.4 agent()

### Purpose

Maps the DUT's HDL partitions into ***e*** code

### Category

Predefined pseudo-method for any unit

### Syntax

**keep** [*unit-exp.*]**agent() ==** ***string*****;**

Syntax example:

```
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
```

### Parameters

| | |
|---|---|
| unit-exp | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |
| string | One of the following predefined agent names: **verilog**, **vhdl**, **mti_vlog**, **mti_vhdl**, **ncvlog** and **ncvhdl**. Specifying the agent name as **verilog** or **vhdl** is preferred because it makes the ***e*** code portable between simulators. In contrast, if a unit is bound to a specific agent, for example to **mti_vhdl**, an error is issued if it is ported to NC Simulator. The predefined names are case-insensitive; in other words, **verilog** is the same as **Verilog.** |

### Description

Specifying an agent identifies the simulator that is used to simulate the corresponding DUT component. Once a unit instance has an explicitly specified agent name then all other unit instances instantiated within it are implicitly bound to the same agent name, unless another agent is explicitly specified.

An agent name may be omitted in a single-HDL environment but it must be defined implicitly or explicitly in a mixed HDL environment for each unit instance that is associated with a non-empty **hdl_path()**. If an agent name is not defined for a unit instance with a non-empty **hdl_path()** in a mixed HDL environment, an error message is issued.

Given the **hdl_path()** and **agent()** constraints, a correspondence map is established between the unit instance HDL path and its agent name. Any HDL path below the path in the map is associated with the same agent unless otherwise specified. This map is further used internally to pick the right adapter for each accessed HDL object.

It is possible to access Verilog signals from a VHDL unit instance code and vice-versa. Every signal is mapped to its HDL domain according to its full path, regardless of the specified agent of the unit that the signal is accessed from.

When the **agent()** method is called procedurally, it returns the agent of the unit. The spelling of the agent string is exactly as specified in the corresponding constraint.

### Notes

- Agents are bound to unit instances during the generation phase. Consequently, there is no way to map between HDL objects and agents before generation. As a result of this limitation, HDL objects in a mixed Verilog/VHDL environment cannot be accessed before generation from **sys.setup()**.
- An unsupported agent name causes an error message during the test phase.

### Example 1

In the following example, the driver instance inherits an agent name implicitly from the enclosing router unit instance.

```
extend sys {
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
    keep router.hdl_path() == "top.rout";
};

extend XYZ_router {
    driver: XYZ_router_driver is instance;

};
```

### Example 2

In this example, the signal ‘top.rout.packet_valid’ is sampled using the Verilog PLI because the path “top.rout” is specified as a Verilog path. In contrast, the signal ‘top.rout.chan.mux.data_out’ is sampled using a VHDL foreign interface because the closest mapped path is “top.rout.chan” and it is mapped as a VHDL path.

```
extend sys {
    router: XYZ_router is instance;
    keep router.agent() == "Verilog";
    keep router.hdl_path() == "top.rout";
};
```

```
unit XYZ_router {
    channel: XYZ_channel is instance;
    keep channel.agent() == "VHDL";
    keep channel.hdl_path() == "chan";

    run() is also {
        print 'packet_valid';
    };
};
unit XYZ_channel {
    run() is also {
        print 'mux.data_out';
    };
};
```

### 23.3.5 get_parent_unit()

#### Purpose

Return a reference to the unit containing the current unit instance

#### Category

Predefined method for any unit

#### Syntax

[***unit-exp***.]**get_parent_unit()**: unit type

Syntax example:

```
out(sys.unit_core.channels[0].get_parent_unit());
```

#### Parameters

| | |
|---|---|
| *unit-exp* | An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed. |

#### Description

Returns a reference to the unit containing the current unit instance.

#### Example

**out(sys.unit_core.channels[0].get_parent_unit())**
```
XYZ_router-@2
```

### See Also

- "get_unit()" on page 670
- "get_enclosing_unit()" on page 672
- "try_enclosing_unit()" on page 674

## 23.4 Unit-Related Predefined Methods of Any Struct

The unit-related predefined methods of any struct are:

- "get_unit()" on page 670
- "get_enclosing_unit()" on page 672
- "try_enclosing_unit()" on page 674
- "set_unit()" on page 676

### See Also

- "Predefined Methods of Any Unit" on page 662
- "Unit-Related Predefined Routines" on page 732

### 23.4.1 get_unit()

#### Purpose

Return a reference to a unit

#### Category

Predefined method of any struct

#### Syntax

[***exp***.]**get_unit()**: unit type

Syntax example:

```
out ("Mutex violation in ", get_unit().full_hdl_path());};
```

#### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed. |

#### Description

When applied to an allocated struct instance, this method returns a reference to the parent unit—the unit to which the struct is bound. When applied to a unit, it returns the unit itself.

Any allocated struct instance automatically establishes a reference to its parent unit. If this struct is generated during pre-run generation it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct the enclosing method belongs to. See Example 3 on page 672 for an illustration of this point.

This method is useful when you want to determine the parent unit instance of a struct or a unit. You can also use this method to access predefined unit members, such as **hdl_path()** or **full_hdl_path()**. To access user-defined unit members, use **get_enclosing_unit()**. See Example 1 on page 671 for an illustration of this point.

### Example 1

This example shows that **get_unit()** can access predefined unit members, while **get_enclosing_unit()** must be used to access user-defined unit members.

```
struct instr {
    %opcode     : cpu_opcode ;
    %op1        : reg ;
    kind        : [imm, reg];

    post_generate() is also {
--      get_unit().print_msg() ; -- COMPILE-TIME ERROR
        get_enclosing_unit(XYZ_cpu).print_msg();
        out("Destination for this instruction is ",
            get_unit().hdl_path()) ;
    };
};

unit XYZ_cpu {
    instrs[3]  :  list of instr;
    print_msg() is {out("Generating instruction for \
        XYZ_cpu...");};
};

extend sys {
    cpu1: XYZ_cpu is instance;
    keep cpu1.hdl_path()=="'TOP/CPU1";
};
'>
```

### Result

```
Generating the test using seed 1...
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
Generating instruction for XYZ_cpu...
Destination for this instruction is 'TOP/CPU1
```

### Example 2

The first call to **get_unit()** below shows that the parent unit of the struct instance “p” is **sys**. The second call shows that the parent unit has been changed to “XYZ_router”.

```
var p: XYZ_packet = new
out(p.get_unit())
  sys-@0
p.set_unit(sys.unit_core)
out(p.get_unit())
  XYZ_router-@1
```

### Example 3

In this example, the trace_inject() method displays the full HDL path of the “XYZ_dlx” unit (not the “XYZ_tb” unit) because “instr_list” is generated by the run method of “XYZ_dlx”.

```
extend sys {
    tb: XYZ_tb is instance;
    keep tb.hdl_path()=="`TOP/tb";
};
unit XYZ_tb {
    dlx: XYZ_dlx is instance;
        keep dlx.hdl_path()=="dlx_cpu";
    !instr_list: list of instruction;
    debug_mode: bool;
};
unit XYZ_dlx {
    run() is also {
        gen sys.tb.instr_list keeping { .size() < 30;};
    };
};
extend instruction {
    trace_inject() is {
        if get_enclosing_unit(XYZ_tb).debug_mode == TRUE {
            out("Injecting next instruction to ",
                get_unit().full_hdl_path());
        };
    };
};
```

### Result

```
sys.tb.instr_list[0].trace_inject()
Injecting next instruction to `TOP/tb.dlx_cpu
```

### See Also

— “get_parent_unit()” on page 669
— “get_enclosing_unit()” on page 672
— “try_enclosing_unit()” on page 674

## 23.4.2 get_enclosing_unit()

### Purpose

Return a reference to nearest unit of specified type

### Category

Predefined pseudo-method of any struct

### Syntax

[*exp*.]**get_enclosing_unit(*unit-type***: exp**)**: unit instance

Syntax example:

```
unpack(p.get_enclosing_unit(XYZ_router).pack_config,
  'data', current_packet);
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed.<br><br>NOTE— If **get_enclosing_unit()** is called from within a unit of the same type as ***exp***, it returns the present unit instance and not the parent unit instance. |
| *unit-type* | The name of a unit type or unit subtype. |

### Description

Returns a reference to the nearest higher-level unit instance of the specified type, allowing you to access fields of the parent unit in a typed manner.

You can use the parent unit to store shared data or options such as packing options that are valid for all its associated subunits or structs. Then you can access this shared data or options with the **get_enclosing_unit()** method.

### Notes

— The unit type is recognized according to the same rules used for the **is a** operator. This means, for example, that if you specify a base unit type and there is an instance of a unit subtype, the unit subtype is found.
— If a unit instance of the specified type is not found, a runtime error is issued.

### Example 1

In the following example, **get_enclosing_unit()** is used to print fields of the nearest enclosing unit instances of type “XYZ_cpu” and “tbench”. Unlike **get_unit()**, which returns a reference only to its immediate parent unit, **get_enclosing_unit()** searches up the unit instance tree for a unit instance of the type you specify. A runtime error is issued unless an instance of type “XYZ_cpu” and an instance of type “tbench” are found in the enclosing unit hierarchy.

```
struct instr {
    %opcode     : cpu_opcode ;
    %op1        : reg ;
    kind        : [imm, reg];

    post_generate() is also {
       out("Debug mode for CPU is ",
           get_enclosing_unit(XYZ_cpu).debug_mode);
       out("Memory model is ",
           get_enclosing_unit(tbench).mem_model);
    };
};
unit XYZ_cpu {
    instr:  instr;
    debug_mode: bool;
};
unit tbench {
    cpu: XYZ_cpu is instance;
```

```
        mem_model: [small, big];
    };

    extend sys {
        tb: tbench is instance;
    };
```

### Result

```
    Generating the test using seed 1...
    Debug mode for CPU is FALSE
    Memory model is small
```

### Example 2

```
    extend XYZ_router {
        pack_config:pack_options;

        keep pack_config == packing.low_big_endian;
    };
```

### Result

```
    var p: XYZ_packet = new
    print p.data
      p.data = (empty)
    out(p.get_unit())
      sys-@0
    p.set_unit(sys.unit_core)
    out(p.get_unit())
      XYZ_router-@1
    show unpack(
        p.get_enclosing_unit(XYZ_router).pack_config, data, p)
      1  0  9  8  7  6  5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0|  +0
                                                                                                   |
      0  0  0  0  1  1  0  1  0  0  0  0  1  1  0  0  0  0  0  0  1  0  1  1  0  0  0  0  1  0  1  0|
                                                                                                   |
      data                                                                                         |

                                     |5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0|  +32
                                     +                                               |
                                     |0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  0
                                     +                                               |
                                     |                                               |
```

### See Also

- "get_unit()" on page 670
- "set_unit()" on page 676
- "try_enclosing_unit()" on page 674
- "get_parent_unit()" on page 669

## 23.4.3 try_enclosing_unit()

### Purpose

Return a reference to nearest unit instance of specified type or NULL

### Category

Predefined method of any struct

### Syntax

[***exp***.]**try_enclosing_unit(*unit-type***: exp**)**: unit instance

Syntax example:

```
var MIPS := source_struct().try_enclosing_unit(MIPS);
```

### Parameters

| | |
|---|---|
| *exp* | An expression that returns a unit or a struct. If no expression is specified, the current struct or unit is assumed.<br><br>NOTE— If **try_enclosing_unit()** is called from within a unit of the same type as ***exp***, it returns the present unit instance and not the parent unit instance. |
| *unit-type* | The name of a unit type or unit subtype. |

### Description

Like **get_enclosing_unit()**, this method returns a reference to the nearest higher-level unit instance of the specified type, allowing you to access fields of the parent unit in a typed manner.

Unlike **get_enclosing_unit()**, this method does not issue a runtime error if no unit instance of the specified type is found. Instead, it returns NULL. This feature makes the method suitable for use in extensions to global methods such as **dut_error_struct.write()**, which may be used with more than one unit type.

### Example

```
<'
extend dut_error_struct {
    write() is also {
        var MIPS := source_struct().try_enclosing_unit(MIPS);
        if MIPS != NULL then {
            out("Status of ", MIPS.e_path(),
              " at time of error:");
            MIPS.show_status();
        };
    };
};
'>
```

### See Also

— “get_unit()” on page 670
— “get_enclosing_unit()” on page 672
— “get_parent_unit()” on page 669

### 23.4.4 set_unit()

#### Purpose

Change the parent unit of a struct

#### Category

Predefined method of any struct

#### Syntax

[***struct-exp***.]**set_unit(*parent***: exp**)**

Syntax example:

```
p.set_unit(sys.unit_core)
```

#### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. If no expression is specified, the current struct is assumed. |
| *parent* | An expression that returns a unit instance. |

#### Description

Changes the parent unit of a struct to the specified unit instance.

NOTE— This method does not exist for units because the unit tree cannot be modified.

#### Example

```
var p: XYZ_packet = new
out(p.get_unit())
  sys-@0
p.set_unit(sys.unit_core)
out(p.get_unit())
  XYZ_router-@1
```

## 23.5 Pseudo-Methods

Pseudo-methods calls look like method calls, but unlike methods they are not associated with structs and are applied to other types of expressions, such as lists.

Pseudo-methods cannot be changed or extended through use of the **is only**, **is also** or **is first** constructs.

The following sections provide descriptions of the pseudo-methods:

— "declared_type()" on page 677
— "type()" on page 677
— "field()" on page 678
— "unsafe()" on page 678
— "source_location()" on page 679

— "source_method()" on page 680

### See Also

— Chapter 19, "List Pseudo-Methods Library"
— Chapter 26, "Predefined File Routines Library"
— Chapter 24, "Predefined Routines Library"

## 23.5.1 declared_type()

### Purpose

Get a handle for the declared type of an expression

### Category

Pseudo-method

### Syntax

***exp*.declared_type()**: type_descriptor

Syntax example:

```
if pkt1.declared_type() != pkt1.type() then {out("Got a mismatch!")};
```

### Parameters

| | |
|---|---|
| *exp* | Any legal ***e*** expression. |

### Description

Returns a handle for the declared type of an expression

The use of this pseudo-method is strongly discouraged.

## 23.5.2 type()

### Purpose

Get a handle for the type of an expression

### Category

Pseudo-method

### Syntax

***exp*.type()**: type_descriptor

Syntax example:

```
if pkt1.type() == pkt2.type() then {out("Got a match!")};
```

### Parameters

| | |
|---|---|
| *exp* | Any legal ***e*** expression. |

### Description

Returns a handle for the type of an expression.

The use of this pseudo-method is strongly discouraged.

## 23.5.3 field()

### Purpose

Get the handle for a field

### Category

Pseudo-method

### Syntax

***struct-exp*.field(*field-name*)**: field

### Parameters

| | |
|---|---|
| *struct-exp* | An expression that returns a struct. |

### Description

Returns the handle for the specified field.

The use of this pseudo-method is strongly discouraged.

## 23.5.4 unsafe()

### Purpose

Bypass type checking

### Category

Pseudo-method

### Syntax

***exp*.unsafe()**: type

### Parameters

| | |
|---|---|
| *exp* | Any legal ***e*** expression. |

### Description

Passes the expression with no type checking or auto-casting.

This method should be used only when calling C routines that perform their own type checking.

### See Also

— "declared_type()" on page 677
— "type()" on page 677
— "field()" on page 678
— Chapter 3, "Data Types"

## 23.5.5 source_location()

### Purpose

Get source reference string

### Category

Pseudo-method

### Syntax

**source_location()**: string

Syntax example:

```
print source_location();
```

### Description

Returns the source location string. The string describes the line number and the module name in which the **source_location()** method was invoked. The format of the string is:

`at line` ***line-number*** `in @`***module-name***

### Example

```
<'
extend sys {
    m() is {
        out("I'm ",source_location());
    };
};
'>
```

### Result

```
sys.m()
  I'm at line 4 in @xxx
```

### See Also

— "dut_error_struct" on page 444

— “source_method()” on page 680

### 23.5.6 source_method()

#### Purpose

Get name of executing method

#### Category

Pseudo-method

#### Syntax

**source_method()**: string

Syntax example:

```
print source_method();
```

#### Description

Returns the name of the enclosing method. The string describes the line number and the module name in which the **source_method()** method was invoked. The format of the string is:

***method-name*** `at line` ***line-number*** `in @`***module-name***

#### Example

```
<'
extend sys {
    run() is also {
        out("location = '", source_location(),
            "'\nmethod = '", source_method(), "'");
    };
};
'>
```

#### Result

```
run
    location = 'at line 4 in @method'
    method = 'sys.run()    at line 3 in @method'
```

#### See Also

— “source_location()” on page 679

## 23.6 Semaphore Methods

The ***e*** language provides three predefined structs that are useful in controlling resource sharing between TCMs:

— semaphore

This is the typical semaphore. The maximum value ranges between 1 and MAX_INT. By default, it is MAX_INT, and the initial value (number of available resources) is 0. (See "Example 1" on page 683.)

— rdv_semaphore
A rendezvous semaphore is a semaphore with no resources. It requires the meeting of a producer and a consumer for either to proceed. When they finally proceed, the **up()** thread always runs first, followed immediately by the **down()**. (See "Example 2" on page 684.)

— locker
The methods of this struct provide a fair, FIFO ordered sharing of resources between multiple competing methods.

A locker is useful when a single entity needs to prevent others from a shared resource. lock() and release() must be issued by the same entity. A semaphore is more flexible. You could implement the locker functionality with semaphore by initializing the semaphore count to 1, then changing all locker.lock() to semaphore.down() and all locker.release() to semaphore.up(). You can also use a semaphore when the need is for a producer and consumer, i.e. one entity "locks" and the other one "releases". Finally, you could also use a semaphore if you had more than one shared resource available by initializing the count

Table 23-1 gives a brief description of the predefined methods of the **semaphore** and **rdv_semaphore** structs. Table 23-2 describes the predefined methods of the **locker** struct.

**Table 23-1—Semaphore Methods**

| Method | Description |
|---|---|
| up() | Increments the semaphore's value. Blocks if the value is already the maximum possible. |
| down() | Decrements the semaphore's value. Blocks if the value is already 0. |
| try_up() | Increments the semaphore's value. If the value is already the maximum possible, returns without blocking. |
| try_down() | Decrements the semaphore's value. If the value is already 0, returns without blocking. |
| set_value() | Sets the initial value of the semaphore. |
| get_value() | Returns the current value of the semaphore. |
| set_max_value() | Sets an upper limit to the possible value of the semaphore. |
| get_max_value() | Returns the maximum possible value. |

**Table 23-2—Locker Methods**

| Method | Description |
|---|---|
| lock() | The first TCM to call the **lock()** method of a field of type **locker** gets the lock and can continue execution. The execution of the other TCMs is blocked. |

**Table 23-2—Locker Methods *(continued)***

| Method | Description |
|---|---|
| release() | When a TCM that has the lock calls **release()**, control goes to the next TCM serviced by the scheduler that is waiting on the locker. The order in which the lock is granted is by a FIFO (First In First Out) order of client **lock()** requests. |

### See Also

— "How to Use the Semaphore Struct" on page 682
— "up() and down()" on page 682
— "try_up() and try_down()" on page 685
— "set_value() and get_value()" on page 687
— "set_max_value() and get_max_value()" on page 688
— "lock() and release()" on page 689

## 23.7 How to Use the Semaphore Struct

A field of type semaphore typically serves as a synchronization object between two types of TCMs: producer and consumer.

Any consumer TCM uses the predefined **down()** time-consuming method of the semaphore to gain control of a new resource managed by the semaphore. If no resources are available at the time **down()** is called, the consumer TCM is blocked until such a resource is available.

Any producer TCM uses the predefined **up()** time consuming method of the semaphore to increase the amount of available resources of the semaphore. This resource is made available for consumer TCMs. If the semaphore already contains the maximum number of resources at the time **up()** is called, the producer TCM is blocked until a semaphore resource is consumed.

The amount of available resources is zero by default but can be set otherwise using the **set_value()** method. The current amount of available resources can be obtained using the **get_value()** method.

There is a limit to the possible number of available resources. Typically, the maximum is MAX_INT, but it can be set to other values between 0 and MAX_INT using the **set_max_value()** method. The current limit for available resources can be obtained using the **get_max_value()** method.

Any producer TCM is blocked if the semaphore already holds the maximum number of available resources.

### 23.7.1 up() and down()

#### Purpose

Synchronize producer and consumer TCMs

#### Category

Predefined TCM of **semaphore** struct

### Syntax

***semaphore*.up()**

***semaphore*.down()**

Syntax example:

```
sem1.up();
sem1.down();
```

### Parameters

| | |
|---|---|
| *semaphore* | An expression of type **semaphore** or **rdv_semaphore** |

### Description

The **up()** time-consuming method increases the number of available resources of the semaphore by 1. If the number of available resources is already the maximum, the TCM is blocked. Blocked calls to **up()** are serviced according to their request order (on a First In First Out basis).

The **down()** time consuming method decreases the number of resources of the semaphore by 1. If no resources are available, the TCM is blocked. Blocked calls to **down()** are serviced according to their request order (on a First In First Out basis).

With an **rdv_semaphore**, **up()** and **down()** are blocked unless they coincide. The **down()** TCM always breaks the block first.

### Example 1

The following example shows how you can use a semaphore to handle concurrent requests for exclusive access from multiple clients in an orderly manner. In this example there are two client structs and one server struct. The server has a semaphore to ensure that all requests are granted and that there are no simultaneous grants.

When both clients issue a request at the same time the semaphore keeps track of the order of the requesting TCMs. The first client to issue a request is granted the single resource, making it unavailable to the other client. When this client is done with the resource, it uses the **up()** method of the semaphore to make the resource available to the other requesting client.

```
<'
struct server {
  event clk;
  sem: semaphore;

  run() is also {
    sem.set_value(1);
  };
};

struct client {
  id: string;
  s: server;

  handshake()@s.clk is { // A single-cycle delay handshake
```

```
        out(id,": Starting handshake at time ",sys.time);
        s.sem.down();
        out(id,": Granted at time ",sys.time);
        wait [1];
        out(id,": Releasing at time ",sys.time);
        s.sem.up();
      };

      run()is also {start handshake();};
    };

    extend sys {
      s: server;
      c1: client; keep {c1.s==s; c1.id=="client 1"};
      c2: client; keep {c2.s==s; c2.id=="client 2"};

      go()@any is {
        for i from 0 to 40 do {
          wait cycle;
          emit s.clk;
        };
        stop_run();
      };
      run()is also {start go()};
    };
    '>
```

### Result:

```
    client 1: Starting handshake at time 1
    client 1: Granted at time 1
    client 2: Starting handshake at time 1
    client 1: Releasing at time 2
    client 2: Granted at time 2
    client 2: Releasing at time 3
```

### Example 2

The following example shows how to use an **rdv_semaphore** to synchronize several reading TCMs that share a common input source.

In this example there is one writer and two readers. It takes the writer 3 cycles to write its data, and then it calls the **up()** method. When a reader appears, it calls the **down()** method and waits for the data to be ready. When both the writer and the reader are ready, at the two sides of the channel, the data exchange takes place: the reader breaks the block first, which allows it to read the data before it is overwritten by the writer. Then comes the writer and starts preparing for the next data exchange.

```
    <'
    extend sys {
        rdv_semaphore;
        reg :int;

        writer(id : int) @any is {
            var v : int = 0;
            while TRUE {
               out(time, " -> writer ", dec(id), ": start preparing data.");
               wait [3];
               v += 1;
```

```
                reg = v;
                out(time, " -> writer ", dec(id),
                    ": done writing to register [ v = ",hex(v),"].");
                rdv_semaphore.up();
            };
        };

        reader(id : int) @any is {
            while TRUE {
                out(time, " -> reader ", dec(id),
                    ": ready to read from register.");
                rdv_semaphore.down();
                out(time, " -> reader ", dec(id),
                    ": reading from register [ reg = ",hex(reg),"].");
                wait [id];
            };
        };

        run() is also {
           start writer(1);
           start reader(2);
           start reader(3);
        };

        event terminate is [10] exec {stop_run();};
    };
    '>
```

### Result:

```
    0 -> writer 1: start preparing data.
    0 -> reader 2: ready to read from register.
    0 -> reader 3: ready to read from register.
    3 -> writer 1: done writing to register [ v = 0x1].
    3 -> reader 2: reading from register [ reg = 0x1].
    3 -> writer 1: start preparing data.
    5 -> reader 2: ready to read from register.
    6 -> writer 1: done writing to register [ v = 0x2].
    6 -> reader 3: reading from register [ reg = 0x2].
    6 -> writer 1: start preparing data.
    9 -> reader 3: ready to read from register.
    9 -> writer 1: done writing to register [ v = 0x3].
    9 -> reader 2: reading from register [ reg = 0x3].
    9 -> writer 1: start preparing data.
```

### See Also

— “How to Use the Semaphore Struct” on page 682
— “try_up() and try_down()” on page 685
— “set_value() and get_value()” on page 687
— “set_max_value() and get_max_value()” on page 688

## 23.7.2 try_up() and try_down()

### Purpose

Synchronize producer and consumer methods

### Category

Predefined method of **semaphore** struct

### Syntax

***semaphore*****.try_up()**: bool

***semaphore*****.try_down()**: bool

Syntax example:

```
compute sem1.try_up();
compute sem1.try_down();
```

### Parameters

| | |
|---|---|
| *semaphore* | An expression of type **semaphore** or **rdv_semaphore**. |

### Description

The **try_up()** and **try_down()** methods try to increment or decrement the number of available resources by 1, respectively. If the number of available resources is already at its maximum or minimum respectively, these methods return immediately without any effect (in particular, no blocking). If the number of resources was changed, the returned value is TRUE. If the number of resources was not changed, the returned value is FALSE. The FIFO ordered of service of the semaphore is kept even when the **try_up()** and **try_down()** methods are involved. For example, a **try_up()** will never succeed if there are pending calls to **up()**.

NOTE— Being regular methods (not TCMs), **try_up()** and **try_down()** never generate a context switch.

### Example

The following example shows a driver that sends information at each clock. If there is valid data in ***reg*** that is protected by the semaphore reg_sem, it sends its contents. Otherwise, it sends 0.

```
driver()@any is {
    while TRUE {
        if sem.try_down() {
        send( reg );
        } else {
            send( 0 );
        };
        wait cycle;
    };
};
```

### See Also

— "How to Use the Semaphore Struct" on page 682
— "up() and down()" on page 682
— "set_value() and get_value()" on page 687
— "set_max_value() and get_max_value()" on page 688

### 23.7.3 set_value() and get_value()

#### Purpose

Set and get the number of available resources of a semaphore

#### Category

Predefined method of **semaphore** struct

#### Syntax

***semaphore*.set_value(*new_value*:** int**)**

***semaphore*.get_value()**: int

Syntax example:

```
sem1.set_value(7);
cur_value = sem1.get_value();
```

#### Parameters

| | |
|---|---|
| *new_value* | An expression of type signed int |

#### Description

The **set_value()** method sets the number of available resources of the semaphore. By default, a semaphores are initialized with zero available resources.

The new value must be a non-negative integer, no larger than MAX_INT. If the **set_max_value()** method of the struct was used, the new value must also be smaller or equal to the last setting of the maximum number of resources. If these conditions do not hold, a runtime error is issued.

**set_value()** cannot be called if either **up()** or **down()** was previously called. In such case, an erroris issued. Setting the value of an **rdv_semaphore** to something other than zero also results in a runtime error.

The **get_value()** method returns the current number of available resources of the semaphore.

#### Example

```
<'
extend sys {
    sem : semaphore;
    run() is also {
        sem.set_value(5);
        print sem.get_value();
    };
};
'>
```

#### Result:

```
sem.get_value() = 5
```

### See Also

— “How to Use the Semaphore Struct” on page 682
— “up() and down()” on page 682
— “try_up() and try_down()” on page 685
— “set_max_value() and get_max_value()” on page 688

## 23.7.4 set_max_value() and get_max_value()

### Purpose

Set and get the maximum number of available resources of a semaphore

### Category

Predefined method of **semaphore** struct

### Syntax

***semaphore*.set_max_value(*new_value*:** int**)**

***semaphore*.get_max_value()**: int

Syntax example:

```
sem1.set_max_value(17);
cur_max_value = sem1.get_max_value();
```

### Parameters

| | |
|---|---|
| *new_value* | An expression of type signed int |

### Description

The **set_max_value()** method sets the maximum number of available resources of the semaphore. By default, a semaphore is initialized with a maximum of MAX_INT available resources.

The new value must be a positive integer, no larger than MAX_INT. If **set_value()** was used, the new value must not be smaller than the number of available resources. If these conditions do not hold, a runtime error is issued.

The value of an **rdv_semaphore** is constantly zero. Therefore its default maximum value is zero, and it cannot be set to a value other than that. Trying to do so also results in a runtime error.

**set_max_value()** cannot be called if either **up()** or **down()** was previously called. In such case, an error is issued.

It is safer to invoke the **set_max_value()** method before any other semaphore method.

The **get_max_value()** method returns the current limit for available resources of the semaphore.

### Example

```
<'
```

```
extend sys {
    sem : semaphore;
    run() is also {
        sem.set_max_value(5);
        print sem.get_max_value();
    };
};
'>
```

**Result:**

```
sem.get_max_value() = 5
```

**See Also**

— “How to Use the Semaphore Struct” on page 682
— “up() and down()” on page 682
— “try_up() and try_down()” on page 685
— “set_value() and get_value()” on page 687

## 23.7.5 lock() and release()

**Purpose**

Control access to a shared resource

**Category**

Predefined TCM of **locker** struct

**Syntax**

*locker-exp*.lock()

*locker-exp*.release()

Syntax example:

```
lckr.lock();
lckr.release();
```

**Parameters**

| | |
|---|---|
| *locker-exp* | An expression of type **locker**. |

**Description**

**locker** is a predefined struct with two predefined methods, **lock()** and **release()**. These methods are TCMs.

Once a field is declared to be of type **locker**, that field can be used to control the execution of TCMs by making calls from the TCMs to ***locker*.lock()** and ***locker*.release()**.

If you call ***locker*.lock()** from multiple TCMs, the first TCM gets the lock and can continue execution. The execution of the other TCMs is blocked. Thus any resources that are shared between the TCMs will be available only to the TCM that gets the lock.

When a TCM calls **release()**, control goes to the next TCM serviced by the scheduler that is waiting on the locker. The order in which the lock is granted is by a FIFO (First In First Out) order of client **lock()** requests.

An ***e*** program uses non-preemptive scheduling, which means that thread execution is interrupted only when the executing thread reaches a **wait**, **sync**, TCM call, **release()**, or **lock()** request. This has two implications:

— You do not need to use locks unless the code to be executed between the **lock()** and the **release()** contains a **wait**, **sync**, or TCM call.
— Code that is not time-consuming and is used by multiple threads should be put in a regular method so no locks are needed.

NOTE—

— Calling **lock()** again before calling **release()** results in a deadlock. The TCM attempting to acquire the locker stops and waits for the locker to be released. This TCM never executes because it cannot release the locker. Naturally none of the other TCMs that wait for the locker is executed.
— The release of the locker must be explicit. If the locking thread ends (either normally or abnormally) without a call to **release()**, the locker is not released. Again, none of the other TCMs that wait for the locked is executed.

## Example 1

This example illustrates how the execution of two TCMs are controlled using a field of type **locker**.

```
<'
struct foo {
    lckr: locker;
    tcm1(id: uint) @sys.any is {
        all of {
            {
                lckr.lock();
                out("first branch got the lock");
                wait [2];
                out("first branch releases the lock");
                lckr.release();
            };
            {
                lckr.lock();
                out("second branch got the lock");
                wait [2];
                out("second branch releases the lock");
                lckr.release();
            };
        };
        wait [10] * cycle;
        out("******After 10 cycles*********");
        stop_run();
    };
    tcm2() @sys.any is {
        lckr.lock();
        out("tcm2 got the lock");
        wait [2] * cycle;
        out("tcm2 releases the lock");
        lckr.release();
    };
    run() is also {
        start tcm2();
        start tcm1(1);
    };
};

extend sys {
    foo_inst: foo;
};
'>
```

## Result

Note that tcm2 gets the lock first, then the first **all of** branch of tcm1 and finally the last **all of** branch of tcm1.

```
    tcm2 got the lock
    tcm2 releases the lock
    first branch got the lock
    first branch releases the lock
    second branch got the lock
    second branch releases the lock
    ******After 10 cycles*********
```

## Example 2

```
    <'
    struct st {
        lckr: locker;
        ftcm(id: uint) @sys.any is {
            for i from 0 to 2 do {
                lckr.lock();
                out("Id ", id, " got the resource");
                wait cycle;
                lckr.release();
            };
        };
        run() is also {
            start ftcm(1);
            start ftcm(2);
        };
    };
    extend sys {
        sti: st;
    };
    '>
```

## Result

```
    Doing setup ...
    Generating the test using seed 1...
    Starting the test ...
    Running the test ...
    Thread 1 got the resource
    Thread 2 got the resource
    Thread 1 got the resource
    Thread 2 got the resource
    Thread 1 got the resource
    Thread 2 got the resource
```

```
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

## 23.8 TCM Related Methods

The **scheduler** is a predefined struct containing methods that let you access active TCMs and terminate them.

A TCM that is invoked with a **start** action is a thread. If a started TCM calls other TCMs, those TCMs are considered subthreads of the started TCM thread. If a struct has more than one started TCM, each TCM runs on a separate, parallel thread. Each thread shares a unique identifier, or thread handle, with its subthreads. The thread handle is automatically assigned by the scheduler.

The following sections describe how to retrieve the handle for active threads:

— “get_current_handle()” on page 692
— “get_handles_by_name()” on page 693
— “get_handles_by_type()” on page 695

The following sections describe how to terminate active threads:

— “kill()” on page 696
— “terminate_branch()” on page 698
— “terminate_thread()” on page 699

### 23.8.1 get_current_handle()

#### Purpose

Obtain the handle of the current TCM

#### Category

Predefined method

#### Syntax

**scheduler.get_current_handle()**: thread handle

Syntax example:

```
out ("(started) I = ",i," in Thread " ,
    scheduler.get_current_handle(),".");
```

#### Description

Returns the handle of the currently running TCM. The handle is of the predefined type named “thread_handle”.

This method must ultimately be invoked from a TCM. You can call it from a non-TCM method, but that method must, at some point, be called from a TCM. That is, you can call **get_current_handle()** from a non-

TCM, which, in turn, is called from another non-TCM, and so on, but at the top of the chain of method calls must be a TCM.

NOTE— A runtime error is produced if **get_current_handle()** is called from within a regular (not time-consuming) method, if that method is not ultimately called from a TCM.

