OpenVera Language Reference Manual: Native Testbench

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Contents

OpenVera NTB Basics

NTB-OV Programming Overview

NTB-OV Statements, Assignments, and Control

NTB-OV Classes

Concurrency Control

Interfaces and Signal Operations

Aspect Oriented Extensions

Predefined Methods and Procedures

Randomization in NTB-OV

Random Sequence Generation

Functional Coverage Groups

1

OpenVera NTB Basics

This chapter explains the basic lexical elements of the OpenVera Native Testbench (NTB-OV) language. NTB-OV is one flavor of testbench language that you can use natively in VCS to verify your designs (the other flavor is SystemVerilog). NTB-OV supports most (but not all) of the features available in the OpenVera standard. The supported features are documented in this manual. OpenVera Native Testbench is referred to simply as NTB-OV in the remainder of this manual. This chapter explains NTB-OV basics in the following major sections:

- Lexical Elements
- Data Types and Variable Declarations
- Operators
- Arrays

Lexical Elements

NTB-OV source code consists of a stream of the following lexical elements:

- White Space
- Comments
- Identifiers
- Keywords
- Strings
- Numbers

White Space

White space is any sequence of spaces, tabs, newlines, and formfeeds. NTB-OV uses white space as a token separator. The tool ignores extra white space except within strings.

Comments

NTB-OV supports single-line and block comments. A single-line comment starts with a double slash (`//`) and finishes out the line. The syntax is:

```
any_NTB-OV_statement; // One-line comment
```

A block comment starts with a `/*` and ends with a `*/`. Everything between the start and end tags is a comment. For example:

```
/*Blocks of comments
can take up
multiple lines */
```

You cannot nest block comments.

Identifiers

An identifier is a sequence of letters `[a-zA-Z]`, digits `[0-9]`, and underscores[_]. Identifiers are case-sensitive and `cannot` begin with a digit.

Note:

Keywords cannot be used as identifiers.

Keywords

The NTB-OV keywords are listed in Table 1-1.

Table 1-1 NTB-OV Keywords

after	coverage_group	join	reg
all	coverage_option	little_endian	repeat
any	coverage_val	local	return
around	default	m_bad_state	rules
assoc_index	depth	m_bad_trans	shadow
assoc_size	dist	m_state	soft
async	do	m_trans	state
bad_state	else	negedge	static
bad_trans	end	new	string
before	enum	newcov	super

Table 1-1 NTB-OV Keywords (Continued)

begin	event	non_rand	task
big_endian	export	none	terminate
bind	extends	not	this
bind_var	extern	null	trans
bit	for	or	typedef
bit_normal	foreach	output	unpacked
bit_reverse	fork	packed	var
break	function	port	vca
breakpoint	hdl_node	posedge	vector
case	hdl_task	proceed	verilog_node
casex	hide	prod	verilog_task
casez	if	prodget	vhdl_node
class	illegal_self_transition	prodset	vhdl_task
CLOCK	illegal_state	program	virtual
constraint	illegal_transition	protected	virtuals
continue	in	public	void
coverage_block	inout	rand	while
coverage_def	input	randc	wildcard
coverage_depth	integer	randcase	with
coverage_goal	interface	randseq	

NTB-OV has predefined system macro constants whose names must not be used or redefined without proper system tasks. Table 1-2 lists the NTB-OV predefined constant identifiers.

Table 1-2 NTB-OV Predefined Constants

stderr	EC_RETURN	LIC_PRWARN	ONE_BLAST
stdin	EC_RGNTMOUT	LIC_WAIT	ONE_SHOT
stdout	EC_SCONFLICT	LO	ORDER
ALL	EC_SEMTMOUT	LOAD	PAST_IT

Table 1-2 NTB-OV Predefined Constants (Continued)

ANY	EC_SEXPECT	LOW	PERCENT
BAD_STATE	EC_SFULLEXPECT	LT	POSEDGE
BAD_TRANS	EC_SNEXTPECT	MAILBOX	PROGRAM
CALL	EC_EVNTIMOUT	MAX_COM	RAWIN
CHECK	EC_USERSET	NAME	REGION
CHGEDGE	EQ	NEGEDGE	REPORT
CLEAR	EVENT	NE	SAMPLE
COPY_NO_WAIT	FAIL	NEXT	SAVE
COPY_WAIT	FIRST	NO_OVERLAP	SEMAPHORE
CROSS	FORK	NO_OVERLAP _STATE	SET
CROSS_TRANS	GE	NO_OVERLAP _TRANS	SILENT
DEBUG	GOAL	NO_VARS	STATE
DELETE	GT	NO_WAIT	STR
EC_ARRAYX	HAND_SHAKE	NUM	STR_ERR_O UT_OF_RAN GE
EC_CODE_END	HI	NUM_BIN	STR_ERR_R EGEXP_SYN TAX
EC_CONFLICT	HIGH	NUM_DET	SUM
EC_EXPECT	HNUM	OFF	TRANS
EC_FULLEXPECT	LE	OK	VERBOSE
EC_MBXTMOUT	LIC_EXIT	OK_LAST	WAIT
EC_NEXPECT	LIC_PRERR	ON	

Strings

A string is a sequence of characters enclosed by double quotes. A string must be contained in a single line unless the new line is immediately preceded by a backslash.

Numbers

In NTB-OV, you can form a number using two different formats. The syntax for the first format is a simple decimal integer:

```
[0123456789_]+
```

where:

`+` matches the preceding pattern one or more times

`[...]` matches a character from the enclosed set

The syntax for the second format is:

size '*base number*

`size`

specifies the number of bits in the number. If the `size` is omitted, the number of bits for `number` defaults to the host machine word size. A plus or minus sign before the `size` specification signifies the number's polarity.

`base`

is always preceded by a single quote (`'`). The `base` can be one of the following: `d`(ecimal), `h`(exadecimal), `o`(ctal), or `b`(inary). The base identifier can be either upper- or lower-case.

`number`

The valid elements of `number` for each `base` are:

```
'b (binary): [01xXzZ_]
'd (decimal): [0123456789_]
'o (octal): [01234567xXzZ_]
'h (hexadecimal): [0123456789abcdefABCDEFxXzZ_]
```

The X and x represent unknown values, and Z and z represent high-impedance values in binary, octal, or hexadecimal form. Underscores are ignored.

If the most-significant specified digit of a number representation is an X or a Z, NTB-OV extends the X or Z to fill the higher-order bits or digits.

For example, 8'bx is equivalent to 8'bxxxxxxxx, and 8'bz00 is equivalent to 8'bzzzzzz00.

If not all the bits are specified and the highest specified bit is not X or Z, then NTB-OV zero fills.

Data Types and Variable Declarations

The NTB-OV standard data types are:

- integer
- reg
- string
- event

Note:

In NTB-OV, you cannot give a variable the same name as a function.

The NTB-OV user-defined data types are:

- Enumerated Types
- Virtual Ports
- Bit and Array Word Selects

You can declare all basic data types as class members and use them to form associative and non-associative arrays.

Standard Data Types

integer

Integers are variables that span the range of signed values from -2,147,483,648 (32'h8000_0000) through 0 to 2,147,483,647 (32'h7FFF_FFFF).

The integer data type is similar to a bit field with MSB = 31 and LSB = 0, except that arithmetic performed on integers is signed (unlike regs). An uninitialized integer has the value `32'bx`.

The syntax to declare an integer is:

```
integer variable_name [=initial_value];
```

`variable_name`

can be any valid identifier.

`initial_value`

is optional.

For expressions involving both unsigned (for example, reg) and integer types, NTB-OV first converts integer types to 32-bit unsigned integers.

reg

Use the `reg` data type for 4-valued, unsigned, single-bit, or bit vectors. You must specify the MSB and LSB, and the LSB must always be 0. The syntax for declaring a reg type variable is the same as Verilog 2001. You can optionally initialize variables of type reg in the declaration.

Any expression that is legal on the right hand-side of a reg assignment is valid as the expression for initializing a reg. Initialization has the same semantics as assignment. If no explicit initialization is done, the variable is automatically initialized to `n'bx` where `n` is the size in bits of the variable. The syntax to declare a `reg` is:

```
reg [[high:0]]variable_name [=initial_value];
```

`variable_name`

must be a valid identifier.

`initial_value`

is optional.

`high`

specifies the upper limit on the field. The high specifier must be a constant.

string

Use a variable of the string data type to store a C-like null terminated character string. You can assign a quoted string literal to a variable of type string. In Verilog 2001 and NTB-OV string literals behave like bit vectors (of a width that is a multiple of 8 bits).

You can manipulate strings using a wide range of string methods. In Verilog 2001, a quoted string assigned to a bit vector (reg) is truncated to the size of the bit vector. In NTB-OV, strings of arbitrary length can be assigned to a string type variable. The syntax to declare a string is:

```
string variable_name [=initial_value];
```

variable_name

must be a valid identifier.

initial_value

is optional.

For example:

```
string a;// a initialized to a special string null
string b = "Hi"; // b initialized to a string containing "Hi"
string c = b;// c initialized to a string containing "Hi"
// modifying characters in c does not affect b
```

Null versus Empty Strings

The special string `null` is different from the string obtained from `“”`, as shown in the following examples.

```
string a;
string b = "";
```

```
a == null is true
b == "" is true
a == "" is false
a == b is false
b == null is false
"" == null is false
```

Strings and String Literals in Assignment and Expressions

You can assign string type variables to string literals or any expression of type `string`. Note that NTB-OV (like Verilog) treats string literals as numeric constants consisting of the sequence of 8-bit ASCII codes of the characters in the literal. The string data type is thus different from a string literal, which is of type `reg`. However, in NTB-OV a string literal is implicitly converted to type `string` when it is assigned to a string type variable or used in an expression involving string type operands. String literals and concatenation/ replication of string literals are the only types of regs that can be assigned to a string type variable. Consider the string/variable assignments shown in Example 1-1, some of which are valid and some of which are not.

Example 1-1 Valid and Invalid String/Variable Assignments

```
reg [15:0] r;
integer i = 1;
string b;
string a = {"Hi", b};
r = a;   // valid
b = r;   // invalid!
b = "Hi";// valid
b = {5{"Hi"}};// valid
a = {i{"Hi"}};// valid
r = {i{"Hi"}};// invalid
a = {i{b}};// valid
a = {a,b};// valid
a = {"Hi",b};// valid
```

In NTB-OV, a string has different semantics than a char pointer in C, as shown in Example 1-2.

Example 1-2 String Assignment Semantics

```
string a, b;
a = "Hello World!";
b = a;
b.putc(5, "*");
printf("%s\n", a);
printf("%s\n", b);
```

Note that in Example 1-2 only `b` is modified; `a` is unmodified in spite of the change on `b`. This example prints:

```
Hello World!
Hello*World!
```

event

An event variable is a handle to a synchronization object. A synchronization object can be either ON or OFF. NTB-OV provides `sync()` and `trigger()` system tasks to control objects. The syntax to declare an event variable is:

```
event variable_name [=initial_value];
```

`variable_name`

can be a valid identifier.

`initial_value`

can be either null or another event variable. The default `initial_value` is a new synchronization object set to `OFF`.

User-Defined Data Types

NTB-OV supports the following kinds of user-defined types:

- Enumerated Types
- Virtual Ports

Enumerated Types

Enumerated types are user-defined lists of named integer constants. You can define them in the global name space or in a class. The same name, therefore, can be used in two different name spaces. For example, enum COLOR can be defined in a class and defined in the global name space (see Example 1-3 on page 34).

The width of an enumerated data type is the same as the width of an integer data type (32 bits). Enumerated type names, and the element names defined in them, have global scope, unless they are defined in a class. Different enumerated types cannot share the same element name. Elements within enumerated type definitions are numbered consecutively, starting from 0 (or, alternatively, the explicit value assigned to the first element). Each element can have an optional explicit value assigned to it. The explicit value assigned to an element affects values of subsequent elements that do not have an explicit assignment. That is, the subsequent element is set to the value of the previous element plus one. The syntax to declare an enumerated type is:

```
enum enum_type {list};
```

`enum_type`

is the name of the enumerated type. It is used to assign list values to variables.

`list`

is a list of category values separated by commas. They are assigned sequential integer values in the order listed.

The syntax to declare an enumerated variable is:

```
enum_type variable_name [=initial_value];
```

`enum_type`

is the category name of an enumerated type declaration.

`variable_name`

is any valid identifier.

`initial_value`

The enumerated type declaration lists the valid values. If this specification is omitted, the initial value of the declared variable is the first element in the enumerated type declaration.

Example 1-3 shows an enumerated type definition that assigns a unique number to each of the color identifiers, creating a new data type named `colors`.

Example 1-3 Enumerated Type Declaration

```
enum colors {red, green, blue, yellow, white, black};
colors new_color; // value of new_color is red.
new_color = green;
new_color = 1; // Invalid assignment.
```

Before new_color is initialized, its value is red. Next, the enumerated value green is assigned to the colors variable new_color. The second assignment is invalid because of the strict typing rules used for enumerated types.

Enumerated values specified in the list of an enumerated type declaration are assigned consecutive integers, starting from 0. In the above example, red is assigned 0, green is assigned 1, and so on. The enumerated values can be specified in several ways:

- name: an enumerated value (name), is defined and assigned the next consecutive integer after the previous enumerated value.
- name=N: this assigns the constant N to name.

This next example:

```
enum instructions {add=10, sub, div, mul=4, jmp};
```

creates the following enumerated values with corresponding integers for the enum_type instructions.

Table 1-4

Enumerated Value	**Integer**
add	10
sub	11
div	12
mul	4
jmp	5

As shown in the above example, any explicit constant integer assignment in an enumerated value specification changes all integer assignments for subsequent enumerated values.

NTB-OV issues a compilation error if the enumerated value specifications create a situation where the same integer is being assigned to two different enumerated values, as shown in this next example:

```
enum instructions {add=10, sub, jmp=11};
```

In this example, NTB-OV issues a compilation error because the enumerated values (`sub` and `jmp`) are both assigned the same integer (`11`).

An initial value of `X` or `Z` is not allowed in enumerated data type definitions. Also, different categories cannot share the same enumerated value. Consider this next example:

```
enum color {red, orange, green};
enum fruit {apple, orange, banana};
```

Here, NTB-OV issues a compilation error because the enumerated value `orange` is used in both declarations.

You cannot declare enumerated types inside tasks or functions.

For details on the semantics of assignments to enumerated types and the operators defined in them, see “Assignment Semantics” on page 82, “Compound Assignment” on page 85 and “Increment and Decrement Operators” on page 50. NTB-OV implicitly converts an enumerated type variable used in an expression to the integral value of the enum element.

Class Scope Enumerations

Enumerated types defined in a class are part of the class name space and not the global name space. The same name, therefore, can be used in the two different name scopes (for example, enum

COLOR can be defined in a class and also defined in the global scope). Here are some examples that show how to use enumerated types.

1. You can define enumerated types in a class as shown in the following example:

```
class Bus {
     enum TRAFFIC = PCI, AHB, USB;
     enum MS = MASTER, SLAVE = 5;
     enum RW = READ, WRITE;
     TRAFFIC traffic;
     task set_traffic(TRAFFIC inp);
     function TRAFFIC get_traffic();
     }

task Bus::set_traffic(Bus::TRAFFIC inp)
{    // accepts a class scope enumerated type
     traffic = inp;
}

function Bus::TRAFFIC Bus::get_traffic()
{ // returns a class scope enumerated type
     get_traffic = traffic;
}

program main {
     Bus bus_inst0;
     bus_inst0 = new();

     // Assigns object data to USB
     bus_inst0.traffic = Bus::USB;

     // Assign object data to PCI
     bus_inst0.set_traffic(Bus::PCI);

     printf(" bus_inst0.traffic = %0s \n",
     bus_inst0.get_traffic()); }
```

2. You can use enumerated types defined in a class to declare variables outside the class scope using the scope resolution operator(`::`).

```
program main {
     // enum defined in class Bus.
     Bus::TRAFFIC external_bus;

     // Assign PCI to variable external_bus.
     Bus bus_inst0;
     external_bus = Bus::PCI;
     // Assign V=USB to object data member
     bus_inst0 = new();
     bus_inst0.traffic = Bus::USB;
}
```

3. You can use enumerated types defined in a base class in a derived class.

```
class HyperBus extends Bus {
     // TRAFFIC is an enum defined in base class Bus.
     TRAFFIC internal_traffic;
}
```

4. You can override enumerated types defined in a base class using an enum defined in a derived class.

```
class HyperBus extends Bus {
// Overriding original enum definition RW
enum RW = MEMREAD, MEMWRITE;
}
```

5. You cannot reuse an enumerated value name as a variable name of a class member.

```
class HyperBus extends Bus {
enum RW = MEMREAD, MEMWRITE;
     TRAFFIC internal_traffic;
// Member variable MEMREAD conflicts with member of
// enum HyperBus::RW.
integer MEMREAD;
```

```
}
```

Virtual Ports

A virtual port is a user-defined data type that contains a set of port signal members grouped together under a given user-defined port name. You can pass virtual port variables to tasks and functions. The syntax is:

```
port virtual_port_name {
      port_signal_member_name1;
      ...
      port_signal_member_nameN;
}
```

For example:

```
port rcv_port {
      frame_n;
      valid_n;
      busy;
      packet;
}
```

This example creates a new data type named rcv_port which contains port signal members frame_n, valid_n, busy, and packet.