### Example

```
<'
struct sch {
    run() is also {
        start started_tcm(13);
        start started_tcm(29);
    };

    started_tcm(i:int) @sys.any is {
       out ("(started) I = ",i," in Thread " ,
          scheduler.get_current_handle(),".");
       called_tcm(i);
       wait [1];
       out ("(cont ..) I = ",i," in Thread " ,
       scheduler.get_current_handle(),". (same thread handle)");
    };

    called_tcm(i:int) @sys.any is {
       out ("(called ) I = ",i," in Thread " ,
       scheduler.get_current_handle(),". (same thread handle)");
    };
};

extend sys {
    pmi: sch;
};
'>
```

### Result

```
(started) I = 13 in Thread 1.
(called ) I = 13 in Thread 1. (same thread handle)
(started) I = 29 in Thread 2.
(called ) I = 29 in Thread 2. (same thread handle)
(cont ..) I = 13 in Thread 1. (same thread handle)
(cont ..) I = 29 in Thread 2. (same thread handle)
```

### See Also

— “get_handles_by_name()” on page 693
— “get_handles_by_type()” on page 695

## 23.8.2 get_handles_by_name()

### Purpose

Get list of thread handles on a struct instance basis

### Category

Predefined method

### Syntax

**scheduler.get_handles_by_name(*struct-inst***: exp, ***method-name***: string**)**: list of thread handle

Syntax example:

```
hs = scheduler.get_handles_by_name(a1,"yy");
```

### Parameters

| | |
|---|---|
| *struct-exp* | NULL, or an expression of type struct that specifies the owning struct instance for the started TCMs with the specified name. |
| *method-name* | NULL, or the name of a method in the specified struct, enclosed in double quotes. |

### Description

Returns a list of handles of all started TCMs of the specified name associated with the specified struct instance.

When the struct expression is NULL the resulting list contains handles for all the started TCMs of the given name. When the method name is NULL, the returned list contains thread handles for all the currently running threads for the specified struct. In the case when both parameters are NULL, the list of handles for all currently running threads is returned.

A thread is any started TCM. If a started TCM calls other TCMs, those TCMs are considered subthreads of the started TCM thread. If a struct has more than one started TCM, each TCM runs on a separate, parallel thread. Each thread shares a unique identifier, or thread handle, with its subthreads. The thread handle is automatically assigned by the scheduler.

### Example

```
<'
struct a {
    xx() @sys.any is {
        wait [1]*cycle;
    };
    yy() @sys.any is {
        wait [1]*cycle;
    };
};

extend sys {
    a1:a;
    a2:a;

    run() is also {
        var hs:list of thread_handle;
        start a1.xx();
        start a1.yy();
        start a2.yy();
        hs = scheduler.get_handles_by_name(a1,"yy");
```

```
            print hs;
            print scheduler.get_handles_by_name(NULL,NULL);
        };
    };
    '>
```

### Result

```
      hs =
    0.      2
      scheduler.get_handles_by_name(NULL,NULL) =
    0.      1
    1.      2
    2.      3
```

### See Also

— “get_current_handle()” on page 692
— “get_handles_by_type()” on page 695

## 23.8.3 get_handles_by_type()

### Purpose

Get list of thread handles on a struct type basis

### Category

Predefined method

### Syntax

**scheduler.get_handles_by_type(*struct-inst***: exp, ***method-name***: string**)**: list of thread handle

Syntax example:

```
    hs = scheduler.get_handles_by_type("a","yy");
```

### Parameters

| | |
|---|---|
| *struct-exp* | NULL, or an expression of type struct that specifies the owning struct type for the top-level TCMs of the specified method. |
| *method-name* | NULL, or the name of a method in the specified struct, enclosed in double quotes |

### Description

Returns handles to all TCMs associated with the specified struct type.

When the struct expression is NULL the resulting list contains handles for all the started TCMs of the given name. When the method name is NULL, the returned list contains thread handles for all the currently running threads for the specified struct. If both struct expression and method name are NULL, then all handles of all currently running threads are returned.

A thread is any started TCM. If a started TCM calls other TCMs, those TCMs are considered subthreads of the started TCM thread. If a struct has more than one started TCM, each TCM runs on a separate, parallel thread. Each thread shares a unique identifier, or thread handle, with its subthreads. The thread handle is automatically assigned by the scheduler.

### Example

```
<'
struct a {
    xx() @sys.any is {
        wait [1]*cycle;
    };
    yy() @sys.any is {
        wait [1]*cycle;
    };
};

extend sys {
    a1:a;
    a2:a;

    run() is also {
        var hs:list of thread_handle;
        start a1.xx();
        start a2.yy();
        start a2.yy();
        hs = scheduler.get_handles_by_type("a","yy");
        print hs;
        print scheduler.get_handles_by_type("a",NULL);
    };
};
'>
```

### Result

```
  hs =
0.      2
1.      3
  scheduler.get_handles_by_type("a",NULL) =
0.      1
1.      2
2.      3
```

### See Also

— “get_current_handle()” on page 692
— “get_handles_by_name()” on page 693

## 23.8.4 kill()

### Purpose

Kill a specified thread

### Category

Predefined method

### Syntax

**scheduler.kill(*handle***: thread handle**)**

Syntax example:

```
for each (h) in hs {
    {scheduler.kill(h)};
};
```

### Parameters

| | |
|---|---|
| *handle* | The handle for the thread, as returned by **scheduler.get_current_handle()**, **scheduler.get_handles_by_name()**, or **scheduler.get_handles_by_type()**. |

### Description

Kills a started TCM (a thread) and any TCMs that it has called (its subthreads). A killed method cannot be revived.

### Notes

- Killing a method before it releases an active lock can result in a dead lock.
- A thread cannot kill itself. Use "terminate_thread()" on page 699 instead.

### Example

```
<'
struct p_agent {
    killer_tcm() @sys.any is {
        wait [1]*cycle;
        var hs :=
            scheduler.get_handles_by_type("p_agent","send");
        for each (h) in hs {
            out("Killing thread ",h);
            {scheduler.kill(h)};
        };
    };

    send() @sys.any is {
        wait [5]*cycle;
        out ("this line is never executed. ");
    };
};

extend sys {
    x:p_agent;
    run() is also {
        start x.killer_tcm();
        start x.send();
        start x.send();
        start x.send();
    };
};
'>
```

### Result

```
Killing thread 2
Killing thread 3
Killing thread 4
```

### See Also

— “terminate_branch()” on page 698
— “terminate_thread()” on page 699

## 23.8.5 terminate_branch()

### Purpose

Terminate a specific branch in a **first of** action

### Category

Predefined method

### Syntax

scheduler.terminate_branch()

Syntax example:

```
scheduler.terminate_branch();
```

### Description

This method can be used only within a **first of** action to terminate the branch. When a branch is terminated using this method, the rest of the branches within the **first of** action remain active.

### Example

The TCM “monitor()” in the example below begins several threads. Each waits for a sequence of events. Under some conditions, some of the sequences should be halted.

```
<'
extend sys {
    monitor() @sys.any is {
        wait until rise('top.started');
        first of {
            {
                wait [4] * cycle;
                if 'top.out_of_sync' == 1 then {
                    out("### Went out of sync");
                    scheduler.terminate_branch();
                };
                wait until rise('top.ended');
                out("Normal end");
            };
            {
                wait until rise('top.aborted');
                out("Aborted");
            };
        };
        // continue monitoring...
        stop_run();
    };

    run() is also {
        start monitor();
    };
};
extend sys {
    drive() @sys.any is {
        wait [2] * cycle;
        'top.started' = 1;
        wait [5] * cycle;
        'top.out_of_sync' = 1;
    };

    run() is also {
        start drive();
    };
};
'>
```

### Result

```
### Went out of sync
```

### See Also

— “kill()” on page 696
— “terminate_thread()” on page 699

## 23.8.6 terminate_thread()

### Purpose

Terminate the current thread

### Category

Predefined method

### Syntax

scheduler.terminate_thread()

Syntax example:

```
scheduler.terminate_thread();
```

### Description

Terminates the current thread immediately, not at the end of the current tick. To terminate the current thread at the end of the current tick, use **quit()**.

A thread is any started TCM. If a started TCM calls other TCMs, those TCMs are considered subthreads of the started TCM thread. If a struct has more than one started TCM, each TCM runs on a separate, parallel thread. Each thread shares a unique identifier, or thread handle, with its subthreads. The thread handle is automatically assigned by the scheduler.

### Example

The TCM inject() in the example below assigns the DUT signals. There are some conditions during the run that indicate that the injection should stop.

```
<'
extend sys {
    inject() @sys.any is {
        'top.data' = 1;
        wait [10] * cycle;
        // continue reading/writing...
        if ('top.status' == 0) then {
            out("'top.status' == 0, Stop injection");
            scheduler.terminate_thread();
        };
        // continue reading/writing...
    };

    run() is also {
        start inject();
    };
};
'>
```

### Result

```
'top.status' == 0, Stop injection
```

### See Also

— “The quit()  Method of any_struct” on page 659
— “kill()” on page 696
— “terminate_branch()” on page 698

## 23.9 Coverage Methods

The **covers** struct is a predefined struct containing methods that you use for coverage and coverage grading. With the exception of the **set_external_cover()** and **write_cover_file()** methods, all of these methods are methods of the **covers** struct.

— “include_tests()” on page 701
— “set_weight()” on page 702
— “set_at_least()” on page 702
— “set_cover()” on page 703
— “get_contributing_runs()” on page 705
— “get_unique_buckets()” on page 706
— “set_external_cover()” on page 707
— “write_cover_file()” on page 708
— “get_overall_grade()” on page 709
— “get_ecov_name()” on page 710

— "get_test_name()" on page 711
— "get_seed()" on page 711

### See Also

The following section describes another predefined method you use for coverage:

— "set_check()" on page 448

## 23.9.1 include_tests()

### Purpose

Specify which test runs coverage information will be displayed for.

### Category

Predefined method

### Syntax

**covers.include_tests(*full-run-name*:** string**, *bool*:** exp**)**

Syntax example:

```
covers.include_tests("tests_A:run_A_10", TRUE);
```

### Parameters

| | |
|---|---|
| *full-run-name* | The name of the test to include or exclude. |
| *bool-exp* | Set to TRUE to include the specified test, FALSE to exclude it. |

### Description

This method allows you to specify which test runs you want to see coverage information for.

If you are reading in .ecov files to load coverage information, this method should be called only after the .ecov files have been read.

### Example

The following example shows several ways to specify test runs for display by **show coverage**.

```
<'
extend sys {
    setup() is also {
        covers.include_tests("tests_A:...", FALSE);
        covers.include_tests("...crc_test", TRUE);
        covers.include_tests("/.*crc_test.*/", TRUE);
    };
};
'>
```

### 23.9.2 set_weight()

#### Purpose

Specify the coverage grading weight of a group or item

#### Category

Predefined method

#### Syntax

**covers.set_weight(*entity-name*:** string**, *value*:** int**, *bool*:** exp**)**

Syntax example:

```
covers.set_weight("inst.done", 4, FALSE);
```

#### Parameters

| | |
|---|---|
| *entity-name* | The group or item to set the weight for. May include wild cards. |
| *value* | The integer weight value to set. |
| *bool* | When this is FALSE, it changes the weights of all matching groups or items to ***value***. When this is TRUE, the weights of all matching groups or items are multiplied by ***value***. |

#### Description

Coverage grading uses weights to emphasize the affect of particular groups or items relative to others. The weights can be specified in the coverage group or item definitions. This method sets the weights procedurally. It overrides the weights set in the group or item definitions. Weights can be set explicitly, or can be multiplied by a given value.

If you are reading in .ecov files to load coverage information, this method should be called only after the .ecov files have been read.

#### See Also

— "Defining Coverage Groups: cover" on page 373
— "item" on page 378
— "set_at_least()" on page 702

### 23.9.3 set_at_least()

#### Purpose

Set the minimum number of samples needed to fill a bucket

#### Category

Predefined method

### Syntax

**covers.set_at_least(*entity-name*:** string**, *value*:** int**, *exp*:** bool**)**

Syntax example:

```
covers.set_at_least("inst.done", 4, FALSE);
```

### Parameters

| | |
|---|---|
| *entity-name* | The group or item to set the at_least number for. May include wild cards. |
| *value* | The "at-least" integer value to set. |
| *bool* | When this is FALSE, it sets the "at-least" number for all matching items to ***value***. When this is TRUE, it multiplies the "at-least" number for all matching items by ***value***. |

### Description

The minimum number of samples required to fill a bucket can be set in the coverage group or item definitions. This method can be used to set the number procedurally. It overrides the numbers set in the group or item definitions.

If the ***entity-name*** is a coverage group name, all items in the group are affected. If the ***entity-name*** matches items within a coverage group, only those items are affected.

If you are reading in .ecov files to load coverage information, this method should be called only after the .ecov files have been read.

### See Also

— "Defining Coverage Groups: cover" on page 373
— "item" on page 378

## 23.9.4 set_cover()

### Purpose

Turns coverage data collection and display on or off for specified items or events

### Category

Predefined method

### Syntax

**covers.set_cover(*item*|*event*:** string**, *bool*:** exp**)**

Syntax example:

```
covers.set_cover("packet.*", FALSE);
```

## Parameters

| | |
|---|---|
| *item* | A string, enclosed in double quotes, specifying the coverage item you want to turn on or off. This may include wild cards. |
| *event* | A string, enclosed in double quotes, specifying the event you want to turn on or off. This may include wild cards.<br><br>Enter the name of the event using the following syntax:<br><br>session.events.*struct_type__event_name*<br><br>where the struct type and the event name are separated by two underscores. Wild cards may be used.<br><br>If you enter only one name, it is treated as a struct type, and the method affects all events in that struct type. |
| *bool* | Set to TRUE to turn on coverage for the item or FALSE to turn coverage off. |

## Description

By default, coverage data is collected for all defined coverage items and groups, and for all user-defined events. This method selectively turns data collection on or off for specified items, groups, or events.

After coverage data has been collected and written by a test or set of tests, this method can be used to selectively turn on or off the display of the coverage data for specified items, groups, or events.

Although this method can be used to filter samples during periods in which they are not valid, for performance reasons, filtering should be done using **when** subtypes instead.

Additionally, if the test ends while coverage collection is turned off by **set_cover()** for one or more coverage groups, then **set_cover()** must be called again to re-enable sampling before the .ecov file is written, in order to include the previously collected samples for those groups in the .ecov file.

## Example 1

The following example turns off coverage data collection for all items in all coverage groups defined in the “inst” struct, and then turns back on the collection of data for the “len” item in the “done” group in that struct.

```
<'
extend sys {
    setup() is also {
        covers.set_cover("inst.*.*", FALSE);
        covers.set_cover("inst.done.len", TRUE);
    };
};
'>
```

## Example 2

The following example turns off coverage data collection for an event named “my_event” defined in the “inst” struct.

```
<'
extend sys {
    setup() is also {
        covers.set_cover("session.events.inst__my_event", FALSE);
    };
};
'>
```

### Example 3

The following example turns off coverage data collection for all events in the struct named my_struct:.

```
<'
extend sys {
    setup() is also {
        covers.set_cover("my_struct", FALSE);
    };
};
'>
```

## 23.9.5 get_contributing_runs()

### Purpose

Return a list of the test runs that contributed samples to a bucket

### Category

Predefined method

### Syntax

**covers.get_contributing_runs(*item-name***: string**, *bucket-name***: string**)**: list of string

Syntax example:

```
bkl=covers.get_contributing_runs("inst.done.len", "[0..4]");
```

### Parameters

| | |
|---|---|
| *item-name* | A string, enclosed in double quotes, specifying the coverage item that contains ***bucket-name***. |
| *bucket-name* | A string, enclosed in double quotes, specifying the bucket for which contributing test run names are to be listed. |

### Description

This method returns a list of strings that are the full run names of the test runs that placed samples in a specified bucket. For a cross item, the ***bucket-name*** may be a bucket of any level, with the bucket set names separated by slashes, for example: “ADD/REG1” or “ADD/REG1/[0xC0..0xCF]”.

### Example

The following example shows several ways to list test runs that put samples in particular buckets.

```
<'
type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
type cpu_reg: [reg0, reg1, reg2, reg3];
struct inst {
    opcode: cpu_opcode;
    op1: cpu_reg;
    op2: byte;
    event done;
    cover done is {
        item opcode;
        item op1;
        item op2;
        cross opcode, op1 using name = opcode_op1;
    };
};
extend sys {
  pre_generate() is also {
    var bl_1: list of string =
      covers.get_contributing_runs("inst.done.opcode",
          "ADD");
    var bl_2: list of string =
      covers.get_contributing_runs("inst.done.opcode_op1",
          "ADD/reg2");
    var bl_3: list of string =
      covers.get_contributing_runs("inst.done.opcode_op1",
          "SUB");
  };
};
'>
```

### 23.9.6 get_unique_buckets()

#### Purpose

Return a list of the names of unique buckets from specific tests.

#### Category

Predefined method

#### Syntax

**covers.get_unique_buckets(*file-name*:** string**)**: list of string

Syntax example:

```
print covers.get_unique_buckets("test_rx")
```

#### Parameters

| | |
|---|---|
| *file-name* | A string, enclosed in double quotes, specifying the coverage database files for which you want to see unique buckets. You cannot use wild cards in the file name. |

### Description

A unique bucket is a bucket that is covered by only one test. This method reports, for each specified test, the full names of its unique buckets, if there are any.

NOTE— You must rank the tests with the before calling **covers.get_unique_buckets()**.

### Example

The following example shows how to display a list of unique buckets that are covered by a test. The results of the test ranking show that the test "CPU_tst2_1" has three unique buckets. Passing this test name to **covers.get_unique_buckets()** retrieves the names of the buckets.

```
Coverage Ranking Report
========================

Coverage Test-Ranking report
============================

Command:              rank cover -sort_only *.*.*
Grading_formula:      linear
Ranking Cost:         cpu_time
At least multiplier:  1
Number of tests:      5
Maximal Grade:        0.87
Note:                 All test grades are given as precentage from Maximal
Grade

The Sorted Test List:
-----------------------
                                Cum.   Rel.   Cum.          Abs.   Effic-
Num Test name                   Grade  Grade  Cost   Cost   Grade  iency  UB
--- ------------------------- ------ ------ ----- ----- ---- ------ ----
1   CPU_tst3_7                  N/A    N/A    N/A    7       0.86   0.74  0
2   CPU_tst4_7                  N/A    N/A    N/A    7       0.86   0.74  0
3   CPU_tst8_7                  N/A    N/A    N/A    7       0.86   0.74  0
4   CPU_tst2_1                  N/A    N/A    N/A    7       0.85   0.72  3
5   CPU_tst1_1                  N/A    N/A    N/A    6       0.66   0.66  0
--- ------------------------- ------ ------ ------ ------ ------
```
**covers.get_unique_buckets("CPU_tst2_1")**
```
  covers.get_unique_buckets("CPU_tst2_1") =
0.      "instr.start_drv_DUT.cross__opcode__carry:      ANDI/1"
1.      "instr.start_drv_DUT.cross__opcode__carry:      XORI/1"
2.      "instr.start_drv_DUT.cross__opcode__carry:      NOP/1"
```

### See Also

## 23.9.7 set_external_cover()

### Purpose

Enable or disable the import and display of SureCov data

### Category

Predefined method

### Syntax

set_external_cover("surecov", *bool*);

Syntax example:

```
covers.set_external_cover("surecov", FALSE);
```

### Parameters

| | |
|---|---|
| *bool* | TRUE, the default, enables the import of all SureCov data. FALSE disables the import. |

### Description

By default, coverage data is collected for all defined SureCov coverage items and groups. This method disables all import of SureCov data, even if SureCov coverage groups are defined. If disabled, all additional behavior regarding SureCov is cancelled and the SureCov groups are not displayed.

You can call **set_external_cover()** at any time during a run.

### Example

This example shows how to turn off the import and display of SureCov data.

```
<'
extend sys {
    setup() is also {
        covers.set_external_cover("surecov", FALSE);
    };
};
'>
```

### See Also

— "cover ... using external=surecov" on page 408

## 23.9.8 write_cover_file()

### Purpose

Write the coverage results during a test

### Category

Predefined method

### Syntax

write_cover_file();

Syntax example:

```
write_cover_file();
```

### Description

This method writes the coverage results .ecov file during a test run. It can only be invoked during a test, not before the run starts nor after it ends.

The coverage file written by this method does not contain the **session.end_of_test** or **session.events** coverage groups.

### Example

This example writes the current coverage results to the .ecov file whenever the event named sys.cntr is emitted.

```
<'
struct top {
    event wr_cov is @sys.cntr;
    on wr_cov {
        write_cover_file();
    };
};
'>
```

## 23.9.9 get_overall_grade()

### Purpose

Return the normalized overall coverage grade

### Category

Predefined method

### Syntax

**covers.get_overall_grade()**: int**;**

Syntax example:

```
grade = covers.get_overall_grade();
```

### Description

This method returns an integer that represents the overall coverage grade for the current coverage results. Since ***e*** does not handle floating point types, the value is a normalized value between 1 and 100M. To obtain a value equivalent to the overall grade, divide the returned value by 100M.

### Example

```
<'
struct top{
    event e;
    a: uint(bits: 4);
    b: uint(bits: 4);
    cover e is {
        item a;
```

```
            item b;
        };
        run() is also {emit e;};
    };

    extend sys {
        toplist[10]: list of top;
        finalize() is also {
            var grade: int;
            grade = covers.get_overall_grade();
            print grade;
        };
    };
    '>
```

### See Also

— "get_ecov_name()" on page 710
— "get_test_name()" on page 711
— "get_seed()" on page 711

## 23.9.10 get_ecov_name()

### Purpose

Return the name of the .ecov file

### Category

Predefined method

### Syntax

**covers.get_ecov_name()**: string**;**

Syntax example:

```
    ecov_file = covers.get_ecov_name();
```

### Description

This method returns the name of the .ecov file in which the current coverage results will be stored.

### Example

```
    <'
    extend sys {
        finalize() is also {
            var ecov_file: string;
            ecov_file = covers.get_ecov_name();
            print ecov_file;
        };
    };
    '>
```

### See Also

— "get_overall_grade()" on page 709
— "get_test_name()" on page 711
— "get_seed()" on page 711

## 23.9.11 get_test_name()

### Purpose

Return the name of the current test

### Category

Predefined method

### Syntax

**covers.get_test_name()**: string**;**

Syntax example:

```
ecov_file = covers.get_test_name();
```

### Description

This method returns the identifier of the current test run.

### Example

```
<'
extend sys {
    finalize() is also {
        var test_id: string;
        test_id = covers.get_test_name();
        print test_id;
    };
};
'>
```

### See Also

— "get_overall_grade()" on page 709
— "get_ecov_name()" on page 710
— "get_seed()" on page 711

## 23.9.12 get_seed()

### Purpose

Return the value of the seed for the current test

### Category

Predefined method

### Syntax

**covers.get_seed()**: int

Syntax example:

```
seed_val= covers.get_seed();
```

### Description

This method returns the current test seed.

### Example

```
<'
extend sys {
    finalize() is also {
        var test_seed: int;
        test_seed = covers.get_seed();
        print test_seed;
    };
};
'>
```

### See Also

— “get_overall_grade()” on page 709
— “get_ecov_name()” on page 710
— “get_test_name()” on page 711

# 24 Predefined Routines Library

Predefined routines are ***e*** macros that look like methods. The distinguishing characteristics of predefined routines are:

— They are not associated with any particular struct
— They share the same name space for user-defined routines and **global** methods
— They cannot be modified or extended with the **is only**, **is also** or **is first** constructs
— They have no debug information

The following sections describe the predefined routines:

— "Deep Copy and Compare Routines" on page 713
— "Arithmetic Routines" on page 721
— "Bitwise Routines" on page 730
—
— "String Routines" on page 736
— "Output Routines" on page 762
— "Configuration Routines" on page 766
— "OS Interface Routines" on page 785
— "On-the-Fly Garbage Collection Routine: do_otf_gc()" on page 792
— "Calling Predefined Routines: routine()" on page 793

### See Also

— "Invoking Methods" on page 474
— Chapter 23, "Predefined Methods Library"
— Chapter 26, "Predefined File Routines Library"

## 24.1 Deep Copy and Compare Routines

The following routines perform recursive copies and comparisons of nested structs and lists:

— "deep_copy()" on page 713
— "deep_compare()" on page 716
— "deep_compare_physical()" on page 720

### 24.1.1 deep_copy()

#### Purpose

Make a recursive copy of a struct and its descendants

#### Category

Predefined routine

#### Syntax

**deep_copy(*struct-inst*:** exp**)**: struct instance

Syntax example:

```
var pmv: packet = deep_copy(sys.pmi);
```

## Parameters

| | |
|---|---|
| *struct-inst* | An expression that returns a struct instance. |

## Description

Returns a deep, recursive copy of the struct instance. This routine descends recursively through the fields of a struct and its descendants, copying each field by value, copying it by reference, or ignoring it, depending on the **deep_copy** attribute set for that field.

The return type of **deep_copy()** is the same as the declared type of the struct instance.

The following table details how the copy is made, depending on the type of the field and the **deep_copy** attribute (**normal**, **reference**, **ignore**) set for that field. For an example of how field attributes affect deep_copy(), see "attribute field" on page 139.

| Field Type/ Attribute | Normal | Reference | Ignore |
|---|---|---|---|
| scalar | The new field holds a copy of the original value. | The new field holds a copy of the original value. | The new field holds a copy of the original value. |
| string | The new field holds a copy of the original value. | The new field holds a copy of the original value. | The new field holds a copy of the original value. |
| scalar list | A new list is allocated with the same size and same elements as the original list. | The new list field holds a copy of the original list pointer. * | A new list is allocated with zero size. |
| struct | A new struct instance with the same type as the original struct is allocated. Each field is copied or ignored, depending on its **deep_copy** attribute. | The new struct field holds a pointer to the original struct. * | A new struct instance is allocated and it is NULL. |
| list of structs | A new list is allocated with the same number of elements as the original list.<br><br>New struct instances are also allocated and each field in each struct is copied or ignored, depending on its **deep_copy** attribute. | The new list field holds a copy of the original list pointer. * | A new list is allocated with zero size. |

*If the list or struct that is pointed to is duplicated (possibly because another field with a normal attribute is also pointing to it) the pointer in this field is updated to point to the new instance. This duplication applies only to instances duplicated by the **deep_copy()** itself, and not duplications made by the extended/overridden **copy()** method.

## Notes

— A deep copy of a scalar field (numeric, boolean, enumerated) or a string field is the same as a shallow copy performed by a call to **copy()**.

— A struct or list is duplicated no more than once during a single call to **deep_copy()**.
If there is more than one reference to a struct or list instance, and that instance is duplicated by the call to **deep_copy()**, every field that referred to the original instance is updated to point to the new instance.

— The **copy()** method of the struct is called by **deep_copy()**.
The struct's **copy()** method is called before its descendants are deep copied. If the default **copy()** method is overwritten or extended, this new version of the method is used.

— You should apply the **reference** attribute to fields that store shared data and to fields that are back-pointers (pointers to the parent struct). Shared data in this context means data shared between objects inside the deep copy graph and objects outside the deep copy graph. A deep copy graph is the imaginary directed graph created by traversing the structs and lists duplicated, where its nodes are the structs or lists, and its edges are deep references to other structs or lists.

## Example

```
<'
struct packet {
    header: header;
    data[10]: list of byte;
    protocol: [ATM, ETH, IEEE];
};
struct header {
    code: uint;
};
extend sys {
    pmi: packet;
    m1() is {
        var pmv: packet = deep_copy(sys.pmi);
        pmv.data[0] = 0xff;
        pmv.header.code = 0xaa;
        pmv.protocol = IEEE;
        print pmi.data[0], pmi.header.code, pmi.protocol;
        print pmv.data[0], pmv.header.code, pmv.protocol;
    };
};
'>
```

## Result

This example shows that any changes in value to lists and structs contained in the copied struct instance (pmv) are not propagated to the original struct instance (pmi) because the struct has been recursively duplicated.

**sys.m()**
```
  pmi.data[0] = 0x1b
  pmi.header.code = 0x43dfc545
  pmi.protocol = ATM
  pmv.data[0] = 0xff
```

```
    pmv.header.code = 0xaa
    pmv.protocol = IEEE
```

### See Also

- "The copy() Method of any_struct" on page 647
- "deep_compare()" on page 716
- "deep_compare_physical()" on page 720
- "attribute field" on page 139

## 24.1.2 deep_compare()

### Purpose

Perform a recursive comparison of two struct instances

### Category

Predefined routine

### Syntax

**deep_compare(*struct-inst1*: exp, *struct-inst2*: exp, *max-diffs*: int)**: list of string

Syntax example:

```
    var diff: list of string = deep_compare(pmi[0], pmi[1], 100);
```

### Parameters

| | |
|---|---|
| ***struct-inst1***, ***struct-inst2*** | An expression returning a struct instance. |
| *max-diffs* | An integer representing the maximum number of differences you want reported. |

### Description

Returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a struct and its descendants, comparing each field or ignoring it, depending on the **deep_compare** attribute set for that field.

The two struct instances are "deep equal" if the returned list is empty.

"Deep equal" is defined as follows:

- Two struct instances are deep equal if they are of the same type and all their fields are deep equal.
- Two scalar fields are deep equal if an equality operation applied to them is TRUE.
- Two list instances are deep equal if they are of the same size and all their items are deep equal.

Topology is taken into account. If two non-scalar instances are not in the same location in the deep compare graphs, they are not equal. A deep compare graph is the imaginary directed graph created by traversing the structs and lists compared, where its nodes are the structs or lists, and its edges are deep references to other structs or lists.

The following table details the differences that are reported, depending on the type of the field and the **deep_compare** attribute (**normal**, **reference**, **ignore**) set for that field. For an example of how field attributes affect deep_copy(), see "attribute field" on page 139.

| Field Type/ Attribute | Normal | Reference | Ignore |
|---|---|---|---|
| scalar | Their values, if different, are reported. | Their values, if different, are reported. | The fields are not compared. |
| string | Their values, if different, are reported. | Their values, if different, are reported. | The fields are not compared. |
| scalar list | Their sizes, if different, are reported. All items in the smaller list are compared to those in the longer list and their differences are reported. | The fields are equal if their addresses are the same. The items are not compared. | The fields are not compared. |
| struct | If two structs are not of the same type, their type difference is reported. Also, any differences in common fields is reported. [*]<br><br>If two structs are of the same type, every field difference is reported. | The fields are equal if they point to the same struct instance. [†]<br><br>If the fields do not point to the same instance, only the addresses are reported as different; the data is not compared. | The fields are not compared and no differences for them or their descendants are reported. |
| list of structs | Their sizes, if different, are reported. All structs in the smaller list are deep compared to those in the longer list and their differences are reported. | The fields are equal if their addresses are the same and they point to the same struct instance. [†] | The fields are not compared and no differences for them or their descendants are reported. |

[*]Two fields are considered common only if the two structs are the same type, if they are both subtypes of the same base type, or if one is a base type of the other.

[†]If the reference points inside the deep compare graph, a limited topological equivalence check is performed, not just an address comparison.

## Difference String Format

The difference string is in the following format:

```
Differences between <inst1-id> and <inst2-id>
---------------------------------------------
<path>:    <inst1-value>   !=   <inst2-value>
```

| | |
|---|---|
| *path* | A list of field names separated by periods (.) from (and not including) the struct instances being compared to the field with the difference. |
| *value* | For scalar field differences, ***value*** is the result of **out(*field*)**.<br><br>For struct field type differences, **type()** is appended to the path and ***value*** is the result of **out(*field*.type())**.<br><br>For list field size differences, **size()** is appended to the path and ***value*** is the result of **out(*field*.size())**.<br><br>For a shallow comparison of struct fields that point outside the deep compare graph, ***value*** is the struct address.<br><br>For a comparison of struct fields that point to different locations in the deep compare graphs (topological difference), ***value*** is **struct#** appended to an index representing its location in the deep compare graph. |

NOTE— The same two struct instances or the same two list instances are not compared more than once during a single call to **deep_compare()**.

## Example

This example uses **deep_compare()** to show the differences between copying nested structs by reference (with **copy()**) and copying nested structs by allocation (with **deep_copy()**).

```
<'
struct a {
    x: byte;
};
struct b {
    as: list of a;
    keep as.size() in [2 .. 3];
};
struct c {
    bs: list of b;
    keep bs.size() in [2 .. 3];

    print() is {
        var idx: uint;
        for each b (b) in bs {
            idx = index;
            for each a (a) in b.as {
                out ("b[",idx,"] - a[",index,"] : ",
                hex(bs[idx].as[index].x));
            };
        };
    };
};

extend sys {
    c1: c;
    post_generate() is also {
        var c2: c = new;
```

```
            var c3: c = new;

            out("C1:");
            c1.print();

            // copy by allocation

            out("C2:");
            c2 = deep_copy(c1);
            c2.print();
            print deep_compare(c1,c2,20);

            // copy by reference

            out("C3:");
            c3 = c1.copy();
            c3.print();
            print deep_compare(c1,c3,20);

            // demonstrate difference - change original

            out("Change C1:");
            c1.bs[0].as[0].x = 0;
            c1.print();
            print deep_compare(c1,c2,20);
            print deep_compare(c1,c3,20);
        };
    };
    '>
```

## Result

The results show the differences between the two ways of copying. The c2 instance is copied by **deep_copy()**, so when the value of x is changed in the original instance, c1, the value of x in c2 is not changed. In contrast, the value of x in c3 is changed because c3 was copied by reference. Note that “deep_compare() = empty” means that the two struct instances are deep equal.

```
C1:
b[0x0] - a[0x0] : 0x75
b[0x0] - a[0x1] : 0x8a
b[0x1] - a[0x0] : 0x0a
b[0x1] - a[0x1] : 0x2a
C2:
b[0x0] - a[0x0] : 0x75
b[0x0] - a[0x1] : 0x8a
b[0x1] - a[0x0] : 0x0a
b[0x1] - a[0x1] : 0x2a
  deep_compare(c1,c2,20) = (empty)
C3:
b[0x0] - a[0x0] : 0x75
b[0x0] - a[0x1] : 0x8a
b[0x1] - a[0x0] : 0x0a
b[0x1] - a[0x1] : 0x2a
  deep_compare(c1,c3,20) = (empty)
Change C1:
b[0x0] - a[0x0] : 0x00
b[0x0] - a[0x1] : 0x8a
b[0x1] - a[0x0] : 0x0a
```

```
b[0x1] - a[0x1] : 0x2a
  deep_compare(c1,c2,20) =
0.      "Differences between c-@0 and c-@1
---------------------------------------------------"
1.      "bs[0x0].as[0x0].x:    0x00   !=   0x75"
  deep_compare(c1,c3,20) = (empty)
```

### See Also

- — "deep_compare_physical()" on page 720
- — "deep_copy()" on page 713
- — "attribute field" on page 139

## 24.1.3 deep_compare_physical()

### Purpose

Perform a recursive comparison of the physical fields of two struct instances

### Category

Predefined routine

### Syntax

**deep_compare_physical(*struct-inst1*:** exp**, *struct-inst2*:** exp**, *max-diffs*:** int**)**: list of string

Syntax example:

```
var diff: list of string = deep_compare_physical(pmi[0],
    pmi[1], 100);
```

### Parameters

| | |
|---|---|
| ***struct-inst1***, ***struct-inst2*** | An expression returning a struct instance. |
| *max-diffs* | An integer representing the maximum number of differences you want reported. |

### Description

Returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a struct and its descendants, ignoring all non-physical fields and comparing each physical field or ignoring it, depending on the **deep_compare_physical** attribute set for that field.

This routine is the same as the **deep_compare()** routine except that only physical fields (indicated with the % operator prefixed to the field name) are compared.

The two struct instances are "deep equal" if the returned list is empty.

"Deep equal" is defined as follows:

- — Two struct instances are deep equal if they are of the same type and all their fields are deep equal.
- — Two scalar fields are deep equal if an equality operation applied to them is TRUE.

— Two list instances are deep equal if they are of the same size and all their items are deep equal.

NOTE— Adding a field under a when construct causes the parent type and the when subtype to be different, even if the field added under the when is a virtual field.

### Example

```
<'
struct packet {
    %header: header;
    %data[10] :list of byte;
    protocol: [ATM, ETH, IEEE];
};
struct header {
    %code: uint;
};
extend sys {
    pmi[2]: list of packet;
    post_generate() is also {
    var diff: list of string = deep_compare_physical(pmi[0],
      pmi[1], 100);
        if (diff.size() != 0) {
            out(diff);
        };
    };
};
'>
```

### Result

This example shows the differences between the physical fields of the packet instances. The differences between the protocol fields are not reported.

```
Differences between packet-@0 and packet-@1
--------------------------------------------------
header.code:    1138738501   !=   3071222567
data[0]:    27   !=   37
data[1]:    132   !=   56
data[2]:    163   !=   69
data[3]:    71   !=   236
data[4]:    178   !=   120
data[5]:    230   !=   100
data[6]:    116   !=   239
data[7]:    241   !=   216
data[8]:    238   !=   144
data[9]:    150   !=   253
```

### See Also

— “deep_compare()” on page 716
— “deep_copy()” on page 713
— “attribute field” on page 139

## 24.2 Arithmetic Routines

The following sections describe the predefined arithmetic routines in *e*:

### 24.2.1 min()

#### Purpose

Get the minimum of two numeric values

#### Category

Pseudo method

#### Syntax

**min(*x***: numeric-type**, *y***: numeric-type**)**: numeric-type

Syntax example:

```
print min((x + 5), y);
```

#### Parameters

| | |
|---|---|
| *x* | A numeric expression. |
| *y* | A numeric expression. |

#### Description

Returns the smaller of the two numeric values.

#### Example

```
<'
extend sys {
    m1() is {
        var x:int = 5;
        var y: int = 123;
        print min((x + 5), y);
    };
};
'>
```

#### Result

**sys.m1()**
```
  min((x + 5), y) = 10
```

### See Also

— "Arithmetic Routines" on page 721

## 24.2.2 max()

### Purpose

Get the maximum of two numeric values

### Category

Pseudo method

### Syntax

**max(*x***: numeric-type**, *y***: numeric-type**)**: numeric-type

Syntax example:

```
print max((x + 5), y);
```

### Parameters

| | |
|---|---|
| *x* | A numeric expression. |
| *y* | A numeric expression. |

### Description

Returns the larger of the two numeric values.

### Example

```
<'
extend sys {
    m1() is {
        var x:int = 5;
        var y: int = 123;
        print max((x + 5), y);
    };
};
'>
```

### Result

**sys.m1()**
```
  max((x + 5), y) = 123
```

### See Also

— "Arithmetic Routines" on page 721

### 24.2.3 abs()

#### Purpose

Get the absolute value

#### Category

Routine

#### Syntax

**abs(*x***: numeric-type**)**: numeric-type

Syntax example:

```
print abs(x);
```

#### Parameters

| | |
|---|---|
| *x* | A numeric expression. |

#### Description

Returns the absolute value of the expression.

#### Example

```
<'
extend sys {
    m1() is {
        var x: int = -5;
        print abs(x);
    };
};
'>
```

#### Result

**sys.m1()**
```
  abs(x) = 5
```

#### See Also

— “Arithmetic Routines” on page 721

### 24.2.4 odd()

#### Purpose

Check if an integer is odd

#### Category

Routine

### Syntax

**odd(***x*: int**)**: bool

Syntax example:

```
print odd(x);
```

### Parameters

| | |
|---|---|
| *x* | An integer expression. |

### Description

Returns TRUE if the integer is odd, FALSE if the integer is even.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = -5;
        print odd(x);
    };
};
'>
```

### Result

**sys.m1()**
```
odd(x) = TRUE
```

### See Also

— “Arithmetic Routines” on page 721

## 24.2.5 even()

### Purpose

Check if an integer is even

### Category

Routine

### Syntax

**even(***x*: int**)**: bool

Syntax example:

```
print even(x);
```

### Parameters

| | |
|---|---|
| *x* | An integer expression. |

### Description

Returns TRUE if the integer passed to it is even, FALSE if the integer is odd.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = -5;
        print even(x);
    };
};
'>
```

### Result

**sys.m1()**
```
even(x) = FALSE
```

### See Also

— "Arithmetic Routines" on page 721

## 24.2.6 ilog2()

### Purpose

Get the base-2 logarithm

### Category

Routine

### Syntax

**ilog2(*x*:** uint**)**: int

Syntax example:

```
print ilog2(x);
```

### Parameters

| | |
|---|---|
| *x* | An unsigned integer expression. |

### Description

Returns the integer part of the base-2 logarithm of x.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = 1034;
        print ilog2(x);
    };
};
'>
```

### Result

**sys.m1()**
```
  ilog2(x) = 10
```

### See Also

— “Arithmetic Routines” on page 721

## 24.2.7 ilog10()

### Purpose

Get the base-10 logarithm

### Category

Routine

### Syntax

**ilog10(*x***: uint**)**: int

Syntax example:

```
print ilog10(x);
```

### Parameters

| | |
|---|---|
| *x* | An unsigned integer expression. |

### Description

Returns the integer part of the base-10 logarithm of x.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = 1034;
        print ilog10(x);
    };
};
'>
```

### Result

**sys.m1()**
```
ilog10(x) = 3
```

### See Also

— "Arithmetic Routines" on page 721

## 24.2.8 ipow()

### Purpose

Raise to a power

### Category

Routine

### Syntax

**ipow(*x*:** int**, *y*:** int**):** int

Syntax example:

```
print ipow(x, y);
```

### Parameters

| | |
|---|---|
| *x* | An integer expression. |
| *y* | An integer expression. |

### Description

Raises x to the power of y and returns the integer result.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = 4;
        var y: int = 3;
        print ipow(x, y);
    };
};
'>
```

### Result

**sys.m1()**
```
  ipow(x, y) = 64
```

### See Also

— "Arithmetic Routines" on page 721

### 24.2.9 isqrt()

#### Purpose

Get the square root

#### Category

Routine

#### Syntax

**isqrt(*x***: uint**)**: int

Syntax example:

```
print isqrt(x);
```

#### Parameters

| | |
|---|---|
| *x* | An unsigned integer expression. |

#### Description

Returns the integer part of the square root of x.

#### Example

```
<'
extend sys {
    m1() is {
        var x: int = 67;
        print isqrt(x);
    };
};
'>
```

#### Result

**sys.m1()**
```
  isqrt(x) = 8
```

#### See Also

— “Arithmetic Routines” on page 721

### 24.2.10 div_round_up()

#### Purpose

Division rounded up

#### Category

Routine

### Syntax

**div_round_up(*x*:** int**, *y*:** int**):** int

Syntax example:

```
print div_round_up(x, y);
```

### Parameters

| | |
|---|---|
| *x* | An integer expression. to use as the dividend. |
| *y* | An integer expression to use as the divisor. |

### Description

Returns the result of x / y rounded up to the next integer.

### Example

```
<'
extend sys {
    m1() is {
        var x: int = 5;
        var y: int = 2;
        print div_round_up(x, y);
    };
};
'>
```

### Result

**sys.m1()**
```
  div_round_up(x, y) = 3
```

### See Also

— "Arithmetic Routines" on page 721
— "/" arithmetic operator in "+ - * / %" on page 41

## 24.3 Bitwise Routines

### 24.3.1 Overview

The predefined bitwise routines perform boolean operations bit-by-bit and return a single-bit result.

### 24.3.2 bitwise_op()

### Purpose

Perform a Verilog-style unary reduction operation

### Category

Pseudo-method

### Syntax

**bitwise_*op*(*exp*:** int|uint**)**: bit

Syntax example:

```
print bitwise_and(b);
```

### Parameters

| | |
|---|---|
| *op* | One of **and**, **or**, **xor**, **nand**, **nor**, **xnor**. |
| *exp* | A 32-bit numeric expression. |

### Description

Performs a Verilog-style unary reduction operation on a single operand to produce a single bit result. There is no reduction operator in ***e***, but the **bitwise_*op*()** routines perform the same functions as reduction operators in Verilog. For example, you can use **bitwise_xor()** to calculate parity.

For **bitwise_nand()**, **bitwise_nor()**, and **bitwise_xnor()**, the result is computed by inverting the result of the **bitwise_and()**, **bitwise_or()**, and **bitwise_xor()** operation, respectively.

Table 24-1 shows the predefined pseudo-methods for bitwise operations.

**Table 24-1—Bitwise Operation Pseudo-Methods**

| Pseudo-Method | Operation |
|---|---|
| bitwise_and() | Boolean AND of all bits |
| bitwise_or() | Boolean OR of all bits |
| bitwise_xor() | Boolean XOR of all bits |
| bitwise_nand() | !bitwise_and() |
| bitwise_nor() | !bitwise_or() |
| bitwise_xnor() | !bitwise_xor() |

NOTE— These routines cannot be used to perform bitwise operations on unbounded integers.

### Example 1

```
<'
struct nums {
    m1() is {
        var b: uint = 0xffffffff;
        print bitwise_and(b);
        print bitwise_nand(b);
    };
};

extend sys {
    pmi:nums;
};
'>
```

### Result

**sys.pmi.m()**
```
  bitwise_and(b) = 1
  bitwise_nand(b) = 0
```

### Example 2

```
<'
struct nums {
    m1() is {
        var b: uint = 0xcccccccc;
        print bitwise_or(b);
        print bitwise_nor(b);
    };
};

extend sys {
    pmi:nums;
};
'>
```

### Result

**sys.pmi.m()**
```
  bitwise_or(b) = 1
  bitwise_nor(b) = 0
```

### Example 3

```
<'
struct nums {
    m1() is {
        var b: uint = 0x1;
        print bitwise_xor(b);
        print bitwise_xnor(b);
    };
};

extend sys {
    pmi:nums;
};
'>
```

### Result

**sys.pmi.m()**
```
  bitwise_xor(b) = 1
  bitwise_xnor(b) = 0
```

### See Also

— “Arithmetic Routines” on page 721

## 24.4 Unit-Related Predefined Routines

The predefined routines that are useful for units include:

— "set_config_max()" on page 733
— "get_all_units()" on page 735

### 24.4.1 set_config_max()

#### Purpose

Increase values of numeric global configuration parameters

#### Category

Predefined routine

#### Syntax

**set_config_max(*category***: keyword**, *option***: keyword**, *value***: exp [**, *option***: keyword, ***value***: exp...]**)**

Syntax example:

```
set_config_max(memory, gc_threshold, 100m);
```

## Parameters

*category* Is one of the following: **coverage**, **generation**, **memory**, and **run**.

*option* The valid **coverage** options are:

— absolute_max_buckets, described in the **coverage** option of "set_config()" on page 766

The valid **generation** options are:

- absolute_max_list_size
- max_depth
- max_structs

These options are described in the **generation** option of "set_config()" on page 766.

The valid **memory** options are:

- gc_threshold
- gc_increment
- max_size
- absolute_max_size

These options are described in the **memory** option of "set_config()" on page 766.

The valid **run** options are:

- tick_max, described in the **run** option of "set_config()" on page 766.

*value* The valid values are different for each option and are described in "set_config()" on page 766.

## Description

Sets the numeric options of a particular category to the specified maximum values.

If you are creating a modular verification environment, it is recommended to use **set_config_max()** instead of **set_config()** in order to avoid possible conflicts that may happen in an integrated environment. For example, if two units are instantiated and both of them attempt to enlarge configuration value of **absolute_max_size** then the recommended way to it is via set_config_max, so that no unit decrements the value set by another one.

## Example

```
<'
extend sys {
    setup() is also {
        set_config_max(memory, gc_threshold, 100m);
        };
    };
    '>
```

### See Also

- — "Predefined Methods of Any Unit" on page 662
- — "Unit-Related Predefined Methods of Any Struct" on page 670

## 24.4.2 get_all_units()

### Purpose

Return a list of instances of a specified unit type

### Category

Routine

### Syntax

**get_all_units(*unit-type*:** exp**)**: list of unit instances

Syntax example:

```
print get_all_units(XYZ_channel);
```

### Parameters

| | |
|---|---|
| *unit-type* | The name of a unit type. The type must be defined or an error occurs. |

### Description

This routine receives a unit type as a parameter and returns a list of instances of this unit type as well as any unit instances contained within each instance.

### Example

This example uses **get_all_units()** to print a list of the instances of XYZ_router. Note that the display also shows that this instance of XYZ_router contains "channels", which is a list of three unit instances.

```
<'
unit XYZ_router {
    channels: list of XYZ_channel is instance;

    keep channels.size() == 3;
    keep for each in channels {
        .hdl_path() == append("chan", index);
        .router == me
    };

};

unit XYZ_channel {
    router:XYZ_router;
};

extend sys {
    router:XYZ_router is instance;

    run() is also {
        print get_all_units(XYZ_router);
    };
};
'>
```

### Result

```
  get_all_units(XYZ_router) =
item   type        channels
---------------------------------------------------------------
0.     XYZ_router  (3 items)
```

### See Also

— “Predefined Methods of Any Unit” on page 662
— “Unit-Related Predefined Methods of Any Struct” on page 670

## 24.5 String Routines

None of the string routines in *e* modify the input parameters. When you pass a parameter to one of these routines, the routine makes a copy of the parameter, manipulates the copy, and returns the copy.

You can use the **as_a()** casting operator to convert strings to integers or bytes. See Table 3-5 on page 107.

Routines that convert expressions into a string:

— “append()” on page 737
— “appendf()” on page 739
— “quote()” on page 743
— “to_string()” on page 761

Routines that manipulate substrings:

— “str_join()” on page 748
— “str_split()” on page 756
— “str_split_all()” on page 758
— “str_sub()” on page 759

Routines that manipulate regular expressions:

— “str_match()” on page 751
— “str_replace()” on page 754
— “str_insensitive()” on page 747

Routines that change the radix of a numeric expression:

— “bin()” on page 740
— “dec()” on page 741
— “hex()” on page 742

Routines that manipulate the length of an expression:

- “str_chop()” on page 744
- “str_empty()” on page 745
- “str_exactly()” on page 745
- “str_len()” on page 749
- “str_pad()” on page 753

Routines that are useful within macros:

- “quote()” on page 743
- “str_expand_dots()” on page 746

Routines that manipulate the case of characters within a string:

- “str_lower()” on page 750
- “str_upper()” on page 760
- “str_insensitive()” on page 747

### See Also

- “String Matching” on page 51
- “The string Type” on page 86
- Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107

## 24.5.1 append()

### Purpose

Concatenate expressions into a string

### Category

Routine

### Syntax

**append()**: string

**append(*item***: exp**, ...)**: string

Syntax example:

```
    message = append(list1, " ", list2);
```

### Parameters

| | |
|---|---|
| *item* | A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. If no items are passed to **append()**, it returns an empty string. |

### Description

Calls "to_string()" on page 761 to convert each expression to a string using the current radix setting for any numeric expressions, then concatenates them and returns the result as a single string.

NOTE— If you are concatenating a very large number of strings (for example, a long list of strings) into a single string, it is better to use **str_join()**, for performance reasons. If there are 10000 items in my_list, the for loop shown below makes 10000 copies of the lengthening string, hence creating an n**2 effect:

```
foo = "";
for each in l {
    foo = append(foo, it);
};
```

In contrast, the following will create a string of the right length and do a single copy operation:

```
foo = str_join(l, "");
```

### Example

```
extend sys {
    m1() is {
        var message: string;
        var u1:uint = 55;
        var u2:uint = 44;
        message = append(u1, " ", u2);
        print message;
        var list1:list of bit = {0;1;0;1};
        var list2:list of bit = {1;1;1;0};
        message = append(list1, " ", list2);
        print message;
    };
};
```

### Result

The radix setting for this example was hex.

**sys.m1()**
```
  message = "0x37 0x2c"
  message = "0x0 0x1 0x0 0x1 0x1 0x1 0x1 0x0"
```

### See Also

— "String Routines" on page 736
— Table 3-5 on page 107, for information about type conversion between strings and scalars

### 24.5.2 appendf()

#### Purpose

Concatenate expressions into a string according to a given format

#### Category

Routine

#### Syntax

**appendf(*format***: string**, *item***: exp**, ...)**: string

Syntax example:

```
message = appendf("%4d\n %4d\n %4d\n", 255, 54, 1570);
```

#### Parameters

| | |
|---|---|
| *format* | A string expression containing a standard C formatting mask for each ***item***. See "Format String" on page 765 for more information. |
| *item* | A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. |

#### Description

Converts each expression to a string using the current radix setting for any numeric expressions and the specified format, then concatenates them and returns the result as a single string.

NOTE— If the number and type of masks in the format string does not match the number and type of expressions, an error is issued.

#### Example

```
<'
extend sys {
    m1() is {
        var message: string;
        message = appendf("%4d\n %4d\n %4d\n", 255, 54, 1570);
        out(message);
    };
};
'>
```

#### Result

The radix setting for this example was **DEC**.

**sys.m1()**
```
 255
  54
1570
```

### See Also

— "String Routines" on page 736
— Table 3-5 on page 107, for information about type conversion between strings and scalars

## 24.5.3 bin()

### Purpose

Concatenate expressions into string, using binary representation for numeric types

### Category

Routine

### Syntax

**bin(*item***: exp, ...**)**: string

Syntax example:

```
var my_string: string = bin(pi.i, " ", list1, " ",8);
```

### Parameters

| | |
|---|---|
| *item* | A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. |

### Description

Concatenates one or more expressions into a string, using binary representation for any expressions of numeric types, regardless of the current radix setting.

### Example

```
<'
struct p {
    i:int;
};

extend sys {
    pi:p;
    m1() is {
        pi = new;
        pi.i = 11;
        var list1:list of bit = {0;1;1;0};
        var my_string: string = bin(pi.i, " ", list1, " ",8);
        print my_string;
    };
};
'>
```

### Result

**sys.m1()**
```
  my_string = "0b1011 0b0 0b1 0b1 0b0 0b1000"
```

### See Also

- "String Routines" on page 736
- Table 3-5 on page 107, for information about type conversion between strings and scalars

## 24.5.4 dec()

### Purpose

Concatenate expressions into string, using decimal representation for numeric types

### Category

Routine

### Syntax

**dec(*item***: exp, **...)**: string

Syntax example:

```
var my_string: string = dec(pi.i, " ", list1, " ",8);
```

### Parameters

| | |
|---|---|
| *item* | A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. |

### Description

Concatenates one or more expressions into a string, using decimal representation for any expressions of numeric types, regardless of the current radix setting.

### Example

```
<'
struct p {
    i:int;
};

extend sys {
    pi:p;
    m1() is {
        pi = new;
        pi.i = 11;
        var list1:list of bit = {0;1;1;0};
        var my_string: string = dec(pi.i, " ", list1, " ",8);
        print my_string;
    };
};
'>
```

### Result

**sys.m1()**

```
  my_string = "11 0 1 1 0 8"
```

### See Also

— "String Routines" on page 736
— Table 3-5 on page 107, for information about type conversion between strings and scalars

## 24.5.5 hex()

### Purpose

Concatenate expressions into string, using hexadecimal representation for numeric types

### Category

Routine

### Syntax

**hex(*item***: exp, ...**)**: string

Syntax example:

```
var my_string: string = hex(pi.i, " ", list1, " ",8);
```

### Parameters

*item* A list of one or more legal ***e*** expressions. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed.

### Description

Concatenates one or more expressions into a string, using hexadecimal representation for any expressions of numeric types, regardless of the current radix setting.

### Example

```
<'
struct p {
    i: int;
};

extend sys {
    pi: p;
    m1() is {
        pi = new;
        pi.i = 11;
        var list1:list of bit = {0;1;1;0};
        var my_string: string = hex(pi.i, " ", list1, " ",8);
        print my_string;
    };
};
'>
```

### Result

**sys.m1()**
```
  my_string = "0xb 0x0 0x1 0x1 0x0 0x8"
```

### See Also

— "String Routines" on page 736
— Table 3-5 on page 107, for information about type conversion between strings and scalars

## 24.5.6 quote()

### Purpose

Enclose a string in double quotes

### Category

Routine

### Syntax

**quote(*text*:** string**)**: string

Syntax example:

```
out(quote(message));
```

### Parameters

| | |
|---|---|
| *text* | An expression of type **string**. |

### Description

Returns a copy of the text, enclosed in double quotes (" "), with any internal quote or backslash preceded by a backslash (\).

This routine is useful when creating commands with **define**.

### Example

```
<'
extend sys {
    m1() is {
        var message: string = "Error occurred in \"D\" block...";
        out(message);
        out(quote(message));
    };
};
'>
```

### Result

**sys.m1()**
```
  Error occurred in "D" block...
  "Error occurred in \"D\" block..."
```

### See Also

— "String Routines" on page 736

— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

### 24.5.7 str_chop()

#### Purpose

Chop the tail of a string

#### Category

Routine

#### Syntax

**str_chop(*str***: string**, *length***: int**)**: string

Syntax example:

```
var test_dir: string = str_chop(tmp_dir, 13);
```

#### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *length* | An integer representing the desired length. |

#### Description

Returns ***str*** (if its length is <= ***length***), or a copy of the first ***length*** chars of ***str***.

Removes characters from the end of a string, returning a string of the desired length. If the original string is already less than or equal to the desired length, this routine returns a copy of the original string.

#### Example

```
<'
extend sys {
    m1() is {
        var tmp_dir: string = "/rtests/test1/tmp";
        var test_dir: string = str_chop(tmp_dir, 13);
        print test_dir;
    };
};
'>
```

#### Result

**sys.m1()**
```
  test_dir = "/rtests/test1"
```

#### See Also

— “String Routines” on page 736
— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

### 24.5.8 str_empty()

#### Purpose

Check if a string is empty

#### Category

Routine

#### Syntax

**str_empty(*str***: string**)**: bool

Syntax example:

```
print str_empty(s1);
```

#### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |

#### Description

Returns TRUE if the string is uninitialized or empty.

#### Example

```
<'
extend sys{
    m1() is {
        var s1: string;
        var s2: string = "";
        print str_empty(s1);
        print str_empty(s2);
    };
};
'>
```

#### Result

**sys.m1()**

```
  str_empty(s1) = TRUE
  str_empty(s2) = TRUE
```

#### See Also

— “String Routines” on page 736

### 24.5.9 str_exactly()

#### Purpose

Get a string with exact length

### Category

Routine

### Syntax

**str_exactly(*str*:** string**, *length*:** int**):** string

Syntax example:

```
var long: string = str_exactly("123", 6);
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *length* | An integer representing the desired length. |

### Description

Returns a copy of the original string, whose length is the desired length, by adding blanks to the right or by truncating the expression from the right as necessary. If non-blank characters are truncated, the * character appears as the last character in the string returned.

### Example

```
<'
extend sys {
    m1() is {
        var short: string = str_exactly("123000",3);
        print short;
        var long: string = str_exactly("123", 6);
        print long;
    };
};
'>
```

### Result

**sys.m1()**
```
  short = "12*"
  long = "123   "
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.10 str_expand_dots()

### Purpose

Expand strings shortened by the parser

### Category

Routine

### Syntax

**str_expand_dots(*str***: string**)**: string

Syntax example:

```
out("After expand: ",str_expand_dots(<1>));
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |

### Description

Returns a copy of the original string, expanding any dot place holders into the actual code they represent.

This routine is useful only in context of **define as [computed]** statements. When preprocessing an ***e*** file, any sequence of characters between matching brackets or quotes (such as (), [], {}, or " ") is conveted to compressed code, with dots as place holders. If you want to retrieve the original string, not just the compressed code, you can use this routine.

### Example

To retrieve the original string passed to my_macro, the **str_expand_dots()** routine is called.

```
<'
define <my_macro'command> "my_macro (\
          \"<string>\")" as computed {
        out("Before expand: ", <1>);
        out("After expand: ",str_expand_dots(<1>));
        result = "{}"
};
'>
```

### Result

**my_macro "hello world"**
```
  Before expand: "10"
  After expand: "hello world"
```

### See Also

— “String Routines” on page 736
— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

## 24.5.11 str_insensitive()

### Purpose

Get a case-insensitive AWK-style regular-expression

### Category

Routine

### Syntax

**str_insensitive(*regular_exp*:** string**)**: string

Syntax example:

```
var insensitive: string = str_insensitive("/hello.*/");
```

### Parameters

| | |
|---|---|
| *regular-exp* | An AWK-style regular expression. |

### Description

Returns an AWK-style regular expression string which is the case-insensitive version of the original regular expression.

### Example

```
<'
extend sys {
    m1() is {
        var insensitive: string = str_insensitive("/hello.*/");
        print insensitive;
    };
};
'>
```

### Result

**sys.m1()**
```
  insensitive = "/[Hh][Ee][Ll][Ll][Oo].*/"
```

### See Also

— “AWK-Style String Matching” on page 52
— “String Routines” on page 736
— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

## 24.5.12 str_join()

### Purpose

Concatenate a list of strings

### Category

Routine

### Syntax

**str_join(*list***: list of string**, *separator***: string**)**: string

Syntax example:

```
var s := str_join(slist," - ");
```

### Parameters

| | |
|---|---|
| *list* | A list of type **string**. |
| *separator* | The string that will be used to separate the list elements. |

### Description

Returns a single string which is the concatenation of the strings in the list of strings, separated by the separator.

### Example

```
<'
extend sys {
    m1() is {
        var slist: list of string = {"first";"second";"third"};
        var s := str_join(slist," - ");
        print s;
    };
};
'>
```

### Result

**sys.m1()**

```
  s = "first - second - third"
```

### See Also

— “String Routines” on page 736

— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

## 24.5.13 str_len()

### Purpose

Get string length

### Category

Routine

### Syntax

**str_len(*str***: string**)**: int

Syntax example:

```
var length: int = str_len("hello");
```

#### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |

#### Description

Returns the number of characters in the original string, not counting the terminating NULL character \0.

#### Example

```
<'
extend sys {
    m1() is {
         var length: int = str_len("hello");
        print length;
    };
};
'>
```

#### Result

**sys.m1()**
```
  length = 5
```

#### See Also

— "String Routines" on page 736

### 24.5.14 str_lower()

#### Purpose

Convert string to lowercase

#### Category

Routine

#### Syntax

**str_lower(*str***: string**)**: string

Syntax example:

```
var lower: string = str_lower("UPPER");
```

### Parameters

*str* An expression of type **string**.

### Description

Converts all upper case characters in the original string to lower case and returns the string.

### Example

```
<'
extend sys {
    m1() is {
       var lower: string = str_lower("UPPER");
        print lower;
    };
};
'>
```

### Result

**sys.m1()**
```
  lower = "upper"
```

### See Also

— “String Routines” on page 736
— Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalars

## 24.5.15 str_match()

### Purpose

Match strings

### Category

Routine

### Syntax

**str_match(*str*:** string**, *regular-exp*:** string**):** bool

Syntax example:

```
print str_match("ace", "/c(e)?$/");
```

## Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *regular-exp* | An AWK-style or native ***e*** regular expression. If not surrounded by slashes, the expression is treated as a native style expression. See "String Matching" on page 51 for more information on these types of expressions. |

## Description

Returns TRUE if the strings match, or FALSE if the strings do not match. The routine **str_match()** is fully equivalent to the operator ~.

## Example 1

After doing a match, you can use the local pseudo-variables $1, $2...$27, which correspond to the parenthesized pieces of the match. $0 stores the entire matched piece of the string.

This example uses AWK-style regular expressions. See "AWK-Style String Matching" on page 52 for more information on these types of expressions.

```
<'
extend sys {
    m1() is {
        print str_match("a", "/^(a)?(b)?$/"); -- matches "ab", "a", or "b"
        print $0, $1, $2;
        print str_match("hello","/^\\d*$/"); -- matches string with 0
                                             -- or more digits
        print $0, $1, $2;
        print str_match("ab", "/^(a)?(b)?$/");
        print $0, $1, $2;
        print str_match("ace", "/c(e)?$/"); -- matches string ending
                                            -- in "c" or "ce"
        print $0, $1;
    };
};
'>
```

## Result

**sys.m1()**

```
str_match("a", "/^(a)?(b)?$/") = TRUE
  $0 = "a"    -- stores the entire matched portion of the string
  $1 = "a"    -- stores the first match
  $2 = ""
str_match("hello","/^\\d*$/") = FALSE
  $0 = "a"    -- the values from the previous call to str_match() persist
  $1 = "a"
  $2 = ""
str_match("ab", "/^(a)?(b)?$/") = TRUE
  $0 = "ab"
  $1 = "a"
  $2 = "b"
str_match("ace", "/c(e)?$/") = TRUE
  $0 = "ce"
  $1 = "e"
```

### Example 2

This example uses native ***e*** regular expressions. See "Native e Elite String Matching" on page 51 for more information on these types of expressions. The * character matches any sequence of non-white characters, while the "..." string matches any sequence of characters, including white space.

```
<'
extend sys {
    m1() is {
        var s:string = "a bc";
        print str_match(s, "a*");
        print $0, $1, $2;
        print str_match(s, "* *");
        print $0, $1, $2;
        print str_match(s, "a...");
        print $0, $1, $2;
    };
};
'>
```

### Result

**sys.m1()**

```
str_match(s, "a*") = FALSE
  $0 = ""
  $1 = ""
  $2 = ""
str_match(s, "* *") = TRUE
  $0 = "a bc"
  $1 = "a"
  $2 = "bc"
str_match(s, "a...") = TRUE
  $0 = "a bc"
  $1 = " bc"
  $2 = ""
```

### See Also

- "String Routines" on page 736
- "~ !~" on page 47

## 24.5.16 str_pad()

### Purpose

Pad string with blanks

### Category

Routine

### Syntax

**str_pad(*str*:** string**, *length*:** int**):** string

Syntax example:

```
var s: string = str_pad("hello world",14);
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *length* | An integer representing the desired length, no greater than 4000. |

### Description

Returns a copy of the original string padded with blanks on the right, up to desired length. If the length of the original string is greater than or equal to the desired length, then the original string is returned (not a copy) with no padding.

### Example

```
<'
extend sys {
    m1() is {
        var s: string = str_pad("hello world",14);
        print s;
    };
};
'>
```

### Result

**sys.m1()**
```
s = "hello world   "
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.17 str_replace()

### Purpose

Replace a substring in a string with another string

### Category

Routine

### Syntax

**str_replace(*str***: string**, *regular-exp***: string**, *replacement***: string**)**: string

Syntax example:

```
var s: string = str_replace("crc32", "/(.*32)/", "32_flip");
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *regular-exp* | An AWK-style or native ***e*** regular expression. If not surrounded by slashes, the expression is treated as a native style expression. See "String Matching" on page 51 for more information on these types of expressions. |
| *replacement* | The string that you want to replace all occurrences of the regular expression. |

### Description

A new copy of the original string is created, and then all the matches of the regular expression are replaced by the replacement string. If no match is found, a copy of the source string is returned.

To incorporate the matched substrings in the ***replacement*** string, use back-slash escaped numbers \1, \2, ...

In native ***e*** regular expressions, the portion of the original string that matches the * or the ... characters is replaced by the replacement string.

In AWK-style regular expressions, you can mark the portions of the regular expressions that you want to replace with parentheses. For example, the following regular expression marks all the characters after "on", up to and including "th" to be replaced.

```
"/on(.*th)/"
```

### Example 1

This example uses AWK-style string matching.

```
<'
extend sys {
    m1() is {
        var new_str : string = str_replace("testing one two \
          three", "/on(.*th)/" , "f");
        print new_str;
    };
};
'>
```

### Result

```
sys.m1()
new_str = "testing free"
```

### Example 2

Another AWK-style example, using \1, \2:

```
<'
extend sys {
    m1() is {
        var new_str : string = str_replace("A saw B",
            "/(.*) saw (.*)/" , "\2 was seen by \1");
        print new_str;
    };
};
```

```
'>
```

### Result

**sys.m1()**
```
new_str = "B was seen by A"
```

### Example 3

This example uses a ***e*** -style regular expression.

```
<'
extend sys {
    m1() is {
        var new_str: string=str_replace("abc z", "a* " , "xy");
        print new_str;
    };
};
'>
```

### Result

**sys.m1()**
```
new_str = "xyz"
```

### Example 4

This example uses a ***e*** -style regular expression with \1.

```
<'
extend sys {
    m1() is {
var new_str: string=str_replace("abc ghi", "* " , "\1 def ");
    print new_str;
    };
};
'>
```

### Result

**sys.m1()**
```
new_str = "abc def ghi"
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.18 str_split()

### Purpose

Split a string to substrings

### Category

Routine

### Syntax

**str_split(*str***: string**, *regular-exp***: string**)**: list of string

Syntax example:

```
var s: list of string = str_split("first-second-third", "-");
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *regular-exp* | An AWK-style or native ***e*** -style regular expression that specifies where to split the string. See "String Matching" on page 51 for more information on these types of expressions. |

### Description

Splits the original string on each occurrence of the regular expression, and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned.

If the regular expression is an empty string, it has the effect of removing all blanks in the original string.

### Example 1

```
<'
extend sys {
    m1() is {
        var s: list of string=str_split("one-two-three", "-");
        print s;
    };
};
'>
```

### Result

**sys.m1()**
```
s =
0.      "one"
1.      "two"
2.      "three"
```

### Example 2

```
<'
extend sys {
    m1() is {
        var s: list of string = str_split(" A B  C", "/ +/");
        print s;
    };
};
'>
```

### Result

```
sys.m1()
s =
0.      ""
1.      "A"
2.      "B"
3.      "C"
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.19 str_split_all()

### Purpose

Split a string to substrings, including separators

### Category

Routine

### Syntax

**str_split_all(*str***: string**, *regular-exp***: string**)**: list of string

Syntax example:

```
var s: list of string = str_split_all(" A B C", "/ +/");
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *regular-exp* | An AWK-style or native ***e*** -style regular expression that specifies where to split the string. See "String Matching" on page 51 for more information on these types of expressions. |

### Description

Splits the original string on each occurrence of the regular expression, and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned.

This routine is similar to **str_split**(), except that it includes the separators in the resulting list of strings.

### Example

```
<'
extend sys {
    m1() is {
        var s: list of string = str_split_all(" A B  C", "/ +/");
        print s;
    };
};
```

```
'>
```

### Result

**sys.m1()**

```
s =
0.      ""
1.      " "
2.      "A"
3.      " "
4.      "B"
5.      "  "
6.      "C"
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.20 str_sub()

### Purpose

Extract a substring from a string

### Category

Routine

### Syntax

**str_sub(*str***: string**, *from***: int**, *length***: int**)**: string

Syntax example:

```
var dir: string = str_sub("/rtests/test32/tmp", 8, 6);
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |
| *from* | The index position from which to start extracting. The first character in the string is at index 0. |
| *length* | An integer representing the number of characters to extract. |

### Description

Returns a substring of the specified length from the original string, starting from the specified index position.

### Example

```
<'
extend sys {
    m1() is {
            var dir: string = str_sub("/rtests/test32/tmp", 8, 6);
            print dir;
        };
    };
    '>
```

### Result

**sys.m1()**
```
dir = "test32"
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.21 str_upper()

### Purpose

Convert a string to uppercase

### Category

Routine

### Syntax

**str_upper(*str***: string**)**: string

Syntax example:

```
var upper: string = str_upper("lower");
```

### Parameters

| | |
|---|---|
| *str* | An expression of type **string**. |

### Description

Returns a copy of the original string, converting all lower case characters to upper case characters.