The syntax to create a port_variable_name is:

```
virtual_port_name port_variable_name;
```

`virtual_port_name`

is the name of the user-defined virtual port.

`port_variable_name`

is the port variable of type virtual_port_name.

For more on information on virtual ports, see “Virtual Ports” on page 161 .

To enable reuse, you can define tasks and functions that reference port signal members instead of specific interface signals.

You can create multiple port variables of the same virtual port type and assign the port signal members of each virtual port variable to different sets of interface signals. When a task or function is called and a virtual port variable is passed as an argument, NTB-OV performs operations on the set of interface signals assigned to the port signal members.

Class

The class user-defined data type is composed of data members of valid NTB-OV data types (known as properties) and tasks or functions (known as methods) for manipulating the data members. The properties and methods define the contents and capabilities of a class instance or object.

Class declarations are top-level constructs that have global scope. Class declarations cannot occur in tasks or functions, or nested in another class. Following is the class declaration syntax:

```
class class_name {
      [property_declaration]...
      [method_declaration]...
}
```

`class_name`

is the name of the class, and must be a valid identifier.

`property_declaration`

is any valid variable declaration of integer, reg, string, event, enumerated type, virtual port, array, or class.

`method_declaration`

is any valid task and function declaration.

Note:

Empty class declarations result in compilation errors.

Example 1-5 shows how to declare a class.

Example 1-5   Class Declaration

```
class Packet{
      reg [3:0] command; // property declarations
      reg [40:0] address;
      reg [4:0] master_id;
      integer time_requested;
      integer time_issued;
      integer status;
      Packet next;

      task new() // initialization {
            command = IDLE;
            address = 4'b0;
            master_id = 5'bx;
            next = null;
      }

      task clean() //method declaration {
            command = 0; address = 0; master_id = 5'bx;
      }

      task issue_request(integer delay){ //method declaration
            // send request to bus
      }

      function integer current_status(){ //method declaration
            current_status = status;
      }
}

program test {
      ....
```

```
        packet p = new();
        ...
    }
```

NTB-OV does not require complex memory allocation and deallocation, so you do not get memory leaks like you do in C. Construction of an object is straightforward, and garbage collection is implicit and automatic.

Casting

You can use the `cast_assign()` system function to assign values to variables that might not ordinarily be valid because of type checking rules. The syntax is:

`function integer cast_assign` `(`**`scalar`** *`dest_var`*`,` **`scalar`** *`source_exp`* `[,CHECK])`**`;`**

`dest_var`

is the variable to which the assignment is made. It can be any non-array scalar type (bit, integer, string, enumerated type, virtual port, event, or object handle).

`source_exp`

is the source expression that is assigned to the destination variable.

`CHECK`

The CHECK predefined macro is optional. When you call the `cast_assign()` system function without CHECK, the function assigns the source expression to the destination variable. If the assignment is illegal, a fatal runtime error occurs. When you call the `cast_assign()` system function with CHECK, the function makes the assignment and returns a 1 if the casting is successful. If the casting is unsuccessful, the function does not make the assignment and returns a 0. In this case, no runtime error occurs, and the destination variable is set to null, X, or uninitialized, depending on the data type.

Note:

The compiler only checks that the destination variable and source expression are scalars. Otherwise, NTB-OV does no type checking at compile time.

Example 1-6 shows a `cast_assign()` system function:

Example 1-6   Casting Expression to Enumerated Type

```
cast_assign(my_enum, 12*7);
```

This example assigns the expression to the enumerated type. Without `cast_assign()`, this assignment is illegal because of the strong typing of enumerated types.

Example 1-7 shows how to use `cast_assign()` to cast a base type into an extended type:

Example 1-7   Casting Base to Extended Type

```
// virtual class V function allocate_v returns the base type V
virtual class V {
      virtual function V allocate_v() ;
}
// function allocate_v() defined in class W returns an object
 // of the extended class W
class W extends V {
      reg i = 1 ;
      virtual function V allocate_v();
}
function V W :: allocate_v() {
      W w = new() ;
      allocate_v = w ;
}
```

```
// cast assign is used to cast the base type into the extended
 // type so integer i can be accessed in the extended class W

program test {
      W ww ;
      W tmp ;
      ww = new() ;
      cast_assign(tmp, ww.allocate_v()) ;
      tmp.i = 0 ;
}
```

Operators

The following sections explain NTB-OV operators:

- NTB-OV Operators
- Operator Precedence

NTB-OV Operators

All NTB-OV operators shown in Table 1-3 are also defined in Verilog and have the same semantics as described in the Verilog 2001 LRM.

Any extensions of these semantics for NTB-OV data types not in Verilog 2001 are described in the corresponding section for that data type in this document.

Table 1-3   NTB-OV Operators

Operator	Description	Semantics
[ ]	Select operator	Same as Verilog 2000. Both bit-select and part-select are supported in NTB-OV.
{}	RHS numeric concatenation	Same as Verilog 2001.
'{}	LHS numeric concatenation	Same as LHS {} in Verilog 2001.
{{}}	Numeric replication	Same as Verilog 2001.
{}	String concatenation	Not in Verilog 2001.
{{}}	String replication	Not in Verilog 2001.
+ - * /	Arithmetic	Same as Verilog 2001.
%	Modulus	Same as Verilog 2001.
++ --	Increment, decrement (pre or post)	Not in Verilog 2001. See section on "Increment and Decrement Operators" on page 50.
+= -= *= /= %= <<= >>= &= |= ^=	Compound assignment	See section on "Compound Assignment" on page 85.
=	Simple assignment	Same as Verilog 2001.
> >= < <=	Relational	Same as Verilog 2001.
! && || == !=	Logical operators	Same as Verilog 2001.
=== !==	Case equality/ inequality	Same as Verilog 2001.
><	Bit-reverse	See "Bit Reverse Operator" on page 52.
=?= !?=	Wild equality/ inequality	Not in Verilog 2001. See section on "Wild Equality and Inequality Operators" on page 50.
~	Bit-wise negation	Same as Verilog 2001.

Table 1-3 NTB-OV Operators (Continued)

Operator	Description	Semantics
& | ^ ^~	Bit-wise operators	Same as Verilog 2001.
& | ^ ~& ~| ~^	Reduction operators	Same as Verilog 2001.
<< >>	Logical shift	Same as Verilog 2001.
?:	Conditional	Same as Verilog 2001. See "Conditional (Ternary) Operator" on page 51.

Bit and Array Word Selects

Bit select and array word select operators have the same semantics in NTB-OV as in Verilog 2000. However, NTB-OV does not support selects into array words, as shown in the following example:

```
reg[20:0] arr[7];
...
arr[1][7]   // is not supported
arr[1][1:0]//is not supported
```

Part Selects

NTB-OV supports part selects with constant boundary specifications the same way as in Verilog, as shown in the following example:

```
vec[5:0] // is a valid reference
```

NTB-OV also provides limited support for variable width selects:

```
reg [31:0] vec;
integer i, j;
...
vec[i:j] //is a legal reference
```

The NTB-OV concept of a variable part select differs from the Verilog 2000 expression evaluation approach, and it is not a part of the 1364-2001 standard. In Verilog, all expression widths are statically known before simulation. There are three situations involving variable part selects that are processed differently by NTB-OV:

1. In situations where NTB-OV can statically determine the width of the variable part select, it performs the expression in the same way as in Verilog 2000, as shown in this next example:

```
      reg[5:0] vec;
      integer i, j;
      i = 0;
      vec = 5'b11111;
      printf("%b", vec[i+3 : i]);
// The width of the select is detectable
// statically - 4 bit. The printed result is 1111.
```

2. The width of the part select is not statically known. In this case the width of the select is statically assumed to be the maximal width of the whole selected vector. For self-determined expressions, this approach may produce an expression width wider than it would be when if the width was known statically:

```
      i = 0;
      j = 3;
      ...
      printf("%b", vec[j : i]);
// The width of the select is unknown statically.
// 001111 is printed
```

   Zero padding derives from the fact that the expression is statically assumed to be 6 bits (that is, the length of the whole vector). This width adjustment scheme produces correct results for most operators and assignments, as shown in the following example:

```
      reg [3:0] res;
      i = 0;
      j = 3;
...
```

```
res = vec[j:i] + 4'b0001;
```

Here, the `res` variable gets the correct binary `0000` even though addition is performed on the 6-bit entities.

3. NTB-OV does not support variable part-select in the following contexts:

   - As a member of concat/replicate operators. For example, in the following expressions:

     ```
     {m,inp[m:l],l}
     m + {8{inp[m:l]}} + l;
     ```

   - As an argument to reductional operators. For example, in following expressions:

     unary bit negation:

     ```
     res = ~inp[m:l]
     ```

     unary bit AND:

     ```
     res = &inp[m:l]
     ```

     unary bit NAND

     ```
     res = ~&inp[m:l]
     ```

     unary bit XOR

     ```
     res = ^inp[m:l]
     ```

     unary bit XNOR

     ```
     res = ~^inp[m:l]
     ```

When NTB-OV sees a variable part select in an unsupported context, it issues an error message and terminates the compile.

NTB-OV does not allow the use of variable part selects in the following contexts:

- at `var` ports of functions or tasks
- in `signal_connect()` to connect design and testbench ports
- in `@()` constructs
- variable part select based on memory/MDA words.

There two other areas where NTB-OV's processing of variable part selects differs from the constant boundary part selects semantics specified in the 1364-2001 standard.

1. Out of Bounds Selects

   NTB-OV terminates the simulation when any of the specified boundaries of the part select appear to be out of range or unknown at runtime. However, in Verilog, out-of-bounds bits are filled with X values for right-hand side references, and extra bits for the left-hand side out-of-bound assignments are ignored.

2. Wrong Way Selects

NTB-OV allows wrong way selects in variable range expressions; the tool normalizes them based on the declaration of the ascending or descending range specification. However, NTB-OV does not allow wrong way constant bounded part selects; the tool issues an error message and stops the compile when it sees them. For example, for declared variable `reg [31:0], r` reference `r[10:20]` is illegal and results in a compilation error. However, `r[i:j]` is legal in NTB-OV. If at runtime it appears that `i = 10` and `j = 20` (that is, `r[10:20]` is dynamically referenced) the tool swaps ranges and computes `r[20:10]` select instead.

Increment and Decrement Operators

NTB-OV supports increment and decrement operators similar to those used in C++ when they are used with integer, reg, and enumerated data types. The ordering of increment/decrement operations relative to any other operation within an expression is not guaranteed. Consider the following example:

```
i = 10;
j = i++ * i;
```

After execution of these two statements, the value of `j` can be either 100 or 110, since the order in which the operands of `*` are evaluated is not guaranteed.

Wild Equality and Inequality Operators

NTB-OV handles wild and inequality operators as shown in the following examples:

- `a =?= b` (`a` equals `b`, X and Z values are wildcards)
- `a !?= b` (`a` not equal to `b`, X and Z values are wildcards)

The wild equality (`=?=`) and inequality operators (`!?=`) treat an X or Z value in a given bit position (for bit values) as a wildcard. They match any bit value (0, 1, Z, or X) in the value of the expression being compared against it, as shown in the following example:

```
reg [3:0] r = 4'b10xz;
reg [3:0] s = 4'b1xz0;

if (r =?= s) {
     printf("r matches s\n");
}
```

Conditional (Ternary) Operator

In NTB-OV, as in Verilog, all three operands of the ternary operator are evaluated. The conditional operator follows this syntax:

```
conditional_expression ::=  expression1 ? expression 2:
expression3
```

If `expression1` evaluates to true (known value other than 0), then `expression2` is evaluated and used as the result. If `expression1` evaluates to false (0), then `expression3` is evaluated and used as the result. If `expression1` evaluates to an ambiguous value (X or Z), then both `expression2` and `expression3` are evaluated and their results are combined, bit by bit, using the `?:` truth table to calculate the final result. If the lengths of `expression2` and `expression3` differ, the shorter operand is lengthened to match the longer and zero filled from the left. Table 1-4 shows the results of unknown conditional statements.

Table 1-4 Results of Unknown Conditionals

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

When `expression2` and `expression3` are class pointers or strings, the bit-by-bit resolution does not apply to these types. For these types, if the condition contains X or Z bits, NTB-OV returns a null value, as shown in the following example:

```
MyClass c1, c2, c3;
integer cond;
...
c1 = cond ? c2 : c3;
// if cond is known non-zero value c = c2;
// if cond is zero, c = c3;
// if cond is unknown value, c = null
```

The case with string type operands works the same way.

Bit Reverse Operator

NTB-OV supports the bit-reverse operator for bit-vector, integer, and enumerated data types, as shown in the following example:

```
reg [3:0] s1;
reg [7:0] s2;
reg [15:0] res;
s1 = 1000;
s2 = 11001010;
res = s1 + (><s2);
```

With this example, the value of res is `0000000001011011`. NTB-OV reverses the operand to the `><` reversed and then depending on the context where it is used, either pads or truncates.

Corner Case Scenario

NTB and OpenVera behave differently in the following corner case scenario. In NTB, the behavior of `(~a_bit)*64` is different from `{~a_bit}*64`, but in OpenVera the behavior is the same. Note the use of parentheses around the expression in the first example and curly braces around the expression in the second example.

In NTB, when parentheses are used around this expression, `a_bit` is 1 bit wide and `64` is a 32 bits wide. Therefore, `a_bit` is sized to 32 bits (it is context-dependent). This context dependency determines the following results.

In NTB, when `a_bit` is 0:

```
(~a_bit)*64 => ~(32 0's)*64 => (~(32 0's)*64)
```

In NTB, when `a_bit` is 1:

```
(~a_bit)*64 => ~(32 1's)*64 => (~(32 1's)*64)
```

In NTB, when curly braces are used around the expression, the context width does not propagate into `a_bit`. That's because `{ }` is a self-determined operator. Therefore:

In NTB, when `a_bit` is 0:

```
{~a_bit}*64 => 1*64
```

In NTB, when `a_bit` is 1:

```
{~a_bit}*64 => 0*64
```

Operator Precedence

The precedence order of NTB-OV operators is defined in Table 1-5.

Table 1-5 NTB-OV Operator Precedence

<table>
<tr><td>!</td><td>~</td><td>++</td><td>--</td><td>&</td><td>~&</td><td>|</td><td>~|</td><td>^|</td><td>^</td><td>~^</td><td>unary</td></tr>
<tr><td>*</td><td>/</td><td>%</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>+</td><td>-</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td><<</td><td></td><td>>></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td><</td><td><=</td><td>></td><td>>=</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>==</td><td>!=</td><td>===</td><td>!==</td><td>=?=</td><td>!?=</td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>^</td><td>~^</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>|</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>&&</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>||</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>binary</td></tr>
<tr><td>?:</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>ternary</td></tr>
<tr><td>=</td><td>+=</td><td>*=</td><td>/=</td><td>%=</td><td>&=</td><td>|=</td><td>^=</td><td><<=</td><td>>>=</td><td></td><td>binary</td></tr>
</table>

All binary operators with the same precedence associate from left to right. Unary and ternary operators associate from right to left. Therefore, in the following examples:

```
1.  a ? b: c ? d: e;
2.  (a ? b: c) ? d: e;
3.  a ? b: (c ? d: e);
```

lines (`1`) and (`3`) are equivalent, but lines (`1`) and (`2`) are not.

If you use multiple operators with the same precedence (as in `A + B*C`), NTB-OV evaluates the expression from left to right (for example, `B*C`, then `+A`). When operators differ in precedence, NTB-OV executes the highest precedence operator first. Parentheses change the operator precedence.

NTB-OV provides a set of basic operators that you can use to manipulate combinations of string variables and string constants. Table 1-6 lists the valid operators..

Table 1-6 NTB-OV String Operators

Operator	Meaning
`Str1 == Str2`	Checks the equality of the two strings. If they are equal, the result is 1; otherwise, the result is 0. Both strings may be of type string, or one of them may be a string literal. If both are string literals, the expression is equivalent to the `==` operator for numeric types.
`Str1 != Str2`	Logical negation of `==`.
`{Str1, Str2 ...,Strn}`	Each Str_i may be of type string or may be a string literal (it is implicitly converted to string). If at least one `Stri` is of type string, then the expression evaluates to the concatenated string and is of type string. Note that if all the `Stri` are string literals, the expression behaves like a Verilog concatenation for numeric types.
`{multiplier {Str}}`	`Str` may be of type string or a string literal. `multiplier` is a numeric type and must be a constant. If `Str` is a literal, the expression behaves like numeric replication in Verilog.
`Str.method(...)`	The dot (.) operator is used to invoke a specified method on strings. See the “String Methods” section for descriptions of various methods.