### Example

```
<'
extend sys {
    m1() is {
       var upper: string = str_upper("lower");
        print upper;
    };
};
'>
```

### Result

**sys.m1()**

```
    upper = "LOWER"
```

### See Also

- — "String Routines" on page 736
- — Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.5.22 to_string()

### Purpose

Convert any expression to a string

### Category

Method

### Syntax

***exp*.to_string()**: string

Syntax example:

```
    print pkts[0].to_string();
```

### Parameters

| | |
|---|---|
| *exp* | A legal ***e*** expression. |

### Description

This method can be used to convert any type to a string.

If the expression is a struct expression, the **to_string()** method returns a unique identification string for each struct instance. You can use this ID to refer to the struct. By default, the identification string is of the form ***type*-@ *num***, where ***num*** is a unique struct number over all instances of all structs in the current run.

If the expression is a list of strings, the **to_string()** method is called for each element in the list. The string returned contains all the elements with a newline between each element.

If the expression is a list of any type except **string**, the **to_string()** method returns a string containing all the elements with a space between each element.

If the expression is a numeric type, the expression is converted using the current radix with the radix prefix.

If the expression is a string, the **to_string()** method returns the string.

If the expression is an enumerated or a boolean type, the **to_string()** method returns the value.

### Example

```
    <'
    struct pkt {
        protocol: [ethernet, atm, other];
        legal : bool;
        data[2]: list of byte;
    };

    extend sys {
        pkts[5]: list of pkt;
        m1() is {
            var slist: list of string = {"strA";"strB";"strC"};
            print pkts[0].to_string();
            print pkts[0].data.to_string();
            print pkts[0].protocol.to_string();
            print pkts[0].legal.to_string();
            print slist.to_string();
        };
    };
    '>
```

### Result

```
    sys.m1()
      pkts[0].to_string() = "pkt-@0"
      pkts[0].data.to_string() = "52 218"
      pkts[0].protocol.to_string() = "atm"
      pkts[0].legal.to_string() = "TRUE"
      slist.to_string() = "strA
    strB
    strC"
```

### See Also

— "String Routines" on page 736
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalars

## 24.6 Output Routines

The predefined output routines print formatted and unformatted information to the screen and to open log files. The following sections describe the output routines:

— "out()" on page 762
— "outf()" on page 764

### 24.6.1 out()

#### Purpose

Print expressions to output, with a new line at the end

#### Category

Routine

#### Syntax

**out()**

**out(*item*: exp, ...)**

Syntax example:

```
out("pkts[1].data is          ", pkts[1].data);
```

## Parameters

*item* A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. If no item is passed to **out()**, an empty string is printed, followed by a new line.

## Description

Calls **to_string()** to convert each expression to a string and prints them to the screen (and to the log file if it is open), followed by a new line.

## Example

This first example shows that if the expression is a struct, **out()** prints the ID of the struct. If the expression is a list, **out()** prints each element of the list from left to right, starting with the element with the lowest index. If no expression is passed, **out()** prints a new line.

```
struct pkt {
    protocol: [ethernet, atm, other];
    legal : bool;
    data[2]: list of byte;

};

extend sys {
    pkts[5]: list of pkt;
    m1() is {
        out();
        out("ID of first packet is   ", pkts[0]);
        out("pkts[1].data is         ", pkts[1].data);
        out("pkts[1].data[0] is      ", pkts[1].data[0]);
        out();
    };
};
```

## Result

**sys.m1()**

```
ID of first packet is   pkt-@0
pkts[1].data is         142 170
pkts[1].data[0] is      142
```

## See Also

— “outf()” on page 764

### 24.6.2 outf()

#### Purpose

Print formatted expressions to output, with no new line at the end

#### Category

Routine

#### Syntax

**outf(*format*:** string**, *item*:** exp, ...**)**

Syntax example:

```
outf("%s %#08x","pkts[1].data[0] is      ", pkts[1].data[0]);
```

#### Parameters

| | |
|---|---|
| *format* | A string containing a standard C formatting mask for each ***item***. See "Format String" on page 765 for more information. |
| *item* | A legal ***e*** expression. String expressions must be enclosed in double quotes. If the expression is a struct instance, the struct ID is printed. If the expression is a list, an error is issued. |

#### Description

Converts each expression to a string using the corresponding format string and then prints them to the screen (and to the log file if it is open).

To add a new line, add the \n characters to the format string.

#### Notes

— If the number and type of masks in the format string does not match the number and type of expressions, an error is issued.

— The ***e*** program adds a new line by default after 80 characters. To disable this, set the line size to UNDEF or to a very large number with the **set_config()** routine.

```
set_config(print, line_size, UNDEF);
```

#### Example 1

This example has similar results to the in the description of "out()" on page 762. Note that **outf()** allows you to add the new lines where needed. Printing of lists is not supported with **outf()**.

```
extend sys {
    pkts[5]: list of pkt;
    m1() is {
        outf("%s %s\n","pkts[0] ID is      ", pkts[0]);
        outf("%s %#x","pkts[1].data[0] is  ", pkts[1].data[0]);
    };
};
```

### Result

**sys.m1()**
```
pkts[0] ID is             pkt-@0
pkts[1].data[0] is        0x8e
```

### Example 2

This example shows the control over formatting of strings that **outf()** allows. The fields have been enclosed in double ":" characters so that you can easily see how wide the field is.

```
<'
extend sys {
    m1() is {
    outf(":%s:\n", "hello world");
    outf(":%15s:\n", "hello world");
    outf(":%-15s:\n", "hello world");
    outf(":%.8s:\n", "hello world");
    outf(":%08d:\n", 12);
    };
};
'>
```

### Result

**sys.m1()**
```
:hello world:
:    hello world:
:hello world    :
:hello wo:
:00000012:
```

### See Also

— "out()" on page 762

## 24.6.3 Format String

The format string for the **outf()** and for the **appendf()** routine uses the following syntax:

**"%**`[0|-][#][min_width][.[max_chars]](s|d|x|b|o|u)`**"**

where:

| | |
|---|---|
| 0 | Pads with 0 instead of blanks. Padding is only done when right alignment is used, on the left end of the expression. |
| - | Aligns left. The default is to align right. |
| # | Adds 0x before the number. Can be used only with the x (hexadecimal) format specifier. Examples: %#x, %#010x |
| min_width | A number that specifies the minimum number of characters. This number determines the minimum width of the field. If there are not enough characters in the expression to fill the field, the expression is padded to make it this many characters wide. If there are more characters in the expression than this number (and if max_chars is set large enough), this number is ignored and enough space is used to accommodate the entire expression. |

| | |
|---|---|
| max_chars | A number that specifies the maximum number of characters to use from the expression. Characters in excess of this number are truncated. If this number is larger than min_width, then the min_width number is ignored.<br><br>In the following example, min_width is 7 and max_chars is 5, so the **outf()** method prints five characters from “abcdefghi” in a field seven characters wide, padding the two unused spaces with blanks:<br><br>`outf("%7.5s\n", "abcdefghi")`<br>`  abcde`<br><br>In the following example, min_width is 8 and max_chars is 3, so the **outf()** method prints three characters from “abcdefghi” in a field eight characters wide, padding the five unused spaces with zeros, since the 0 padding option is present immediately after the %:<br><br>`outf("%08.3s\n", "abcdefghi")`<br>`00000abc` |
| s | Converts the expression to a string. The routine “to_string()” on page 761 is used to convert a non-string expression to a string. |
| d | Prints a numeric expression in decimal format. |
| x | Prints a numeric expression in hex format. With the optional # character, adds 0x before the number. |
| b | Prints a numeric expression in binary format. |
| o | Prints a numeric expression in octal format. |
| u | Prints integers (**int** and **uint**) in **uint** format. |

## 24.7 Configuration Routines

The configuration routines set options that allow you to control printing, memory usage, runtime characteristics, coverage, generation, and debug from within the ***e*** code.

— “set_config()” on page 766
— “get_config()” on page 781
— “write_config()” on page 782
— “read_config()” on page 784

### 24.7.1 set_config()

#### Purpose

Set configuration options

#### Category

Routine

### Syntax

**set_config(*category***: keyword**, *option***: keyword**, *value***: exp [**, *option***: keyword, ***value***: exp...]**)** [[**with**] {***action***; ...}]

Syntax example:

```
set_config(print, radix, bin);
```

### Parameters

*category* Is one of the following: **cover**[**age**], **data**[**browser**], **debug**[**ger**], **gen**[**eration**], **gui**, **mem**[**ory**], **misc**, **print**, **run**, and **wave**.

*option* The valid options are different for each category. The options for each category are listed in the following tables:

- Coverage Configuration Options: Table 24-2, “Coverage Configuration Options”, on page 768
- Generation Configuration Options: Table 24-2, “Coverage Configuration Options”, on page 768
- Memory Configuration Options: Table 24-4, “Memory Configuration Options”, on page 775
- Run Configuration Options: Table 24-5, “Run Configuration Options”, on page 778

*value* The valid values are different for each option and are described in the tables listed above.

**Table 24-2—Coverage Configuration Options**

| Option | Description |
|---|---|
| at_least_multiplier, *n* | An integer that is multiplied by the ***at-least*** specification for each coverage bucket to determine the smallest number of samples are not considered a hole. The default of this number is 1. If several tests were run you might want to increase this number. The range is 0 to MAX_INT. |
| grading_formula, *type* | Selects the type of grading formula used to calculate bucket grades, either **linear** (the default) or **root_mean_square**. |
| mode, *value* | Legal values are **default**, **off**, **on**, **on_interactive**. The default is **on** if coverage groups exist in the ***e*** code. If no coverage groups exist, the default is **off**.<br><br>**default**<br>If coverage groups exist in the ***e*** code, the mode switches to **on** just before simulation (in the setup phase). If no coverage groups exist, the mode switches to **off**.<br><br>**off**<br>No coverage collection is done.<br><br>**on**<br>Collects coverage summary only. Using this mode, post run cross coverage is disabled. This is the default mode. It can be overridden by the **no_collect** option of each cover group.<br><br>**on_interactive**<br>Coverage information is fully collected. It can be overridden by the **no_collect** option of each cover group. |
| **verbose_interface,** TRUE \| FALSE | Turns on verbose mode, in which a prompt is issued to confirm the effects of coverage actions and methods. The affected methods are:<br><br>• covers.set_cover()<br>• covers.set_weight()<br>• covers.set_get_contributing_runs()<br>• covers.include_tests()<br>• covers.set_at_least() |
| dir,*string* | The directory in which to save coverage files. |
| **file_name,*string*** | The file in which to store coverage data at the end of the test. The default is **run_name** (see above). If no extension is given with the string, the .ecov extension is added automatically. |
| test_name,*string* | With **run_name** and **tag_name**, an identifier for the coverage data for a particular test run.<br><br>The default is the name of the top module (the ***e*** module loaded last). |

| | |
|---|---|
| run_name, *string* | With **test_name** and **tag_name**, an identifier for the coverage data for a particular test run.<br><br>The default is **test_name_seed**. |
| tag_name, *string* | With **test_name** and **run_name**, an identifier for the coverage data for a particular test run.<br><br>The initial default is dir (see **-dir** option, below). |
| **max_int_buckets, *n*** | An integer that applies to integer items such as **int** or **byte**, with no explicit ranges. If the maximum number of possible values is less than or equal to ***n***, a bucket is created for every possible value. If, however, the number of possible values is greater than ***n***, a bucket is created only for each new sampled value. The result is that no buckets are defined for values which were never sampled. The default value is 16. The range is 0 to MAX_INT. |
| absolute_max_buckets, *n* | An integer that specifies the maximum number of buckets that can be created for a range specified using the **range** cover item option.<br><br>This limit prevents problems due to excessive memory usage when there are too many buckets to display. For example, if a cover item definition includes a range specification such as **range**([1..64K], “”, 1), then 64K buckets are created, which can cause a memory explosion when coverage is displayed. Setting this option to a number smaller than 64K causes an error message for this case. If you really intend to display 64K buckets, you can set this option to 64K.<br><br>The default is 4096. The range is MIN_INT to 0. |
| ~~max_gui_buckets, *n*~~ | ~~An integer that specifies the maximum number of buckets that will be shown for any coverage item.~~<br><br>~~The Coverage GUI can display up to about 50,000 buckets. Since that is a very large number of buckets to browse interactively, and since exceeding that number of buckets might cause the Coverage GUI to crash, this option allows you to limit the number of buckets shown in the GUI.~~<br><br>~~For any coverage item whose number of buckets exceeds this value, the item’s buckets are not shown in the Coverage GUI. However, the item name and its grade are shown.~~<br><br>~~The range is 0 to 200,000. The default is 10,000.~~ |

| | |
|---|---|
| show_mode, *value* | Determines whether holes are shown. Possible value are **holes_only**, **full**, and **both**. The default is **both**.<br><br>**holes_only**<br>Shows only buckets whose number of samples is less than the ***at-least*** number for this bucket multiplied by the **at_least_multiplier** factor.<br><br>**full**<br>Shows all buckets with a grade of 1. Invalid buckets are not reported.<br><br>**both**<br>Shows all full buckets and buckets with holes. Invalid buckets are not reported |
| sorted, *bool* | If TRUE displayed buckets are sorted by decreasing samples count. The default value is FALSE. |
| show_file_names, bool | If TRUE (the default), then all coverage file names are displayed as the coverage files are read in.<br><br>If a coverage file contains data from more than one test, the number of test runs is displayed with the file name. |
| show_sub_holes | A cross coverage item might have empty subtrees if any of the combinations of cross items consists only of empty buckets. When this option is TRUE, all of the item values for empty subtrees are displayed even though they have only empty buckets.<br><br>When this option is FALSE (the default), the item value which has only empty buckets is displayed as a dash (-). |
| **show_instances_only** | Determines what information is shown for per-instance coverage.<br><br>If this option is TRUE, for cover groups that have **per_instance** items, coverage data is only shown for the instances. The grading does not include data for the original item.<br><br>If this option is FALSE, coverage data is shown for the original item and for all instances.<br><br>The default is FALSE. |
| **show_partial_grade** | Enables displaying the fully calculated grade for part of the coverage database. When this option is set to TRUE, if you use wild cards for coverage groups or item when you view the coverage, you see the total grades for the particular items and groups that match the wild cards. |

surecov_config

Sets filtering options for SureSight. The options are:

- --filter_reset – Filters out FSM reset transitions in coverage reports.
- --filter_file = filter-file-name – Applies the specified filter file to the coverage database. The specified filter file must be an existing file.

You can enter either option alone or both of them, in either order.

NOTE— The --filter_reset and --filter_file options are available in SureCov version 3.1 or later.

**ranking_cost, *option***

Use this option to specify a metric for the cost of coverage. The metric option can be one of the following:

- **constant_cost** – Test ranking is based on the coverage database grade only, disregarding the cost in terms of time.
- **system_cpu_time** – Test ranking is based on the coverage database grade divided by the system CPU time consumed by the coverage run. The system CPU time is stored in the predefined **session.end_of_test.system_time** field.
- **user_cpu_time** – Test ranking is based on the coverage database grade divided by the user CPU time consumed by the coverage run. The user CPU time is stored in the predefined **session.end_of_test.user_time** field.
- **cpu_time** – Test ranking is based on the coverage database grade divided by the sum of the system CPU time and the user CPU time (see above).
- **test_time** – Test ranking is based on the coverage database grade divided by the test time consumed by the run (the value of **sys.time** at the end of the run). The test time is stored in the predefined **session.end_of_test.test_time** field.

**ranking_precision, *value***

Use this option to specify the number of digits after the floating point that are to be used for test ranking results. The value can be from 0 to 4, and has a default of 2.

This option affects only the test ranking output display, and not the actual computation of the results.

**Table 24-3—Generation Configuration Options**

| Option | Description |
|---|---|
| **seed, *n* \| random** | The initial seed used for every generation in the run. **random** specifies a new initial seed that is based on the time of day, the user id, and so on. The range for ***n*** is MIN_INT to MAX_INT. The default is 1. |
| default_max_list_size, *n* | An integer value that specifies the maximum number of items in a generated list. The default value is 50. You can override the default max list size with a hard value constraint, or cancel it with a **reset_soft()** constraint. The range is 0 to MAX_INT. |
| absolute_max_list_size, *n* | This is a watchdog field, which simply helps you get a clear error message rather than undesirable results. If while generating a list, its range extends to **absolute_max_list_size**, then an error message will result: this means that the size of this list has not been constrained properly (via either hard or soft constraints).<br><br>The default is 524282. The range is 0 to MAX_INT. |
| max_depth, *n* | This is another watchdog field. It guards against an infinite generation loop, which usually results from not constraining a back-pointer to point back at the parent struct. The default value is 30. The range is 0 to MAX_INT. |
| max_structs, *n* | This is another watchdog field. If the total number of structs generated for any particular field during a single test (both pre-generation and on the fly) exceeds this number, an error message is issued. Often you will get the **max_depth** message first, but in some cases (e.g.very long lists of struct) you will get this error message first. The default is 10,000. The range is 0 to MAX_INT. |
| collect_gen, *bool* | Starts or stops tracing of generation data. The default is FALSE. |
| collect_all, *bool* | This option has no effect when the **collect_gen** option is FALSE. When this option is TRUE, all generation executions are traced. When it is FALSE, only the last unnested generation execution is traced. The default is FALSE. |
| reorder_fields, *bool* | By default this is TRUE and the following algorithm is used to determine the order in which it generates the fields within a struct:<br><br>• By default, use the order in which the fields are declared.<br><br>• However, if a **gen before** constraint requests a different order, obey it.<br><br>• Finally, if the constraints themselves force a different order, this takes highest precedence. For instance, a constraint such as the following always causes y to be generated before x:<br><br>`keep x == my_method(y);`<br><br>Setting **reorder_fields** to FALSE causes the third rule above to be ignored, and generation to always be done by field order and **gen before** order. Use this flag to help debugging. |

check_unsatisfied_cons

If TRUE (the default), at the end of generation for each struct, checks that the constraints defined in this struct are satisfied. If FALSE, some constraints may be ignored by the generator without warning.

long_max_width, *n*

Determines the maximum size (in bits) of a long integer that is used with bit-constraints. By default ***n*** is 128.

You should increase the maximum size of long integers if you have bit slice constraints on integers that are longer than 128 bits and you get a generation contradiction. For example, the following constraint causes a contradiction if the maximum size is not increased:

```
<'
extend sys {
    i: int (bits: 256);
    keep i[143:72] == 1;
};
'>
```

NOTE— Increasing the size has performance impact.

determinants_before_ sub-types, *bool* | *option*

To turn on global subtype generation optimization, setting this option to TRUE activates subtype optimization for all structs.

For more specific subtype generation optimization control, the ***option*** can be one of the following:

- **force_off** – Subtype optimization is not activated for any determinants.
- **default_off** – Subtype optimization is not activated for any determinant except those determinants specified in **gen_before_subtypes()** constraints in the ***e*** code. This is the default.
- **default_on** – Subtype optimization is activated for all structs or units, except those that contain a **reset_gen_before_subtypes()** constraint, and for all determinants within a struct or unit, except those specified explicitly by **gen_before_subtypes()** constraints.
- **force_on** – Subtype optimization is activated for all determinants in the ***e*** code.

See "keep gen_before_subtypes()" on page 287 for information about the purpose and usage of subtype generation optimization.

warn, *bool*

Turns on warnings if set to TRUE. Generation warnings, including warnings on single constraints, on ordering, and on size, are issued only if the **-warn** generation option is set to TRUE. The default is FALSE, except during debug, when all warnings are shown.

resolve_cycles, *bool*

A constraint cycle occurs when two or more unidirectional constraints impose order relations inconsistent with each other. With this option set to FALSE (the default), a cycle causes a run-time contradiction error. With this option set to TRUE, whenever the generator encounters a cycle, it resolves it as follows:

- The generator chooses the item that needs the least number of other items.
- If there are multiple items with the same least number of needs relations, the generator chooses the one lowest in the soft order.

bool_exp_is_bidir, *bool*

When TRUE (the default), constraints of the form are bidirectional:

**keep *bool-field* == *complex-bool-exp***

where ***complex-bool-exp*** is an expression involving only a single comparison operator between fields or constants.

To maintain compatibility with previous releases, you can set this flag to FALSE.

Setting this option to FALSE makes such constraints unidirectional, which might cause an order cycle or a contradiction.

unit_reference_rule, *bool*

When TRUE (the default), requires the generator to recognize constraints on fields within a unit reference and set those constraints to be unidirectional, with the unit reference being generated before other fields in the constraint. For example, the following constraint, where “driver” is a field of type unit (not unit instance), causes “driver” to be generated before “name”:

keep name == driver.name;

NOTE— This option does not apply to unit instances used to point to other unit instances or to unit references that are generated and then assigned.

**static_analysis_opt,** TRUE

Performs optimizations to decrease the memory consumption and run time of static analysis.

### Notes

Random stability is not preserved when the optimizations are activated.

If it is used, the **static_analysis_opt** configuration option must be set no later than the **setup()** test phase.

The configuration option is FALSE by default, and cannot be set to FALSE. The only possible user setting is TRUE.

This option is for beta testing purposes only, and will be deprecated in future releases.

**Table 24-4—Memory Configuration Options**

| Option | Description |
|---|---|
| absolute_max_size, *n* | An integer value that determines the maximum amount of memory that the program will consume before issuing a fatal error message and terminating execution. This value should be at least 2Mb larger than the **max_size** option value in order to provide sufficient memory for handling and debugging memory overflow errors. In any case, the value of **absolute_max_size** must be less than the total memory resources of the machine.<br><br>The default for **absolute_max_size** is 200Mb. The range is 100M to MAX_UINT. There is no automatic update for the value of **absolute_max_size**. |
| gc_increment, n | An integer value that determines the automatic increase in the value of the gc_threshold option after a regular garbage collection. gc_increment effectively sets a minimum amount of memory allocation between successive garbage collections. If you set gc_increment to a large number, 100Mb for example, garbage collections occur at a lower rate. Depending on your hardware resources, however, the ***e*** might run out of memory if you increase gc_increment to a large amount.<br><br>The default is 16Mb. The range is 0 to MAX_UINT.<br><br>the ***e*** automatically updates the value of gc_increment after performing a regular garbage collection (and before updating the value of gc_threshold). The updated value of gc_increment is as follows:<br><br>max (gc_increment, (memory_size / 5))<br><br>Where memory_size is the amount of memory after garbage collection.<br><br>The value of gc_increment is not updated after on-the-fly garbage collection. |

| | |
|---|---|
| gc_threshold, n | An integer value that determines how much memory the program can allocate before performing a regular garbage collection. Increasing gc_threshold to 500Mb, for example, is a good way to avoid regular garbage collections. Depending on your hardware resources, however, the program might run out of memory if you increase gc_threshold to a large amount.<br><br>The default is 80Mb. The range is 0 to MAX_UINT.<br><br>The program automatically updates the value of gc_threshold after performing a regular garbage collection. The updated value of gc_threshold is as follows:<br><br>max (gc_threshold, (memory_size + updated_gc_increment))<br><br>Where memory_size is the amount of memory and updated_gc_increment is the updated value of gc_increment after the garbage collection takes place.<br><br>The value of gc_threshold is not updated after on-the-fly garbage collection. |
| max_size, n | Maximum amount of memory that the program can use. When the memory to be allocated reaches this limit, the program performs on-the-fly garbage collection in an attempt to find unused memory segments. If no additional memory is found, an error is issued. the program does not terminate execution, however, until the amount of memory reaches the value of absolute_max_size.<br><br>The default for max_size is 180Mb. The range is 100M to MAX_UINT. There is no automatic update for the value of max_size. |
| print_msg, bool | When TRUE, causes the program to print garbage collection information whenever regular (non-on-the-fly) garbage collection occurs.<br><br>The default is TRUE |
| print_otf_msg, bool | When TRUE, causes the program to print garbage collection information whenever on-the-fly garbage collection occurs.<br><br>The default is FALSE. |

| | |
|---|---|
| retain_printed_structs, bool | When FALSE, allows normal garbage collection to be performed on printed struct instances—unless pointed to by existing objects or retained by another option (such as retain_trace_structs).<br><br>Printed struct instances are instances whose value you can print. In other words, they are struct instances for which to_string() or vt.instance_to_string() has been called to create a unique type-@num ID, for example "pkt-@0". (See "to_string()" on page 761.<br><br>When TRUE, retain_printed_structs prevents the garbage collector from collecting and destroying printed struct instances.<br><br>Setting this option to FALSE speeds up simulation. Setting it to TRUE retains printed struct instances, which can increase runtime memory consumption.<br><br>The default is TRUE. |
| retain_trace_structs, bool | When FALSE, allows normal garbage collection to be performed on structs displayed in the waveform viewer—unless pointed to by existing objects or retained by another option (such as retain_printed_structs).<br><br>When TRUE, prevents the garbage collector from collecting and destroying any struct that is displayed in the waveform viewer.<br><br>Setting this option to FALSE speeds up simulation.<br><br>The default is FALSE |

**Table 24-5—Run Configuration Options**

| Option | Description |
|---|---|
| error_command, *string* | Specifies a command that is executed if an error occurs during the run. The command is executed for all types of errors including: <br>• DUT errors (defined with **check that**, **expect**, or **dut_error()**) whose check-effect is ERROR, ERROR_AUTOMATIC, ERROR_BREAK_RUN, or ERROR_CONTINUE <br>• Pre-run errors such as load errors or errors during the generation phase <br>• Errors defined with **error()** |
| exit_on, *exit-mode* | Specifies the conditions under which teh program exits. ***exit-mode*** is one of the following: <br>**command** (the default) — Exits only when you execute the simulator exit command. This option is recommended for interactive runs. <br>**normal_stop** — Exits only when the run completes without an error. If any error occurs, the program does not exit until you execute the simulator exit command. This option is recommended for batch runs if you want to execute some commands in the simulator, such as debug commands, when the run completes with an error. <br>**error** —Exits when an error of any kind occurs. If the error is a pre-run error, such as a load error or an error during the generation phase, the program exits immediately. If the error is a DUT error, the point at which the program exits depends on the check effect for that error. If no error occurs, the program does not exit until you execute the simulator exit command. This option is recommended for batch runs if you want to execute some commands in the simulator when the run completes without error. <br>**all** — Exits when a call to **stop_run()** is made from ***e*** code or when an error of any kind occurs. If neither of these occurs, the program does not exit until you execute the simulator exit command. This option is recommended for batch runs. |
| tick_max, *n* | Determines the maximum number of ticks before the test stops. The purpose is to keep the simulation from running longer than necessary. When run **tick_max** is reached, an error occurs. The value of ***n*** can be a number in the range 0 to MAX_INT. The default is 10,000. |
| use_manual_tick, *bool* | Adds the ability to use the manually cause a tick. This is usually used for debugging without a simulator. When not running with a simulator, the program runs until **stop_run()** is called, unless an ERROR occurs. When this flag is TRUE, you must execute the tick manually. The default is FALSE. |

## Description

This form of the **set_config()** action sets the specified options of a particular category to the specified values.

If an action block is specified, the options are set only temporarily during the execution of the action block.

For the configuration options to be effective, they must be set before the **run** test phase. The recommended place to use **set_config()** is in the **sys.setup()** method, as follows:

```
extend sys {
    setup() is also {
        set_config(category, option, value, ...);
    };
};
```

## Example 1

In the following example, the **setup()** method is extended to set configuration options.

```
<'
extend sys {
    setup() is also {
        set_config(print, radix, bin);
        set_config(cover, sorted, TRUE);
        set_config(gen, seed, random, default_max_list_size,
            20);
        set_config(run, tick_max, 1000, exit_on, all);
        set_config(memory, gc_threshold, 100m);
        set_config(misc, warn, FALSE);
    };
};
'>
```

## Result

```
configuration options for: print
        -radix                          =       BIN
.
.
.
configuration options for: cover
        -sorted                         =       TRUE
.
.
.
configuration options for: gen
        -seed                           =       187136885
        -default_max_list_size          =       20
.
.
.
configuration options for: run
        -tick_max                       =       1000
        -exit_on                        =       all
.
.
.
configuration options for: memory
        -gc_threshold                   =       104857600
.
.
.
configuration options for: misc
        -warn                           =       FALSE
```

```
.
.
.
```

**Example 2**

```
<'
struct packet {
    protocol: [atm, eth];
    data[10]: list of byte;
};

extend sys {
    !pi: list of packet;
    post_generate() is also {
        set_config(gen, seed, 0xff) with {gen pi;};
    };
};
'>
```

**Example 3**

The following example demonstrates the difference between setting the **show_sub_holes** coverage configuration option to TRUE versus the default setting of FALSE.

Two coverage items, i1 and i2 are defined, both for boolean fields. A cross item is also defined for the two items.

```
<'
extend sys {
    i1: bool;
    i2: bool;
    event cv;
    cover cv is {
        item i1;
        item i2;
        cross i1, i2;
    };
    run() is also {
        emit cv;
    };
};
extend sys {
    setup() is also {
        set_config(cover, mode, on_interactive, show_mode, both);
    };
};
'>
```

In the test, i1 gets FALSE and i2 gets TRUE.

The cross i1, i2 section of the coverage data with **show_sub_holes** = FALSE is shown below. In this example, since i1 = TRUE is an empty top bucket, the coverage report does not show the non-existent sub-buckets for i1 = TRUE, i2 = X.

```
** cross__i1__i2 **
Samples: 1  Tests: 1  Grade: 0.25  Weight: 1
```

```
grade    goal    samples    tests    %p/%       i1/i2

 0.50     -         1         1     100/100     FALSE
 0.00     1         0         0       0/0       FALSE/FALSE
 1.00     1         1         1     100/100     FALSE/TRUE
 0.00     -         0         0       0/0       TRUE
```

The cross i1, i2 section of the coverage data with **show_sub_holes** = TRUE is shown below. In this example, the coverage data shows the tree under the empty top bucket for i1 = TRUE, and expands the two empty sub-buckets under it, i1/i2 = TRUE/FALSE and i1/12=TRUE/TRUE.

```
** cross__i1__i2 **
Samples: 1  Tests: 1  Grade: 0.25  Weight: 1

grade    goal    samples    tests    %p/%t       i1/i2

 0.50     -         1         1     100/100      FALSE
 0.00     1         0         0       0/0        FALSE/FALSE
 1.00     1         1         1     100/100      FALSE/TRUE
 0.00     -         0         0       0/0        TRUE
 0.00     1         0         0       0/0        TRUE/FALSE
 0.00     1         0         0       0/0        TRUE/TRUE
```

### See Also

— "Configuration Routines" on page 766

## 24.7.2 get_config()

### Purpose

Get the configuration option settings

### Category

Routine

### Syntax

**get_config(*category*:** exp**, *option*:** exp**)**

Syntax example:

```
var s: int = get_config(gen, seed);
```

### Parameters

| | |
|---|---|
| *category* | Is one of the following: **cover**, **generation**, **memory**, and **run**. |
| *option* | The valid options are different for each category. See "set_config()" on page 766 for more information: |

### Description

Returns the value of the configuration option. The type of the returned value depends on the specified option.

### Example

```
<'
struct packet {
    protocol: [atm, eth];
    data[10]: list of byte;
};

extend sys {
    !pi: list of packet;

    post_generate() is also {
        set_config(gen, seed, 0xff) with {
            gen pi;
            var seed: int = get_config(gen, seed);
            set_config(print, radix, hex) with {
                print seed;
                var radix: po_radix = get_config(print, radix);
                print radix;
                print get_config(print, radix).type().name;
            };
        };
    };
};
'>
```

### Results

```
  seed = 0xff
  radix = HEX
  get_config(print, radix).type().name = "po_radix"
```

### See Also

— "Configuration Routines" on page 766

## 24.7.3 write_config()

### Purpose

Save configuration options in a file

### Category

Routine

### Syntax

**write_config(*filename*:** string**)**

Syntax example:

```
write_config("test25");
```

### Parameters

| | |
|---|---|
| *filename* | A string enclosed in double quotes and containing the name of a file. If a filename is entered with no filename extension, the default extension .ecfg is used. |

### Description

Creates a file with the specified name and writes the current configuration options to the file.

The **read_config()** method can be used to read the configuration options back in from the file.

### Example

```
<'
struct packet {
    protocol: [atm, eth];
    data[10]: list of byte;
};

extend sys {
    pi: list of packet;
};

extend sys {
    setup() is also {
        set_config(print, radix, bin);
        set_config(cover, sorted, TRUE);
        set_config(gen, seed, random, default_max_list_size, 20);
        set_config(run, tick_max, 1000, exit_on, all);
        set_config(memory, gc_threshold, 100m);
        set_config(misc, warn, FALSE);
        write_config("test25");
    };
};
'>
```

### Result

```
% cat test25.ecfg
-- The top struct
struct: config_all-@0:

print: print config_options-@1:
category: print
print'radix: BIN
print'items: 0b10100
print'list_lines: 0b10100
print'list_from: 0b0
print'title: NULL
.
```

```
    .
    .
```

### See Also

— “Configuration Routines” on page 766

## 24.7.4 read_config()

### Purpose

Read configuration options from a file

### Category

Routine

### Syntax

**read_config(*filename*:** string**)**

Syntax example:

```
read_config("test25");
```

### Parameters

| | |
|---|---|
| *filename* | A string enclosed in double quotes and containing the name of a file. If a filename is entered with no filename extension, the default extension .ecfg is used. |

### Description

Read configuration options from a file with the specified name. The configuration options must previously have been written to the file using the **write_config()** method.

### Example

```
<'
struct packet {
    protocol: [atm, eth];
    data[10]: list of byte;
};

extend sys {
    pi: list of packet;

    setup() is also {
        read_config("test25");
    };
};
'>
```

### Result

```
configuration options for: print
        -radix                          =       BIN
```

```
.
.
.
configuration options for: cover
        -sorted                         =       TRUE
.
.
.
configuration options for: gen
        -seed                           =       908142002
        -default_max_list_size          =       20
.
.
.
configuration options for: run
        -tick_max                       =       1000
        -exit_on                        =       all
.
.
.
configuration options for: memory
        -gc_threshold                   =       104857600
.
.
.
configuration options for: misc
        -warn                      =       FALSE
.
.
.
```

**See Also**

— "Configuration Routines" on page 766

## 24.8 OS Interface Routines

The routines in this section enable use of operating system commands from within the *e* programming language. These routines work on all supported operating systems.

The OS interface routines in this section are:

— "spawn()" on page 786
— "spawn_check()" on page 786
— "system()" on page 787
— "output_from()" on page 788
— "output_from_check()" on page 789
— "get_symbol()" on page 790
— "date_time()" on page 791
— "getpid()" on page 792

### 24.8.1 spawn()

#### Purpose

Send commands to the operating system

#### Category

Pseudo-routine

#### Syntax

spawn()

**spawn(*command*:** string**, ...)**

Syntax example:

```
spawn("touch error.log;","grep Error my.elog > error.log");
```

#### Parameters

| | |
|---|---|
| *command* | An operating system command, with or without parameters and enclosed in double quotes. |

#### Description

Takes a variable number of parameters, concatenates them together, and executes the string result as an operating system command via **system()**.