Note:

You can compare string variables to null.

Arrays

NTB-OV supports the following kinds of arrays:

- Fixed-size Arrays
- Associative Arrays
- Dynamic Arrays

Fixed-size Arrays

Fixed-size arrays are memory efficient and provide fast data access because the access time is constant regardless of the number of elements in an array. The size of fixed-size arrays is set at compile time. Fixed-size arrays can have one or more dimensions.

Single Dimensional Arrays

The syntax to declare a single dimensional array is:

```
data_type array_name[size1];
```

`data_type`

is the data type of the array elements. Supported data types are integer, reg, string, event, enumerated type, virtual port, and class object.

`array_name`

is the name of the array and must be a valid identifier.

`size`

specifies the number of elements in the array. The maximum is 2^{31}-1 elements.

The following example shows a single-dimensional array declaration:

```
integer array_ex[5];
```

Accessing an array with an unknown bit (X) in the index causes a simulation error. Also, writing to an array with an unknown in the index is ignored, and reading with an unknown in the index returns Xs.

You cannot reference a bit field of an array element directly. To reference a bit field of an array element, use a temporary variable, as shown in the following example:

```
tmp = memory[42];
if (tmp[3:2] == 0) ...
```

Initializing Arrays

You can initialize an array of integer, reg, enum, or string when you declare it. The values used for array initialization are subject to the same limitations as the initialization of scalar variables. For example:

```
integer array[5] = {0, 1, 2, 3, 4};
```

Concatenation is not supported in array initialization. An attempt to concatenate results in a compilation error. For example:

```
// illegal declaration
#define OPCODE 8'ha
reg [16:0] array1[3] = { {OPCODE, 8'h00}, {OPCODE, 8'h01},
{OPCODE, 8'h02}};
```

You must specify the size of the array. The following is not supported:

```
integer numbers[]={1, 2, 3};
```

Multidimensional Arrays

The syntax to declare a multidimensional array is:

```
data_type array_name[size1]...[sizeN];
```

`data_type`

of the array elements.

`array_name`

name of the array.

`size`

specifies the number of elements in N^{th} dimension of the array.

For example:

```
integer matrix [2][5];
Color colors [3][4][2];
event myevent [2][2];
```

Reading an array with an index that contains unknown values (X or Z) returns the default value of the array element data type (for example, null for classes and strings; X for regs and integers).

Note that you cannot reference a part select of an array element directly. To reference a part select of an array element, use a temporary variable, as shown in the following example:

```
tmp = memory[42];
if (tmp[3:2] == 0) ...
```

You specify the size of each dimension as shown below:

```
integer a[10][20][30];
// three dimensional array of integers
reg [7:0] b[10][20];
// two dimensional array of 8-bit fields
```

The index of each dimension runs from 0 to *n*-1, where *n* is the size of the dimension specified in the declaration.

You cannot follow array index expressions with a part-select. That is, for the two-dimensional array `b` in the above example, `b[5][5]` is valid, whereas `b[5][5][0]` and `b[5][5][3:0]` are invalid. It is invalid to not specify all dimensions of the array. That is, for the three dimensional array `a` in the above example, `a[5][5]` is not a valid expression in NTB-OV. Example 1-8 illustrates the use of a three-dimensional array.

Example 1-8   Three-Dimensional Array Program

```
task cube_add(integer cube[2][2][2], integer offset) {
      integer i, j, k;
      for (i=0; i<2; ++i){
             for (j=0; j<2; ++j){
                  for (k=0; k<2; ++k){
                        cube[i][j][k] += offset;
                  }
                        }
      }
}

program array {
      integer cube[2][2][2], i, j, k;

      for (i = 0; i < 2; ++i) {
            for (j = 0; j < 2; ++j) {
                  for (k = 0; k < 2; ++k) {
                        cube[i][j][k] = i+j+k;
                  }
            }
      }

cube_add(cube,4);
```

```
            for (i = 0; i < 2; ++i){
                for (j = 0; j < 2; ++j) {
                    for (k = 0; k < 2; ++k) {
                            printf("cube[%d][%d][%d] = %d\n", i, j, k,
                                cube[i][j][k]);
                        }
                    }
                }
}
```

The program shown in Example 1-8 produces the following results:

```
Cube Contents
cube [0][0][0] = 0
cube [0][0][1] = 1
cube [0][1][0] = 1
cube [0][1][1] = 2
cube [1][0][0] = 1
cube [1][0][1] = 2
cube [1][1][0] = 2
cube [1][1][1] = 3

Cube Contents after adding 4
cube[0][0][0] = 4
cube[0][0][1] = 5
cube[0][1][0] = 5
cube[0][1][1] = 6
cube[1][0][0] = 5
cube[1][0][1] = 6
cube[1][1][0] = 6
cube[1][1][1] = 7
```

When referencing elements in a multidimensional array, you must specify multiple indices as follows:

```
vname[index_1]...[index_n]
```

For example:

```
task data_buf (reg[7:0] mem_data[4][4]) {
...
}

program test {
    reg[7:0] mem_bank[4][4];
    ...
    data_buf (mem_bank);
```

```
        ...
}
```

In this example, the `data_buf()` task is declared with a two-dimensional array argument with array sizes of four. When the `data_buf()` task is called, the variable passed as the argument (`mem_bank`) must match the dimension and array sizes of the declaration; otherwise you get a compilation error, as shown in this next example:

```
// compilation error, size must be specified
integer mem_bank[][2];
reg data_bank[][];

// compilation error, bit slicing used
reg[7:0] matrix[3][6];
reg[7:0] temp_array[6];
...
matrix[0] = temp_array[5];  // bit slicing
```

You can pass multidimensional arrays as arguments to a task or function call, but they must be of the same data type and size as the arguments declared in the task or function. For example, the declaration:

```
task fun(integer x[2][2])
```

creates a task `fun` that takes one argument, a two-dimensional array where each dimension is of size two. Any call to `fun` must pass in a two-dimensional array where each dimension is of size two.

Initializing Multidimensional Arrays

You can initialize a multidimensional array of integer, reg, bit, enum, or string in the declaration. The values used for array initialization are subject to the same limitations as the initialization of scalar variables. The order of elements being initialized is identical to C and C++. For example:

```
integer x[2][2]={{0,1},{2,3}};
```

The initial values of the array elements is this example are:

```
x[0][0] = 0
x[0][1] = 1
x[1][0] = 2
x[1][1] = 3:
```

NTB-OV does not support concatenation in array initialization. An attempt to concatenate results in a compilation error, as shown in this next example:

```
// illegal declaration, compilation error issued
#define OFFSET 4'f
reg[7:0] mem_bank[2][2] = { {OFFSET, 4'h0}, {OFFSET,4'h1},
{OFFSET,4'h2},{OFFSET,4'h3} };
```

Associative Arrays

Associative arrays are arrays whose dimensions are not specified in the declaration. Associative arrays offer more flexibility than fixed-size arrays in that they can be sparse, allowing for the modeling of memories. The syntax to declare an associative array is:

```
data_type array_name[];
```

Associative arrays can have only one dimension. The following examples result in compilation errors:

```
integer assoc_matrix[][2]; //Invalid
integer double_assoc_matrix[][]; //Invalid
```

An element in associative array `A` at index `k` is created when `A[k]` is assigned a value. The index of an associative array cannot more than 64-bits wide. If no value is assigned to an associative array at a given index, reading the array element at that index returns the same value a variable of the same type that is not explicitly initialized would return (see Table 1-7).

Table 1-7 Values of Variables that are not Explicitly Initialized

Type	**Implicitly Initialized Value**
reg	1'bx
reg[n:0]	n'bx
integer	32'bx
event	event in the OFF state
class	null
string	null
enum	first member in the enum declaration

Array elements in associative arrays are allocated dynamically when you access a particular element. The array index tracks the elements that have been assigned values and stores those values within the array. The index is an unsigned number with a maximum value of 2^64-2. When using integer and bit associative arrays, if you try to access an element that has not been assigned a value, an X is returned.

Note:

> Using associative arrays slows down simulation time slightly. The effect is usually not noticeable. However, with large arrays, the effect can be significant. You may be able to use dynamic arrays instead.

You can use the `assoc_index()` system function to manipulate and analyze associative arrays. See "assoc_index ()" on page 345 for the syntax, description, and examples.

Dynamic Arrays

Dynamic arrays combine the flexibility of associative arrays by giving you the ability to define the size at runtime, with the fast access and low memory usage of fixed-size arrays. Like associative arrays, dynamic arrays do not support multiple dimensions. The syntax to declare a dynamic array is:

```
data_type array_name[*];
```

`data_type`

of the array elements. Dynamic arrays support the same types as fixed-size arrays.

For example:

```
// Dynamic array of integers
integer mem[*];

// Dynamic array of 4-bit vectors
reg[3:0] nibble[*];
```

To set or change the size of the array during runtime, use the `new[]` operator. To get the current size of the array, use the `size()` function.

The new [ ] Operator

The syntax for the `new[]` operator is:

```
array_name = new[size] [(src_array)];
```

`size`

is the number of elements in the array. It can be any non-negative integral expression.

`src_array`

is the name of the array to be copied into `array_name`. If `src_array` is not specified, `array_name` is initialized with a default value, with the value depending on its data type. The *src_array* must also be a dynamic array of the same data type as *array_name*, but it does not have to be the same size. If `src_array` is smaller than `array_name`, *array_name*'s extra elements are left with their default values. If `src_array` is bigger than `array_name`, `src_array`'s extra elements are ignored.

For example:

```
// Declare 2 arrays.
integer packetA[*], packetB[*];
packetA = new[100]; // Create and load packetA.
...
// Create packetB as a copy of packetA.
packetB = new[100] (packetA);
```

This parameter is usually used to change the size of an array. In this situation, `src_array` is `array_name`, so the previous values of the array elements are preserved, as shown in the following example:

```
integer packet_size[*]; // Declare the dynamic array.
packet_size = new[100]; // Initialize the array.
...
// Resize the array with no source. The elements are
// initialized and the previous contents lost.
packet_size = new[150];
...
// Resize the array while preserving the values.
packet_size = new[200] (packet_size);
```

The size () Function

The `size()` function returns the current size of a dynamic array. The syntax is:

```
function integer array_name.size();
```

The delete () Task

The `delete()` task deletes all existing elements from a dynamic array, making its size zero. The handle of the array still exists. The syntax is:

```
task array_name.delete();
```

For example:

For example:

Note:

Because a pass by value involves the overhead of a full array copy, it is preferable to pass arrays by reference wherever possible for improved performance.

2

NTB-OV Programming Overview

This chapter explains the basic OpenVera Native Testbench (NTB-OV) programming elements, including the fundamental program structure used in all NTB-OV programs. This information is presented in the following major sections:

- Overview
- Tasks and Functions
- External Subroutine Declarations

Overview

An NTB-OV program consists of the following components:

- Program definition

An NTB-OV testbench must contain exactly one program. Testbench execution starts in this program.

- Class declarations

  Classes are the primary user-defined data types in NTB-OV. They provide a way to encapsulate data and the methods that operate on that data.

- Global task and function declarations

  Global task and function declarations are global methods defined outside the scope of any class and the program.

- Enumerated type declarations

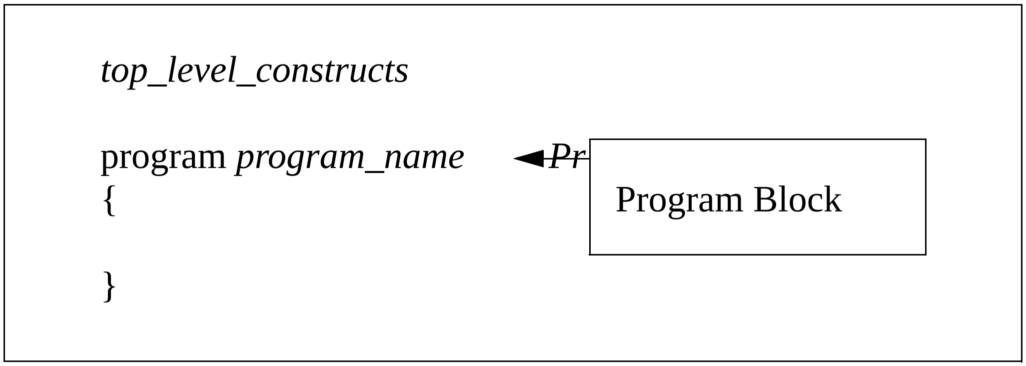
  Enumerated type declarations are user-defined types that represent a finite set of constant integer values, each of which is assigned a unique name and can be referenced using that name. Variables of an enumerated type are strongly typed.

- Interface definitions

  Interface definitions provide the primary mechanism for communicating with the device under test. You instantiate a program with these interface signals connected to appropriate signals in the Verilog design.

An NTB-OV program involves the integration of several key components of a testbench, including a required program and optional top-level constructs, as shown in Figure 2-1.

Figure 2-1 NTB-OV Program Overview



Program Block

You start a program block using the `program` keyword, and add a program name, variable declarations, and code, as shown in the following example:

```
program program_name {
      global variable declarations
      program block code
}
```

The testbench starts execution in the program block. NTB-OV variables that you declare in the program block are global, whereas variables you define in tasks or functions have local scopes.

Note:

> The scope of variables declared in statement blocks is limited to the block in which they are declared. Also, only one program is currently supported in NTB-OV.

Top-level Constructs

You can have any number of the following top-level constructs:

- enumerated type definitions
- class definitions
- out-of-block class method definitions
- global task and function definitions
- Verilog task and function prototypes
- interface declarations

Scope Rules

The following NTB-OV language elements are defined in a global scope and share a global name space. Therefore, no two of them can share the same names:

- Classes
- Global tasks and functions
- Enumerated types
- Elements in each enumerated type
- Interfaces

An NTB-OV program *does not* create a new scope. Therefore, variables declared inside the program block are accessible everywhere.

The following NTB-OV language elements create a new name space:

- Class definition
- Task or function definition (the arguments of the task or function share a name space as the outermost block in the task or function body)
- Statement blocks
- Interfaces

Note:

No two elements defined inside a class definition (for example) can have the same name.

Tasks and Functions

You cannot nest NTB-OV tasks and functions; they are re-entrant. Each task or function call receives a copy of all local, non-static variables. Recursive calls to NTB-OV tasks and functions are allowed, and you can pass in arguments by value or reference.

Note:

It is illegal to call an NTB function from Verilog.

Functions

NTB-OV functions can have zero or more formal arguments and one return value. The syntax is:

```
function data_type function_name (data_type argument_list){statements;}
```

`data_type`

can be any valid NTB-OV built-in or user-defined data type. The returned value has the same data type as the declared function. Functions cannot return arrays.

`function_name`

is the name by which the function is called throughout the program.

`argument_list`

an argument is a variable, including the data type, that is passed to the function when the function is called. You can pass all data types, including interface signals. Array arguments can be regular or associative, as well as `var` (that is, array arguments can be passed by reference). The type and dimension of the array in the call must match the type and dimension of the array in the function declaration. Separate multiple arguments using commas.

`statements`

can be any statement, including function calls and variable assignments

Because functions return a value, they must appear in assignment statements. For example:

```
i = func();
```

But the following is illegal:

```
func();
```

To discard a return value, use the `void` keyword, as shown in the following example:

```
void = myfunc(a, b, c);
```

Return Statement

A return statement causes a function to return immediately to its caller. A return statement is shown in the following example:

```
function integer foo(...) {
     ...
     foo = value;
     return;
}
```

If program execution reaches the end of a function definition, NTB-OV executes an implicit return. If the function’s return value variable is not assigned, NTB-OV returns the default value of the function's return type (see table Table 1-7 for return values).

Tasks

Tasks in NTB-OV can have zero or more arguments. Calls to tasks must not appear in expressions. In other words, a task can only be invoked as a statement. The syntax is:

`task` *`task_name (type formal_argument_list)`*`{`*`statements;`*`}`

`task_name`

is the name by which the task is called throughout the program.

`formal_argument_list`

is a variable, including the data type, that is passed to the task when the task is called. All data types can be passed. Separate multiple arguments using commas.

`statements`

can be any statement.

The following example shows an NTB-OV task declaration:

```
task handshake_port0(reg direction, reg [7:0]
data1, reg[7:0] data2) {

         @0,1000 intf1.req == 1'b1;
         intf1.ack = 1'b1;
         @1 intf1.ack = 1'b0;

         if(direction) port0.data = data1;
         else port0.data = data2;
         }
```

The syntax to invoke a task is:

```
         task_name(argument_list);
```

as shown in the following example:

```
         print_data(new_data);
```

An NTB-OV task can contain statements that suspend execution of the task. Array arguments are strongly typed. The type and dimension of the array in the call must match the type and dimension of the array in the task declaration.

Return Statement

A return statement causes a task to return immediately to its caller, as shown in the following example:

```
task foo(...) {
     ...
     moo = value;
     return;
}
```

If the execution of a task falls through to its end, NTB-OV executes an implicit return.

Task or Function Arguments

You can pass a task or function's arguments by value or by reference. Pass by value is the default method. This way, if the arguments are changed in the body of task or function, the changes do not affect the caller.

To pass an argument by reference, the declaration of the argument in the task or function prototype must be preceded by the keyword `var`. Any changes take effect immediately and are reflected in the actual argument. If the argument is passed by reference, the actual argument must be a valid l-value, but cannot be a part- or bit-select, or a concatenation. Note that an l-value is an expression referring to an object (see "Assignment Semantics" on page 82 for list of forms an l-value can take). The types of the actual and formal arguments for task and function calls must match *`exactly`*.

Default Arguments

You can optionally specify default values for subroutine arguments. These values can be any expressions involving constants and global program variables. When the subroutine is called, you can instruct the compiler to use the default value for any argument by specifying an asterisk (`*`) at that argument position, as shown in the following example. You can omit a trailing list of asterisks.

```
task foo(integer a = 0, integer b = 1, integer c = 2) {
...
}
...
foo();// same as foo(0,1,2);
foo(*,4);// same as foo(0,4,2);
foo(4,*,5);// same as foo(4,1,5);
foo(5);// same as foo(5,1,2);
```

Static Variables

By default, variables are local to the function or task that uses them. They are allocated when the function or task is called. If you want a to share a variable across all invocations of a function or task, use the `static` declaration. You can use static variables to create recursive functions as an application. The syntax to declare a `static` variable is:

```
static data_type variable_name;
```

You can declare any data type as a `static` variable. You can initialize a static variable in the declaration, but the initial value expression must be a constant or an expression containing other static/global variables or global function calls. Note that the ordering of initialization of static variables in different scopes is undefined. Therefore, it is a good idea to initialize static variables to only constant expressions or expressions containing static variables (previously declared and initialized) within the same scope. You can also initialize a static variable using a function call that does not have side effects.

Note:

In the case of concurrent accesses, there may be races if multiple threads assign to the same variable. You cannot declare static variables in a global context.

External Subroutine Declarations

External subroutine declarations enable the use of multiple source files (that is, functions and tasks can be declared in separate source files). This way, you can compile large functions and tasks separately, which facilitates debugging.

Declaring External Subroutines

Subroutines can only be declared as external at the top level. The syntax is:

```
extern task | function subroutine (argument_list);
```

Note:

When using external subroutines, the argument types you pass must match exactly.

External Default Arguments

You can set default values locally, and independently, for each compilation unit using extern declarations. This way, you can customize a general library for a particular user or testbench and implement it using include files with different defaults.

For example, you can define the `write()` task in a separate library and compile it independently. The NTB-OV file in which the `write()` is used must declare `write()` as external. You can set default values in this extern declaration, as shown in the following example:

File A (library)

```
task write (integer i, k, reg[5:0] data) {
     // write definition
}
```

File B (testbench)

```
extern task write(integer i = 10,integer k,reg[5:0]
     data=6'b1);

task xyz () {
     write (*, 5);
     //continue task declaration
}
```

3

NTB-OV Statements, Assignments, and Control

This chapter explains the syntax and semantics of OpenVera Native Testbench (NTB-OV) statements, assignments, and sequential control, in the following major sections:

- Statements and Statement Blocks
- Simple Assignment
- Sequential Control

Statements and Statement Blocks

An NTB-OV statement always includes a terminating semicolon, as shown in the following examples:

```
integer i;
```

```
printf("Local data = %h\n", data);
```

You create a statement block using curly braces. A block groups a sequence of variable declarations and statements. The declared variables are visible to the statements declared within the braces. The syntax for statement blocks is:

```
{
     // variable_declarations
     // NTB-OV_statements
}
```

`NTB-OV_statements`

can be, for example, assignment, if-else, case, and expect statements.

To create an empty statement, use a { } block:

```
if(1)
 {}
else
 {}
```

The following is not legal in NTB-OV, and generates a parse error.

```
if(1)
 ;
else
 ;
```

Simple Assignment

In NTB-OV, the primitive operation to change the value of a variable is an assignment. Assignments are allowed in program, task, and function scopes. A variable declaration can optionally contain an assignment to initialize the variable.

Variable Initialization

The syntax for variable initialization is:

```
type identifier = expression;
```

Declare variables before any executable statements inside a program, task, function, or block statement. Initializations are equivalent to inserting (in source order of declarations) assignments to variables before the first executable statement in the program, task, function, or block. For example:

```
{
    integer a = 10;
    integer b = a;
    integer c;
    c = 100;
}
```

is equivalent to:

```
{
    integer a;
    integer b;
    integer c;
    a = 10;
    b = a;
    c = 100;
```

```
}
```

Assignment Semantics

The right-hand side of an assignment can be any expression that evaluates to an atomic type value. Such expressions are known as l-value expressions. The left-hand side of an assignment must be an expression of an atomic type (reg, integer, string, event, enumerated type, or class reference type) or the keyword `void`.

An l-value expression is one that refers to an object. It can take the following forms:

- an atomic type variable name
- an array index expression that evaluates to an atomic type
- a bit- or part-select of reg type variable
- a class reference that evaluates to an atomic type
- a concatenation of reg/integer type variables, reg /integer type array index expression, bit- or part-selects of regs, or reg/integer type class member access sequence
- the keyword `void`

If the left-hand side of an assignment is `void`, NTB-OV evaluates the right-hand side expression and discards the value.

The left- and right-hand sides of assignments must have compatible data types (see Table 3-1).

Table 3-1   NTB-OV Assignment of Type Compatibility

LHS Type	Compatible RHS Type
integer, reg	integer, reg, enum, string
enum	Same enum type as LHS
class reference	Same class type as LHS
string	string, string-literal
event	event

NTB-OV assignments work like simple procedural assignments in Verilog with no delay or event control. Note the invalid assignment examples in Example 3-1.

Example 3-1   NTB-OV Assignment Examples

```
integer i;
event e;
reg rs;
reg[7:0] rb;
integer ai[2];
integer mai[2][2];
reg [7:0] arb[2];
integer hi[];
enum Fruit { Apple,   Orange,    Banana }
class moo  {
      string s;
}
class boo  {
      moo m[];
}
class foo  {
      integer i;
      boo b;
      moo m[10];
}
moo cm = new;
boo cb = new;
foo cf = new;
Fruit ft;
```

```
ft = Apple;// valid
i = 10;// valid
e = null;// valid
rs = 1'bx;// valid
rb = 8'hff; // valid
rb[3:0] = 4'hf;// valid
rb[i:j] = 4'hf //valid
ai[0] = 10;// valid
mai[0][1] = 100;// valid
hi[10] = 100;// valid
cm.s = "Hello";// valid
cf.m[9] = cm;// valid
cf.m[9].s = "Hi";// valid
cf.b = cb;// valid
cb.m[10] = cm;// valid
'{i, rb } = 40'bx; // valid
'{ i, ft } = { 10, Apple }; // invalid -- LHS concat has enum
'{ s } = { "Hello" }; // invalid - LHS concat has string
'{ cf } = new; // invalid - LHS concat has class reference
'{ cf.i } = 10; // valid
ai = { 100, 1000 }; // invalid - LHS is non-atomic
arb[7][0] = 1'b0; // invalid - bit-sel of array expression
arb[7][3:0] = 4'b0; // invalid - part-sel of array expression
```

Variable Part-Selects

NTB-OV supports variable part-selects on the RHS and LHS of assignments. The following example shows how to use variable part-selects on the RHS of an assignment:

```
bit[31:0] inp;
bit[7:0] sel;
integer m,l;
...
m = 15;
l = 8:
inp = 32'h89abcdef;
sel = inp[m:l];
```

You can also use variable part-selects on the LHS of an assignment, condition expression of if/case statement, member of another expression, or an argument to user-defined functions or tasks.

Compound Assignment

NTB-OV supports compound assignments of the form:

```
LHS bop = RHS;
```

where `bop` is a binary operator. For the list of all valid compound assignment operators, see Table 1-11. The LHS of a compound assignment:

- *must* be of type integer, reg, or enumerated
- *must* be a valid left-hand side for an assignment
- *cannot* be a concatenation

The LHS can be a numeric type variable name, class member access, or array index expression. If the LHS of a compound assignment contains side effects, the results are undefined. For example, in the following example `func()` may contain side effects that affect the RHS:

```
[func()] += rhs;
```

For enumerated data types, the only valid compound assignments are `+=` and `-=` (see Table 3-2.)

Table 3-2 Semantics of += and -=

Operation	Semantics
enum_var += *numeric_expression*	If *numeric_expression* evaluates to val, assign to *enum_var* the valth member in definition order from the current value of *enum_var*. A wrap to the beginning of the enum list occurs if the end of the list is encountered before the valth member.
enum_var -= *numeric_expression*	If *numeric_expression* evaluates to val, assign to *enum_var* the valth member in reverse definition order from the current value of *enum_var*. A wrap to the end of enum list occurs if the beginning of the list is encountered before the valth member.

Sequential Control

This section explains the syntax and semantics of the NTB-OV constructs you can use for sequential flow control in the following subsections:

if-else Statements

The `if-else` statement is the general form of selection statement. The syntax is:

```
if (expression) {
if_statement1;
if_statement2;
if_statementN;
}
[else {
else_statement1;
else_statement2;
else_statementN;
}
```

`expression`

can be any expression that evaluates to a numeric value.

`if_statement`(s) and `else_statement`(s)

are sets of one or more statements.

If the *`expression`* evaluates to true, *`if_statement(s)`* is executed. If it evaluates to false, *`else_statement(s)`* is executed.

If *`else_statement(s)`* is omitted, NTB-OV evaluates the condition and executes the *`if_statement(s)`* only if the condition evaluated to true. Otherwise, program execution continues with the first line after the *`if_statement(s)`*. Consider the following example:

```
program if_else_nesting {
      integer i;
      i = value;

      if (i < 10){
            printf("Warning.\n");
            if (i <4)
                  printf("Action required.\n");
            else
                  printf("No action required\n");
      }
      else {
            printf("Recheck i later.\n");
      }
}
```

When the value for `i` is `3`, NTB-OV executes the nested else statement and generates the following output:

```
Warning.
Action required.
```

When the value for `i` is `11` because the *`if_expression`* is false, NTB-OV executes the final else statement and generates the following output:

```
Recheck i later.
```

NTB-OV does not allow assignments within the condition. The assignment `c=1` is not an expression. You must use `c==1`.

case Statements

You can use `case` statements for multi-way control branching. The syntax is:

```
case (primary_expression) {

case1_expression : statement
case2_expression : statement
...
caseN_expression : statement
[default : statement]
}
```

`primary_expression`

is any expression that evaluates to a numeric value.

`case_expression`

can be one or more numeric expressions separated by commas Expressions separated by commas allow multiple expressions to share the same statement block.

`statement`

can be any valid statement or block of statements. If you use a code block, NTB-OV executes the entire block.

A case statement must have at least one case item aside from the default case, which is optional. The default case must be the last item in a case statement.

NTB-OV first evaluates the `primary_expression` and then successively compares that value against each `case_expression` using the `===` operator. When an exact match is found, NTB-OV executes the statement corresponding to the matching case and control passes to the first line of code after the case block. If other matches exist, they are not executed. Consider the following example:

Example 3-2 Case Block Example

```
program p1 {

reg [3:0] bus;
enum Packet {packet_null, READ, WRITE, UNKNOWN};
Packet packet;
bus = 4'b1001;

case ( bus[3:0] ){
4'b00ZZ: packet = packet_null;
4'b0001, 4'b1001: packet = READ;
// two expressions share the same statement
4'b0010, 4'b1010: packet = WRITE; // ditto
4'b00XX: packet = UNKNOWN;
default: printf("Error: illegal packet %h detected\n", bus[3:0]);
  }
}
```

To use X or Z as don't cares, use the `casex` or `casez` statements, respectively. When you use `casex`, NTB-OV treats X and Z bits in both the *`primary_expression`* and *`case_expressions`* as don't cares. When you use `casez`, only the Z bits in the *`primary_expression`* and *`case_expressions`* are treated as don't cares. If no match is in a case statement found, NTB-OV executes the `default` statement.

repeat Loops

The `repeat` loop executes a statement a fixed number of times. The syntax is:

```
repeat (expression) statement;
```

`expression`

can be any valid numeric expression.

`statement`

can be any valid statement or block of statements. If a code block is used, NTB-OV executes the entire block.

You can use repeat statements to repeat any statement a fixed number of times. NTB-OV evaluates the value of the expression before starting the repetitions. Changing a variable within the expression does not change the number of loops that are executed. Repeat statements are often used to implement a wait or pause in the simulation. For example:

```
repeat (10) @(posedge CLOCK);
```

pauses the simulation for `10` clock cycles.

for Loops

The syntax to declare a `for` loop is:

```
for ([initial];[condition];[increment]) statement;
```

`initial`

is zero or more assignment statements separated by commas.

`condition`

is any numeric expression. If the condition is omitted, NTB-OV assumes the value `1`.

`increment`

is zero or more assignment statements separated by commas.

`statement`

is any valid statement or block of statements. If a code block is used, the entire block is executed.

NTB-OV first executes the `for` loop `initial` section and then evaluates the `condition`. If the `condition` is true (a known, non-zero numeric value), NTB-OV executes the statement once and executes the increment section. NTB-OV then checks the `condition` again and repeatedly executes the loop statement and the `increment` until the `condition` evaluates to false. An exception to the above is a break statement (see break and continue Statements) or a return statement inside the loop statement, which causes the loop to end before the `condition` is false. When the `condition` is false and the loop finishes, NTB-OV continues execution from the statement after the loop. Example 3-3 shows some `for` loop examples:

Example 3-3 For Loop Examples

```
for (i = 0; i < 100; i++) {
      for (j = i; ; j++){
            if (j >= 100) break;
            if (a[j] < a[i]) {
                  temp = a[i];
                  a[i] = a[j];
                  a[j] = temp;
            }
      }
}
```

while Loops

The syntax to declare a while loop is:

```
while (condition) {
statement;
}
```

condition

can be any numeric expression.

statement

can be any valid statement or block of statements. If a code block is used, NTB-OV executes the entire block.

The loop iterates as long as the condition is true. When the condition is false, control passes to the first line of code after the loop. NTB-OV checks the condition at the top of each loop. Note that NTB-OV does not allow assignments within the condition. The assignment c=1 is not an expression. You must use c==1. Example 3-4 shows an example of a while loop:

Example 3-4 While Loop

```
op = 0;
while (op<5) {
      printf("Operator is %d\n", ++op);
}
```

In this example, this loop continues until op equals 5. Each time through the loop, op increases by 1. After five passes through the loop, the loop ends, and control passes to the first line of code after the loop. If the condition is a non-zero constant, the loop becomes infinite. You can only break Infinite loops using the break or return statements.

do-while

The syntax to declare a do-while loop is:

```
do {
statement;
} while (condition);
```

`statement`

can be any valid statement or block of statements. If a code block is used, NTB-OV executes the entire block.

`condition`

can be any expression that evaluates to a boolean value. NTB-OV evaluates the `condition` after the `statement` is executed. You can use `break` or `continue` to break out of or continue the execution of the loop. Example 3-5 shows an example of a `do-while` loop.

Example 3-5 do-while Loop

```
program test {
      integer i = 0;
      do {
           printf("i = %0d \n", i);
           i++;
      } while (i < 10);
}
```

foreach

NTB-OV supports `foreach` loops for single-dimensional, fixed-size arrays, dynamic arrays, . Associative arrays, string mapped associative arrays, and multidimensional arrays are not supported. The syntax is:

```
foreach (name, loop_variable) {
statement;
}
```

`name`

is the array ueue to use in the `foreach` block.

`loop_variable`

is the name of a newly created temporary variable of type integer which acts as an indexing member. This variable only needs to be declared in the argument list and the constraint block. The `loop_variable` name cannot be the same as the array or Smart Queue name. Example 3-6 shows a `foreach` example:

Example 3-6 foreach Loop

```
program example {
      string names[$]={"Hello", "World "};
      foreach (names, i){
            printf("Value at index %0d is %0s\n", i, names[i]);
        }
}
```

This program prints out:

```
Value at index 0 is Hello
Value at index 1 is World
```

break and continue Statements

You can use `break` and `continue` statements to control the flow within loops. A `break` statement forces the immediate termination of a loop, bypassing the normal loop test. The syntax is:

```
break;
```

When a `break` statement is executed from inside a loop, the loop terminates immediately and control passes to the first line of code after the loop. If a `break` statement is used outside of a loop, NTB-OV issues a syntax error. For example:

```
while (test_flag) {
      if (done) break;
      ...
}
```

This example breaks when `done` is true and control passes to the first line after the loop.