#### Example

```
<'
extend sys {
    m1() is {
        spawn("touch error.log;","grep Error my.elog > \
            error.log");
    };
};
'>
```

#### See Also

— “OS Interface Routines” on page 785

### 24.8.2 spawn_check()

#### Purpose

Send a command to the operating system and report error

#### Category

Routine

### Syntax

**spawn_check(*command*:** string**)**

Syntax example:

```
spawn_check("grep Error my.elog >& error.log");
```

### Parameters

| | |
|---|---|
| *command* | A single operating system command, with or without parameters and enclosed in double quotes. |

### Description

Executes a single string as an operating system command via **system()**, then calls **error()** if the execution of the command returned an error status.

### Example

```
<'
extend sys {
    m1() is {
        spawn_check("grep Error my.elog >& error.log");
    };
};
'>
```

### Result

**sys.m1()**

```
*** Error: Error while executing 'spawn_check("grep Error my.elog >&
error.log")'
```

### See Also

— “OS Interface Routines” on page 785

## 24.8.3 system()

### Purpose

Send a command to the operating system

### Category

Routine

### Syntax

**system(*command*:** string**)**: int

Syntax example:

```
stub = system("cat my.v");
```

### Parameters

| | |
|---|---|
| *command* | A single operating system command, with or without parameters and enclosed in double quotes. |

### Description

Executes the string as an operating system command using the C **system()** call and returns the result.

### Example

```
<'
extend sys {
    m1() is {
        var stub: int;
        stub = system("cat my.v");
        print stub;
    };
};
'>
```

### Result

**sys.m1()**
```
module top;
    parameter sn_version_id = 0;  /* Version */
    parameter sn_version_date = 31198;  /* Version date*/
...
```

### See Also

— “OS Interface Routines” on page 785

## 24.8.4 output_from()

### Purpose

Collect the results of a system call

### Category

Routine

### Syntax

**output_from(*command*:** string**)**: list of string

Syntax example:

```
log_list = output_from("ls *log");
```

### Parameters

| | |
|---|---|
| *command* | A single operating system command, with or without parameters and enclosed in double quotes. |

### Description

Executes the string as an operating system command and returns the output as a list of string. Under UNIX, **stdout** and **stderr** go to the string list.

### Example

```
<'
extend sys {
    m1() is {
        var log_list: list of string;
        log_list = output_from("ls *log");
        print log_list;
    };
};
'>
```

### Result

**sys.m1()**
```
log_list =
0.      "my_sn.elog"
1.      "my_sn_sv.elog"
```

### See Also

— “OS Interface Routines” on page 785

## 24.8.5 output_from_check()

### Purpose

Collect the results of a system call and check for errors

### Category

Routine

### Syntax

**output_from_check(***command*: string**)**: list of string

Syntax example:

```
log_list = output_from_check("ls *.log");
```

### Parameters

| | |
|---|---|
| *command* | A single operating system command with or without parameters and enclosed in double quotes. |

### Description

Executes the string as an operating system command, returns the output as a list of string, and then calls **error()** if the execution of the command returns an error status. Under UNIX, **stdout** and **stderr** go to the string list.

### Example

```
<'
extend sys {
    m1() is {
        var log_list: list of string;
        log_list = output_from_check("ls *.log");
        print log_list;
    };
};
'>
```

### Result

**sys.m1()**
```
*** Error: Error from system command "ls *.log":
No match
```

### See Also

— "OS Interface Routines" on page 785

## 24.8.6 get_symbol()

### Purpose

Get UNIX environment variable

### Category

Routine

### Syntax

**get_symbol(*env-variable*:** string**)**: string

Syntax example:

```
current_dir = get_symbol("PWD");
```

### Parameters

| | |
|---|---|
| *env-variable* | A UNIX environment variable enclosed in double quotes. |

### Description

Returns the environment variable as a string.

### Example

```
<'
extend sys {
    m1() is {
        var current_dir: string;
        current_dir = get_symbol("PWD");
        print current_dir;
    };
};
'>
```

### Result

**sys.m1()**
```
current_dir = "/users3/sawa/docs/314/code"
```

### See Also

— "OS Interface Routines" on page 785

## 24.8.7 date_time()

### Purpose

Retrieve current date and time

### Category

Routine

### Syntax

**date_time()**: string

Syntax example:

```
print date_time();
```

### Description

Returns the current date and time as a string.

### Example

```
date_time() = "Tue Jul 27 13:10:43 1999"
```

#### See Also

— “OS Interface Routines” on page 785

### 24.8.8 getpid()

#### Purpose

Retrieve process ID

#### Category

Routine

#### Syntax

**getpid()**: int

Syntax example:

```
print getpid();
```

#### Description

Returns the current process ID as an integer.

#### Example

```
getpid() = 25517
```

#### See Also

— “OS Interface Routines” on page 785

## 24.9 On-the-Fly Garbage Collection Routine: do_otf_gc()

#### Purpose

Perform on-the-fly garbage collection

#### Category

Routine

#### Syntax

do_otf_gc()

Syntax example:

```
do_otf_gc();
```

### Description

This routine performs on-the-fly garbage collection. It can be called at any time from either a regular method or a TCM. It takes no arguments and returns no value.

Use this routine at any point when reducing the ***e*** program memory heap would be beneficial. For example, use this routine between ticks, when significant memory can accumulate. (Regular garbage collection does not occur between ticks.)

### Notes

— The **do_otf_gc()** routine is not related to the regular garbage collection mechanism, although keeping the memory heap low with **do_otf_gc()** helps limit the amount of regular garbage collection that is performed.
— The **do_otf_gc()** routine does consume some CPU time, so it should be used in moderation.
— Use the **print_otf_msg** configuration option to request that on-the-fly GC messages be printed.

### Example

This example calls do_otf_gc() from a TCM. However, it can also be called from a regular method.

```
unit port {
    drive_frames    :  out buffer_port of frame is instance;
      keep drive_frames.buffer_size() == 200;
    num_frames: uint;

    !next_frame: frame;

    in_frames() @sys.any is {
        var in_count : uint ;

        while (in_count < num_frames) {
            gen next_frame;
            drive_frames.put(next_frame);
            in_count += 1;
            // invoke on-the-fly garbage collection every 10 frames
            if (in_count % 10 == 0) then {
                do_otf_gc();
            };
        };
    };
};
```

### See Also

— "set_config()" on page 766

## 24.10 Calling Predefined Routines: routine()

### Purpose

Call a predefined routine

### Category

Action

### Syntax

*routine-name*()

***routine-name*(*param*, ...)**

Syntax example:

```
var s := str_join(slist," - ");
```

### Parameters

| | |
|---|---|
| *routine-name* | The name of the routine. |
| *param* | One or more parameters separated by commas, one parameter for each parameter in the parameter list of the routine definition. Parameters are passed by their relative position in the list, so the name of the parameter being passed does not have to match the name of the parameter in the routine definition. The parentheses around the parameter list are required even if the parameter list is empty. |

### Description

Calls a predefined routine passing it the specified parameters.

### Example

This example shows how to call a predefined routine.

```
<'
extend sys {
    m1() is {
        var slist: list of string = {"first";"second";"third"};
        var s := str_join(slist," - ");
        print s;
    };
};
'>
```

### Result

**sys.m1()**
```
s = "first - second - third"
```

# 25 Simulation-Related Constructs

This chapter describes the following *e* constructs:

## 25.1 Verilog Statements or Unit Members

Some basic functionality of the Verilog simulator interface, such as setting or sampling the values of some Verilog objects, is enabled without any action on your part. However, some features, such as the continuous driving of Verilog signals or calling of Verilog tasks and functions, requires some user-specified declarations - **verilog** statements or unit members. The following sections describe these constructs:

### 25.1.1 verilog code

#### Purpose

Write Verilog code directly to the stubs file

#### Category

Statement or unit member

#### Syntax

verilog code {*list-of-strings*}

Syntax examples:

```
verilog code {"initial clk = 1'b1;"};
unit router {
    verilog code { "initial "; s};
    verilog code ls;
};
```

### Parameters

| | |
|---|---|
| *list-of-strings* | Any list of strings that after concatenation creates any sequence of legal Verilog code. Verilog syntax errors are identified only when you compile or interpret the file with the Verilog compiler.<br><br>The curly braces are required only if the list of strings contains more than one string. |

### Description

Specifies a list of Verilog strings to be included in the Verilog stubs file each time it is generated. The stubs file contains code that enables some Verilog-related features. It is recommended to use **verilog code** statements or unit members to modify this file rather than to modify it by hand.

When **verilog code** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

NOTE— Whenever you add or modify a **verilog code** statement or unit member or add an instance of a unit containing a **verilog code** unit member, you must create a new stubs file.

### Example 1

This example uses a **verilog code** statement to define a Verilog event, clk_rise, in the stubs file. The Verilog clk_rise event is triggered on every positive edge of the HDL signal, top.clk. Deriving an ***e*** event from the Verilog clk_rise event rather than from the HDL signal top.clk itself reduces the number of callbacks by half.

#### top.v

```
module top();
    reg clk;

    initial clk = 0;
    forever begin
        #50 clk = ~clk;
    end

endmodule
```

#### clk.e

```
verilog code {
    "event clk_rise;";
    "always @(posedge top.clk) begin";
    "  ->clk_rise;";
    "end"
};
extend sys {
    event clk_rise is change('top.clk_rise')@sim;

    on clk_rise {
        print sys.time;
    };
};
```

### Result

```
sys.time = 50
  sys.time = 150
  sys.time = 250
  sys.time = 350
  sys.time = 450
  sys.time = 550
  sys.time = 650
...
```

### Example 2

This example initializes an HDL clock by adding code to the stubs file. It uses **verilog code** as a unit member so that the clock name and its initial value can be configured on a unit instance basis.

```
<'
unit channel {
    clk_name: string;
    keep soft clk_name == "clk";
    clk_init: bit;
    keep soft clk_init == 1;
    s: string;
    keep s == append("initial ", full_hdl_path(),".",clk_name,
                        "= ",clk_init,";");
    verilog code {s};
};

extend sys {
    chan: channel is instance;
    keep chan.hdl_path() == "top";
};
'>
```

## 25.1.2 verilog function

### Purpose

Declare a Verilog function

### Category

Statement or unit member

### Syntax

**verilog function '*HDL-pathname*' (*verilog-function-parameter*[, ... ]): *result-size-exp***

Syntax examples:

```
verilog function 'top.write_mem'(addr:32, data:32):32;

extend my_unit {
    verilog function '(read_mem_name)'(addr:addr_width):ret_size;
}
```

## Parameters

| | |
|---|---|
| *HDL-pathname* | The full path to the Verilog function. If this name is not a constant, it is calculated after the final step of pre-run generation. See "'HDL-pathname'" on page 838 for a complete description of HDL path syntax. |
| *verilog-function-parameter* | The syntax for each parameter is ***parameter_name:size_exp***. The parameter name need not match the name in Verilog. All parameters are passed by position, not name. All parameters must be inputs, and the number of parameters must match the number declared in the Verilog function.<br><br>The ***size-exp*** must be a legal unsigned integer expression specifying the size in bits of the parameter. No default size is assumed. If the size expression is not a constant, it is calculated after the final step of pre-run generation. |
| *result-size-exp* | A legal unsigned integer expression specifying the size in bits of the returned value. No default size is assumed. If the size expression is not a constant, it is calculated after the final step of pre-run generation. |

## Description

Declares a Verilog function in ***e*** so that it can be called from a time-consuming method (TCM).

When **verilog function** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

## Notes

- Calls to Verilog functions are value-returning expressions.
- Even though Verilog functions do not consume time in Verilog, they do so on the ***e*** side because they require a context switch between the ***e*** program and the simulator. Thus, Verilog functions can be called only from TCMs.
- The ***e*** language does not support concurrent calls to a Verilog function. In other words, you cannot call the same Verilog function from more than one thread in the same tick.
- You must explicitly pack all structs before passing them in as inputs to the function.
- Whenever you add or modify a **verilog function** statement or unit member or add an instance of a unit containing a **verilog function** unit member, you must create a new stubs file and then load the file into the Verilog simulator.

## Example

The following function has two 32-bit inputs and returns a 32-bit error status. The memory_driver unit calls the top.write_mem() function.

```
unit memory_driver {
    addr_width: uint;
    keep soft addr_width == 32;
    data_width: uint;
    keep soft data_width == 32;
    verilog function 'top.write_mem'(addr:addr_width,data:data_width):32;

    event mem_enable is rise ('top.enable') @sim;
    write() @mem_enable is {
        var error_status: int;
        error_status = 'top.write_mem'(31,45);
        };
    };
```

### See Also

— “'HDL-pathname'” on page 838
— “Rules for Defining and Extending Methods” on page 459

## 25.1.3 verilog import

### Purpose

Read in Verilog text macros

### Category

Statement

### Syntax

**verilog import *file-name***

Syntax example:

```
    verilog import defines_MCP.v;
```

### Parameters

| | |
|---|---|
| *file-name* | The ***e*** program searches for imported files in the directories specified in $PATH. If the file is not found in $PATH, the ***e*** program then searches the directory where the importing file resides. |

### Description

Reads in a file that includes Verilog `define text macros. After reading in the file, you can use these text macros in ***e*** expressions.

An ***e*** program understands several Verilog language constructs when reading the `define macros:

— It recognizes the Verilog `include directive and reads in the specified file.
— It recognizes the Verilog `ifdef, `else, and `endif directives and skips the irrelevant parts of the file according to the currently defined symbols.

### Notes

— **verilog import** statements cannot be used as unit members.
— **verilog import** statements must appear in the ***e*** file before any other statement except **package**, **define**, and **import** statements. **package** statements must appear first in the file.
— Whenever you add or redefine an imported Verilog macro that appears in an HDL declaration, you need to create a new stubs file and then load the file into the simulator.

For example, if you use a Verilog macro to specify the width of the parameters in a task identified with **verilog task**, then you need to recreate the stubs file if the macro is redefined.

### Example 1

To import more than one file, you must use multiple **verilog import** statements or use the Verilog compiler directive `include. You can import the same file more than once. If you define the same macro with different values before using it, the ***e*** program uses the definition that is loaded last. Once you use the macro, however, you cannot redefine it. Subsequent definitions are ignored without warning.

The following example shows how to import multiple files. Note that once X is used in the x.e file, its value of 7 (the last loaded definition before it is used) cannot be changed.

#### a.v

```
'define X 5
```

#### b.v

```
'define X 7
```

#### x.e

```
verilog import a.v  ;
verilog import b.v  ;
extend sys {
    run() is also {
       print 'X;
    };
};
```

#### y.e

```
verilog import a.v ;
extend sys {
    run() is also {
       print 'X;
    };
};
```

#### z.e

```
verilog import b.v ;
```

### Example 2

You can use Verilog macros everywhere an ***e*** macro is allowed. For example, you can use Verilog macros in width declarations or when assigning values to enumerated items.

#### macros.v

```
    `define BASIC_DELAY   2
`ifdef OLD_TECH
    `define TRANS_DELAY   `BASIC_DELAY+3
`else
    `define TRANS_DELAY   `BASIC_DELAY
`endif
    `define TOP   tb
    `define READY `TOP.ready
    `define WORD_WIDTH 8
    `define HIGH_BIT   7
```

### dut_driver.e

```
verilog import macros.v;

extend sys {
    event pclk;
    driver: dut_driver;
    event pclk is only @any; -- for stand-alone mode
};

struct dut_driver {
    ld: list of int(bits: `WORD_WIDTH);
    keep ld.size() in [1..30];

    stimuli() @sys.pclk is {
        '`READY' = 1;
        for each in ld {
            wait until true('`READY' == 1);
            '`TOP.data_in' = it;
            wait [`TRANS_DELAY];
        };
        stop_run();
    };
    run() is also {
        start stimuli();
    };
};
```

## 25.1.4 verilog task

### Purpose

Declare a Verilog task

### Category

Statement or unit member

### Syntax

**verilog task '*HDL-pathname*' (*verilog-task-parameter*[, ...])**

Syntax examples:

```
verilog task 'top.read'(addr: 64, cell: 128:out);
verilog task 'top.(read_task_name)'(addr:addr_width, cell:cell_width:out);
```

### Parameters

| | |
|---|---|
| *HDL-pathname* | The full path to the Verilog task. If this name is not a constant, it is calculated after the final step of pre-run generation. See “"HDL-pathname"” on page 838 for a complete description of HDL path syntax. |

| | |
|---|---|
| *verilog-task-parameter* | The ***verilog-task-parameter*** has the syntax ***name***:***size-exp***[:***direction***].<br><br>The ***name*** need not match the name in Verilog. All parameters are passed by position, not name. The number of parameters must match the number of parameters in the task declaration in Verilog.<br><br>The ***size-exp*** must be a legal unsigned integer expression specifying the size in bits of the parameter. No default size is assumed. If the size expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>The ***direction*** is one of the following keywords: **input** (or **in**), **output** (or **out**), **inout**. The default is **input**. |

## Description

Declares a Verilog user-defined task or system task in ***e*** so that it can be called from a TCM. In the following example, “addr” is a 64-bit input and “cell” is a 128-bit output. $display is a Verilog system task.

```
verilog task 'top.read'(addr: 64, cell: 128:out);
verilog task '$display'(int:32);
```

When **verilog task** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

## Notes

— Calls to Verilog tasks are time-consuming actions and can only be made from TCMs.

— The ***e*** language does not support concurrent calls to a Verilog task. In other words, you cannot call the same Verilog task from multiple threads in the same tick.

— You must explicitly pack all structs before passing them in as inputs to the task.

— Whenever you add or modify a **verilog task** statement or unit member or add an instance of a unit containing a **verilog task** unit member, you must create a new stubs file and then load the file into the Verilog simulator.

## Example

This task is similar to the Verilog function “write_mem” shown in the example for the **verilog function** unit member (“verilog function” on page 797). This task returns an error status as an output parameter.

```
struct mem_w {
    addr: int;
    data: int(bits: 64);
};

unit receiver {
    read_task_name: string;
    addr_width: uint;
    keep soft addr_width == 32;
    data_width: uint;
    keep soft data_width == 64;
    verilog task 'top.(read_task_name)'
```

```
          (addr:addr_width:in,data:data_width:out,status:32:out);
        event mem_read_enable;

        get_mem(mw: mem_w) @mem_read_enable is {
            var error_status: int;

            'top.(read_task_name)'(mw.addr, mw.data, error_status);
            check that error_status == 0;
        };
    };
```

### See Also

- "'HDL-pathname'" on page 838
- "Rules for Defining and Extending Methods" on page 459

## 25.1.5 verilog time

### Purpose

Set the Verilog time resolution

### Category

Statement

### Syntax

verilog time *verilog-time-scale*

Syntax examples:

```
    verilog time 100ns/10ns;
    verilog time num1 ns/num2 ns;
```

### Parameters

| | |
|---|---|
| *verilog-time-scale* | The Verilog specification for time scale has the syntax ***time-exp unit / precision-exp unit***.<br><br>***time-exp*** and ***precision-exp*** must be integer expressions. If the expression is not a constant, it is calculated after the final step of pre-run generation. According to Verilog standards, the legal values for the integer expressions are 1, 10 and 100, but The ***e*** program does not check whether the value is legal or not.<br><br>***unit*** is any unit of time measurement. According to Verilog standards, the valid character strings are s, ms, us, ns, ps, and fs, but the ***e*** program does not check whether the value is legal or not. |

### Description

Sets the time resolution in the Verilog stubs file to the specified ***verilog-time-scale***.

The **verilog time** statement can be used to scale the following:

— # delays in **verilog variable** or **verilog code** statements
— delays in ***e*** temporal expressions
— simulation time as shown in **sys.time,** a 64-bit integer field that stores testbench time.

### Notes

— When the DUT contains more than one `timescale directive, using the **verilog time** statement is strongly recommended. Otherwise, ***e*** program time becomes dependent on the order of the Verilog modules.
— If you need to call the ***e*** program (using **$sn**) from Verilog code explicitly, use the **verilog code** statement to put the ***e*** program call in the stubs file. This ensures that the call to the ***e*** program has the time scale shown in the stubs file instead of a time scale from other parts of the Verilog code.
— The Verilog time scale precision in the **verilog time** statement must be set to the precision required for # delays in **verilog variable** or **verilog code** statements.
— **verilog time** statements cannot be used as unit members.
— If you use a non-constant expression for the time or the precision, this expression is computed in the context of **sys** and so must be visible in that context.
— Whenever you add or modify a **verilog time** statement in an ***e*** module, you must create a new stubs file and then load the file into the Verilog simulator.

### Example 1

```
verilog time 10ns/1ns;
```

### Example 2

```
verilog time (num1)ns/(num2)ns;
extend sys {
    num1:int;
    num2:int;

    keep num1 == 100;
    keep num2 == 10;
};
```

## 25.1.6 verilog variable reg | wire

### Purpose

Identify a Verilog register or wire

### Category

Statement or unit member

### Syntax

**verilog variable '*HDL-pathname*' using *option*,** ...

Syntax examples:

```
verilog variable 'top.dbus[5:0]' using wire,drive="@(posedge top.m)";
verilog variable 'i1.(reset_name)[m:l]' using wire,drive_hold=dh_opt;
```

## Parameters

| | |
|---|---|
| *HDL-pathname* | The complete path to the Verilog object, a register or a net.If this name is not a constant, it is calculated after the final step of pre-run generation. See “'HDL-pathname'” on page 838 for a complete description of HDL path syntax.<br><br>NOTE— If the register or wire has a width greater than 1, the bit range must be explicitly declared in order to create internal temporary registers of the correct width.<br><br>The width range has the syntax<br>[***right-bound-exp***:***left-bound-exp***]<br><br>where ***right-bound-exp*** and ***left-bound-exp*** are any legal integer expressions. The width range must be the same (including the descending or ascending order) as in the Verilog declaration. If these expressions are not constants, they are calculated after the final step of pre-run generation. |
| *option* | A list of one or more of the following options separated by commas. |
| **drive**=***string-exp*** | Specifies that the Verilog object is driven when the event specified by ***string-exp*** occurs. ***string-exp*** is any legal string expression specifying a legal Verilog timing control. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| **drive_hold**=***string-exp*** | Specifies a Verilog event after which the HDL object’s value is set to z. ***string-exp*** is any legal string expression specifying a legal Verilog timing control. If this expression is not a constant, it is calculated after the final step of pre-run generation. The **drive_hold** option requires that you also specify the **drive** option. |
| **net**,<br><br>wire | Specifies that the Verilog object is a net (wire). Some simulators (VCS and ModelSim) require this option anytime you drive a Verilog wire from ***e***. Other simulators require this option only when you drive from within the model as well as from ***e*** and you want the Verilog wire to be resolved. By default the value driven by the ***e*** program is always driven last, overwriting all values driven by Verilog. |
| forcible | Allows forcing of Verilog wires. By default Verilog wires are not forcible. This option requires that you also specify the **net** or **wire** option. |
| **strobe**=***string-exp*** | Specifies that the value of the Verilog object is sampled at the event specified using ***string-exp***. ***string-exp*** is any legal string expression specifying a legal Verilog timing control. If this expression is not a constant, it is calculated after the final step of pre-run generation. |

## Description

Allows access to the Verilog object named in '***HDL-pathname***', a register or a net. In general, you can access Verilog objects using the '***HDL-pathname***' expression. The **verilog variable** statement is necessary only when you want to

— drive or force a wire
— drive or sample a wire or reg with an offset from the ***e*** program callback

Some simulators (VCS and ModelSim) require this option anytime you drive a Verilog wire from ***e***. Other simulators require this option only when you drive from within the model as well as from ***e*** and you want the Verilog wire to be resolved. By default the value driven by the ***e*** program is always driven last, overwriting all values driven by Verilog.

If you have not included **verilog variable** statement and one is required, you will see an error message such as

```
*** Error: Cannot drive ‘top.sig_wire’. Driving a wire which is not declared
    with ‘verilog variable suing wire’ is not allowed by the simulator.
    at line 6 in @your_module
```

**verilog variable** declarations for the same object can be repeated (to allow incremental building of ***e*** code), but each repeated declaration must be identical.

When **verilog variable** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

### Notes

— Assignments to Verilog nets are implemented through the Verilog code in the stubs file. These assignments are not propagated until the start of simulation, so no pre-run initialization of Verilog nets can be performed by the ***e*** program.
— Whenever you add or modify a **verilog variable** statement or unit member or add an instance of a unit containing a **verilog variable** unit member, you must create a new stubs file and then load the file into the Verilog simulator.

### Example 1

The following example sets **drive** and **drive_hold** timing controls on a register:

```
verilog variable 'top.reset_request' using
    drive="#5",drive_hold="@(negedge clock)";

struct controller {
    event reset_request;
    on reset_request {
        'top.reset_request' = 1;
    };
};
```

If, for example, the ***e*** event “reset_request” is emitted at time 100ns, then the Verilog top-level register “reset_request” is set at time 105ns, and it is disconnected (assigned Z value) at the next negative edge of the clock.

### Example 2

The following example sets **strobe** and **drive** delays for a wire using a **verilog variable** unit member and a non-constant expression.

```
unit box {
    bus_name: string;
    bus_width: uint;
    strobe_option: string;
    drive_option: string;
    drive_hold_option: string;

    verilog variable '(bus_name)[bus_width-1:0]' using
                  strobe=strobe_option,
                  drive=drive_option,
                  drive_hold=drive_hold_option,
                  wire;
  };
  extend sys {
    box: box is instance;
      keep box.hdl_path() == "top_box";
      keep box.bus_name == "xx_bus";
      keep box.bus_width == 64;
      keep box.strobe_option == "@(posedge top.clk_b)";
      keep box.drive_option == "@(negedge top.clk_b) #3";
      keep box.drive_hold_option == "@(negedge top.clk_b) #10";
};
```

The **verilog variable ... using strobe** statement creates an additional temporary variable in the Verilog stubs file that strobes the value of the real Verilog object, according to the timing conditions specified for this option. Any read operation of the real Verilog object in ***e*** code accesses the temporary variable and thus gets the latest strobed value. In this example, an additional register, "top__xx_bus", is created and this register gets the current value of "top.xx_bus[63:0]" at every positive edge on "top.clk_b".

### Example 3

Here is a complete example of how to drive a clock wire.

### clk.e

```
verilog time 1ns / 1ns;
verilog variable 'top.ev_clock' using wire ;
define CLOCK_HALF_PERIOD 50;

extend sys {
    dut: dut;
};

struct dut {
    clk: uint (bits:1);
    event dut_evclk is rise ('top.ev_clock')@sim;

    driver() @sys.any is {
        var drive_var: int = 1;
        'top.ev_clock' = 0;
        while(TRUE) {
                wait delay(CLOCK_HALF_PERIOD);
                'top.ev_clock' = drive_var;
                drive_var = ~drive_var;
        };
   };

    wait_and_stop() @sys.any is {
        wait [100] * cycle;
        stop_run();
    };

    run() is also {
        start  driver();
        start wait_and_stop();
    };
};
```

### clk.v

```
    'timescale 1 ns / 1ns
    module top;
       wire ev_clock ;

    endmodule
```

### Example 4

This example shows how to use **verilog variable** as a unit member with a relative path. In this case, the full HDL path of the unit driver is prefixed to the relative path ‘rxd’ in order to access the rxd signal in each of the DUT receivers. For example ‘~/top.chan0.rx.rxd’ accesses the rxd signal in the first channel’s receiver.

### enet_env.e

```
    <'
    unit enet_env {
        keep hdl_path() == "~/top";
        ports : list of enet_port is instance;

        keep soft ports.size() == 4;

        keep for each in ports {
            .number == index;
            .hdl_path() == append("chan", index);
        };
    };

    unit enet_port {
        number        : int;

        injector      : driver is instance;
        collector     : receiver is instance;

        keep injector.parent_port == me;
        keep injector.number == number;
        keep injector.hdl_path() == "rx";
        keep collector.parent_port == me;
        keep collector.number == number;
        keep collector.hdl_path() == "tx";
    };

    unit driver {
        parent_port     : enet_port;
        number          : int;
        event clk is rise ('~/top.clock')@sim;

        verilog variable 'rxd[7:0]' using wire;// Unit member with relative path

        send_data() @clk is {
            out("Injecting data at ", full_hdl_path());
            var pkt: packet;
            gen pkt;
            var total_data     : list of byte;
            total_data = pack(packing.low, pkt);
            print total_data;
            wait cycle;
            for each (data) in total_data {
                'rxd' = data;                // inject data
                 wait cycle;
            };
            wait [50] * cycle;
            stop_run();
        };

        run() is also {
            start send_data();
        };
    };
    unit receiver {
        parent_port     : enet_port;
        number          : int;
    };

    struct packet {
        %data: list of byte;
    };

    extend sys {
        enet: enet_env is instance;
    };
    '>
```

**top.v**

```
    module top ();
        reg clock;

        initial clock = 0;
        always #50 clock = ~clock;

    channel chan0();
    channel chan1();
    channel chan2();
    channel chan3();

    endmodule

    module channel();
        receiver rx();
        transmitter tx();

    endmodule
```

```
module receiver();
    wire [7:0] rxd;

endmodule

module transmitter();

endmodule
```

### Result

```
ncsim> run
Injecting data at ~/top.chan0.rx
  total_data =  (27 items, dec):
        201 120  34  42  158 187  67  96  255 147 240   2       .0
         38  21  43  49  151 151  10  79  181 102 176 145       .12
                                              249 124  77       .24

Injecting data at ~/top.chan1.rx
  total_data =  (12 items, dec):
         70  48 246  35   78 170 156 136   10 133  33 156       .0

Injecting data at ~/top.chan2.rx
  total_data =  (22 items, dec):
        170  70  48 175   50 112  39  26  109  53  65 156       .0
                152 108  147 136  83 239   62 217 131 249       .12

Injecting data at ~/top.chan3.rx
  total_data =  (3 items, dec):
                                              146  71 163       .0

Memory Usage - 24.5M program + 17.7M data = 42.3M total
CPU Usage - 0.4s system + 4.0s user = 4.4s total (87.8% cpu)
Simulation stopped via $stop(2) at time 5450 NS + 1
ncsim> exit
```

### See Also

— “'HDL-pathname'” on page 838
— “force” on page 830
— “release” on page 834

## 25.1.7 verilog variable memory

### Purpose

Identify a Verilog memory

### Category

Statement or unit member

### Syntax

**verilog variable '*HDL-pathname* [*mem-range*] [*verilog-reg-range*]' [using vcs_pli]**

Syntax example:

```
verilog variable 'top.my_mem[1:10000][31:0]';
```

### Parameters

| | |
|---|---|
| *HDL-pathname* | The full path to the Verilog memory. If this name is not a constant, it is calculated after the final step of pre-run generation. See "'HDL-pathname'" on page 838 for a complete description of HDL path syntax. |
| *mem-range* | A legal expression specifying the range of the memory elements. A legal expression has the syntax [***exp***:***exp***]. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| *verilog-reg-range* | A legal expression specifying the width of each memory element. A legal expression has the syntax [***exp***:***exp***]. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| using vcs_pli | Enables the propagation of Verilog memory updates with VCS. If the ***e*** program is not linked to the VCS simulator, this option is ignored.<br><br>NOTE— Due to a limitation of the VCS simulator extension, changes to bits containing x or z values are not propagated. |

### Description

Allows access to a Verilog memory. **verilog variable** declarations for the same memory can be repeated (to allow incremental building of ***e*** code), but each repeated declaration must be identical.

Note that the order of the range specifiers in the ***e*** syntax is the reverse of the Verilog declaration order.

For example, if a memory is defined in Verilog as follows:

```
module top;
    reg[31:0] my_mem[1:10000];
endmodule
```

The corresponding **verilog variable** statement is:

```
verilog variable 'top.my_mem[1:10000][31:0]';
```

When **verilog variable** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

### Notes

— Only a single register element from the Verilog memory can be accessed in one ***e*** expression (such as 'top.my_mem[0]' or 'top.my_mem[j]').

— Verilog memories are always accessed directly, not cached, for reading and writing. You must organize how reading and writing memories occurs within a tick.

— Verilog memories cannot be accessed before simulation starts. This means that memories cannot be initialized during pre-run generation. You must provide code to generate initial memory contents on the fly at time zero.

— For NC Verilog, code that contains memory declarations must be compiled with a NC Verilog command line option, **-nomempack**.
— Access to Verilog memories is not supported when the ***e*** program is linked with SpeedSim.
— Whenever you add or modify a **verilog variable** statement or unit member or add an instance of a unit containing a **verilog variable** unit member, you must create a new stubs file and then load the file into the Verilog simulator.

### Example

This example shows one possible way of initializing a Verilog memory at the beginning of the simulation.

### xmem.v

```
module xmem;
    reg [63:0] bank [1:10];
endmodule

module topmod;
    reg clk;
    initial clk = 0;
    always #10 clk = ~clk;
    xmem m1();
endmodule
```

### memstarter.e

```
verilog variable 'topmod.m1.bank[1:10][63:0]';

struct mem_starter {
    event init_event is rise('topmod.clk')@sim;
    on init_event {
        var rand_s: int (bits: 64);

        for j from 1 to 10 {
            gen rand_s;
            'topmod.m1.bank[j]' = rand_s;
        };
        quit();
    };
};

extend sys {

    ms: mem_starter;
    event clk is fall('topmod.clk')@sim;
};
```

### See Also

— "'HDL-pathname'" on page 838

## 25.2 VHDL Statements and Unit Members

Some basic functionality of the VHDL simulator interface, such as setting or sampling the values of some VHDL objects, is enabled without any action on your part. However, some features, such as the continuous driving of VHDL signals or calling of VHDL subprograms, requires some user-specified declarations - **vhdl** statements. The following sections describe these statements and unit members:

— "vhdl code" on page 813
— "vhdl driver" on page 815
— "vhdl function" on page 819
— "vhdl procedure" on page 822
— "vhdl time" on page 829

### 25.2.1 vhdl code

#### Purpose

Write VHDL code directly to the stubs file

#### Category

Statement or unit member

#### Syntax

vhdl code {*list-of-strings*}

Syntax example:

```
vhdl code {
    "library common_lib;";
    "use common_lib.common_types.all;";
};
unit router {
    vhdl code {
        "library my_lib;";
        "use my_lib.my_types.all;";
    };
};
```

#### Parameters

| | |
|---|---|
| *list-of-strings* | Any list of strings that after concatenation creates any sequence of legal VHDL code. the ***e*** program does not check for VHDL syntax errors. VHDL syntax errors are identified only when you compile the file with the VHDL compiler.<br><br>The curly braces are required only if the list of strings contains more than one string. |

#### Description

Specifies a list of VHDL strings to be included in the stubs file ("*sim*.vhd"). The stubs file contains code that enables some simulation-related features. It is recommended to use **vhdl code** statements or unit members to modify this file rather than to modify it by hand.

When **vhdl code** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of VHDL code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

NOTE— Whenever you add or modify a **vhdl code** statement or unit member or add an instance of a unit containing a **vhdl code** unit member, you must create a new stubs file.