A `continue` statement forces the next iteration of a loop to take place, skipping any code in between. The syntax is:

```
continue;
```

In a `repeat` loop, a `continue` statement passes control back to the top of the loop. If the loop is complete, control passes to the first line of code after the loop.

In a `for` loop, a `continue` statement causes the increment portion and conditional test of the loop to execute.

In a `while` loop, a `continue` statement passes control to the conditional test. For example:

```
for (i=0;i<10;i++) {
      if (skip_loop) continue;
      ...
}
```

4

NTB-OV Classes

This chapter explains how to declare OpenVera Native Testbench (NTB-OV) classes, create and initialize objects of a class, and manipulate such objects, in the following major sections:

- Class Declaration
- Creating an Object of a Class
- Properties
- Methods
- Constructors
- Sharing Class Properties
- Subclasses and Inheritance
- Data Hiding and Encapsulation
- super Keyword

- Chaining Constructors
- Virtual Class
- Finding the Right Method
- Type Casting

Class Declaration

The user-defined data type `class` is composed of data members (also known as properties) and tasks or functions (methods) for manipulating those data members. Class declarations are top-level constructs and have global scope. You cannot declare classes in tasks or functions, or nested in another class. Use the `class` keyword to declare a new class data type. The syntax is:

```
class class_name  {
      [ property_declaration ]...
      [ method_declaration ]...
}
```

`class_name`

is the name of the class declaration, which becomes the name of the new data type. Any valid variable declarations (except enumerated types) are allowed inside class declarations.

Example 4-1 shows an example class declaration.

Example 4-1   Class Declaration

```
class Packet {
      reg [3:0] command; // property declarations
      reg [40:0] address;
      reg [4:0] master_id;
      integer time_requested;
      integer time_issued;
      integer status;
```

```
        Packet next;

        task new(){ // initialization
              command = IDLE;
              address = 4'b0;
              master_id = 5'bx;
              next = null;
        }

        task clean() //method declaration {
              command = 0; address = 0; master_id = 5'bx;
        }

        task issue_request(integer delay ){// method declaration
        }
        function integer current_status(){ // method declaration
              current_status = status;
        }
  }
```

Forward Referencing a Declaration

NTB-OV supports forward referencing of a `class` name before its definition is seen using the `typedef` keyword, as shown in the following example:

```
typedef class X;
class Y {
      X x;
      ...
}
class X {
      ...
}
```

Creating an Object of a Class

To use the methods of a class, you must create an *instance* of the declared class. First, declare a variable that can hold an object handle:

```
Packet p;
```

In this example, the class data type is `Packet`. The declaration of `p` is simply a variable that can hold a handle of a `Packet` object. For `p` to refer to something, you must create an instance of the class using the `new()` task:

```
p = new();  // the () is optional.
```

By default, NTB-OV sets an uninitialized object handle to the special value `null`. You can detect an uninitialized object by comparing its handle to `null`. For example, the `task mytask` statement below checks if the object is initialized. If it is not, it creates a new object using the `new()` task.

```
class obj_example {
    ...
}

task mytask(integer a, var obj_example myexample){
      if (myexample == null) myexample = new;
}
```

Here are some examples of different ways to instantiate a class:

```
class Packet { ... } // class definition defines the data type

Packet p;      // class reference declaration declares a handle
                   // it is initialized to null. (p == null) evaluates
                  // to 1 (true).
Packet q, r;
integer i = 10;
```

```
p = new;      // class instantiation creates an object of type
                // Packet and stores the handle to that object in p.

q = new p;    // class instantiation creates a Packet object
                     // with q containing its handle and the properties of
                   // p's object (shallow) copied into it.

r = new(p,i);// instantiation with custom initialization
                      // is allowed if class Packet contains a method
                         // defined as: task new(Packet p, integer i) { ... }
```

Properties

The properties in a class can be the following atomic types or arrays of these atomic types.

- reg
- reg [msb:0]
- integer
- string
- event
- class type
- enum type

You can precede a property declaration with one of the following keywords:

- `local`—a property designated as `local` is visible only to methods in the class.

- `public`—a property designated as `public` is accessible everywhere.
- `protected`—properties are public by default, but can also be protected. A `protected` property has all of the characteristics of a local member, except that it can be inherited. A `protected` property is also visible to subclasses.

You can initialize a static property to any expression. However, the order of static initializations is not guaranteed. Therefore, if the expression is not a constant, it can lead to unpredictable results.

Accessing Properties

You access an object's data fields using the dot operator (`.`). The variable name for the object precedes the dot, followed by the qualifying property name (for example, address, command).

`instance_name.property_name`

For example, you can use commands for the `Packet` object `p,` as shown in the following example:

```
Packet p = new; // declares a handle and instantiates the class
integer b;
p.command = INIT;
p.address = $random;
time = p.time_requested;

p.address = 41'b0;// sets property address value as 4'b0
p.next = new;    // sets property nextvalue as the handle
                    // to a newly created object
p.next.address = 41'hdeadbeef;
b = p.next.address;
```

Methods

Methods are either tasks or functions. Like properties, you can designate methods as **`local`** or **`public`**; they are public by default. You can declare the body of a method either inside the class definition or outside of it using an out-of-block declaration. The syntax for an out-of-block declaration is similar to that of a regular global task or function declaration *`except`* that the method name is prefixed by *`classname`*:: as shown below.

```
class Packet {
      ...
      function integer send(integer value);
}
function integer Packet::send(integer value) { ... }
```

Accessing Object Methods

You access an object's methods using the dot operator (`.`). The variable name for the object precedes the dot, followed by the method, as shown in the following example:

```
      Packet p = new;
// declares handle and instantiates class
      p.issue_request(100);
// calls method on the first object created above
      b = p.next.current_status();
// calls method on the second object created above
```

Constructors

NTB-OV provides a mechanism for initializing an instance when the object is created. For example:

```
Packet p = new;
```

With this example, NTB-OV executes the `new()` task associated with the class as shown in the following example:

```
class Packet {
      integer command;

      task new() {
            command = IDLE;
      }
}
```

The `Packet p = new` statement creates an object of class `Packet`. You can also pass arguments to the constructor, which allows runtime customization of an object. For example:

```
Packet p = new(STARTUP),
$random,
$time);
```

Now the `new()` initialization task in `Packet` might look like:

```
task new(integer inCommand=IDLE, reg[12:0]
inAddress=0, integer time) {
      command = inCommand;
      address = inAddress;
      time_requested = time;
}
```

The conventions for arguments are the same as other with method calls, including the use of default arguments.

Assignment, Renaming, and Copying

The following example creates a variable `p1` that can hold the handle of an object of class `Packet`.

```
Packet p1;
```

The initial default value of `p1` is null. The object does not yet exist, and `p1` does not contain an actual handle until you create an instance of type `Packet` as shown below:

```
p1 = new;
```

You can declare another variable of type `Packet` and assign the handle `p1` to it as follows:

```
Packet p2;
p2 = p1;
```

In this case, there's still only one object. This single object can be referred to using either the `p1` or `p2` variable. You can then use either variable to create a new object, as shown in the following example.

```
p2 = new p1;
```

This is shorthand for creating a new object with `p2` as its handle and copying all the properties in the `p1` object to the `p2` object.

The instance `p2` is known as a shallow copy because all the variables are copied across, including integers, strings, and instance handles. NTB-OV does not copy objects; only their handles. As before, two names for the same object are created. This is true even if the class declaration includes the instantiation operator `new()` as shown in Example 4-2.

Example 4-2 Class Assignments

```
class A    {
      integer j;
      task new(){ j=5;}
}

class B    {
```

```
        integer i;
        A a;
        task new() {i = 1;}
    }

    task test() {

        B b1 = new; // Create an object of class B
        B b2;
        b1.a = new;
        b2 = new b1; // Create an object that is a copy of b1
        b2.i = 10; // i is changed in b2, but not in b1
        b2.a.j = 50; // change a, shared by both b1 and b2
        test = b1.i; // test is set to 1 (b1.i has not changed)
        test = b1.a.j; // test is set to 50 (a.j has changed)
    }
     printf ("Best is now %d\n", best);
     test ();
     printf ("Best is now %d\n", best);
    }
```

Several things are noteworthy in Example 4-2:

- Properties and instantiated objects can be initialized directly in a class declaration.
- The shallow copy does not copy objects.
- You can chain instance qualifications as needed to reach into or through objects:

```
b1.a.j // reaches into a, which is a property of b1
p.next.next.next.val // chain through a sequence of
                     // handles to get to val
```

To do a full (deep) copy, where everything (including nested objects) is copied, you typically need to use custom code. For example:

```
Packet p1 = new;
Packet p2 = new;
p2.copy(p1);
```

In this example, `copy(Packet p)` is a method written by the user to copy the object specified as its argument into its instance (that is, `p2`).

Sharing Class Properties

Sometimes you need only one version of a variable to be shared by all instances. You can create such class properties using the `static` keyword. For example, in a case where all instances of a class need access to a semaphore id, you can use the `static` keyword as follows:

```
class Packet {
      static integer semId = alloc(SEMAPHORE, 1);
}
```

The `semId` in this example is created and initialized once. Now every `Packet` object can access the semaphore in the usual way:

```
Packet p;
semaphore_get(WAIT, p.semId);
```

this

If you need to unambiguously refer to properties or methods of the current instance, you can use the `this` keyword to write an initialization task, as shown in the following example:

```
class Demo {
      integer x;

      task new (integer x){
            this.x = x;
       }
}
```

In this example, `x` is now both a property of the class and an argument to the task `new()`. In the task `new()`, an unqualified reference to `x` is resolved by looking at the innermost scope, in this case the argument declaration. To access the instance property, you qualify it with the `this` keyword to refer to the current instance. In writing methods, you can always qualify members with `this` to refer to the current instance, but it is usually unnecessary.

Subclasses and Inheritance

In earlier examples in this chapter, we defined a class called `Packet`. If you want to extend this class so the packets can be chained together into a list, you can create a new class called `LinkedPacket` that contains a variable of type `Packet`.

Whenever you refer to a property of `Packet`, you need to reference the variable `packet`, as shown in the following example:

```
class LinkedPacket {
Packet packet;
      LinkedPacket next;

      function LinkedPacket get_next() {
            get_next = next;
      }
}
```

Because `LinkedPacket` is a specialization of `Packet`, a more elegant solution is to extend the class, creating a new subclass that inherits the members of the parent class, as shown in this next example:

```
class LinkedPacket extends Packet {
      LinkedPacket next;
```

```
        function LinkedPacket get_next(){
              get_next = next;
        }
  }
```

Now, all of the methods and properties of `Packet` are part of `LinkedPacket`—as if they were defined in `LinkedPacket`—and `LinkedPacket` has additional properties and methods. You can also override the parent's methods, changing their definitions.

Data Hiding and Encapsulation

In NTB-OV, unlabeled properties and methods are public. That is, an object's properties and methods are, by default, accessible to tasks and functions that are not members of the class. By making certain members `local` you can restrict access to these properties and methods such that they are visible only to members of the same class. You can then modify `local` members without changing any code outside the class. In other words, you can change the underlying implementation of a class without changing any external code that uses the class.

A member identified with the `local` keyword is available only to methods inside the class. A protected property or method has all of the characteristics of a `local` member, except that it can be inherited; it is visible to subclasses.

In summary, use local members whenever possible. Hide members that the outside world doesn't need to know about. And use protected members if the outside world doesn't have a need to know, but subclasses might.

Public access should only be allowed when it is absolutely necessary, and the access should be limited as much as possible. Generally, you should not provide direct access to properties. Instead, it is good practice to use methods to provide this access. For example, provide only read access if a variable should never be written. This gives an extra level of protection and preserves flexibility for future changes.

super Keyword

Use the `super` keyword from within a derived class to refer to properties of the parent class. You need to use `super` when the property of the derived class has been overridden and cannot be accessed directly (see Example 4-3).

Example 4-3 Super Keyword

```
class Packet  { //parent class
      integer value;
      function integer delay(){
            delay = value * value;
      }
}

class LinkedPacket extends Packet { //derived class
      integer value;
      function integer delay() {
       delay = super.delay()+value * super.value;
      printf("super.delay()+value * super.value\n");
               printf(" %0d + %0d * %0d = %0d\n",super.delay(),
                   this.value, super.value, delay);
}
task setSuperValue(integer value){
super.value = value;
   }
}
program test {
LinkedPacket lp = new();
lp.value = 1;
lp.setSuperValue(7);
printf("Result: %0d \n", lp.delay());
```

```
}
```

The property can be a member declared a level up or inherited by the class one level up. There is no way to reach higher (for example, `super.super.count` is not allowed).

Subclasses are classes that are extensions of the current class, whereas `super` classes are classes that the current class is extended from, beginning with the original base class. When using the `super` keyword with `new()`, `super` must be the first statement in the constructor.

Chaining Constructors

When you instantiate a subclass, one of the system's first actions is to invoke the class method `new()`. The first implicit action `new()` takes is to invoke the `new()` method of its superclass, and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the base class and ending with the current class.

If the initialization method of the super-class requires arguments, you have two choices. If you want to always supply the same arguments, you can specify them when you extend the class:

```
class EtherPacket extends Packet(5) {
...
}
```

This example passes `5` to the `new()` routine associated with `Packet`. A more general approach is to use the `super` keyword to call the superclass constructor as the first executable statement of the constructor, as shown in this next example:

```
task new() {
      super.new(5);
{
```

Virtual Class

It is common to create a set of classes that can all be viewed as derived from a common base class. For example, you might start with a common base class of type `BasePacket` that sets out the structure of packets, but is incomplete; you would not want to instantiate it. From this base class, though, you might derive a number of useful subclasses: Ethernet packets, token ring packets, GPSS packets, and satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details. You start by creating the base class that sets out the prototype for these subclasses. Since you don't need to instantiate the base class, declare it to be abstract by declaring the class to be virtual:

```
virtual class BasePacket {}
```

By themselves, abstract classes are not that interesting, but abstract classes can also have virtual methods. Virtual methods provide prototypes for subroutines and all the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type (if needed). Later, when subclasses override virtual methods, they must follow the prototype exactly. Thus, all versions of the virtual method look identical in all subclasses, as shown in Example 4-4.

Example 4-4 Virtual Methods

```
virtual class BasePacket {
      virtual protected function integer send(reg[31:0] data);
}
```

```
class EtherPacket extends BasePacket {
     protected function integer send(reg[31:0] data)
     {
          // body of the function
          ...
}
```

Using this example, `EtherPacket` is now a class you can instantiate. In general, if an abstract class has several virtual methods, all the methods must be overridden for the subclass to be instantiated. If all of the methods are not overridden, the subclass needs to be abstract.

You can also declare methods of normal classes to be virtual. In this case, the method must have a body. Now, the class can be instantiated, as can its subclasses. However, if the subclass overrides the virtual method, the new method must exactly match the superclass's prototype.

Polymorphism

Polymorphism allows you to use superclass variables to hold subclass objects and reference the methods of those subclasses directly from the superclass variable. For example, consider the base class for the packet objects, `BasePacket`. Assume that it defines, as virtual functions, all of the public methods to be generally used by its subclasses; methods such as send, receive, and print. Even though `BasePacket` is abstract, you can still use it to declare a variable, as shown in the following example:

```
BasePacket packets[100];
```

You can now create instances of various packet objects, and put them into the array you just created as follows:

```
EtherPacket ep = new();
TokenPacket tp = new();
GPSSPacket gp = new();
packets[0] = ep;
packets[1] = tp;
packets[2] = gp;
```

If your data types were, for example, integers, regs, and strings, you couldn't store all of these types in a single array, but with polymorphism you can do this with objects. In this example, since the methods are declared as virtual, you can access the appropriate subclass methods from the superclass variable even though the compiler didn't know at compile time what was going to be loaded into, for example, `packets[1]`:

```
packets[1].send();
```

This example invokes the send method associated with the `TokenPacket` class. At runtime, NTB-OV correctly binds the method from the appropriate class.

Finding the Right Method

There are several subtleties that arise when you start using virtual methods, although the underlying rules are simple. At some point, the compiler or the runtime system needs to find the proper method. In a simple case with no inheritance, the answer is straightforward:

```
GigaEtherPacket p = new();
p.send();
```

In this example, the system invokes the `send()` method declared in `GigaEtherPacket`. But if you inherited a virtual method, you need to find the right version. If `GigaEtherPacket` is a subclass, and doesn't declare `send()`, what do you do next?