### Example 1

A common use for the **vhdl code** statement is to include packages. For example, parameter types may be declared in a different package from the one where the subprogram is declared. The example below adds the “common_types” package defined in the “common_lib” library to the ***_sim***.vhd file.

```
vhdl code {
    "library common_lib;";
    "use common_lib.common_types.all;";
};
```

The *sim*.vhd file generated with this statement looks like this:

```
library common_lib;
use common_lib.common_types.all;

entity ... is
end ...;
```

### Example 2

You can also use non-constants in **vhdl code** expressions. The following example configures the DUT clock differently for each channel instance.

```
<'
unit channel {
    chan_number: string;
    keep chan_number == hdl_path();

    s: string;
    keep s == append("library worklib; \n",
                     "configuration phased_clocks of TOP is \n",
                     "    use worklib.all; \n",
                     "    for behavioral \n",
                     "      for CLK: clock \n",
                     "        use entity clkgen(",chan_number,"_clock); \n",
                     "      end for; \n",
                     "    end for; \n",
                     "end phased_clocks;" );
    vhdl code {s};
};
'>
```

This **vhdl code** unit member inserts the following VHDL code into the ***sim***.vhd file:

```
library worklib;
configuration phased_clocks of TOP is
    use worklib.all;
    for behavioral
        for CLK: clock
            use entity clkgen(CHAN0_clock);
        end for;
    end for;
end phased_clocks;
library worklib;
configuration phased_clocks of TOP is
    use worklib.all;
    for behavioral
        for CLK: clock
            use entity clkgen(CHAN1_clock);
        end for;
    end for;
end phased_clocks;
```

### See Also

— “VHDL Statements and Unit Members” on page 812

### 25.2.2 vhdl driver

#### Purpose

Drive a VHDL signal continuously via the resolution function

#### Category

Unit member

#### Syntax

**vhdl driver '*HDL-pathname*' using *option*, ...**

Syntax examples:

```
unit top {
    vhdl driver '~/top/data' using initial_value=32'bz,
      disconnect_value=32'bz;
    vhdl driver '(addr_name)' using initial_value= 1'bz,
      delay= addr_delay;
};
```

#### Parameters

| | |
|---|---|
| *HDL-pathname* | A full or a relative VHDL path to the signal. If the signal has more than one driver, one driver in the DUT and one in the ***e*** program, for example, then the signal must be of a resolved type. If this name is not a constant, it is calculated after the final step of pre-run generation.<br><br>See “'HDL-pathname'” on page 838 for a complete description of HDL path syntax. |
| *option* | A list of one or more of the following options separated by commas. None of the options is required. |

| | |
|---|---|
| ~~**disconnect_value=** [***integer-expression*** \| ***verilog-literal***]~~ | ~~Any legal integer expression specifying the value to be used on the ***e*** program restore to disconnect the driver. The default is z. If this expression is not a constant, it is calculated after the final step of pre-run generation.~~<br><br>~~This value is used when you restore Specman Elite after issuing a **test** command but do not restart the simulator. This value should be set to a value that does not affect the overall value of the resolved signal. For std_logic signals, the value should be z.~~<br><br>~~NOTE— If the VHDL signal name is a computed name then it will be computed again at the before the restore. Thus, in order to correctly assign a disconnect_value, it is important to keep the expressions used in the computed name unchanged during the simulation session.~~<br><br>~~Use a Verilog literal to specify values containing x or z. A Verilog literal is a sized literal that can contain 0, 1, x, and z, for example 16'bx.~~ |
| **delay**= ***integer-expression*** | Any legal integer expression specifying a delay for all assignments. The delay time units are determined by the current time unit setting. See "vhdl time" on page 829 for information on how to set time units for ***e*** programs. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| **mode**=[**INERTIAL** \| **TRANSPORT**] | Used only when **delay** is also specified, this option specifies whether pulses whose period is shorter than the delay are propagated through the driver. INERTIAL specifies that such pulses are not propagated. TRANSPORT, the default, specifies that all pulses, regardless of length, are propagated.<br><br>The mode option also influences what happens if another driver (either VHDL or another unit) schedules a signal change and before that change occurs, this driver schedules a different change. With INERTIAL, the first change never occurs. |
| **initial_value**= [***integer-expression*** \| ***verilog-literal***] | Any legal integer expression specifying an initial value for the signal. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>When the ***e*** program is driving a resolved signal that is also driven from VHDL, unless an initial value is specified, the signal value is X until the first value is driven from he ***e*** program, even if a 0 or a 1 is driven from VHDL.<br><br>Use a Verilog literal to specify values containing x or z. A Verilog literal is a sized literal that can contain 0, 1, x, and z, for example 16'bx. |

## Description

The **vhdl driver** unit member identifies a VHDL signal that, when assigned to by an ***e*** method of that unit, is driven continuously using the resolution function. In contrast, signal assignments made without **vhdl driver** are over-ridden by any subsequent assignment from a VHDL process or from another ***e*** method, rather than the two driven values being resolved.

Any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file.

To require resolution between VHDL process assignments and an ***e*** method assignment or between ***e*** method assignments, you can use the **vhdl driver** unit member. The **vhdl driver** unit member creates a driver that influences any assignments to the specified VHDL signal made by a method in the enclosing unit or by a method in a struct enclosed by this unit.

You can create multiple drivers for the same signal by making multiple instances of a unit that contains a **vhdl driver** unit member. In this case, the driver is created for every unit instance. This may be useful when modeling multiple modules on a tri-state bus, for example. Multiple drivers can be created only for signals of a resolved type.

Note that there is a significant semantic difference between a **verilog variable using drive** statement/unit member and a **vhdl driver** unit member. **verilog variable using drive** creates a single Verilog always block for each actual Verilog signal. This means that there is one and only one driver for this signal.

## Notes

— **vhdl driver** influences only assignments made by methods of the unit in which it was declared or by methods of the structs nested in this unit. Therefore, you must use the **vhdl driver** unit member in each unit that drives a particular signal; adding a **vhdl driver** unit member in only one of the units that drives it is not sufficient.

— **vhdl driver** is a unit member, not a statement. If you need to declare a **vhdl driver** and your verification environment does not have any user-defined units, then you must explicitly extend **sys** (**sys** is a unit):

```
<'

extend sys {

    vhdl driver '/top/comp_io' using initial_value=1'bz;

};

'>
```

— The **vhdl driver** unit member is supported only with ModelSim VHDL. For multi-bit signal support, ModelSim version 5.4c is earliest version that can be used.

— The pathname for multi-bit signals must be specified without a bit range, for example, 'sig'. If you include the bit range, for example, 'sig[1:0]', the ***e*** program does not apply the driver.

— The **vhdl driver** unit member does not affect the VHDL stubs file, so it is not necessary to rewrite the stubs file if you add or modify a **vhdl driver** unit member in your ***e*** code.

## Example

### top.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

entity top is
end top;

architecture arch of top is

signal a : std_logic := '0';
signal b : std_logic := '0';
signal c : std_logic := '0';
signal s : std_logic := '1';
signal t : std_logic ;

signal clk : std_logic := '0';

   component comspec
   end component;
   for all: comspec use entity work.REFERENCE (arch);

begin

      clk_pr:process(clk) begin
         clk <= not clk after 50 ns;
      end process clk_pr;

      shifter:  process(clk) begin
         if (clk'event  AND clk = '1') THEN
            a <= not a ;
            b <= a xor b ;
            c <= (a and b) xor c ;

            if (b = '0' and c = '1') THEN
               t <= '1' ;
            elsif (b = '1' and c = '1') THEN
               t <= '0' ;
            else t <= 'Z' ;
            end if ;
         end if ;
      end process shifter ;

      I: comspec;
end arch ;
```

**driver.e**

```
<'
extend sys {
    event clk is fall('~/top/clk')@sim ;
    drive0 : drive is instance ;
    drive1 : drive is instance ;
    keep drive0.hdl_path() == "~/top";
    keep drive0.name == "t";
    keep drive1.hdl_path() == "~/top";
    keep drive1.name == "t";
};

unit drive {
    name: string;
    vhdl driver '(name)' using disconnect_value=1'bz,
        initial_value = 1'bz;
    driver() @sys.clk is {
        '(name)' = 1 ;
        wait cycle ;
        '(name)' = 1'bz ;
        wait [7] ;
        '(name)' = 0 ;
        wait cycle ;
        stop_run() ;
    };
```

```
        run() is also { start driver() } ;
    };
    '>
```

### Result (ModelSim waveform)

```
    run -all
    #                   0 : t = U
    #                  50 : t = Z
    #                 100 : t = 1
    #                 200 : t = Z
    #                 450 : t = 1
    #                 650 : t = 0
    #                 850 : t = Z
    #                 900 : t = 0
    # Simulation stop requested
    quit
```

### See Also

- “'HDL-pathname'” on page 838
- “VHDL Statements and Unit Members” on page 812

## 25.2.3 vhdl function

### Purpose

Declare a VHDL function defined in a VHDL package

### Category

Statement or unit member

### Syntax

**vhdl function '*identifier*' using *option*, ...**

Syntax examples:

```
    unit calc {
        vhdl function 'max' using
          interface="(op1:word64; op2:word64) return word64",
          library="work", package="arith_pkg";

        vhdl function '(vhdl_min_name)' using
          interface=intf,library=lib_name,package=package_name;
       };
```

### Parameters

| | |
|---|---|
| *identifier* | The operator symbol (for example '"and"') or the name of the VHDL function as specified in the package. If the identifier is not a constant, it is calculated after the final step of pre-run generation. |

| | |
|---|---|
| *option* | A comma-separated list of two or more of the following options. The **interface, library** and **package** options are required. |
| **interface**= **“(*string-exp*) return *return-subtype*”** | A legal string expression that specifies the interface list for the function. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>The legal syntax for this string is<br>[***parameter-class***] ***identifier*** **:*subtype***[**;**...]<br><br>The only output of the function is the returned value.<br><br>The optional parameter class is **constant** or **signal** — as specified in the VHDL declaration — and is transparent for the ***e*** program.<br><br>The identifier does not have to be the name of the parameter as specified in the package but the parameter subtype must match exactly the package specification.<br><br>The type of the returned value must match exactly the package specification. The type must also be one of the types supported by the ***e*** language. |
| library=*string-exp* | A legal string expression specifying the name of the VHDL library containing the package where the function is defined. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| package= *string-exp* | A legal string expression specifying the package name. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| alias=*string-exp* | A legal string expression that specifies the name with which the function will be called from ***e***. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>There are various reasons to use aliases. For example, an alias is required when there are overloaded functions with the same names but different interfaces. It must also be used for VHDL functions that overload operator symbols (for example, function “+”...) because the call from ***e*** cannot contain double quotes. |
| declarative_item=*string-exp* | A legal string expression that specifies one or more **use** clauses that are placed in the stubs file at the local scope of the function call rather than at the global scope of the REFERENCE entity. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>This option may prevent unwanted overwriting of declarations in the stubs file. See Example 1 on page 825 for an example of declarative_item used with **vhdl procedure**. |

### Description

Declares a VHDL function in *e* so that it can be called from a time-consuming method (TCM).

When there is a call to a procedure with a constant or signal parameter, the value of the constant or signal is passed.

If there are functions in a single package that have the same name but different numbers or types of parameters, make a separate declaration in *e* for each one you plan to call and specify a unique alias for each declaration.

When **vhdl function** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

### Notes

— The VHDL function specifiers “pure” and “impure” are not supported.
— If an alias is not used, then a call to a VHDL function requires the full VHDL name in the following format: ***library_name.package_name.function_name***.
— Calls to VHDL functions are value-returning expressions. They are time-consuming and can only be made from TCMs.
— Function calls must explicitly specify actual values—either scalars or lists of scalars—for all parameters. Default values for parameters are not supported.
— Whenever you add or modify a **vhdl function** statement in an *e* module, you must create a new stubs file.

### Example 1

VHDL allows overloading of subprogram names. It is possible, for example, to define two functions with the name “increment” where the number and type of arguments or the return type may differ:

```
function increment (a: integer) return integer is ...
function increment (a: integer; n:integer) return integer is ...
```

In cases like this, you must create an alias for each version that you intend to call.

```
vhdl function 'increment' using
    interface="(a: integer) return integer",
    library="work", package="pkg", alias="integer_inc_1";

vhdl function 'increment' using
    interface="(a: integer; n: integer) return integer",
    library="work", package="pkg",
    alias="integer_inc_n";

extend sys {
    event clk is rise ('top.clk') @sim;

    test(a:int)@clk is {
       check that 'integer_inc_1'(a) == 'integer_inc_n'(a,1);
    };
};
```

### Example 2

Another case where aliasing may be required is when a subprogram contains unconstrained array parameters or unconstrained return values. For example, “ToStdLogicVector” accepts an integer and converts it to a STD_LOGIC_VECTOR of the given length, probably truncating the value.

In order to support subprograms with unconstrained array parameters or return values, the ***e*** program must be notified about all (or maximal) potential lengths in the given test. This is required because the ***e*** program must provide a VHDL stub code that will specify the exact returned value type. Accordingly you may need to use multiple aliases for the same VHDL function.

```
library ieee;
use IEEE.std_logic_1164.all;

package pkg is
    function ToStdLogicVector(oper:INTEGER; length:NATURAL) \
        return STD_LOGIC_VECTOR;
end pkg;
```

You must create an alias for each of the possible lengths of the returned value.

```
vhdl function 'ToStdLogicVector' using
    interface= "(oper:INTEGER; length:NATURAL) \
      return STD_LOGIC_VECTOR(0 to 31)",
    library="lib", package="pkg",
    alias="ToStdLogicVector32";

vhdl function 'ToStdLogicVector' using
    interface="(oper:INTEGER; length:NATURAL) \
      return STD_LOGIC_VECTOR(15 downto 0)",
    library="lib", package="pkg",
    alias="ToStdLogicVector16";
```

### See Also

— “'HDL-pathname'” on page 838
— “VHDL Statements and Unit Members” on page 812

## 25.2.4 vhdl procedure

### Purpose

Declare a VHDL procedure defined in a VHDL package

### Category

Statement or unit member

### Syntax

**vhdl procedure '*identifier*' using *option*, ...**

Syntax example:

```
unit calc {
    vhdl procedure 'do_arith_op' using
      interface="(op1:integer; op2: integer; \
      op_code: func_code; result: out integer)",
      library="work", package="arith_pkg";

    vhdl procedure '(vhdl_min_name)' using
      interface=intf,library=lib_name,package=package_name;
};
```

## Parameters

| | |
|---|---|
| *identifier* | The name of the VHDL procedure as specified in the package. If this identifier is not a constant, it is calculated after the final step of pre-run generation. |
| *option* | A comma-separated list of two or more of the following options. The **library** and **package** options are required. |
| interface= "(*string-exp*)" | A legal string expression that specifies the interface list for the procedure. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>The legal syntax for this string is<br>[ ***parameter_class*** ] ***identifier* :** [***mode***] ***subtype***[**;**...]<br><br>The optional parameter class is **constant**, **variable**, or **signal** — as defined in the VHDL declaration — and is transparent for the ***e*** program.<br><br>Mode is **in**, **inout** or **out**. The default mode is **in.**<br><br>The identifier does not have to be the name of the parameter as specified in the package but the subtype must match exactly the specification in the package. The subtype must also be one of the types supported by ***e***. |
| library= *string-exp* | A legal string expression specifying the name of the library containing the package where the procedure is defined. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| package= *string-exp* | A legal string expression specifying the package name. If this expression is not a constant, it is calculated after the final step of pre-run generation. |
| alias= *string-exp* | A legal string expression that specifies the name with which the procedure will be called from ***e***. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>There are various reasons to use aliases. For example, an alias is required when there are overloaded procedures with the same names but different interfaces. It is also useful when two different procedures have the same name but belong to different packages. |

| | |
|---|---|
| declarative_item= *string-exp* | A legal string expression that specifies a **use** clause that is placed at the local scope of the procedure call rather than at the global scope of the REFERENCE entity in the stubs file. If this expression is not a constant, it is calculated after the final step of pre-run generation.<br><br>This option prevents unwanted overwriting of declarations in the stubs file. See Example 1 on page 825. |

### Description

Declares a VHDL procedure in ***e*** so that it can be called from a time-consuming method (TCM).

When **vhdl procedure** is used as a unit member, any non-constant expressions in the list of strings are calculated within the context of a unit instance. Each unit instance adds a separate fragment of Verilog code to the stubs file. When used as a statement, any non-constant expressions in the list of strings are calculated within the context of **sys**.

When there is a call to a procedure with a constant or variable parameter, the value of the constant or variable is passed. Signal parameters are handled differently.

When there is a call to a procedure with a signal parameter of mode **in**, instead of passing the value of the signal, it passes the signal object itself. Any reference to the formal parameter within the procedure is exactly like a reference to the actual signal itself. A consequence of this is that if the procedure executes a wait statement, the signal value may be different after the wait statement is completed and the procedure resumes.

When there is a call to a procedure with a signal parameter of mode **out**, the procedure receives a reference for the signal. When the procedure performs a signal assignment statement on the formal parameter, the value is propagated to the actual signal parameter. Note that output parameters are assigned in the default ***e*** way; the value is assigned immediately but is not forced and not assigned via a VHDL driver.

If there are procedures in a single package that have the same name but different numbers or types of parameters, you must make a separate declaration in ***e*** for each one you plan to call and specify a unique alias for each declaration.

In order to allow multiple ***e*** program threads to call the same time-consuming VHDL procedure simultaneously and in parallel, you must create multiple unit members - one per such a thread. This may be done, for example, by placing a **vhdl procedure** declaration in a unit, which logically maps the block of the DUT to a ***e*** program thread, and creating multiple instances of that unit. the ***e*** program creates a separate VHDL process in the stubs file for each unit member, and you can then invoke the procedure simultaneously from different unit instances.

There is a significant semantic difference between VHDL procedure/function and Verilog task/function unit members. Verilog tasks are always located within Verilog modules. Thus, there may be multiple instances of those tasks within a DUT. Accordingly, an HDL path of a corresponding unit in ***e*** maps between the unit and a Verilog task instances. On the VHDL side, ***e*** supports only subprograms that are declared in VHDL packages. Such subprograms have nothing to do with HDL paths. Accordingly, the correspondence between ***e*** unit instance and a VHDL procedure unit member is completely abstract—it just creates a separate process in the VHDL stub file. Such correspondence is natural, because in VHDL the time-consuming package subprograms are re-entrant and may be called simultaneously from parallel VHDL component instances.

The fact that VHDL subprogram unit members are not bound to HDL paths causes a difference in how these subprograms may be called from nested unit instances. The essence of this difference is that if some VHDL procedure/function is declared in a parent unit instance, then it may be called from other units, instantiated below, without a duplication of the subprogram declaration.

### Notes

- — Calls to VHDL procedures are time-consuming and can only be made from TCMs.
- — Passing a bit select of a signal parameter (for example, 'signal_name[3:7]') is not supported. (Passing a bit select of a constant or variable is supported.)
- — Passing a signal parameter with @x, @z, or @n (for example, 'signal_name@x') is not supported. (Passing a constant or variable with @x, @n, or @z is supported.)
- — If an alias is not used, then a call to VHDL procedure requires the full VHDL name in the following format: ***library_name.package_name.function_name***
- — You must explicitly pack all structs before passing them in as inputs to a procedure.
- — Procedure calls must explicitly specify actual values for all parameters. Default values for parameters are not supported.
- — Whenever you add or modify a **vhdl procedure** statement in an ***e*** module, you must create a new stubs file.

### Example 1

This example shows how to handle the case where two different types from two different packages have the same name ("state"). Using the **declarative_item** option to specify a **use** clause at the scope local to the procedure avoids name collision.

```
vhdl procedure 'check_state' using
    interface="(s:state)",
    library="lib1",
    package="procedures_pkg",
    declarative_item="use lib1.fsm_types.all;";

vhdl procedure 'show' using
    interface="(s:state)",
    library="lib2",
    package="display_pkg",
    declarative_item="use lib2.widget_types.all;";
```

### Result

The qvh.vhd file has the following structure. Note that the library declarations appear at the global scope. Any **use** clauses included with **vhdl code** declarations would also appear at this scope. In contrast, the **use** clauses declared with **declarative_item** appear at the scope of the appropriate function call.

```
entity ... is
end ...;
architecture fmi_stub of ... is
...
library IEEE;
use IEEE.std_logic_1164.all;

entity  wave is
```

This example shows how to allow parallel invocation of the same procedure. There are two instances of the unit "transactor", which contains a **vhdl procedure** unit member. The TCM "test()" of each unit instance is called during the run phase, simultaneously invoking two different VHDL processes. Notice that the parentheses are required when calling the procedure from ***e***, even though "send_packet()" has no interface.

```
unit transactor {
    vhdl procedure 'send_packet' using library="work",
        package="pkg";

    test() @sys.clk is {
        'work.pkg.send_packet'();
    };
};

extend sys {
    event clk is rise ('top.clk') @sim;

    transactor1: transactor is instance;
    transactor2: transactor is instance;

    run() is also {
        start transactor1.test();
        start transactor2.test();
    };
};
```

### Example 2

This example illustrates the transmitting and receiving of a one-byte packet. The unit “driver” uses the VHDL procedure “transmit_packet_data” to drive one bit of data per cycle. This procedure accepts two input parameters by value and two signal parameters: an input, “clock”, and an output, the data bit.

The unit “receiver” uses the VHDL procedure “receive_packet” for reading the data. This procedure accepts two input signal parameters: the clock and one bit of data (both are changing during the running of the procedure) and one output parameter: the resulting package. The procedure waits for the rise of the clock and then reads the new data.

### The VHDL package:

```
package transmit_receive_pkg is

    subtype data_range is integer range 1 to 8;
    type packet_array is array (data_range) of bit;

    procedure receive_packet (signal rx_data: in bit;
      signal rx_clock: in bit;
      data_buffer: out packet_array);

    procedure transmit_packet_data (packet_type: in bit;
      data_buffer: in bit_vector(6 downto 0);
      signal rx_clock: in bit;
      signal rx_data: out bit);

end transmit_receive_pkg;

package body transmit_receive_pkg is

    procedure receive_packet (signal rx_data: in bit;
      signal rx_clock: in bit;
      data_buffer: out packet_array) is
    begin
        for index in data_range loop
```

```
            wait until rx_clock = '1';
            data_buffer(index) := rx_data;
        end loop;
    end receive_packet;

    procedure transmit_packet_data ( packet_type: in bit;
      data_buffer: in bit_vector(6 downto 0);
      signal rx_clock: in bit;
      signal rx_data: out bit ) is
    begin
        for index in 0 to 6 loop
            wait until rx_clock = '0';
            rx_data <= data_buffer(index);
        end loop;

        wait until rx_clock = '0';
        rx_data <= packet_type;
    end transmit_packet_data;
end;
```

**The entity and architecture:**

```
use work.transmit_receive_pkg.all;
entity receiver is
end receiver;

architecture behavioral of receiver is

    component SN
    end  component;
    for all: SN use entity work.reference (arch);

    signal data : bit;
    signal clock: bit;

  begin

    clock_generator : process
    begin
      clock <= '0' after 2 ns, '1' after 10 ns;
      wait for 10 ns;
    end process clock_generator;

SN_INST: SN;

end behavioral;
```

**The receiver unit**

```
<'

unit receiver {

    vhdl procedure 'receive_packet' using interface= "( \
      signal rx_data: bit; signal rx_clock: bit; \
      data_buffer: out packet_array)", library= "work",
      package = "transmit_receive_pkg";
```

```
        !received_packet : byte;
        event clk is fall ('clock')@sim;

        receive_packet() @clk is {
            'work.transmit_receive_pkg.receive_packet'('data',
              'clock',received_packet);

            wait cycle;
            outf("The packet that was received : \
              %#x\n",received_packet);

            wait cycle;
        };
    };

    unit driver{

        vhdl procedure 'transmit_packet_data' using interface= "( \
          packet_type:in bit; \
          data_buffer : in bit_vector(6 downto 0); \
          signal rx_clock : in bit; signal rx_data : out bit )",
          library= "work", package = "transmit_receive_pkg";

        transmit_data : list of bit;
        keep transmit_data.size() == 7;

        packet_type          : bit;

        event clk is rise ('clock')@sim;

        send_packet()@clk is{
          'work.transmit_receive_pkg.transmit_packet_data'(
          packet_type,transmit_data,'clock','data');
        };
    };

    extend sys{

        driver   :driver   is instance;
        receiver :receiver is instance;

        keep driver.hdl_path() == "/";
        keep receiver.hdl_path() == "/";

        run() is also {
            start driver.send_packet();
            start receiver.receive_packet();
            start do_check();
        };

        do_check()@receiver.clk is {
            var transmit_packet: byte;
            unpack(NULL,driver.transmit_data.reverse(),
              transmit_packet[7:1]);
            transmit_packet[0:0] = driver.packet_type;
            outf("The packet that was transmitted : \
              %#x\n",transmit_packet);
            wait [10]*cycle;
            check that (transmit_packet == receiver.received_packet)
              else dut_error(" the data did not pass correctly");

            stop_run();
        };
    };
    '>
```

### See Also

- "'HDL-pathname'" on page 838
- "VHDL Statements and Unit Members" on page 812

## 25.2.5 vhdl time

### Purpose

Sets VHDL time resolution

### Category

Statement

### Syntax

vhdl time *integer-exp time-unit*

Syntax example:

```
vhdl time 100 ns;
```

### Parameters

| | |
|---|---|
| *integer-exp* | A legal integer expression. If the expression is not a constant, it is calculated after the final step of pre-run generation. |
| *time-unit* | Any valid VHDL time unit. |

### Description

Sets the time resolution to the specified time scale. This time scale is used to scale:

- Delays in ***e*** program temporal expressions
- Delays specified in **vhdl driver** unit members
- Simulation time as shown in **sys.time,** a 64-bit integer field that stores ***e*** program time.

If you use NC simulator and do not specify a time resolution, the default resolution for the ***e*** program is 1ns. (NC VHDL always uses a time scale of 1 fs.)

If you use ModelSim, the default resolution always matches the settings chosen on the simulator side in the initialization file or with the simulator invocation option.

#### Notes

- The **vhdl time** statement does not affect the stubs file, so it is not necessary to rewrite the stubs file if you change the time scale.
- **vhdl time** cannot be used as a unit member.
- If you use a non-constant expression in a **vhdl time** statement, this expression is computed in the context of **sys** and so must be visible in that context.

#### Example 1

The following statement sets the time resolution to 100ns.

```
vhdl time 100 ns;
```

#### Example 2

In this example, the time resolution is not calculated until after the final step of pre-run generation.

```
vhdl time num ns;

extend sys {

    num:int;
    keep num == 100;
};
```

#### See Also

- "VHDL Statements and Unit Members" on page 812
- "vhdl driver" on page 815

## 25.3 Simulation-Related Actions

There are two simulation-related actions in ***e***:

- "force" on page 830
- "release" on page 834

### 25.3.1 force

#### Purpose

Force a value on an HDL object

#### Category

Action

#### Syntax

**force '*HDL-pathname*'** = ***exp***

Syntax example:

```
force '~/top/sig' = 7;
```

## Parameters

| | |
|---|---|
| *HDL-pathname* | The full path name of an HDL object, optionally including expressions. See "'HDL-pathname'" on page 838 for more information. |
| *exp* | Any scalar expression or literal constant, as long as it is composed only of 1's and 0's. No x or z values are allowed. Thus "16'hf0f1" or (sys.my_val + 5) are legal. |

## Description

Forces an HDL object to a specified value, overriding the current value and preventing the DUT from driving any value. The HDL object remains at the specified value until a subsequent force action from e or until freed by a release action.

When an ***e*** program is linked with a Verilog simulator, you can apply a force action to any net or wire that has been declared **forcible** with the **verilog variable** statement.

When an ***e*** program is linked with a VHDL simulator, you can force signals of a scalar integer or enumerated type as well as binary array type, such as arrays of std_logic and bit vectors. No declaration is required for VHDL objects.

When an ***e*** program is linked with ModelSim VHDL simulator, you can force single elements of an array of a scalar integer or enumerated type using the predefined routine **simulator_command()**.

If you force a part of a vectored object, the force action is propagated to the rest of the object. If a **force** action is applied to a Verilog signal from an ***e*** programe, all preceding and subsequent non-forced assignments to the same object —including those within the same tick — are ignored until a subsequent release action from ***e***. The only exception is that if you force different parts of a vectored object in the same tick, the actions are accumulated and applied together at the end of the tick.

If there are multiple assignments to a VHDL object from ***e***, every new (not necessarily forced) assignment overrides the previous one, without requiring you to explicitly release the signal from ***e***. The release action is needed only when a signal must be driven from the ***e*** program and from the DUT by turns.

## Notes

— Forcing of Verilog registers is not supported.
— The interface with SpeedSim does not support **force** or **release**.
— Forcing signals is always costly to performance. In VCS, forcing signals is disabled by default. If it is not enabled and you try to force a signal, you will see an error such as the following:

```
*** Error: acc_set_value: Pli force not enabled.
    Please add capability frc to module your_module.
```

To enable forcing in VCS, you need to add the force option manually to the pli.tab by changing "acc=rw,cbk:*" to "acc=frc,cbk:*". To decrease the performance hit, you can change "*" to just the levels of DUT hierarchy that you want an ***e*** program to access, for example, "acc=frc,cbk:TOP.DMA.*;TOP.USB.*".

## Example 1

This example shows the effect of force and release actions in a Verilog environment.

**top.v**

```
module top();
    reg clk;
    wire [7:0] data;

    initial begin
        clk = 0;
        forever begin
            #50 clk = ~clk;
        end
    end

// monitors
   always @(data) $display ("%t : data = %b", $time, data);

endmodule
```

**test.e**

```
struct sim {
    event clk is rise ('~/top/clk')@sim;

    m() @clk is {
        '~/top/data' = 8'b10101010;
        force '~/top/data[3:0]' =0;
        '~/top/data[7:4]' = 4'b0000;

        wait cycle;
        '~/top/data' = 8'b10011001;

        wait cycle;
        release '~/top/data[7:0]';

        wait cycle;
        '~/top/data' = 8'b11111111;

        wait [2]* cycle;
        stop_run();
    };

};

extend sys {
    sim;

    setup() is also {
        set_config(print, radix, bin);
    };

    run() is also {
        start sys.sim.m();
    };
};
```

**verilog.e**

```
verilog variable '~/top/data[7:0]' using wire, forcible;
```

#### Result (ModelSim transcript)

```
run -all
#                    0 : data = zzzzzzzz
#                   50 : data = 10100000
#                  250 : data = 10011001
#                  250 : data = zzzzzzzz
#                  350 : data = 11111111
# Simulation stop requested
quit
```

### Example 2

This example shows the effect of force and release actions in a VHDL environment.

#### top.vhd

```
use std.textio.all;
library IEEE;
use IEEE.std_logic_1164.all;
entity top is
end top;

architecture arc of top is

signal clk : std_logic := '0';
signal data: std_logic_vector (7 downto 0);

component comspec
end component;

for all: comspec use entity work.REFERENCE (arch);
begin

I: comspec;

  clk <= not clk after 50 ns;

end arc;
```

#### test.e

```
struct sim {
    event clk is rise ('~/top/clk')@sim;

    m() @clk is {
        '~/top/data' = 8'b10101010;
        force '~/top/data[3:0]' =0;
        '~/top/data[7:4]' = 4'b0000;

        wait cycle;
        '~/top/data' = 8'b10011001;

        wait cycle;
        release '~/top/data[7:0]';

        wait cycle;
        '~/top/data' = 8'b11111111;
```

```
            wait [2]* cycle;
            stop_run();
        };

    };

    extend sys {
        sim;

        setup() is also {
            set_config(print, radix, bin);
        };

        run() is also {
            start sys.sim.m();
        };
    };
```

### Result (ModelSim waveform)

```
    run -all
    #                    0 : data = UUUUUUUU
    #                   50 : data = 10100000
    #                  150 : data = 10011001
    #                  350 : data = 11111111
    # Simulation stop requested
    quit
```

### See Also

— “'HDL-pathname'” on page 838
— “release” on page 834

## 25.3.2 release

### Purpose

Remove a force action from an HDL object

### Category

Action

### Syntax

release '*HDL-pathname*'

Syntax example:

```
    release 'TOP.sig';
```

### Parameters

*HDL-pathname* The full path name of an HDL object previously specified in a **force** action.

### Description

Releases the HDL object that you have previously forced.

In a VHDL environment, a **release** action is only required to allow the object to be driven by the DUT. Each new action from ***e*** overrides the previous one without an explicit **release** action.

If the object is a Verilog wire and it has no other driver from within the model, it floats to high-impedance (all z).

NOTE— The interface with SpeedSim does not support **force** or **release**.

### Example 1

This example shows the effect of force and release actions in a Verilog environment.

### top.v

```
module top();
    reg clk;
    wire [7:0] data;

    initial begin
        clk = 0;
        forever begin
            #50 clk = ~clk;
        end
    end

// monitors
   always @(data) $display ("%t : data = %b", $time, data);

endmodule
```

### test.e

```
struct sim {
    event clk is rise ('~/top/clk')@sim;

    m() @clk is {
        '~/top/data' = 8'b10101010;
        force '~/top/data[3:0]' =0;
        '~/top/data[7:4]' = 4'b0000;

        wait cycle;
        '~/top/data' = 8'b10011001;

        wait cycle;
        release '~/top/data[7:0]';

        wait cycle;
        '~/top/data' = 8'b11111111;
```

```
            wait [2]* cycle;
            stop_run();
        };

    };

    extend sys {
        sim;

        setup() is also {
            set_config(print, radix, bin);
        };

        run() is also {
            start sys.sim.m();
        };
    };
```

### verilog.e

```
    verilog variable '~/top/data[7:0]' using wire, forcible;
```

### Result (ModelSim transcript)

```
    run -all
    #                    0 : data = zzzzzzzz
    #                   50 : data = 10100000
    #                  250 : data = 10011001
    #                  250 : data = zzzzzzzz
    #                  350 : data = 11111111
    # Simulation stop requested
    quit
```

### Example 2

This example shows the effect of force and release actions in a VHDL environment.

### top.vhd

```
    use std.textio.all;
    library IEEE;
    use IEEE.std_logic_1164.all;
    entity top is
    end top;

    architecture arc of top is

    signal clk : std_logic := '0';
    signal data: std_logic_vector (7 downto 0);

    component comspec
    end component;

    for all: comspec use entity work.... (arch);
    begin

    I: comspec;
```

```
  clk <= not clk after 50 ns;

end arc;
```

**test.e**

```
struct sim {
    event clk is rise ('~/top/clk')@sim;

    m() @clk is {
        '~/top/data' = 8'b10101010;
        force '~/top/data[3:0]' =0;
        '~/top/data[7:4]' = 4'b0000;

        wait cycle;
        '~/top/data' = 8'b10011001;

        wait cycle;
        release '~/top/data[7:0]';

        wait cycle;
        '~/top/data' = 8'b11111111;

        wait [2]* cycle;
        stop_run();
    };

};

extend sys {
    sim;

    setup() is also {
        set_config(print, radix, bin);
    };

    run() is also {
        start sys.sim.m();
    };
};
```

**Result (ModelSim waveform)**

```
run -all
#                    0 : data = UUUUUUUU
#                   50 : data = 10100000
#                  150 : data = 10011001
#                  350 : data = 11111111
# Simulation stop requested
quit
```

**See Also**

— “'HDL-pathname'” on page 838
— “force” on page 830

## 25.4 Simulation-Related Expressions

This section contains:

- — "'HDL-pathname'" on page 838
- — "specman deferred" on page 839

### 25.4.1 'HDL-pathname'

#### Purpose

Accessing HDL objects, using ***full-path-names***

#### Category

Expression

#### Syntax

**'*HDL-pathname*[*index-exp* | *bit-range*] [@(x | z | n)]'**

Syntax example:

```
'~/top/sig' = 7;
print '~/top/sig';
```

#### Parameters

| | |
|---|---|
| *HDL-pathname* | The full path name of an HDL object, optionally including expressions and composite data. |
| *bit-range* | A bit range has the format [***high-bit-num***:***low-bit-num***] and is extracted from the object from high bit to low bit. Slices of buses are treated exactly as they are in HDL languages. They must be specified in the same direction as in the HDL code and reference the same bit numbers. |
| *index-exp* | Accesses a single bit of a Verilog vector, a single element of a Verilog memory, or a single vector of a VHDL array of vectors. |
| **@x** \| **z** | Sets or gets the x or z component of the value. When this notation is not used in accessing an HDL object, the ***e*** program translates the values of x to zero and z to one.<br><br>When reading HDL objects using **@x** (or **@z**), the ***e*** program translates the specified value (x or z) to one, and all other values to zero. When writing HDL objects, if **@x** (or **@z**) is specified, the ***e*** program sets every bit that has a value of one to x (or z). In this way, **@x** or **@z** acts much like a data mask, manipulating only those bits that match the value of x or z. |
| @n | When this specifier is used for driving HDL objects, the new value is visible immediately (now). The default mode is to buffer projected values and update only at the end of the  tick. If reading a value using **@n** then the projected simulator value can be seen. |

### Description

Accesses Verilog and VHDL objects from ***e***.

NOTE— In general, you can access HDL objects using the '***HDL-pathname***' expression. In order to enable some non-trivial capabilities, however, you must also use **verilog** or **vhdl** statements.

—

## 25.4.2 specman deferred

### Purpose

Identify a deferred Verilog `define

### Category

Verilog expression

### Verilog Syntax

**`define *macro-name default-value* // specman deferred**

Syntax example:

```
`define bus_width 64 // specman deferred
```

### Parameters

| | |
|---|---|
| *macro-name* | Any legal Verilog identifier. |
| *default-value* | A constant expression. This value is used as the definition of the `define unless you over-write it with another value. The size and type of this value is used to determine the expected type and size of the runtime expressions. Thus, if this value is longer than 32 bits, its size must be explicitly specified. |
| // specman deferred | The "// specman deferred" comment must appear on the same line of the file that specifies the `define. This comment can contain any number of spaces or tabs. |

### Description

Deferred Verilog `defines let you use Verilog definitions without specifying their final values at compile time. Instead, you can redefine their values just prior to use.

This feature lets you compile and link a single executable for all tests and then load in different Verilog `defines definitions for different tests.

All non-deferred `defines are substituted in place during parsing. References to deferred `defines are resolved at run time.

### Notes

— Deferred `defines cannot be used in other Verilog `defines.
— You can only redefine the value of a `define; you cannot redefine its type or width. For example:

The following lines define a `define with the default width of 32 bits:

```
`define MY_MASK 0     // specman deferred
```

If later on during the run MY_MASK is defined with a different type or length, for example:

```
`define MY_WRONG_MASK 64'bz
```

an error occurs.

— Whether a `define is deferred or not is established at its very first occurrence and cannot be changed by any subsequent occurrence of this macro.
— The use of deferred macros causes some performance overhead, which is equivalent to accessing an entry in a table when using a function call for this purpose.

### See Also

— “verilog import” on page 799

## 25.5 Simulation-Related Routines

The following routines perform functions related to simulation:

— “simulator_command()” on page 840
— “stop_run()” on page 841

### 25.5.1 simulator_command()

#### Purpose

Issue a simulator command

#### Category

Predefined routine

#### Syntax

**simulator_command(***command*: string**)**

Syntax example:

```
simulator_command("force -deposit memA(31:0)");
```

#### Parameters

| | |
|---|---|
| *command* | A valid simulator command, enclosed in double quotes. Commands that change the state of simulation, such as run, restart, restore, or exit, cannot be passed to the simulator with **simulator_command()**. |

### Description

Passes a command to the HDL simulator from ***e***. The command returns no value. The output of the command is sent to standard output and to the log file.

NOTE— This routine can be used only with the ModelSim, SpeedSim, and NC (VHDL) simulators.

### See Also

— "force" on page 830

## 25.5.2 stop_run()

### Purpose

Stop a simulation run cleanly

### Category

Predefined routine

### Syntax

stop_run()

Syntax example:

```
stop_run();
```

### Description

Stops the simulator and initiates post-simulation phases. The following things occur when **stop_run()** is invoked:

1) The **quit()** method of each struct under **sys** is called. Each **quit()** method emits a "quit" event for that struct instance at the end of the current tick.
2) The scheduler continues running all threads until the end of the current tick.
3) At the end of the current tick, the extract, check, and finalize test phases are performed.
4) If a simulator is linked in to the ***e*** programe, the ***e*** program terminates the simulation cleanly after the test is finalized.

### Notes

— This method must be called by a user-defined method or TCM to stop the simulation run cleanly.
— You should not extend or modify the **stop_run()** method. If you want something to happen right after you stop the run, you can extend **sys.extract()**.
— Executing a tick after calling **stop_run()** is considered an error. This includes executing a tick to call a Verilog function or task.
— The actual threads are not removed until the end of the tick.
— If the simulator exit command is called before **stop_run()**, the global methods for extracting, checking and finalizing the test are called.

## Example

```
verilog import macros.v;

extend sys {
    event pclk is rise ('`TOP.pclk');
    driver: dut_driver;
};

struct dut_driver {
    ld: list of int(bits: `WORD_WIDTH);
    keep ld.size() in [1..30];

    stimuli() @sys.pclk is {
        '`READY' = 1;
        for each in ld {
            wait until true('`READY' == 1);
            '`TOP.data_in' = it;
            wait [`TRANS_DELAY];
        };
        stop_run();
    };
    run() is also {
        start stimuli();
    };
};
```

## See Also

— "The quit() Method of any_struct" on page 659
— The **run** option of "set_config()" on page 766

# 26 Predefined File Routines Library

## 26.1 Overview

The global struct named **files** contains predefined routines for working with files. This chapter contains information about using files and the predefined file routines. Like most global objects, the predefined routines in the **files** struct are not meant to be extended with **is also** or **is only**.

General information about working with files is provided in the following sections.

— "File Names and Search Paths" on page 843
— "File Descriptors" on page 843

Syntax for the predefined file routines is described in the following sections.

— "Low-Level File Routines" on page 843
— "General File Routines" on page 856
— "Reading and Writing Structs" on page 874

### See Also

— "File Iteration Actions" on page 545
— "String Routines" on page 736

## 26.2 File Names and Search Paths

Many of the file routines require a file-name parameter. The following are restrictions on file-name parameters for most routines.

— The file-name must be the exact path to the file.
— You cannot use ~, or wild card patterns, or any environment variable, including $PATH, in the file-name.
— For files that have default extensions, such as .e or .ecom, leave the extension off the file-name.

The exception to the above restrictions is the **files.add_file_type()** routine. This routine accepts ~, wild cards (*), or $PATH as a ***file-name*** parameter (see "add_file_type()" on page 844). Before you use any of the file routines, it is recommended that you use **files.add_file_type()** to make sure you have a valid path to a file.

## 26.3 File Descriptors

For every open file, a file descriptor struct exists which contains information about the file. The routine "open()" on page 848 returns the file descriptor as a variable of type **file**. The name of the file variable is used in low-level file operations such as the **files.read()**, **files.write()** and **files.flush()** routines. These routines are described in "Low-Level File Routines" on page 843.

## 26.4 Low-Level File Routines

This section contains descriptions of the file routines that use file descriptor structs.

To write strings to a file, the simplest routine is "write_string_list()" on page 873.

The following file routines are described in this section.

— "add_file_type()" on page 844
— "close()" on page 846
— "flush()" on page 847
— "open()" on page 848
— "read()" on page 850
— "read_lob()" on page 851
— "write()" on page 852
— "write_lob()" on page 853
— "writef()" on page 855

#### See Also

— "General File Routines" on page 856
— "Reading and Writing Structs" on page 874

### 26.4.1 add_file_type()

#### Purpose

Get a file name

#### Category

Method

#### Syntax

**files.add_file_type(*file-name*:** string**, *file-ext*:** string**, *exists*:** bool): string

Syntax example:

```
var fv: string;
fv = files.add_file_type("fname", ".e", FALSE);
```

#### Parameters

| | |
|---|---|
| *file-name* | The name of the file to access. A wild card pattern can be used. |
| *file-ext* | The file extension, including the dot (.) may be empty. |
| ***exists*** | Sets checking for existence of the file. |

#### Description

Returns a string holding the file name.

This routine assigns a string consisting of ***file-name*** and ***file-ext*** to a **string** variable. If ***file-name*** already contains an extension, then ***file-ext*** is ignored. If ***file-ext*** is empty, the ***file-name*** is used with no extension.

If ***exists*** is FALSE, the routine returns the file-name string without checking for the existence of the file. Wild cards, ~, and $PATH are not evaluated.

If ***exists*** is TRUE, the ***e*** program checks to see if there is a file that matches the ***file-name*** in the current directory. The ***file-name*** can contain ~, $PATH, and * wild cards. The * wild card represents any combination of ASCII characters. If there is one and only one file that matches the ***file-name*** pattern, the file's name is returned. If there is no match in the current directory, then the $PATH directories are searched for the file. If no matching file can be found, or if more than one file is found in a directory that matches a wild card, an error is issued. If there are multiple matching files in different directories in the PATH, the first one found is returned.

## Examples

For the following examples, assume files named “ex_file” and “ex_file.tmp” exist in the current directory, and a file named “ex_file.e” exists under /prog/docs (which is included in the $PATH definition).

The following assigns ex_file.e to the f1 variable, without checking to see if the ex_file.e file exists.

```
struct f_str {
    !file_list: list of string;
    AppendFileToList(ex_file: string) is {
        var f1: string;
        f1 = files.add_file_type(ex_file, ".e", FALSE);
        file_list.add(f1);
    };
};
extend sys {
    fi: f_str;
    run() is also {
        fi.AppendFileToList("ex_file");
    };
};
```

The following statement tries to assign ex_file.e to the f2 variable, but issues an error when it checks for the existence of ex_file.e.

```
AppendFileToList(ex_file: string) is {
    var f2: string;
    f2 = files.add_file_type(ex_file, ".e", TRUE);
    file_list.add(f2);
};
```

The error is shown below.

```
*** Error: No match for file ‘ex_file.e’
```

The following action assigns ex_file to the f3 variable.

```
var f3: string = files.add_file_type("ex_file", "", TRUE);
```

Although ex_file.e does not exist in the current directory, it does exist in the /prog/docs directory, which is in the $PATH. Therefore, the following action assigns /prog/docs/ex_file.e to the f4 variable.

```
var f4: string = files.add_file_type("ex_file", ".e", TRUE);
```

The following action assigns ex* to the f5 variable.

```
var f5: string = files.add_file_type("ex*", "", FALSE);
```

The following action checks for files that match the ex* pattern.

```
var f6: string = files.add_file_type("ex*", "", TRUE);
```

Since more than one file in the current directory matches the pattern, the names of the matching files are printed and an error is issued:

```
ex_file
ex_file.tmp
There is more than one file matching ex*
```

### See Also

— "file_exists()" on page 861

## 26.4.2 close()

### Purpose

Close a file

### Category

Method

### Syntax

**files.close(*file*:** file-descriptor**)**

Syntax example:

```
files.close(f_desc);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to be closed. |

### Description

Flushes the file buffer and closes the file specified by ***file-descriptor.*** The file must previously have been opened using "open()" on page 848. When no further activity is planned for a file, it should be closed to prevent unintentional operations on its contents.

### Example

The WrAndFlush() user-defined method in the following example opens a file named "ex_file.txt" as the m_file variable, writes a line to the file, and then closes the file. The RFile() user-defined method then opens the same file for reading and reads its contents into the m_string variable.

The **files.flush()** routine writes the "AaBaCa 0123" string to the disk immediately, so that the read routine can read it. If there were no **files.close()** routine (or **files.flush()** routine) following the write, the data would not be in disk file when the read was done.

```
struct f_s_1 {
        WrAndFlush(ex_file: string) is {
            var m_file: file;
            m_file = files.open(ex_file, "w", "Text file");
            files.write(m_file, "AaBaCa 0123");
            files.close(m_file);
        };
        RFile(ex_file: string) is {
            var m_file: file;
            m_file = files.open(ex_file, "r", "Text file");
            var r_chk: bool;
            var m_string: string;
            r_chk = files.read(m_file, m_string);
            if r_chk then {print m_string}
                else {out("file not read")};
        };
    };
    extend sys {
        f_si_1: f_s_1;
        run() is also {
            f_si_1.WrAndFlush("ex_file.txt");
            f_si_1.RFile("ex_file.txt");
        };
    };
```

### See Also

— "open()" on page 848
— "flush()" on page 847

## 26.4.3 flush()

### Purpose

Flush file buffers

### Category

Method

### Syntax

**files.flush(*file*:** file-descriptor**)**

Syntax example:

```
    files.flush(a_file);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to flush. |

### Description

Flushes all the operating system buffers associated with ***file*** to the disk.

File data is buffered in memory and only written to disk at certain times, such as when the file is closed. This routine causes data to be written to the disk immediately, instead of later when the file is closed.

This can be useful if two processes are using the same disk file, for example, to make sure that the current data from one process is written to the file before the other process reads from the file.

### Example

The WrAndFlush() user-defined method in the following example opens a file named “ex_file.txt” as the m_file variable, writes a line to the file, and then flushes the file’s buffer to disk. The RFile() user-defined method then opens the same file for reading and reads its contents into the m_string variable.

The **files.flush()** routine writes the “AaBaCa 0123” string to the disk immediately, so that the read routine can read it. If there were no **files.flush()** routine (or **file.close()** routine) following the write, the data would not be in disk file when the read was done.

```
struct f_s_2 {
    WrAndFlush(ex_file: string) is {
        var m_file: file;
        m_file = files.open(ex_file, "w", "Text file");
        files.write(m_file, "AaBaCa 0123");
        files.flush(m_file);
    };
    RFile(ex_file: string) is {
        var m_file: file;
        m_file = files.open(ex_file, "r", "Text file");
        var r_chk: bool;
        var m_string: string;
        r_chk = files.read(m_file, m_string);
        if r_chk then {print m_string}
            else {out("file not read")};
    };
};
extend sys {
    f_si_2: f_s_2;
    run() is also {
        f_si_2.WrAndFlush("ex_file.txt");
        f_si_2.RFile("ex_file.txt");
    };
};
```

### See Also

— “open()” on page 848
— “flush()” on page 847

## 26.4.4 open()

### Purpose

Open a file for reading or writing or both

### Category

Method

## Syntax

**files.open(*file-name***: string, ***mode***: string, ***file-role***: string**)**: file

Syntax example:

```
var m_file: file;
m_file = files.open("a_file.txt", "r", "Text File");
```

## Parameters

| | |
|---|---|
| *file-name* | The name of the file to open. Wild cards, ~, and $PATH are not allowed in the file name. To use them to select files, see "add_file_type()" on page 844. |
| *mode* | The read/write mode for the file. The mode may be one of the following.<br><br>r - open the file for reading.<br><br>w - open the file for writing (overwrite the existing contents)<br><br>rw - open the file for reading and writing (add to the end of the existing contents)<br><br>a - open the file for appending (add to the end of the existing contents) |
| *file-role* | A text description used in error messages about the file. |

## Description

Opens the file for reading, writing, both reading and writing, or append, according to mode (r, w, rw, a) and returns the file descriptor of the file. The ***file-role*** is a description of the file, for example, "source file".

If the file cannot be opened, an error like the following is issued.

```
*** Error: Cannot open file-role 'file-name' for mode
```

## Example 1

The following example opens a file named "/users/a_file.txt" in write mode as file variable m_file, writes a line to the file, and then closes the file.

```
struct f_3_str {
    RdWrFile(ex_file: string) is {
        var m_file: file;
        m_file = files.open(ex_file, "w", "Text file");
        files.write(m_file, "HEADER");
        files.close(m_file);
    };
};
extend sys {
    fi_3: f_3_str;
    run() is also {
        fi_3.RdWrFile("/users/a_file.txt");
    };
};
```

### Example 2

The following actions perform the same operations as Example 1, above.

```
var m_file: file;
m_file = files.open("/users/a_file.txt", "w", "Text file");
files.write(m_file, "HEADER");
files.close(m_file);
```

### See Also

— “add_file_type()” on page 844
— “close()” on page 846
— “write()” on page 852

## 26.4.5 read()

### Purpose

Read an ASCII line from a file

### Category

Method

### Syntax

**files.read(*file*:** file-descriptor, ***string-var***: *string**)**: bool

Syntax example:

```
r_b = files.read(f_desc, m_string);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file that contains the text to read. |
| *string-var* | A variable into which the ASCII text will be read. |

### Description

Reads a line of text from a file into a string variable. The file must have been opened with “open()” on page 848. The line from the file is read into the variable without the final \n newline character.

The routine returns TRUE on success. If the method cannot read a line (for example, if the end of the file is reached), it returns FALSE.

The **files.read()** routine is a low level routine. For performance considerations, it is generally recommended to use the **for each line in *file*** action, rather than this routine.

### Example

The following example opens a file named “a_file.txt” as variable m_f, reads lines one by one from the file into a variable named “m_string”, and displays each string as it reads it.

```
struct f_s_3 {
    RFile(ex_file: string) is {
        var m_file: file;
        m_file = files.open(ex_file, "r", "Text file");
        var r_chk: bool;
        var m_string: string;
        r_chk = files.read(m_file, m_string);
        out("The first line is: ", m_string);
        while files.read(m_file, m_string) {
            out("The next line is: ", m_string);
        };
        files.close(m_file);
    };
};
extend sys {
    f_si_3: f_s_3;
    run() is also {
        f_si_3.RFile("ex_file.txt");
    };
};
```

### See Also

— "for each line in file" on page 545
— "open()" on page 848
— "read_lob()" on page 851
— "read_ascii_struct()" on page 874
— "read_binary_struct()" on page 875
— Table 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105, for information about type conversion between scalar types

## 26.4.6 read_lob()

### Purpose

Read from a binary file into a list of bits

### Category

Method

### Syntax

**files.read_lob(*file***: file-descriptor, ***size-in-bits***: int**)**: list of bit

Syntax example:

```
var m_file: file = files.open("a_file.dat", "r", "Data");
var b_l: list of bit;
b_l = files.read_lob(m_file, 32);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to read from. |
| *size-in-bits* | The number of bits to read. Should be a multiple of 8 |

### Description

Reads data from a binary file into a list of bits and returns the list of bits. The file must already have been opened with "open()" on page 848. To read an entire file, use UNDEF as the ***size-in-bits***.

### Example 1

The following example opens a file named "a_file.dat" with the file descriptor m_f, and reads the first 16 bits from the file into a list of bits named "b_list".

```
struct f_4 {
    b_list: list of bit;
    RdLOB(ex_file: string) is {
        var m_f: file = files.open(ex_file, "r", "Data");
        b_list = files.read_lob(m_f, 16);
        files.close(m_f);
    };
};
extend sys {
    fi_4: f_4;
    run() is also {
        fi_4.RdLOB("a_file.dat");
    };
};
```

### Example 2

The following actions perform the same operations as Example 1, above.

```
var m_f: file = files.open("a_file.dat", "r", "data file");
var b_list: list of bit = files.read_lob(m_f, 16);
files.close(m_f);
```

### See Also

— "close()" on page 846
— "open()" on page 848
— "write_lob()" on page 853
— Table 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105, for information about type conversion between scalar types

## 26.4.7 write()

### Purpose

Write a string to file

### Category

Method

### Syntax

**files.write(*file*:** file-descriptor**, *text*:** string**)**

Syntax example:

```
files.write(m_file, "Test Procedure");
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to write into. |
| *text* | The text to write to the file. |

### Description

Adds a string to the end of an existing, open file. A new-line \n is added automatically at the end of the string. The file must already have been opened using "open()" on page 848. If the file is not open, an error message is issued.

If the file is opened in write mode (w), this routine overwrites the existing contents. To avoid overwriting the existing file, open it in append mode (a).

NOTE— The >> concatenation operator can be used to append information to the end of a file.

### Example

The following example opens a file named "/users/a_file.txt" in write mode as file variable m_file, writes two lines to the file, and then closes the file.

```
struct f_s_5 {
    WrFile(ex_file: string) is {
        var m_file: file;
        m_file = files.open(ex_file, "w", "Text file");
        files.write(m_file, "FILE 1");
        files.write(m_file, "Test 1");
        files.close(m_file);
    };
};
extend sys {
    f_si_5: f_s_5;
    run() is also {
        f_si_5.WrFile("/users/a_file.txt");
    };
};
```

### See Also

— "close()" on page 846
— "open()" on page 848

## 26.4.8 write_lob()

### Purpose

Write a list of bits to a binary file

### Category

Method

### Syntax

**files.write_lob(*file***: file-descriptor**, *bit-list***: list of bit**)**

Syntax example:

```
var m_file: file = files.open("a_f.dat", "w", "My data");
var b_l: list of bit;
files.write_lob(m_file, b_l);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to write into. |
| *bit-list* | A list of bits to write to the file. The size of the list must be a multiple of 8 bits. |

### Description

Writes all the bits in the bit list (whose size should be a multiple of 8) to the end of the file specified by ***file***. The file must already have been opened with "open()" on page 848.

Lists of bits are always written in binary format.

### Example

The following example opens a file named "a_file.dat" as file descriptor m_f_1 in write mode (w). The **files.write_lob**() routine writes the contents of a list of bits named "b_list" into the file.

The **files.read_lob**() routine reads the contents of the file into a variable named "b_2" as a list of bits, which is then printed.

```
struct f_5_str {
    RdWrLOB(ex_file: string) is {
        var b_list: list of bit = {1; 0; 1; 1; 0; 1 ;1; 1};
        var m_f_1: file = files.open(ex_file, "w", "Data");
        files.write_lob(m_f_1, b_list);
        files.close(m_f_1);

        var b_2: list of bit;
        var m_f_2: file = files.open(ex_file, "r", "Data");
        b_2 = files.read_lob(m_f_2, 8);
        print b_2 using radix=bin, list_starts_on_right=FALSE;
    };
};
extend sys {
    fi_5: f_5_str;
    run() is also {
        fi_5.RdWrLOB("a_file.dat");
    };
};
```

The **print** action in the example above displays the following.

```
    b_2 =  (8 items, bin):
         0.  1 0 1 1  0 1 1 1
```

### See Also

— "close()" on page 846
— "open()" on page 848
— "read_lob()" on page 851

## 26.4.9 writef()

### Purpose

Write to a file in a specified format

### Category

Pseudo-method

### Syntax

**files.writef(*file***: file-descriptor**,** ***format***: string**,** ***item***: exp, ...**)**

Syntax example:

```
    var m_file: file = files.open("a_f.txt", "w", "My data");
    var m_i := new m_struct_s;
    writef(m_file, "Type: %s\tData: %s\n", m_i.s_type, m_i.data);
```

### Parameters

| | |
|---|---|
| *file* | The file descriptor of the file to write into. |
| *format* | A string containing a standard C formatting mask for each ***item***. See "Format String" on page 765 for information about formatting masks. |
| *item* | An ***e*** expression to write to the file. |

### Description

Adds a formatted string to the end of the specified file. No newline is automatically added. (Use "\n" in the formatting mask to add a newline).

The file must already have been opened with "open()" on page 848, otherwise an error is issued.

If the number of items in the formatting mask is different than the number of item expressions, an error is issued.

How the data is written to the file is affected by the **open()** mode "w" or "a" option and by whether or not the file already exists, as follows:

— If the file did not previously exist and the "w" (write) option is used with **open()**, then **writef()** writes the data into a new file.
— If the file did not previously exist and the "a" (append) option is used with **open()**, then no data is written.

— If the file did previously exist and the “w” (write) option is used with **open()**, then **writef()** overwrites the contents of the file.
— If the file did previously exist and the “a” (append) option is used with **open()**, then **writef()** appends the data to the existing contents of the file.

### Example

In the following example, a file named “pkts_file.txt” is opened in write (“w”) mode, with file descriptor mypkts. The **writef**() pseudo-method writes the contents of a list of pkt_s structs to the pkts_file, using a new line (\n) for each struct instance, with tabs (\t) after the struct instance name and the ptype field. The struct instance and the ptype are written as strings (%s) and pdata is written as a number (%d).

```
<'
type pkt_t: [PKT1, PKT2, PKT3];
struct pkt_s {
    ptype: pkt_t;
    pdata: byte;
};
extend sys {
    pkt_l[5]: list of pkt_s;
    run() is also {
        var mypkts: file = files.open("pkts_file.txt", "w", "");
        for each (pkt) in pkt_l {
            writef(mypkts, "Struct: %s\tType: %s\tData: %d\n",
                pkt, pkt.ptype, pkt.pdata);
        };
        files.close(mypkts);
    };
};
'>
```

### Result

This is what the contents of the pkts_file.txt file look like:

```
Struct: pkt_s-@0        Type: PKT2      Data: 148
Struct: pkt_s-@1        Type: PKT3      Data: 198
Struct: pkt_s-@2        Type: PKT3      Data: 61
Struct: pkt_s-@3        Type: PKT2      Data: 18
Struct: pkt_s-@4        Type: PKT1      Data: 82
```

### See Also

— “close()” on page 846
— “open()” on page 848
— “write()” on page 852
— “write_lob()” on page 853
— “Format String” on page 765

## 26.5 General File Routines

This section contains descriptions of the following routines.

— “file_age()” on page 857
— “file_append()” on page 858

- “file_copy()” on page 859
- “file_delete()” on page 860
- “file_exists()” on page 861
- “file_extension()” on page 862
- “file_is_dir()” on page 863
- “file_is_link()” on page 864
- “file_is_readable()” on page 865
- “file_is_regular()” on page 866
- “file_is_temp()” on page 868
- “file_is_text()” on page 869
- “file_rename()” on page 870
- “file_size()” on page 871
- “new_temp_file()” on page 872
- “write_string_list()” on page 873

### See Also

- “Low-Level File Routines” on page 843
- “Reading and Writing Structs” on page 874

## 26.5.1 file_age()

### Purpose

Get a file’s modification date

### Category

Method

### Syntax

**files.file_age(*file-name*: string)**: int

Syntax example:

```
var f_data: int;
f_data = files.file_age("f.txt");
```

### Parameters

| | |
|---|---|
| *file-name* | The file whose age is to be found. |

### Description

Returns the modification date of the file as an integer. This routine can be used to compare the modification dates of files. The integer returned by the routine is not recognizable as a date, but is a unique number derived from the file’s modification date. If the modification date includes the time of day, the time is factored into the number the routine returns. Newer files produce larger numbers than older files.

If the file does not exist, an error like the following is issued.

```
*** Error: Internal error in file_age: 'file-name' does not exist
```

### Example

In the following example, the **files.file_age()** routine derives a number from the modification date of a file whose variable is my_f. The routine is called twice in the **run()** method in the **sys** extension, once for each of two files. The age numbers are printed and compared to find the largest.

```
struct f_6_str {
    FAge(ex_file: string): int is {
        var my_f: string;
        var my_age: int;
        my_f = files.add_file_type(ex_file, "", TRUE);
        my_age = files.file_age(my_f);
        outf("file name: %s,  age: %d\n", ex_file, my_age);
        return my_age;
    }
};
extend sys {
    fi_6: f_6_str;
    run() is also {
        var my_age_1: int = fi_6.FAge("f_1.e");
        var my_age_2: int = fi_6.FAge("f_2.e");
        var oldest: int = max(my_age_1, my_age_2);
        print oldest;
    };
};
```

The example above prints the following.

```
file name: f_1.e,  age: 927860670
file name: f_2.e,  age: 927860675
  oldest = 927860675
```

### See Also

— “add_file_type()” on page 844

## 26.5.2 file_append()

### Purpose

Append files

### Category

Method

### Syntax

**files.file_append(*from-file-name*:** string**, *to-file-name*:** string**)**

Syntax example:

```
files.file_append(f_1, f_2);
```

### Parameters

| | |
|---|---|
| *from-file-name* | The name of the file that will be appended to the to-file. |
| *to-file-name* | The name of the file to which the from-file will be appended. |

### Description

Adds the contents of the file named ***from-file-name*** to the end of the file named ***to-file-name***. If either of the files does not exist, an error is issued.

NOTE— The >> concatenation operator can be used to append information to the end of a file.

### Example

The following example appends the contents of f_2.txt to the end of f_1.txt.

```
struct f_7_str {
    FAppend(ex_file_1: string, ex_file_2: string) is {
        var my_f_1: string;
        my_f_1 = files.add_file_type(ex_file_1, ".txt", TRUE);
        var my_f_2: string;
        my_f_2 = files.add_file_type(ex_file_2, ".txt", TRUE);
        files.file_append(my_f_1, my_f_2);
    }
};
extend sys {
    fi_7: f_7_str;
    run() is also {
        fi_7.FAppend("f_2.txt", "f_1.txt");
    };
};
```

### See Also

— "add_file_type()" on page 844

## 26.5.3 file_copy()

### Purpose

Create a copy of a file

### Category

Method

### Syntax

**files.file_copy(*from-file-name*:** string**, *to-file-name*:** string**)**

Syntax example:

```
files.file_copy("file_1.txt", "tmp_file.txt");
```

### Parameters

| | |
|---|---|
| *from-file-name* | The name of the file to copy. |
| *to-file-name* | The name of the copy of the file. |

### Description

Makes a copy of the file named ***from-file-name***, with the name ***to-file-name***. If a files already exists with the ***to-file-name***, the contents of that file are replaced by the contents of the file named ***from-file-name***. If the file named ***from-file-name*** does not exist, an error is issued.

### Example

The following example copies the contents of f_1.txt into f_1.bak.

```
struct f_str_8 {
    FCp(ex_file_1: string, ex_file_2:string) is {
        files.file_copy(ex_file_1, ex_file_2);
    };
};
extend sys {
    fi_8: f_str_8;
    run() is also {
        fi_8.FCp("f_1.txt", "f_1.bak");
    };
};
```

### See Also

— “file_rename()” on page 870

## 26.5.4 file_delete()

### Purpose

Delete a file

### Category

Method

### Syntax

**files.file_delete(*file-name*:** string**)**

Syntax example:

```
files.file_delete("run_1.log");
```

### Parameters

| | |
|---|---|
| *file-name* | The file to be deleted. |

### Description

Deletes a specified file. If the file cannot be found, an error like the following is issued.

```
*** Error: No match for file 'run_1.log'
```

### Example

The following example deletes the f_1.txt file.

```
struct f_str_9 {
    FDel(ex_file: string) is {
        var my_f_1: string;
        my_f_1 = files.add_file_type(ex_file, ".txt", TRUE);
        files.file_delete(my_f_1);
    };
};
extend sys {
    fi_9: f_str_9;
    run() is also {
        fi_9.FDel("f_1.txt");
    };
};
```

### See Also

— "file_exists()" on page 861

## 26.5.5 file_exists()

### Purpose

Check if a file exists

### Category

Method

### Syntax

**files.file_exists(*file-name*:** string**):** bool

Syntax example:

```
var f_e: bool;
f_e = files.file_exists("file_1.e");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

### Description

Check if the ***file-name*** exists in the file system. Return TRUE if the file exists or issues an error if it does not exist. Also returns TRUE if the file is a directory. The routine does not check whether the file is readable or not.

NOTE— This routine only checks for the existence of a file with the exact name you specify. For a routine that can check for multiple similarly named files, see "add_file_type()" on page 844.

### Example

The following example prints "file f_1.txt exists" if there is a file named "f_1.txt" in the current directory. If the file does not exist, an error is issued.

```
struct f_str_10 {
    FEx(ex_file: string) is {
        var my_f_1: string;
        my_f_1 = files.add_file_type(ex_file, ".txt", TRUE);
        var f_exists: bool;
        f_exists = files.file_exists(my_f_1);
        if f_exists then {outf("file %s exists\n", my_f_1)};
    };
};
extend sys {
    fi_10: f_str_10;
    run() is also {
        fi_10.FEx("f_1.txt");
    };
};
```

### See Also

— "add_file_type()" on page 844
— "file_is_dir()" on page 863
— "file_is_readable()" on page 865
— "file_is_regular()" on page 866
— "file_is_link()" on page 864
— "file_is_text()" on page 869
— Table 3-4 on page 105, for information about type conversion between scalar types

## 26.5.6 file_extension()

### Purpose

Get the extension of a file

### Category

Method

### Syntax

**files.file_extension(*file-name*:** string**)**: string

Syntax example:

```
var f_ext: string;
f_ext = files.file_extension("f_1.exa");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file. |

### Description

Returns a string containing the file extension, which is the sequence of characters after the last period (.).

### Example

The following example prints “get_ext = “.bak””.

```
struct f_str_11 {
    FExten(ex_file: string) is {
        var get_ext: string;
        get_ext = files.file_extension(ex_file);
        print get_ext;
    };
};
extend sys {
    fi_11: f_str_11;
    run() is also {
        fi_11.FExten("f_1.bak");
    };
};
```

### See Also

- “add_file_type()” on page 844
- “file_exists()” on page 861
- Table 3-4, “Type Conversion Between Scalars and Lists of Scalars”, on page 105, for information about type conversion between strings and scalar types

## 26.5.7 file_is_dir()

### Purpose

Check if a file is a directory

### Category

Method

### Syntax

**files.file_is_dir(*file-name*:** string**):** bool

Syntax example:

```
var is_d: bool;
is_d = files.file_is_dir("a_fil");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

### Description

Returns TRUE if the file exists and is a directory. Returns FALSE if the file does not exist or is not a directory.

### Example

The following example prints TRUE if f_1 is a directory, or FALSE if f_1 does not exist or if it is not a directory.

```
struct f_str_12 {
    F_is_Dir(ex_file: string) is {
        var is_dir: bool;
        is_dir = files.file_is_dir(ex_file);
        outf("%s is_dir = %s\n", ex_file, is_dir);
    };
};
extend sys {
    fi_12: f_str_12;
    run() is also {
        fi_12.F_is_Dir("f_1");
    };
};
```

### See Also

— "file_exists()" on page 861
— "file_is_link()" on page 864
— "file_is_readable()" on page 865)
— "file_is_regular()" on page 866
— "file_is_temp()" on page 868
— "file_is_text()" on page 869
— Table 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105, for information about type conversion between scalar types
— Table 3-5, "Type Conversion Between Strings and Scalars or Lists of Scalars", on page 107, for information about type conversion between strings and scalar types

## 26.5.8 file_is_link()

### Purpose

Check if a file is a symbolic link

### Category

Method

### Syntax

**files.file_is_link(*file-name***: string**)**: bool

Syntax example:

```
var is_l: bool;
is_l = files.file_is_link("a_fil");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

### Description

Returns TRUE if the file exists and is a symbolic link. Returns FALSE if the file does not exist or is not a symbolic link.

### Example

The following example prints TRUE if f_1 is a symbolic link, or FALSE if f_1 does not exist or if it is not a symbolic link.

```
struct f_str_13 {
    F_is_Link(ex_file: string) is {
        var is_link: bool;
        is_link = files.file_is_link(ex_file);
        outf("%s is_link = %s\n", ex_file, is_link);
    };
};
extend sys {
    fi_13: f_str_13;
    run() is also {
        fi_13.F_is_Link("f_1");
    };
};
```

### See Also

— “file_exists()” on page 861
— “file_is_dir()” on page 863
— “file_is_readable()” on page 865)
— “file_is_regular()” on page 866
— “file_is_temp()” on page 868
— “file_is_text()” on page 869
— Table 3-4, “Type Conversion Between Scalars and Lists of Scalars”, on page 105, for information about type conversion between scalar types

## 26.5.9 file_is_readable()

### Purpose

Check if a file is readable

### Category

Method

### Syntax

**files.file_is_readable(*file-name*:** string**)**: bool

Syntax example:

```
var is_rd: bool;
is_rd = files.file_is_readable("a_fil");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

### Description

Returns TRUE if the file exists and is readable. Returns FALSE if the file does not exist or is not readable.

### Example

The following example prints TRUE if f_1.dat is readable, or FALSE if f_1.dat does not exist or if it is not readable.

```
struct f_str_14 {
    F_is_Readable(ex_file: string) is {
        var is_readable: bool;
        is_readable = files.file_is_readable(ex_file);
        outf("%s is_readable = %s\n", ex_file, is_readable);
    };
};
extend sys {
    fi_14: f_str_14;
    run() is also {
        fi_14.F_is_Readable("f_1.dat");
    };
};
```

### See Also

— “file_exists()” on page 861
— “file_is_dir()” on page 863
— “file_is_link()” on page 864**)**
— “file_is_regular()” on page 866
— “file_is_temp()” on page 868
— “file_is_text()” on page 869
— Table 3-4, “Type Conversion Between Scalars and Lists of Scalars”, on page 105, for information about type conversion between scalar types

## 26.5.10 file_is_regular()

### Purpose

Check if a file is a regular file (not a directory or link)

### Category

Method

### Syntax

**files.file_is_regular(*file-name*:** string**)**: bool

Syntax example:

```
var is_rg: bool;
is_rg = files.file_is_regular("a_fil");
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

### Description

Returns TRUE if the file exists and is a regular file. Returns FALSE if the file does not exist or if it is a directory or a symbolic link.

### Example

The following example prints TRUE if f_1 is a regular file, or FALSE if f_1 does not exist or if it is a link or directory.

```
struct f_str_15 {
    F_is_Regular(ex_file: string) is {
        var is_regular: bool;
        is_regular = files.file_is_regular(ex_file);
        outf("%s is_regular = %s\n", ex_file, is_regular);
    };
};
extend sys {
    fi_15: f_str_15;
    run() is also {
        fi_15.F_is_Regular("f_1");
    };
};
```

### See Also

— “file_exists()” on page 861
— “file_is_dir()” on page 863
— “file_is_link()” on page 864
— “file_is_readable()” on page 865**)**
— “file_is_temp()” on page 868
— “file_is_text()” on page 869
— Table 3-4, “Type Conversion Between Scalars and Lists of Scalars”, on page 105, for information about type conversion between scalar types

### 26.5.11 file_is_temp()

#### Purpose

Check if a file name starts with "/tmp"

#### Category

Method

#### Syntax

**files.file_is_temp(*file-name*:** string**)**: bool

Syntax example:

```
var is_tmp: bool;
is_tmp = files.file_is_temp("a_fil");
```

#### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

#### Description

Returns TRUE if the file name starts with "/tmp", otherwise returns FALSE.

#### Example

The following example prints "/tmp/f_1.dat is_temp = TRUE".

```
struct f_str_16 {
    F_is_Temp(ex_file: string) is {
        var is_temp: bool;
        is_temp = files.file_is_temp(ex_file);
        outf("%s is_temp = %s\n", ex_file, is_temp);
    };
};
extend sys {
    fi_16: f_str_16;
    run() is also {
        fi_16.F_is_Temp("/tmp/f_1.dat");
    };
};
```

#### See Also

— "new_temp_file()" on page 872
— "file_exists()" on page 861
— "file_is_dir()" on page 863
— "file_is_link()" on page 864
— "file_is_readable()" on page 865**)**
— "file_is_regular()" on page 866
— "file_is_text()" on page 869

— Table 3-4, “Type Conversion Between Scalars and Lists of Scalars”, on page 105, for information about type conversion between scalar types

### 26.5.12 file_is_text()

#### Purpose

Check if a file is a text file

#### Category

Method

#### Syntax

**files.file_is_text(*file-name*:** string**)**: bool

Syntax example:

```
var is_txt: bool;
is_txt = files.file_is_text("a_fil");
```

#### Parameters

| | |
|---|---|
| *file-name* | The name of the file to be checked. |

#### Description

Returns TRUE if the file is a text file (that is, if it contains more than 20% printable characters). Returns FALSE if the file does not exist or if it is a not a text file.

#### Example

The following example prints TRUE if f_1.dat is a text file, or FALSE if f_1.dat does not exist or if it is not a text file.

```
struct f_str_17 {
    F_is_Text(ex_file: string) is {
        var is_text: bool;
        is_text = files.file_is_text(ex_file);
        outf("%s is_text = %s\n", ex_file, is_text);
    };
};
extend sys {
    fi_17: f_str_17;
    run() is also {
        fi_17.F_is_Text("f_1.dat");
    };
};
```

#### See Also

— “file_exists()” on page 861
— “file_is_dir()” on page 863
— “file_is_link()” on page 864

— "file_is_readable()" on page 865**)**
— "file_is_regular()" on page 866
— "file_is_temp()" on page 868
— Table 3-4, "Type Conversion Between Scalars and Lists of Scalars", on page 105, for information about type conversion between scalar types

## 26.5.13 file_rename()

### Purpose

Rename a file

### Category

Method

### Syntax

**files.file_rename(*from-file-name*:** string**, *to-file-name*:** string**)**

Syntax example:

```
files.file_rename("f_1.exa", "b_1.exa");
```

### Parameters

| | |
|---|---|
| *from-file-name* | The file to rename. |
| *to-file-name* | The new file name. |

### Description

Renames the file named ***from-file-name*** to ***to-file-name***. If any files already exists with ***to-file-name***, that file is overwritten by the contents of the file named ***from-file-name***.

If the file or directory is not writable, an error is issued.

### Example

The following example changes the name of the f_1.dat file to f_old.dat. If the f_1.dat file does not exist, the **files.add_file_type**() routine issues an error.

```
struct f_str_18 {
    FRename(ex_file_1: string, ex_file_2:string) is {
        var m_f: string;
        m_f = files.add_file_type(ex_file_1, "", TRUE);
        files.file_rename (m_f, ex_file_2);
    };
};
extend sys {
    fi_18: f_str_18;
    run() is also {
        fi_18.FRename("f_1.dat", "f_old.dat");
    };
};
```

### See Also

- "add_file_type()" on page 844
- "file_copy()" on page 859

## 26.5.14 file_size()

### Purpose

Get the size of a file

### Category

Method

### Syntax

**files.file_size(*file-name***: string**)**: int

Syntax example:

```
var f_s: int;
f_s = files.file_size("a_file.txt");
```

### Parameters

|  |  |
|---|---|
| *file-name* | The name of the file. |

### Description

Returns the integer number of bytes in the file. If the file does not exist, an error is issued.

### Example

The following example gets and displays the number of bytes in the file named "f_1.dat".

```
struct f_str_19 {
    FGetSize(ex_file: string) is {
        var m_f: string;
        m_f = files.add_file_type(ex_file, "", TRUE);
        var f_size: int;
        f_size = files.file_size (m_f);
        outf("%s size is %d\n", m_f, f_size);
    };
};
extend sys {
    fi_19: f_str_19;
    run() is also {
        fi_19.FGetSize("f_1.dat");
    };
};
```

### See Also

- "add_file_type()" on page 844

— “file_age()” on page 857

### 26.5.15 new_temp_file()

#### Purpose

Create a unique temporary file name

#### Category

Method

#### Syntax

**files.new_temp_file()**: string

Syntax example:

```
var t_name: string;
t_name = files.new_temp_file()
```

#### Description

Computes a file name. Each file name this routine produces contains the name of the process, so names are unique across processes. Returns a string with a period at the end.

The files are saved in the /tmp directory.

This routine only creates a file name. To create a file with this name, use the **files.open()** routine.

#### Example

The example below creates two file names in the /tmp directory and prints them.

```
struct f_str_20 {
    FMkTmp() is {
        var t_name_1: string;
        t_name_1 = files.new_temp_file();
        print t_name_1;
        var t_name_2: string;
        t_name_2 = files.new_temp_file();
        print t_name_2;
    };
};
extend sys {
    fi_20: f_str_20;
    run() is also {
        fi_20.FMkTmp();
    };
};
```

The example above prints the following.

```
  t_name_1 = "/tmp/proc5924/snt_5924_12698."
  t_name_2 = "/tmp/proc5924/snt_5924_12699."
```

### See Also

- “file_is_temp()” on page 868
- “open()” on page 848
- Table 3-5, “Type Conversion Between Strings and Scalars or Lists of Scalars”, on page 107, for information about type conversion between strings and scalar types

## 26.5.16 write_string_list()

### Purpose

Write a list of strings to a file

### Category

Method

### Syntax

**files.write_string_list(*file-name***: string**, *strings***: list of string**)**

Syntax example:

```
var s_list:= {"a string"; "another string"};
files.write_string_list("a_file.txt", s_list);
```

### Parameters

| | |
|---|---|
| *file-name* | The file name to write into. |
| *strings* | A list of strings to write to the file. |

### Description

Writes a list of strings into a file. Every string is written on a separate line in the file, with \n appended to the end of the string. If the file already exists, it is overwritten.

If the list of strings contains a NULL, an error is issued.

### Example

The following example writes three lines of text into a file named “f_1.txt”.

```
struct f_str_21 {
    FWrStr(ex_file: string, str_list: list of string) is {
        var m_f: string;
        m_f = files.add_file_type(ex_file, "", TRUE);
        files.write_string_list(ex_file, str_list);
    };
};
extend sys {
    fi_21: f_str_21;
    run() is also {
        var fname:string;
        fname = "f_1.txt";
```

```
            var strlist: list of string;
            strlist = {"first line"; "second line"; "third line"};
            fi_21.FWrStr(fname, strlist);
        };
    };
```

### See Also

— "open()" on page 848
— "write()" on page 852
— "write_lob()" on page 853
— "write_ascii_struct()" on page 877
— "write_binary_struct()" on page 880

## 26.6 Reading and Writing Structs

Structs in ***e*** can be read from files and written to files in either binary or ASCII format.

The routines that read structs from files and write structs to files are listed below and described in this section.

— "read_ascii_struct()" on page 874
— "read_binary_struct()" on page 875
— "write_ascii_struct()" on page 877
— "write_binary_struct()" on page 880

### See Also

— "General File Routines" on page 856
— "Low-Level File Routines" on page 843

### 26.6.1 read_ascii_struct()

#### Purpose

Read ASCII file data into a struct

#### Category

Method

#### Syntax

**files.read_ascii_struct(*file-name***: string**, *struct***: struct-type**)**: struct

Syntax example:

```
var a_str: s_struct;
a_str = files.read_ascii_struct("a_s.out",
    "s_struct").as_a(s_struct);
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to read from. The file may have been created either with **files.write_ascii_struct()** or in a similar format with an editor. |
| *struct* | The struct type to read data into. |

### Description

Reads the ASCII contents of ***file-name*** into a struct of type ***struct***, and returns a struct. The struct being read must be cast to the correct data type (see "as_a()" on page 104). If the file does not exist, an error is issued.

### Example

The following example creates a variable named "str" to hold an instance of the s_st struct type, reads ASCII contents of a file named "a_s.out" into the struct variable, and prints the contents.

```
struct s_st {
    len: int;
    hdr: string;
    b_l: list of bool;
};
struct r_st {
    r_file() is {
        var str: s_st;
        str = files.read_ascii_struct("a_s.out",
            "s_st").as_a(s_st);
        print str;
    };
    run() is also {
        r_file();
    };
};
extend sys {
    ri: r_st;
};
```

### See Also

— "read()" on page 850
— "read_lob()" on page 851
— "file_is_readable()" on page 865
— "read_binary_struct()" on page 875
— "write_ascii_struct()" on page 877

## 26.6.2 read_binary_struct()

### Purpose

Read the contents of a binary file into a struct

### Category

Method

### Syntax

**files.read_binary_struct(*file-name***: string**, *struct***: struct-type**,**
**    *check-version***: bool**)**: struct

Syntax example:

```
var b_str: s_struct;
b_str = files.read_binary_struct("b.out", "s_struct",
     TRUE).as_a(s_struct);
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to read from. The file must have been created by "write_binary_struct()" on page 880. |
| *struct* | The struct type to read data into. |
| *check-version* | Set to TRUE to compare the contents of the file being read with the definition of the struct in the currently running module. Set to FALSE to allow minor changes. |

### Description

Reads the binary contents of ***file-name*** into a struct of the specified type, and returns a struct. The struct being read must be cast to the correct data type (see "as_a()" on page 104).

If ***check-version*** is FALSE, the routine can run even if the order of fields in the file struct is different from the order of fields in the currently running ***e*** module. If ***check-version*** is TRUE, an error is issued if the struct definition has been changed in any way since the struct was written to the file.

### Example

The following example creates a variable named "str" to hold an instance of the s_st struct type, reads binary contents of a file named "b_s.out" into the struct variable, and prints the contents. The ***check-version*** parameter is set to TRUE to issue an error if the b_s.out file struct does not exactly match the s_st definition. The b_s.out binary struct file was created previously by "write_binary_struct()" on page 880.

```
struct s_st {
    len: int;
    hdr: string;
    b_l: list of bool;
};
struct r_st {
    r_file() is {
        var str: s_st;
        str = files.read_binary_struct("b_s.out", "s_st",
            TRUE).as_a(s_st);
        print str;
    };
    run() is also {
        r_file();
    };
};
extend sys {
    ri: r_st;
};
```

### See Also

- — "read()" on page 850
- — "read_lob()" on page 851
- — "file_is_readable()" on page 865
- — "read_ascii_struct()" on page 874
- — "write_ascii_struct()" on page 877

## 26.6.3 write_ascii_struct()

### Purpose

Write the contents of a struct to a file in ASCII format

### Category

Method

### Syntax

**files.write_ascii_struct(*file-name*:** string**, *struct*:** struct**, *comment*:** string**,**
   ***indent*:** bool**, *depth*:** int**, *max-list-items*:** int**)**

Syntax example:

```
   files.write_ascii_struct("a_file.dat", a_str, "my_struct",
       TRUE, 2, 10);
```

### Parameters

| | |
|---|---|
| *file-name* | The name of the file to write into. If you do not specify a file name extension, the default extension is .erd, which stands for ***e***-readable data. |
| *struct* | The name of the struct instance to write to the file. |
| *comment* | A string for a comment at the beginning of the file. |
| *indent* | Boolean selector for indentation to the struct's field depth. |
| *depth* | The number of levels of nested structs to write. |
| *max-list-items* | For lists, how many items from each list to write. |

### Description

Recursively writes the contents of the ***struct*** to the ***file-name*** in ASCII format. If the struct contains other structs, those structs are also written to the file. If the number of hierarchical levels contained in the ***struct*** is greater than the specified ***depth***, levels below the ***depth*** level are represented by ellipses (...) in the ASCII file.

If the file already exists, it is overwritten.

This routine will not write any of the ***e*** program internal structs. It will write the **sys** struct, but not any predefined structs within **sys**.

The .erd default file name extension is automatically added to the file name only if the file name you specify has no extension and does not end with "." (a period). That is, if you enter "myfile", the file name becomes “myfile.erd”. If you enter “myfile.”, the file is named “myfile.”. If you enter “myfile.out”, the file is named “myfile.out”.

## Example

In the following example, there are three levels of hierarchy under the sys struct: the w_st struct contains a s_st struct, which contains a list of dat_s structs. The ss_i instance of the s_st struct is written to an ASCII file with these options:

— The comment “My ASCII struct” is placed at the top of the file.
— Indentation of the struct’s fields is TRUE.
— Only the highest hierarchical level of structs is written (***depth*** = 1).
— The first three items in lists are written.

```
struct dat_s {
    dat_l: list of uint;
    keep dat_l.size() == 5;
};
struct s_st {
    ds_l: list of dat_s;
    keep ds_l.size() == 6;
    len: int;
    hdr: string;
    b_l: list of bool;
};
struct w_st {
    ss_i: s_st;
    wr_file() is {
        files.write_ascii_struct("a_s.out", ss_i,
            "My ASCII struct", TRUE, 1, 3);
    };
    run() is also {
        wr_file();
    };
};
extend sys {
    wi: w_st;
};
```

The following is the a_s.out file created by the example above.

```
-- My ASCII struct
-- The top struct
struct: s_st-@0{
    ds_l:
        -- struct: ...
        -- struct: ...
        -- struct: ...
        -- ds_l[3..5] ...
    };
    len: -2025306869
    hdr: ""
    b_l:
        FALSE
        TRUE
        FALSE
        -- b_l[3..3] ...
    };
    };
```

Changing ***depth*** from 1 to 2 in the example above adds a level of hierarchy to the results, which produces the following file.

```
    -- My ASCII struct
    -- The top struct
    struct: s_st-@0{
        ds_l:
            -- ds_l[0]
            struct: dat_s-@1{
                -- root___unit: ...
                dat_l:
                    4166871515
                    381462224
                    2293917550
                    -- dat_l[3..4] ...
                };
            };
            -- ds_l[1]
            struct: dat_s-@2{
                -- root___unit: ...
                dat_l:
                    3680934570
                    495418143
                    1152908095
                    -- dat_l[3..4] ...
                };
            };
            -- ds_l[2]
            struct: dat_s-@3{
                -- root___unit: ...
                dat_l:
                    1924257378
                    1889370393
                    3534009340
                    -- dat_l[3..4] ...
                };
            };
            -- ds_l[3..5] ...
        };
        len: -2025306869
        hdr: ""
        b_l:
            FALSE
            TRUE
            FALSE
            -- b_l[3..3] ...
        };
    };
```

### See Also

— “write()” on page 852
— “write_lob()” on page 853
— “read_ascii_struct()” on page 874
— “write_binary_struct()” on page 880

### 26.6.4 write_binary_struct()

#### Purpose

Write the contents of a struct to a file in binary format

#### Category

Method

#### Syntax

**files.write_binary_struct(*file-name***: string**, *struct***: struct**)**

Syntax example:

```
files.write_binary_struct("b_file.dat", b_str);
```

#### Parameters

| | |
|---|---|
| *file-name* | The name of the file to write structs into. |
| *struct* | The name of the struct instance to write to the file. |

#### Description

Recursively writes the contents of the ***struct*** to the ***file-name*** in binary format. If the struct contains other structs, those structs are also written to the file. If the file already exists, it is overwritten.

#### Example

The following example creates a struct instance named “str” and writes the struct’s contents in binary format to a file named “b_s.out”.

```
struct s_st {
    len: int;
    hdr: string;
    b_l: list of bool;
};
struct w_st {
    wr_file() is {
        var str := a new s_st with {
            .len = 1;
            .hdr = "top";
            .b_l = { TRUE; FALSE; TRUE };
        };
        files.write_binary_struct("b_s.out", str);
    };
    run() is also {
        wr_file();
    };
};
extend sys {
    wi: w_st;
};
```

### See Also

— "write()" on page 852
— "write_lob()" on page 853
— "read_binary_struct()" on page 875
— "write_ascii_struct()" on page 877

# 27 State Machines Library

This chapter contains descriptions of how to create state machines and of the constructs used in them. It contains the following sections.

— “State Machine Overview” on page 883
— “State Machine Constructs” on page 883
— “Sample State Machine” on page 888
— “Using State Machines” on page 889

### See Also

— “Invoking Methods” on page 474
— “e Data Types” on page 75

## 27.1 State Machine Overview

The ***e*** language **state machine** action provides constructs for modeling state machines in ***e*** .

A state machine definition consists of the **state machine** action followed by a state holder expression and a block that specifies the ways the state machine can get from one state to another (see “state machine” on page 883).

State machines can be defined only within time-consuming methods (TCMs). When the execution of a TCM reaches a **state machine** action, the appropriate series of state transitions occurs, and then the state machine exits. At this point the TCM continues from the action following the **state machine** action.

### See Also

— “State Machine Constructs” on page 883
— “Sample State Machine” on page 888
— “Using State Machines” on page 889

## 27.2 State Machine Constructs

The ***e*** state machine constructs are used to define state machines and the transitions between their states. This section contains descriptions of the following constructs.

— “state machine” on page 883
— “state => state” on page 886
— “* => state” on page 887
— “state action” on page 887

### 27.2.1 state machine

#### Purpose

Define a state machine

### Category

Action

### Syntax

**state machine** ***state-holder-exp*** [**until** ***final-state***]
  {(***state-transition*** | ***state***) {***action***; ...}; ...}

Syntax example:

```
!c_proc: [st, dn];
s_m()@sys.clk is {
    state machine c_proc until dn {
        *=> st {wait rise('top.rst'); wait [2]*cycle};
        st => dn {out("going to dn"); wait [3]*cycle;};
    };
};
```

### Parameters

| | |
|---|---|
| *state-holder-exp* | Stores the current state of the state machine. This can be a variable in the current TCM, a field under **sys**, or any assignable expression. It typically is an enumerated type field of the struct in which the TCM is defined. |
| *final-state* | The state at which the state machine terminates. |
| *state-transition* | A state transition, which occurs when the associated action block finishes. See "state => state" on page 886 and "* => state" on page 887. |
| *state* | A state. When this state is entered, the associated action block is invoked. See "state action" on page 887. |
| ***action***; ... | One of the following:<br>— An action block which, upon completion, causes the transition to occur, if the state-transition syntax is used.<br>— An action block that is to be performed when the given state is entered, if the state syntax is used. |

### Description

Defines a state machine using an enumerated ***state-holder-exp*** to hold the current state of the machine.

The state machine must be defined in a time-consuming method (TCM). When the **state machine** action is reached, the state machine starts, in the first state listed in the enumerated state holder expression type definition.

During the execution of the **state machine** action the current state is stored in the ***state-holder-exp***.

If the optional **until** ***final-state*** exit condition is used, the state machine runs until that state is reached. The final state must be one of the enumerated values declared with the state machine name.

If the **until** clause is not used, the state machine runs until the TCM is terminated, or, if the state machine is in an **all of** or **first of** action, it runs until the **all of** or **first of** action completes (see "Terminating a State Machine" on page 890).

The ***state-transition*** block is a list of the allowed transitions between states of the state machine. Each state transition contains an action block that defines conditions that cause the transition to occur. Typically, the action block contains a single **wait until** action. However, it can contain any block of actions. For example,

```
x => y {wait until @named_event_1; wait until @named_event_2};
```

The transition only occurs after both events happen, in order.

The action block can contain a regular method, as in the following.

```
x => y {wait until change(p_clk); me.packet.bb_operations()};
```

Once **change(**p_clk**)** happens, the method executes immediately, and then the transition occurs.

## Example

In the following example, the struct field expression used for the state machine is the "status" field declaration. The state machine name is "status", and its possible states are "start" and "done". The state machine is defined in the "sm_meth()" TCM. It has a final state of "done", meaning the state machine terminates when it enters the "done" state.

Since the "start" field is listed first in the list of states, that is the initial state for the state machine. The state changes from "start" to "done" two "sys.smclk" cycles after it enters the "start" state. Upon entering the "done" state, the state machine exits. The **out()** action is executed after the state machine exits.

```
struct smp_state_machine{
    !status: [start, done];
    sm_meth() @sys.smclk is {
        state machine status until done {
            start => done {wait [2]*cycle};
        };
        out("The status state machine is done");
    };
};
```

A more complex state machine is shown below. The name of the state machine is "arbiter_state", and it is declared with states "idle", "busy", "grant", and "reject".

This state machine has no "**until** ***finish-state***" exit condition, so it runs until it the "watcher()" TCM is terminated.

The "* => idle" syntax means "from any other state to the idle state". The condition for this transition is that 10 cycles of "sys.pclk" have elapsed since the state machine entered the any state.

```
struct bus {
    !arbiter_state: [idle, busy, grant, reject];
    watcher() @sys.pclk is {
        wait [3]*cycle;
        state machine arbiter_state {
            idle => busy {wait @sys.req};
            busy => grant {wait [2]*cycle};
            busy => reject {wait @sys.bad_pkt};
            * => idle {wait [10]*cycle};
        };
    };
};
```

### See Also

— "State Machine Overview" on page 883
— "state => state" on page 886
— "* => state" on page 887
— "state action" on page 887

## 27.2.2 state => state

### Purpose

One-to-one state transition

### Category

State transition

### Syntax

***current-state*=>*next-state*** {***action***; ...}

Syntax example:

```
begin => run {wait [2]*cycle; out("Run state entered")};
```

### Parameters

| | |
|---|---|
| *current-state* | The state from which the transition starts. |
| *next-state* | The state to which the transition changes. |
| ***action***; ... | The sequence of actions that precede the transition. It usually contains at least one time-consuming action. |

### Description

Specifies how a transition occurs from one state to another. The action block starts executing when the state machine enters the current state. When the action block completes, the transition to the next state occurs.

### Example

The example below shows a definition of a transition for the "initial" state to the "running" state. If the 'top.start' HDL signal changes while the state machine is in the "initial" state, the state changes to "running".

```
initial => running {wait until change('top.start')@sim};
```

### See Also

— "State Machine Overview" on page 883
— "state machine" on page 883
— "* => state" on page 887
— "state action" on page 887

### 27.2.3 * => state

#### Purpose

Any-to-one state transition

#### Category

State transition

#### Syntax

***=>*next-state* {*action*; ...}**

Syntax example:

```
* => pause {wait @sys.restart; out("Entering pause state");};
```

#### Parameters

| | |
|---|---|
| *next-state* | The state to which the transition changes. |
| ***action***; ... | The sequence of actions that precede the transition. It usually contains at least one time-consuming action. |

#### Description

Specifies how a transition occurs from any defined state to a particular state. The action block starts executing when the state machine enters a new state. When the action block completes, the transition to the next state occurs.

#### Example

The example below shows a definition of a transition for any state to the “running” state. From any state, if the 'top.start' HDL signal rises and later the 'top.hold' signal falls, the state changes to “running”.

```
* => running { wait until rise('top.start')@pclk;
               wait until fall('top.hold')@pclk };
```

#### See Also

— “State Machine Overview” on page 883
— “state machine” on page 883
— “state => state” on page 886
— “state action” on page 887

### 27.2.4 state action

#### Purpose

Execute actions upon entering a state, with no state transition

#### Category

State action block

### Syntax

***current-state*** **{*action*; ...}**

Syntax example:

```
* => run {out("* to run"); wait cycle};
run {out("In run state"); wait cycle; out("Still in run");};
run => done {out("run to done"); wait cycle};
```

### Parameters

| | |
|---|---|
| *current-state* | The state for which the action block is to be executed. |
| ***action***; ... | The sequence of actions that is executed upon entering the current state. It usually contains at least one time-consuming action. |

### Description

Specifies an action block that is executed when a specific state is entered. No transition occurs when the action block completes. The state machine stays in the current state until some other transition takes place.

### Example

The last two lines in the following example contain an action block that is to be executed when the state machine enters the "running" state. The "**while** TRUE ..." action means that as long as the state machine is in the "running" state, the **out()** action is executed every cycle.

```
state machine sm_1 until done {
    initial => running { wait until rise('top.a') };
    initial => done { wait until change('top.r1');
                      wait until rise('top.r2') };
    running => initial { wait until rise('top.b') };
running {
    out("Entered running state");
        while TRUE {wait cycle; out("still running");}
    };
};
```

### See Also

— "State Machine Overview" on page 883
— "state machine" on page 883
— "state => state" on page 886
— "* => state" on page 887

## 27.3 Sample State Machine

The following example shows a single state machine. The state machine is declared in the "sm_1" field, with possible states named "initial", "running", and "done".

```
struct exa_1_state_machine {
    !sm_1: [initial, running, done];
    tcm_1()@sys.sm_clk is {
        wait [10]*cycle;
```

```
            state machine sm_1 until done {
               initial => running { wait until rise('top.a') };
               initial => done { wait until change('top.r1');
                   wait until rise('top.r2') };
               running => done {wait until fall('top.b')};
               running {while TRUE {out("Running"); wait cycle}};
            };
            out("tcm_1 finished");
        };
    };
```

The “sm_1” state machine is defined in the “tcm_1()” TCM. Note that the TCM contains other actions besides the state machine. There is a 10-cycle **wait** before the state machine starts, and an **out()** that is executed after the state machine is finished.

The “**until** done” clause means that the state machine will run until it reaches the “done” state.

The transition definitions are as follows:

| | |
|---|---|
| initial => running | A rise of the 'top.a' HDL signal causes a transition from “initial” to “running”. |
| initial => done | A change in the 'top.r1' signal followed eventually by a rise in the 'top.r2' signal causes a transition from “initial” to “done”. |
| running => done | A fall of the 'top.b' signal causes a transition from “running” to “done”. |
| running | When the state machine enters the “running” state, continuously execute the “{**out**(“Running”); **wait cycle**};” action block until the state changes. |

### See Also

— “State Machine Overview” on page 883
— “State Machine Constructs” on page 883
— “Using State Machines” on page 889

## 27.4 Using State Machines

This section contains the following topics.

— “Initializing a State Machine” on page 889
— “Terminating a State Machine” on page 890
— “Rules for State Transitions” on page 891
— “Nested State Machines” on page 892
— “Parallel State Machines” on page 892

### 27.4.1 Initializing a State Machine

State machines start by default in the first state specified in the enumerated type definition of the ***state-holder-exp*** (see “State Machine Overview” on page 883). In the following, the starting state for state machine “sm_2” is “initial” because that is the first state listed in the “sm_2” type definition.

```
    struct exa_2_state_machine{
        !sm_2: [initial, running, done];
        tcm_2()@sys.sm_clk is {
            state machine sm_2 until done {
                // ...
            };
        };
    };
```

If the state machine is entered several times in the same TCM, it is initialized to the starting state each time it is entered. A state machine can be re-entered if it is nested in another state machine or if it is enclosed in a loop. Conditional initialization of the state machine can be performed within the state machine as shown in the following.

```
    struct exa_2_state_machine{
        !sm_2: [initial, init_cond, init_no_cond, running, done];
        tcm_2()@sys.sm_clk is {
            state machine sm_2 until done {
                initial => init_cond {sync true(cond);};
                initial => init_no_cond {sync true(not cond);};
                // ...
            };
        };
    };
```

### See Also

— "State Machine Overview" on page 883

## 27.4.2 Terminating a State Machine

You can terminate a state machine in any of the following ways.

— Specify a final state in an **until** clause.
— Enclose the state machine within a **first of** action.
— Terminate the TCM using the **quit()** method.

A state machine defined as follows will exit when it reaches the "done" state. The TCM continues execution.

```
    struct exa_3_state_machine{
        !sm_3: [initial, running, done];
        tcm_3()@sys.tclk is {
            state machine sm_3 until done {
                // ...
            };
            // ...
        };
    };
```

The following state machine is enclosed in a **first of** action. The other thread of the **first of** action terminates after **wait [**MAXN**] * cycle**. If the state machine runs for MAXN cycles, the **wait** thread finishes and the TCM terminates.

```
    struct exa_4_state_machine {
        !sm_4: [initial, running, done];
        tcm_4()@sys.tclk is {
            first of {
                {wait [MAXN]*cycle;};
                {state machine sm_4 {
                        // ...
                    };
                };
                // ...
            };
        };
    };
```

The **quit()** method of the struct can be used in another TCM to terminate all active TCMs and their state machines. This method cannot be used to terminate only one of several active TCMs, nor can it terminate a state machine while allowing the TCM to continue executing. In the following example, a TCM in **sys** calls the **quit()** method of the “exa_4_state_machine” instance, which terminates the “tcm_4()” TCM and the state machine.

```
    struct exa_4_state_machine{
        !sm_4: [initial, running, done];
        tcm_4()@sys.tclk is {
            state machine sm_4 {
                // ...
            };
        };
    };
```

### See Also

— “State Machine Overview” on page 883

## 27.4.3 Rules for State Transitions

— A transition takes place when its action block finishes.
— If there are several contending transitions (for example, several transitions with the same ***current-state***), their action blocks are executed in parallel. The transition whose action block finishes first is the one that occurs.
— When the action blocks for two transitions can complete during the same cycle, it is not possible to determine which transition will prevail. One will occur successfully and the other will not occur.
— Action blocks can take time, but transitions themselves take no time.

If the state machine specifies:

```
    x => y {sync true('cpu.clock' == 1)};
    y => z {sync true('alu.clock' == 1)};
```

and both ‘cpu.clock’ and ‘alu.clock’ are high within the same cycle, the two transitions both occur within the same cycle.

### See Also

— “State Machine Overview” on page 883

### 27.4.4 Nested State Machines

A state machine is just an action like all others, so it can appear anywhere an action can appear. This makes nested and parallel state machines possible. For example, the following contains a state machine named “run_to_finish” nested within another state machine named “sm_5”.

```
struct exa_5_state_machine {
    !sm_5: [begin, run, finish];
    run_to_finish: [s_b1, s_b2, s_b3];
    tcm_5() @sys.pclk is {
        state machine sm_5 until finish {
            begin => run {wait [2]*cycle};
            run => finish {
                state machine run_to_finish until s_b3 {
                    s_b1 => s_b2 {wait [2]*cycle};
                    s_b2 => s_b1 {wait [3]*cycle};
                    s_b2 => s_b3 {wait @sys.s_reset};
                };
            };
            * => begin {wait @sys.reset};
        };
    };
};
```

Whenever the “sm_5” state machine enters the “run” state, it starts the nested “run_to_finish” state machine. When that machine finally reaches its “s_b3” state it exits, and the “sm_5” state machine enters its “finish” state.

If “sys.reset” becomes TRUE, the “sm_5” state machine enters its “begin” state regardless of the current state of the “run_to_finish” state machine. This is an example of preempting a state machine from the outside.

#### See Also

— “State Machine Overview” on page 883

### 27.4.5 Parallel State Machines

An example of parallel state machines is shown below.

```
struct exa_6_state_machine {
    !sm_6a: [x1, x2, x3];
    !sm_6b: [y1, y2];
    tcm_6() @sys.sclk is {
        all of {
            state machine sm_6a until x3 {
                // ...
            };
            state machine sm_6b until y2 {
                // ...
            };
        };
        out("sm_6a and sm_6b state machines are both done");
    };
};
```

The two state machines in the example above are entered at the same time, and each proceeds independently of the other. Because they are started in an **all of** construct, both state machines must exit before the **out()** action can be executed.

In the following example, the two state machines are started in a **first of** rather than **all of** construct.

```
struct exa_6_2_state_machine {
    !sm_6a_2: [x1, x2, x3];
    !sm_6b_2: [y1, y2];
    tcm_6_2() @sys.sclk is {
        first of {
            state machine sm_6a_2 until x3 {
                // ...
            };
            state machine sm_6b_2 until y2 {
                //...
            };
        };
        out("either sm_6a_2 or sm_6b_2 state machine is done");
    };
};
```

Parallel state machines can be nested within another state machine, as in the following.

```
a => b {
    all of {
        state machine x until end_x {...};
        state machine y until end_y {...};
    };
};
```

## See Also

— “State Machine Overview” on page 883

# Index