The first step is to decide where in the class hierarchy to begin searching for the method. Begin searching from the class associated with the handle for the method you need to find. In the case above, this is the `GigaEtherPacket` class.

If the method is invoked from inside another method, then the handle is the invoking method's class; if `send()` now invokes `setup()`, start with the class containing `send()`:

```
task send() {
setup();
}
```

The handle for this reference to `setup()` is the implicit handle `this`.

The second step is to search the hierarchy. If the method is not defined at this level, begin going up the inheritance tree unless the method is defined in the superclass as local—in that case the inheritance chain is broken (you can't inherit this method) and you have an error.

You are at the base class, in which case the method was not found. If the method is defined but is not virtual, use it. Otherwise, it must be virtual. Search from this class down the inheritance tree. If you find a non-virtual method, use it. If you hit the bottom of the tree, use the most recent virtual method with a body; this is the method closest to the bottom of the tree. If there is none, the search failed.

So go up the inheritance tree until you find a method you can use. If it is virtual, go back down the tree until you find a non-virtual method. If you hit bottom without finding a method, use the last virtual method with a body you came across. For example, with the following:

```
class BasePacket {
     virtual task send (integer value);
     task init() { // calls a virtual task
```

```
      send(0);
      }
}
class EtherPacket extends BasePacket {
      // no re-declaration of send() or init()
            ...
      }

class Ether100 extends EtherPacket {
      {
            task send(integer value)
            ...
      }
      Ether100 ep = new();
}
```

With this example, if you invoke `ep.init()`, NTB-OV does not execute the version of `init()` defined in `BasePacket`, but the version of `send()` declared in `Ether100`.

Note that if you declare a method as virtual in a base class, it is usually a good idea to declare it virtual in the subclasses, too.

Type Casting

You can use the `cast_assign()` system function to assign values to variables that might not ordinarily be valid because of differing data types. The syntax is:

```
function integer cast_assign(scalar dest_var, scalar
source_exp [, CHECK]);
```

`dest_var`

is the variable to which the assignment is made. It can be any non-array scalar type (reg, integer, string, enumerated type, virtual port, event, or object handle).

`source_exp`

is the source expression assigned to the destination variable.

`CHECK`

The predefined CHECK macro is optional. Its use determines how the function handles invalid assignments. When you call `cast_assign()` without CHECK, the function assigns the source expression to the destination variable. If the assignment is illegal, a fatal runtime error occurs. When you call the `cast_assign()` system function with CHECK, the function makes the assignment and returns a 1 if the casting is successful. If the casting is *un*successful, the function does not make the assignment and returns a 0. In the latter case, no runtime error occurs, and the destination variable is set to null, X, or uninitialized, depending on the data type.

The NTB-OV compiler only checks that the destination variable and source expression are scalars. Otherwise, no type checking is done at compile time. Example 4-5 shows how to use the `cast_assign()` function to assign an extended object to an extended handle.

Example 4-5 Type Casting

```
class Base {
      integer p;
      virtual task display() {
            printf("\nBase: p=%0d\n", p);
      }
}

class Extended extends Base {
      integer q;
      virtual task display() {
            super.display();
            printf("Extended: q=%0d\n", q);
       }
}

program sample {
      Base b1 = new(), b2;
```

```
            Extended e1 = new(), e2;

            b1.p = 1;
            b1.display(); // Just shows base property
            e1.p = 2;
            e1.q = 3;
      e1.display(); // Shows base and extended properties

// Have the base handle b2 point to the extended object
      b2 = e1;
      b2.display();  // Calls Extended.display

// Try to assign extended object in b2 to extended
// handle e2
      if (cast_assign(e2, b2, CHECK))
            e2.display();  // Calls Extended.display
      else
            printf("cast_assign of b2 to e2 failed\n");
      } // program sample
```

The output of Example 4-5 is:

```
Base: p=1

Base: p=2
Extended: q=3

Base: p=2
Extended: q=3

Base: p=2
Extended: q=3
```

Notice that `cast_assign()` was successful in assigning the extended object `b2` to the extended handle `e2`.

Note:

The `cast_assign()` system function does not currently support the `enum` data type.

5

Concurrency Control

This chapter explains how OpenVera Native Testbench (NTB-OV) handles concurrency, including how to model parallel, independent activities. It also explains the NTB-OV constructs you can use to control concurrent threads. This information is presented in the following major sections:

- fork/join Block
- Synchronizing Concurrent Processes with Event Variables
- Semaphores
- Mailboxes

fork/join Block

Fork/join blocks provide the primary mechanism for creating concurrent processes. The syntax to declare a `fork/join`block is:

```
fork {
    statement1;
}

{
    statement2;
}

...

{
    statementN;
}
join [all | any | none]
```

`statement`

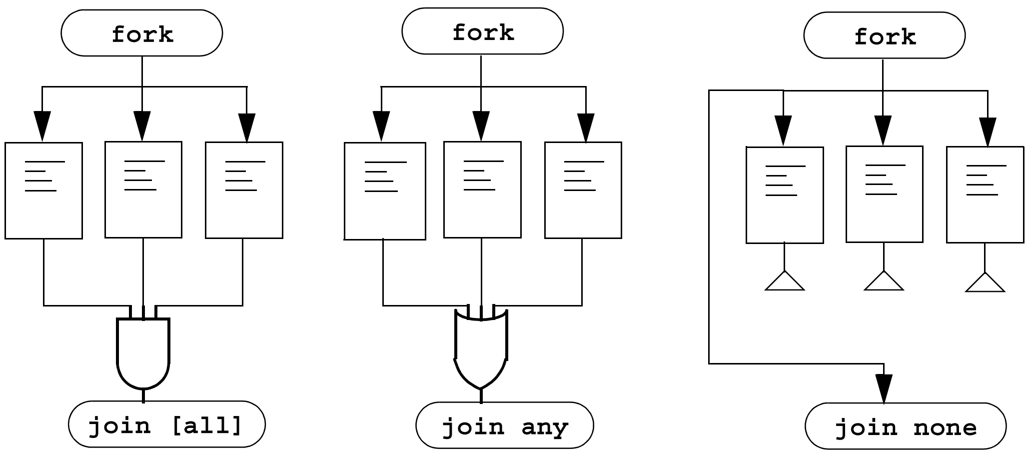
can be any valid statement or sequence of statements.

The `all | any | none` options specify when the code after the fork/join block executes. They are optional.

- The default is `all`. Code after the block executes after all of the concurrent processes are completed.
- When `any` is used, code after the block executes after any single concurrent process is completed.
- When `none` is used, code after the block executes immediately, without waiting for any of the processes to complete.

You don't need to specify more than one forked thread. If only a single thread is specified in a `fork/join` block and that thread consists of a single statement, the thread does not need to be encapsulated with braces ({ }). Figure 5-1 illustrates the flow for a `fork/join` block.

Figure 5-1   fork/join Flow Diagram



When defining a `fork/join` block, encapsulating the entire fork inside braces ({ }) results in the entire block being treated as a single thread, and the code executes consecutively. For example, don't use `fork/join` to execute `statement1` and `statement2` concurrently as shown in the following example:

```
fork  {
      statement1;
      statement2;
}
join // executes as a single sequential process
```

Example 5-1 shows a basic `fork/join` example that takes advantage of the default `all` option.

Example 5-1   Basic fork/join Construct with Default all

```
fork   {
      @1,100 bus.ack == 1'b0;
```

```
        printf("First Block: bus.ack is sampled\n");
    }

    {
        @5 bus.req = 1'b0;
        @1 bus.req <= 1'b1;
        printf("Second Block: bus.req is driven\n");
    }
    join
```

With this example, the concurrent block executes all the statements in parallel. The beginning of each statement executes at the same time. Subsequent statements are executed based on any timing considerations within the process.

Shadow Variables

By default, all child processes have access to the parent´s variables. However, if multiple processes independently use the same variable, races can occur. To avoid races within `fork/join` blocks, use shadow variables. This syntax is:

`shadow` `data_type variable_name;`

`data_type`

Allowed data types are integer, reg, string, event, enum, arrays.

You cannot:

- declare a shadow variable in a task or function's formal argument list, so the following is not allowed:

  ```
  task T(shadow integer i)
  ```

- declare shadow variables in a global context. Recall that variables declared in the NTB-OV main program block are global.

Using the `shadow` keyword forces the NTB-OV compiler to create a copy of the variable local to each child process, which eliminates race conditions. Any descendants of the child processes also have a copy of the variable local to that descendant. Example 5-2 shows a sample NTB-OV program that does not use a shadow variable:

Example 5-2   Program without Shadow Variable

```
task process_value(integer i) {
      printf("[ = %0d\n", i);
}

task spawn_process () {
      integer n;
      printf("[ In spawn_process\n");
      for(n=0; n<3 ; n++) {
            fork
                  process_value(n) ;
            join none
      }
}

program test {
      spawn_process() ;
      @(posedge CLOCK) ;
}
```

The program shown in Example 5-2 produces the following output:

```
[ In spawn_process
[ = 3
[ = 3
[ = 3
```

In this example, the `test` program calls the `spawn_process()` task, where a `for` loop spawns three processes called `process_value`. The `fork/join none` schedules each process for execution but does execute them. After the `for` loop completes it exits with the iterator `n` equal to 3 and `spawn_process` returns to the calling program, where the `@(posedge CLOCK);` statement advances the simulation. Now, the scheduled processes are allowed to execute. Since each process shares the variable `n`, each `process_value` task´s argument is assigned the value of variable `n`, which is 3.

If you modify the task shown in Example 5-2 to attach the shadow attribute to the variable `n`, each process gets its own copy. That's because the `shadow` keyword instructs the NTB-OV compiler to create a copy of the variable local to each child process. Now, the sample program looks like Example 5-3.

Example 5-3 Program with Shadow Variable

```
task process_value(integer i)
{
  printf("[ = %0d\n", i);
}

task spawn_process ()
{
shadow integer n;
printf("[ In spawn_process\n");

for(n=0; n<3 ; n++){
  fork
      process_value(n);
  join none
  }
}

program test {
 spawn_process();
 @(posedge CLOCK);
}
```

The program shown in Example 5-3 produces the following output:

```
[ In spawn_process
[ = 0
[ = 1
[ = 2
```

Compare these results to the output produced by the same program without using the shadow variable (Example 5-2).

Controlling fork/join Blocks

NTB-OV has several constructs and a system task you can use to control `fork/join` blocks.

wait_child ()

Use the `wait_child()` system task to halt execution of the current process until all descendant processes are executed. The syntax is:

```
task wait_child();
```

By default, simulation terminates when the end of the program is reached, regardless of the status of any child processes. Using the `wait_child()` task causes the simulation to wait until all child processes in the current context are completed before executing the next line of code, as shown in Example 5-4.

Example 5-4 Program with wait_child() Construct

```
program test {
      start_monitors(); /* Starts monitors that loop
                           forever in background */
      do_test(); // Performs the actual test
}
task start_monitors() {
      fork
      {...}
      join none
}
task do_test() {
      //Code to do testing
      fork
      {...}
      join none /* Creates child processes that take an
            indeterminate amount of time to complete */
```

```
        wait_child();
}
```

Example 5-4 calls two separate tasks:

- The `do_test()` task forks off several child processes that take an indeterminate amount of time to complete.
- The `wait_child()` task call waits for the threads called in the `do_test()` task to complete before executing the subsequent NTB-OV code.

Note that the `wait_child()` task call does not wait for child processes created outside of its context. Here, a context is a node in the simulator's call stack. NTB-OV constructs that create a new context are:

- the program block
- task
- function
- each process inside the `fork/join`

To see how `fork/join all` and `wait_child()` differ, consider the following examples:

fork/join all

```
fork
{statement3};
{statement4};
join none
fork
{statement1};
{statement2};
join all
```

wait_child ()

```
fork
{statement3};
{statement4};
join none
fork
{statement1};
{statement2};
join none
wait_child();
```

In the `fork/join all` example, code following the block executes after the `statement1` and `statement2` concurrent processes complete. However, code after the block executes immediately, without waiting for `statement3` and `statement4` to complete. Compare Example 5-4, where `statement3` and `statement4` are waited for.

terminate

The `terminate` statement terminates all descendants of the process in which it was called. The syntax is:

```
terminate;
```

If any of the child processes have other descendants, the `terminate` command terminates them as well. If used at the top level, `terminate` terminates all child processes. When the main program completes, NTB-OV executes an implicit `terminate` statement. Example 5-5 shows an NTB-OV program that uses a `terminate` statement.

Example 5-5 Using terminate with Simple fork/join Block

```
task do_test(){
      // Code to do testing
      fork
      {...}
```

```
        join any /* Creates child processes that take an
                   indeterminate amount of time to complete
                   Code to do more testing */
        terminate;
    }
```

This example forks off several child processes within a task. After any of the child processes complete, the code continues to execute. Before the task is completed, NTB-OV terminates all remaining child processes.

suspend_thread ()

Use the `suspend_thread()` system task to temporarily suspend the current thread. The syntax is:

```
task suspend_thread();
```

This suspends the current thread and allows other ready concurrent threads to run. When all ready threads have had one chance to block, the suspended thread resumes execution. Example 5-6 shows part of an NTB-OV program that uses `suspend_thread()`.

Example 5-6   Using suspend_thread ()

```
    for (i=0;i<10;i++) {
        fork
        my_task(i);
        join none
        suspend_thread();
    }
```

Example 5-6 forks multiple threads that call `my_task()`. The thread is forked, the task is called, and then the calling thread is suspended. The forked thread calling `my_task(0)` completes and passes control back to the `for` loop. The next iteration of the loop occurs and forks the next thread. That thread begins and completes execution. All 10 threads are created and executed in sequence. Using this construct, you do not need to declare i as a shadow variable.

Note:

Suspended threads execute after all other current threads execute. However, relative to simulation time, the thread is still executed concurrently with the other threads.

wait_var ()

Use the wait_var() system task to block the calling process until one of the variables in its argument list changes value. The syntax is:

```
task wait_var(integer|reg|string|enum variable_list);
```

variable_list

consists of one or more variables (separated by commas) of type integer, reg, string, or enumerated type.

Only true value changes unblock the process. Reassigning the same value does not unblock. If you specify more than one variable, a change to any of the variables unblocks the process.

If multiple threads are blocked by the same wait_var() variable at the same NTB-OV time stamp, the threads are serviced in LIFO order. Example 5-7 shows an NTB-OV program that uses a wait_var() task.

Example 5-7 Program with wait_var () Task

```
reg[7:0] data [100];
integer i;
fork {
      wait_var(data[2]);
      printf("Data[2] has changed to: %d\n", data[2]);
```

```
}
{
      for (i=0;i<100;i++){
            data[i]=random();
            @(posedge CLOCK);
      }
}
join
```

Example 5-7 forks off concurrent processes. The first thread is suspended until the second element of the `data` array changes. The second process randomly changes the values within the `data` array. When `data[2]` changes, the first process prints its message.

Synchronizing Concurrent Processes with Event Variables

Event is a basic NTB-OV data type you can use to synchronize concurrent processes using `sync()` or `trigger()` tasks. When you call `sync ()` with an `event` argument, a process blocks until another process sends a trigger to unblock it.

sync () Task or Function

Use a `sync()` to synchronize statement execution to one or more triggers. You can use `sync()` as a task or a function. The syntax is:

```
task sync(ALL | ANY | CHECK, event event_name1,
      ..., event event_nameN);
```

`event_name`

is the event variable name on which the `sync()` is activated.

`ALL`

suspends the process until all of the specified events are triggered. For example:

```
sync(ALL, event_a, event_b, event_c);
```

This example suspends the thread until all of the events trigger. Then NTB-OV executes the statement immediately following the `sync()` call.

`ANY`

suspends the process until any of the specified events is triggered. For example:

```
sync(ANY, event_a, event_b, event_c);
```

This example suspends the thread until any of the specified events trigger. Then NTB-OV executes the statement immediately following the `sync()` call.

`CHECK`

is called as a function. It does not suspend the thread. `CHECK` returns a `1` if the event is `ON` or null; else, it returns a `0`. You can only use this `sync()` type with `ON` and `OFF` trigger types and a single event per call. For example:

```
if (sync(CHECK, event_a)
      printf("The event is ON.\n");
```

This `sync()` call returns a `1` if the event is `ON` or null, and then prints the message. If the event is `OFF`, it returns a `0`.

trigger () Task

Use the `trigger()` task to change the state of an event. Triggering an event unblocks waiting syncs, or blocks subsequent syncs. By default, all events are `OFF`. The syntax is:

```
task  trigger([ONE_SHOT | ONE_BLAST |HAND_SHAKE | ON |
      OFF,] event event_name1 , ... ,event event_nameN);
```

`event_name`

is the event variable name on which the sync is activated.

`ONE_SHOT`

is the default trigger type. If you use a `ONE_SHOT` trigger, any process waiting for a trigger receives it. If there are no processes waiting for the trigger, NTB-OV discards the trigger (see Figure 5-2).

Figure 5-2 ONE_SHOT Triggers

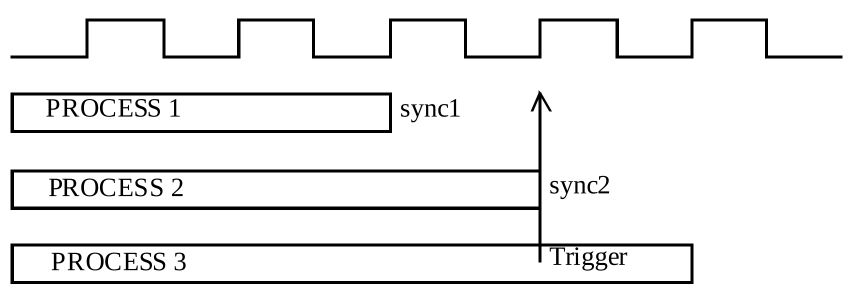


Figure 5-2 shows a trigger called in process 3. The trigger unblocks `sync1`. However, because the trigger and `sync2` execute simultaneously, whether or not process 2 is unblocked depends on the execution order. The `sync()` must be called before the trigger is executed when using `ONE_SHOT` triggers. If the `sync()` is called after the trigger is executed, the process waits indefinitely.

`ONE_BLAST`

triggers work just like `ONE_SHOT` triggers except that they trigger any `sync()` called within a simulation time step, regardless of whether it was called before the trigger was executed.

`HAND_SHAKE`

triggers unblock only one `sync()`, even if multiple syncs are waiting for triggers. The syncs are unblocked on a FIFO basis. If the order of triggering the unblocking of a sync is important, use a `semaphore_get()` and `semaphore_put()` around the `sync()` to maintain order. For more information, see "Semaphores" on page 137.

If a `sync()` was already called and is waiting for a trigger, the `HAND_SHAKE` trigger unblocks the sync. If no `sync()` was called when the trigger occurs, the `HAND_SHAKE` trigger is stored. When a `sync()` is called, NTB-OV immediately unblocks the `sync()` and removes the trigger.

`ON`

Use the `ON` trigger to turn on an event. When an event is turned on, all syncs waiting for that event immediately trigger. Also, all future `sync()` operations on that event immediately trigger until there is a trigger (`OFF`) call.

`OFF`

Use the `OFF` trigger to turn off an event. When an event is turned `OFF`, all future `sync()` operations on the event are blocked.

event Variables

An `event` variable contains a handle to a event object. You can pass `event` variables as arguments to tasks and functions. Event variables are bidirectional when used as arguments in `sync()` and `trigger()` calls. You can use the same `event` variable to pass and receive triggers. Example 5-8 shows an `event` variable used in an NTB-OV program.

Example 5-8   Event Variable

```
      task T1 (event trigger_a) {
            printf("\nT1 syncing at cycle=%0d",get_cycle());
            sync(ALL, trigger_a);
// Blocked: proceed after receiving trigger
            printf("\nT1 event trigger_a received at cycle=%0d",
                  get_cycle());
            repeat (7) @(posedge CLOCK);
            printf("\nT1 triggering trigger_a at cycle=%0d",
                  get_cycle() );
            trigger (trigger_a);
      }

      program trigger_play {
            event trigger1;
            breakpoint;
      // top block code starts here

            fork
                              T1(trigger1);// start T1 and go on
            join none
            {
            repeat(8) @(posedge CLOCK); // T1 is blocked waiting
                                        // for trigger
            }
            fork              {
                  printf("\nPROGRAM triggering trigger1 @cycle=%0d",
                        get_cycle());
                  printf("\nPROGRAM This unblocks T1");
                  trigger(trigger1); // unblock the waiting T1
            }
            {

                  repeat (5) @(posedge CLOCK);
                  printf("\nPROGRAM syncing @cycle=%0d\n\n",
                        get_cycle());
```

```
            sync (ALL, trigger1) ;// wait for T1 to unblock me
        }
        join
        wait_child();
        printf("Trigger play done!");
}
```

In Example 5-8, the `T1` task is defined and then called in a thread forked off from the main program. The program continues without waiting for the child thread to complete. Because `T1` contains a `sync()` in its definition, the child thread blocks, waiting for a trigger. Then another `fork` is used to fork off a trigger, which unblocks the suspended `T1`. A second thread in that `fork` then calls a `sync()`. This `sync()` occurs as `T1` is unblocked. `T1` then continues execution, including execution of the trigger that unblocks the final child thread.

Disabling Events

If an event variable is assigned a null value, subsequent `sync()` calls are immediately unblocked. Previously waiting `syncs()` remain blocked. Consider the following example:

```
event E1 = null;
sync (ALL, E1);
```

Here, the `sync()` is immediately satisfied because of the `null` value assigned to `E1`. Note that assigning `null` to an `event` variable does not cause currently waiting syncs to unblock. Those syncs are waiting to synchronize on a different event object. If the handle to that event object is not available in another `event` variable, it is impossible to change the state of that object again—the syncs would then block forever (see Example 5-9).

Example 5-9   Assigning null to event

```
event E1, E2;
fork
     { sync(ALL, E1); } // will never unblock due to
```

```
                        // E1=null in the sibling process
      { delay(1); E1 = null; }
join any
trigger(ON, E1);
wait_child();
```

Merging Events

When you assign an `event` to another `event`, they merge, which causes triggers on either one to affect both, as shown in Example 5-10.

Example 5-10 Event Merge Effect on Triggers

```
event E1, E2, E3;
E1 = E2;
trigger(E3);
trigger(E1); // this will trigger E2, as well
trigger(E2); // this will trigger E1, as well
E1 = E3;
E2 = E1;
trigger(E1); // this will trigger E2 and E3, as well
trigger(E2); // this will trigger E1 and E3, as well
trigger(E3); // this will trigger E1 and E2, as well
```

Be careful when merging events. The assignment only affects subsequent triggers and syncs. For example, if a process is blocked waiting for `event1` when you assign another event to `event1`, the `sync()` never unblocks. Consider the following example:

Example 5-11 Threads that Never Unblock

```
fork {
      while (1) {sync (ALL, E2);}
}
{
      while (1) {sync (ALL, E1);}
}
{
      E2 = E1;
      while (1) {trigger (E2);}
}
join
```

Example 5-11 forks off three concurrent threads. Each starts at the same simulation time. When threads `1` and `2` are blocked, thread `3` assigns event `E1` to `E2`. This means that thread `1` can never unblock, because event `E2` is now `E1`. To unblock both threads `1` and `2`, you must merge `E2` and `E1` before the fork.

Semaphores

You can use semaphores for mutual exclusion and synchronization, as explained in the following subsections:

- Conceptual Overview
- Allocating Semaphores
- Obtaining Semaphore Keys
- Returning Semaphore Keys
- Semaphore Example

Conceptual Overview

Think of a semaphore as a bucket. When you allocate a semaphore, you create a virtual bucket. Inside the bucket are a number of keys. No process can be executed without first having a key. So, if a specific process requires a key, only a finite number of occurrences of that process can be in progress simultaneously. All others must wait until a key is returned to the virtual bucket.

The NTB-OV system functions you use to create and manage semaphores are `alloc`, `semaphore_get`, and `semaphore_put`.

Allocating Semaphores

To allocate a semaphore, use the `alloc()` system function. The syntax is:

```
function integer alloc(SEMAPHORE, integer semaphore_id,
integer semaphore_count, integer key_count);
```

`semaphore_id`

> is the ID number of the semaphore being created. It must be an integer value. You should generally use 0, because then NTB-OV automatically generates the ID for you. Using any other number explicitly assigns an ID to the semaphore being created.

`semaphore_count`

> specifies how many semaphore buckets you want to create. It must be an integer value.

`key_count`

> specifies the number of keys initially allocated to each semaphore bucket you are creating. The number of keys in the bucket can increase if more keys are put into the bucket than are removed. Therefore, `key_count` is not necessarily the maximum number of keys in the bucket.

The `alloc()` function returns the base semaphore ID if the semaphores are successfully created. Otherwise, it returns `0`.

Obtaining Semaphore Keys

To obtain keys from a semaphore, use the `semaphore_get()` system function. You can use `semaphore_get()` as a system function and or a system task. The syntax is:

```
function integer semaphore_get(NO_WAIT | WAIT,
integer semaphore_id, integer key_count);
```

`NO_WAIT`

this predefined macro means continue code execution even if there are not enough keys available.

`WAIT`

this predefined macro means suspend the process until there are enough keys available, and then continue execution.

`semaphore_id`

specifies which semaphore to get keys from.

`key_count`

specifies the number of keys to take from the semaphore.

When the `semaphore_get()` function is called, it checks the specified semaphore for the number of required keys:

- If enough keys are available, a 1 is returned and execution continues.
- If not enough keys are available, a 0 is returned and the process is suspended depending on the wait option.

The semaphore waiting queue is FIFO based. By default, a process waits at a semaphore without timing out.

When you allocate multiple semaphores, you access the *N*th semaphore using this method:

```
semID=alloc(SEMAPHORE, 0, 4, 2);
if (semaphore_get(WAIT, semID+2, 1))
     printf("The semaphore was successful.");
```

This example allocates four semaphores with IDs 0 to 3, each with two keys. The example then checks to see if there is a key in the third semaphore; if there is, it prints a message.

Returning Semaphore Keys

To return keys to a semaphore, use the `semaphore_put()` system task. The syntax is:

```
task semaphore_put(integer semaphore_id, integer key_count);
```

`semaphore_id`

specifies which semaphore to return the keys to.

`key_count`

specifies the number of keys being returned to the semaphore.

When the `semaphore_put()` system task is called, the specified number of keys is returned to the semaphore. If a process is suspended to wait for a key, that process resumes execution when enough keys are returned.

Semaphore Example

Example 5-12 shows how to use semaphores to prevent conflicts between threads.

Example 5-12 Using Semaphores to Prevent Thread Conflicts

```
class gen {
     local reg[2:0] bit_field_a;
     static integer gu=alloc(SEMAPHORE, 0,1,1);
     task m1(){
          printf("The value of guard %0d, in gen\n", gu);
      }
```

```
    task new(){
        bit_field_a = random();
    }
}

program test {
    integer i, j;
    gen g1, g2, g3;

    printf("This is program test beginning.\n");
    g1=new();g2=new();g3=new();
    printf("The gnd in main %0d %0d %0d\n", g1.gu, g2.gu,g3.gu);
    fork {
        @(posedge CLOCK);
        semaphore_get(WAIT, g1.gu, 1);
     printf("This is g1 semaphore_get at cycle %0d\n", get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g1.gu, 1);
     }
    {
        @(posedge CLOCK);
        semaphore_get(WAIT, g2.gu, 1);
       printf("This is g2 semaphore_get at cycle  %0d\n",get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g2.gu, 1);
    }
    {
        @(posedge CLOCK);
        semaphore_get(WAIT, g3.gu, 1);
       printf("This is g3 semaphore_get at cycle %0d\n", get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g3.gu, 1);
     }
    join all
}
```

Example 5-12 creates class `gen` and allocates a single semaphore with one key inside the class declaration. The main program instantiates three instances of the `gen` class: `g1`, `g2`, and `g3`. But there is only one semaphore because of the static declaration in the semaphore allocation.

Next, the main program forks off three separate threads. Each thread tries to get a key from the semaphore. If a thread cannot get a key, it waits until a one is available. Once it gets the key, the thread prints a message, advances the simulation clock, and returns the key to the semaphore. This way, each thread is suspended until a semaphore key is available. You can use this technique to prevent conflicts between threads.

Mailboxes

You use mailboxes to exchange messages between processes. You can send data to a mailbox with one process and retrieve it with another.

Conceptual Overview

NTB-OV mailboxes work like real mailboxes. When a letter is delivered and put into a mailbox, you can retrieve the letter (and any data stored within). But if the letter has not been delivered when you check the mailbox, you can either wait for the letter or get it on a subsequent trip to the mailbox. Similarly, NTB-OV mailboxes allow you to transfer and retrieve data in a controlled manner. The mailbox system functions are `alloc()`, `mailbox_put()`, and `mailbox_get()`.

Mailbox Allocation

To allocate a mailbox, use the `alloc()` system function. The syntax is:

```
function integer alloc(MAILBOX, integer mailbox_id, integer mailbox_count);
```

`mailbox_id`

is the ID number of the mailbox being created. It must be an integer value. You should generally use 0 because then NTB-OV automatically generates a mailbox ID for you.

`mailbox_count`

is the number of mailboxes you want to create. It must be an integer value.

The `alloc()` function returns the base mailbox ID if the mailboxes are successfully created; otherwise, it returns 0. The maximum number of mailboxes that can be created is determined by `vera_mailbox_size`.

Sending Data to a Mailbox

Use the `mailbox_put()` system task to send data to a mailbox. The syntax is:

```
task mailbox_put(integer mailbox_id, any_scalar_type data);
```

`mailbox_id`

is the destination mailbox.

`data`

is any general expression that evaluates to a scalar type.

Note:

The term scalar is short for the following data types: integer, enum, reg, reg[ ], string, or user-defined class.

The `mailbox_put()` system task stores data in a mailbox in a FIFO manner. Note that when passing objects, only object handles are passed through the mailbox.

Retrieving Data from a Mailbox

Use the `mailbox_get()` system function to retrieve data stored in a mailbox. The syntax is:

```
function integer mailbox_get (NO_WAIT | WAIT | COPY_NO_WAIT | COPY_WAIT,
integer mailbox_id [, data dest_var [, CHECK]]);
```

`NO_WAIT`

This predefined macro dequeues mailbox data if it is available. Otherwise, it returns an empty status (`0`).

`WAIT`

This predefined macro suspends the calling thread until data is available in the mailbox, and then dequeues the data.

`COPY_NO_WAIT`

This predefined macro copies mailbox data without dequeuing it if it is available. Otherwise, it returns an empty status (`0`).

`COPY_WAIT`

This predefined macro suspends the calling thread until data is available in the mailbox, and then copies the data without dequeuing it.

Note:

The `NO_WAIT`, `WAIT`, `COPY_NO_WAIT`, and `COPY_WAIT` arguments to `mailbox_get()` are macro constants. The `mailbox_get()` system function expects its first argument to be a compile-time constant.

`mailbox_id`

is the mailbox to retrieve data from.

`dest_var`

is the variable to hold the retrieved data.

`data`

The term scalar is short for the following data types: integer, enum, or reg. The `data` parameter can be either a scalar or a handle to an object.

`CHECK`

optionally specifies whether type checking occurs between the mailbox data and the destination variable.

The `mailbox_get()` system function assigns any data stored in the mailbox to the destination variable and returns the number of entries in the mailbox, including the entry just received.

- If there is a type mismatch between the data sent to the mailbox and the destination variable, a runtime error occurs unless the `CHECK` option is used.
- If the `CHECK` option is active, a -1 is returned, and the message is left in the mailbox.
- If the mailbox is empty, the function waits for a message to be sent, depending on the wait option. If the wait option is `NO_WAIT`, the function returns a `0`.
- If no destination variable is specified, the function returns the number of entries in the mailbox, but it does not dequeue an item from the mailbox. You can use this feature to continue generating mailbox entries until a specified number are generated, as shown in Example 5-13.

Example 5-13   Using mailbox_put to Put Random Numbers in a Mailbox

```
mboxID=alloc(MAILBOX, 0, 1);
while (mailbox_count <11)
{
      mb_data=random();
      mailbox_put(mboxID, mb_data);
      mailbox_count=mailbox_get(NO_WAIT, mboxID);
}
```

This example generates random numbers and puts them in the mailbox. The loop continues while the number of entries is less than 11.

Mailbox Example

Example 5-14 shows how to use mailboxes in NTB-OV:

Example 5-14   Mailbox Usage Example

```
mboxID=alloc(MAILBOX, 0, 1);
fork {
repeat(256)     {
randomVar=random();
            address0=randVar[17:0];
            @1 memsys.request[0]=1'b1;
            @2,20 memsys.grant==2'b01;
            data0=randVar[7:0];
            writeOp(address0, data0);
            mailbox_put(mboxID, {address0, data0});
            @1 memsys.request[0]=1'b0;
            @2,20 memsys.grant==2'b00;
            random_wait();
      }
}
{     repeat(256)
      {     mailbox_get(WAIT, mboxID, message, CHECK);
            address1=message[15:8];
            data1=message[7:0];
            @1 memsys.request[1]=1'b1;
            @2,20 memsys.grant==2'b10;
            readOp(address1,data1);
            @1 memsys.request[1]=1'b0;
            @2,20 memsys.grant==2'b00;
            random_wait();
```

```
        }
    }
    join
```

Example 5-14 allocates a mailbox before the `fork`. The first thread then randomly assigns values to `address0` and `data0`. The data is passed through a mailbox to the second thread, which is waiting for the data. That data is then read into `message` and used for the `readOp` call.

6

Interfaces and Signal Operations

The interface is an important component of the OpenVera Native Testbench (NTB-OV) environment. An interface declaration specifies a set of testbench signals that connect the testbench to internal nodes or design ports. An interface may also specify a clock. You define the interface separately from the program block.

This chapter explains the syntax and semantics of NTB-OV interfaces and signal operations in the following major sections:

- Interfacing to the Design Under Test
- Signal Operations
- Retrieving Signal Properties

Interfacing to the Design Under Test

NTB-OV interfaces provide an autonomous domain that simplifies management of signal connections to the design you are testing. Interfaces separate timing-related information from the functional information in the body of your NTB-OV program, thus enabling you to develop clear, concise testbench programs. Because the timing for sampling and driving the testbench signals is usually relative to the clock declared in the interface, grouping the testbench signals with a clock in an interface allows you to drive or sample signals within the program without explicitly calling a clock or specifying timing.

Your interface specification can group signals by clock domains for multiclock designs. There is no limit to the number of interface declarations you can create.

Interface Declaration

You use an interface specification to group NTB-OV signals by clock domain. Each interface can include only one input signal of type CLOCK. NTB-OV samples and drives non-clock design signals on the edges of this clock. If you don't specify an input signal of type CLOCK, NTB-OV synchronizes the interface signals using its SystemClock.

Figure 6-1 shows the syntax for interface declarations, including the syntax for port-connected interface signals and directly connected (HDL) signals.

Figure 6-1 The NTB-OV Interface

```
interface interface_name
{
    signal_direction [signal_width] signal_name signal_type
        [skew] [depth value][hdl_node "hdl_path"];

}
```

Port-connected interface signals cause the program to have a corresponding escaped port name. For example:

```
\interface_name.signal_name
```

Note that you must add a space after the port name in the Verilog code.

Interface Signal Declarations

This section explains the properties of:

- Port-connected Interface Signals
- Making Direct HDL Node Connections
- Interface Signal of Type CLOCK

Port-connected Interface Signals

You use port-connected interface signals to connect port-level signals to your testbench. The syntax is:

```
signal_direction [signal_width] signal_name signal_type [skew][depth];
```

`signal_direction`

specifies the direction of the signal from the perspective of the testbench, not from the perspective of the design under test (see Figure 6-2 on page 153).

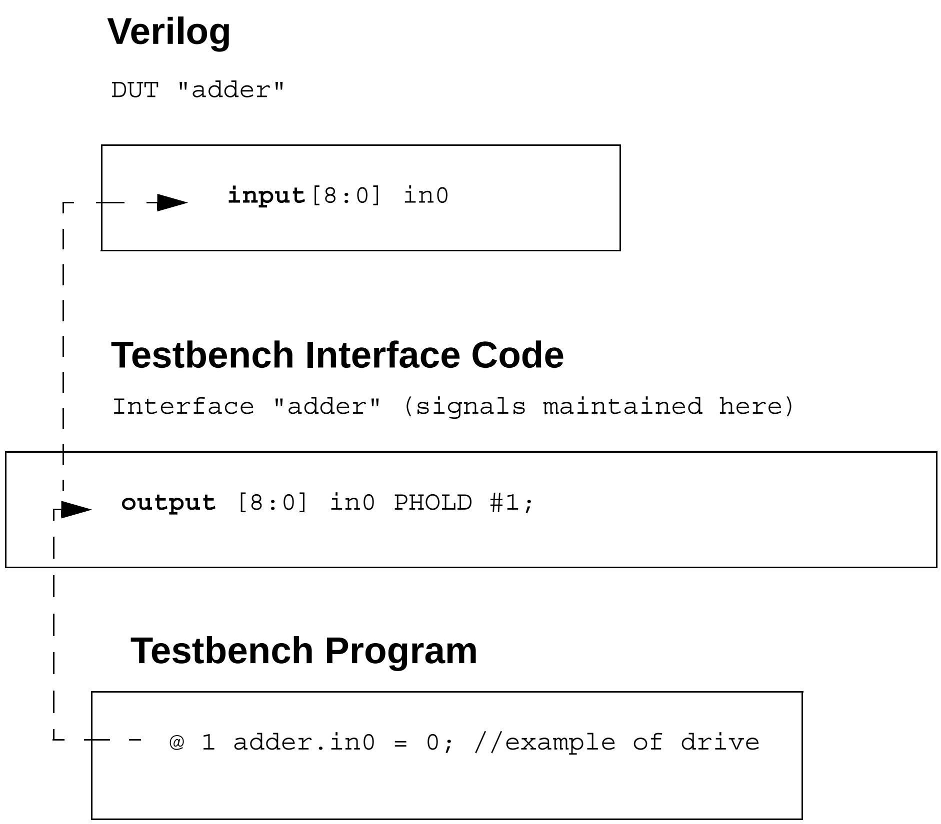
- `input` indicates that the signal goes from the design under test to the testbench.

- `output` indicates that the signal goes from the testbench to the design under test.

- `inout` specifies a bidirectional signal. Bidirectional signals have two `signal_types` and an optional `skew` for each `signal_type`. For example:

```
inout[signal_width] signal_name input_signal_type
     [skew][depth] output_signal_type [skew][depth];
```

Figure 6-2 Representing signal in0 in the Design, Interface, and Testbench Program



`signal_width`

is a bit vector specifying the width of the signal. It must be in the form `[msb:0]`.

`signal_name`

identifies the signal being defined. This is the testbench name for the HDL signal that is a port member of the design.

`signal_type`

The valid signal types and their definitions are listed in Table 6-1.

Table 6-1 NTB-OV Signal Types

Signal Type	Operation
NHOLD	Output signal is driven on the negative edge of the interface clock.
PHOLD	Output signal is driven on the positive edge of the interface clock.
PHOLD NHOLD	Output signal is driven on both edges of the interface clock.
NR0 NR1 NRX NRZ	Signal is driven on the falling edge of the interface clock for 1 cycle. Then it returns to value, which can be 0, 1, X, or Z.
PR0 PR1 PRX PRZ	Signal is driven on the rising edge of the interface clock for 1 cycle. Then it returns to value, which can be 0, 1, X, or Z.
NSAMPLE	Input signal is sampled (evaluated) on the negative edge of the interface clock.
PSAMPLE	Input signal is sampled (evaluated) on the positive edge of the interface clock.
PSAMPLE NSAMPLE	Input signal can be sampled on both edges of the interface clock. (DDR signal type.)
PSAMPLE PHOLD	Inout signal is sampled and driven on the positive edge of the interface clock
NSAMPLE NHOLD	Inout signal is sampled and driven on the negative edge of the interface clock.
PHOLD NHOLD PSAMPLE NSAMPLE	Inout signal can be sampled and driven on both edges of the interface clock. Double-data rate (DDR) signal type.
CLOCK	Specifies the clock to which the interface signals synchronize.

Unidirectional signals can be either double data rate(DDR) (for example, PSAMPLE PHOLD) or non-DDR (for example, PHOLD) or double signal types.

For a DDR signal designated as PHOLD NHOLD, NTB-OV can drive the signal at both the positive and negative edges of the interface clock.

Input signals are only sampled, and output signals are only driven. A bidirectional signal can be both sampled and driven.

Here are some example signal declarations with various signal types:

```
input clk CLOCK;
input [31:0] address PSAMPLE #-1;
inout [7:0] data PSAMPLE #-1 PHOLD #1;
output rdy PHOLD #1;
output sig_out PHOLD NHOLD #1;
inout [7:0] data PSAMPLE NSAMPLE #-1 PHOLD NHOLD #1;
```

Note:

If you want to be able to sample an output signal, declare it as an `inout` signal type.

`skew`

The *`skew`* must be an integer value; it determines how long before the clock edge the signal is sampled, or how long after the clock edge the signal is driven. You define the `skew` units in terms of the timescale of the Verilog design. It is an error to specify a `skew` on a signal of type CLOCK. The `skew` must be in this format:

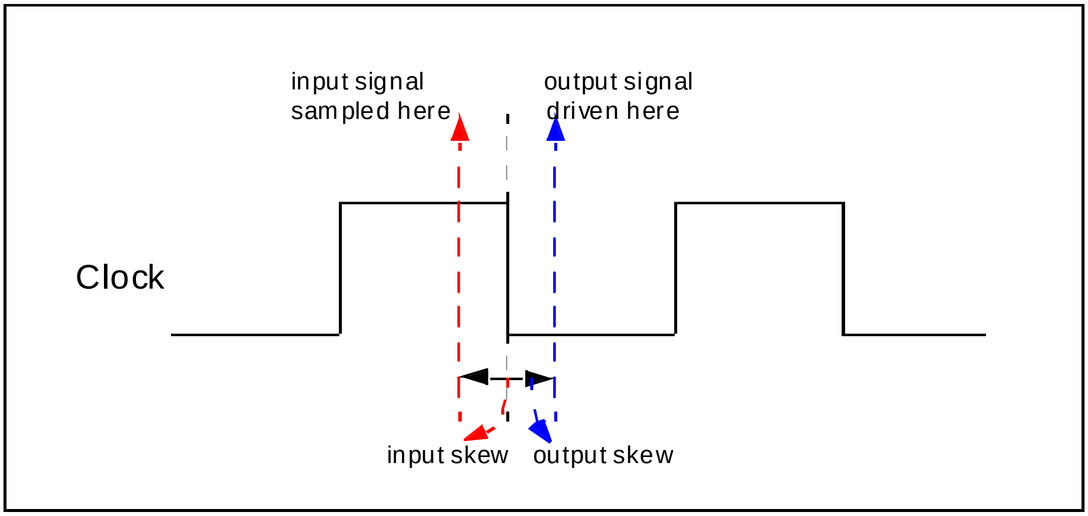
```
#value
```

Output signal types only take positive `skew` values. Input signal types only take negative `skew` values. An inout signal should have a positive `skew` when driven and a negative `skew` when sampled.

Best Practice

You should use a `skew` of at least -1 for all signal sampling, and +1 for all signal driving. This ensures that you retrieve the value before (for sampling), or after (for driving) the clock changes.

Figure 6-3 Driving and Sampling on the Negative Edge of the Clock



For DDR signals, NTB-OV uses a single `skew` for all transactions. This means that synchronous (positive edge and negative edge) and asynchronous transactions use the same `skew` to schedule the transaction. So more than one `skew` specified in the interface input or output signal results in an error. For example:

```
output outp PHOLD #1 NHOLD;
output outp PHOLD #1 NHOLD #2; // results in an error
```

The behavior of existing signal types is not affected by these new signal types and semantics based on synchronized edges.

`depth`

The *depth* value must be a non-negative integer. The default is 0. Input signals are typically sampled on the synchronized edge in the current cycle. However, an input signal can be sampled on a previous synchronized edge. To reference such a signal value, you must specify the signal depth in the signal declaration. The `depth` value specifies the number of synchronized edges that are stored for back-reference. If you want to reference a signal three edges back, this value must be at least 3. You cannot specify a `depth` value for an output signal.

Here is an example of an input signal in the interface where the signal is sampled with an input `skew` of -1, and a `depth` of 5:

```
input [7:0] data PSAMPLE #-1 depth 5;
```

The syntax to back reference such a signal is:

```
interface_name.signal_name.N
```

`interface_name`

is the name of the interface in which the signal is defined.

`signal_name`

is the name of the signal being referenced.

`N`

specifies how many previous synchronized edges to look back when evaluating the signal. The synchronized edge is 0, the previous synchronized edge is 1, and so on. If no signal depth is specified, the default is the synchronized edge.

Here is an example of how to use a back reference to sample a signal value two synchronized edges back:

```
arb.data.2; //interface_name is "arb," signal_name is "data"
```

Making Direct HDL Node Connections

You can connect an interface signal to any signal in the design using the `hdl_node` option followed by the `hdl_path` argument. This is a handy way to monitor and drive internal design signals.

hdl_node "*hdl_path*"

Use the `hdl_path` argument with the `hdl_node` option to specify the full path to a design signal. Enclose the path in double quotes. You can specify any form of cross-module reference (XMR) that VCS supports; for example, absolute path, relative path, or *module_name*.x.y. Concatenations in `hdl_path` are supported, but bit and part selects are not supported. Here are some example `hdl_node` declarations:

```
input[31:0] grant PSAMPLE #-2 hdl_node "sys.cpu2.p0_d1";
      /* sys is the top-level Verilog module, and cpu is the DUT */

output request PHOLD hdl_node "sys.arb.p0_strb";
      /* sys is the top-level Verilog module, and arb is the DUT */
```

Direct Clock Connection

By default, the shell has an extra port for the SystemClock signal. The following declaration creates a direct connection for SystemClock and removes the SystemClock port from the shell. Put this top-level declaration in the program file. The syntax for the declaration is:

```
hdl_node CLOCK "hdl_path";
```

You can connect the SystemClock to any `hdl_path`, but it is usually connected to an existing interface clock. Example 6-1 shows how to connect the SystemClock to the `my_ifc_clk` shell port. Note that shell ports follow an *`interfacename_signalname`* naming convention. If you don't use the `NTB-OV_opts vera_portname` compilation option, precede the *`interfacename_signalname`* with a backslash (\); for example, `output [7:0] \adder.in0;`.

Example 6-1 Connecting the SystemClock

```
task random_wait () {
printf ("Hello World!\n");
}

program p1 {

        @1 my_ifc.io=1'b1;
        random_wait();
};
```

Interface Signal of Type CLOCK

Each NTB-OV interface can have only one input signal of type `CLOCK`. If you don't specify a `CLOCK`, NTB-OV synchronizes interface signals using SystemClock. SystemClock is available inside the testbench program using a `CLOCK` variable. For the above-defined operation, the SystemClock port of the testbench program must be connected to a clock signal in the top-level module, where the testbench program and design under test are instantiated. The syntax for a `CLOCK` declaration is:

```
input clock_name CLOCK;
```

All other signals defined in an NTB-OV interface are governed by this clock. NTB-OV samples and drives interface signals on the specified edge of this clock. Example 6-2 shows how to instantiate your NTB-OV testbench program and design under test in a top-level module.

Example 6-2   Instantiating Testbench Program and Design Under Test

File test.vr

```
interface dff_int {
      input q PSAMPLE #-1;
      output d PHOLD #2;
      input clk CLOCK;
}

program dff_test {
      @5 dff_int.d = 1;
      @1 dff_int.q == 1;
      printf("%d\n",dff_int.q);
}
```

File top.v

```
module dff_top;
parameter simulation_cycle = 100 ;
reg  SystemClock ;
wire  clk ;
wire  d, q ;
assign  clk = SystemClock ;

dff_test test(
.\dff_int.clk (clk),
.\dff_int.q (q),
.\dff_int.d (d),
.SystemClock(SystemClock)
);

dff dut(
.clk ( clk ),
.d ( d ),
.q ( q )
);
initial begin
SystemClock = 0 ;
      forever begin
      #(simulation_cycle/2)
        SystemClock = ~SystemClock ;
       end
      end
endmodule
```

Virtual Ports

NTB-OV `port` variables are user-defined constructs that contain groups of port signal members organized into virtual ports. When you assign virtual ports to interface signals, NTB-OV samples and drives the virtual port signals, which in turn sample and drive the corresponding interface signals. The syntax to declare a **`virtual port`** is:

```
port virtual_port_name {
      port_signal_member_name1;
      ...
      port_signal_member_namen;
}
```

`virtual_port_name`

is a user-defined virtual port name.

`port_signal_member`

is a member of the user-defined virtual port.

Port variables are handles to port instances in the same way that class handles are handles to class instances. You must initialize port variables before you can use them because they have **`null`** values prior to initialization. You can initialize port variables in three different ways:

- Assigned a bind
- Assigned a new port instance
- Assigned a previously initialized port variable

You must also initialize (assign) individual port signal members of a port variable before you use them. You can assign these interface signals to port signal members using the `bind` construct or `signal_connect` to the interface or port signal (static signal connect).

In all instantiations of the port, port signals must be bound to signals of the same size and direction or left unbound (`void`). This applies to `signal_connects` as well. The approach shown in Example 6-3 is not supported and results in a compilation warning.

Example 6-3 Unsupported Port Signal Assignments

```
interface intf {
     input  [4:0] sig1 PSAMPLE;
     output [4:0] sig2 PHOLD;
}

port p {
     sig1;
}
bind p b1 {
sig1  intf.sig1[1:0];  // size is 2
bind p b2 {
     sig1 intf.sig1[2:0];  // size is 3
}
```

The following assignments are not supported because of direction mismatches.

```
bind p b1 {
     sig1 intf.sig1;  // direction of intf.sig1 is input
}

bind p b2 {
     sig1 intf.sig2;  // direction of intf.sig2 is output
}
```

The bind Construct

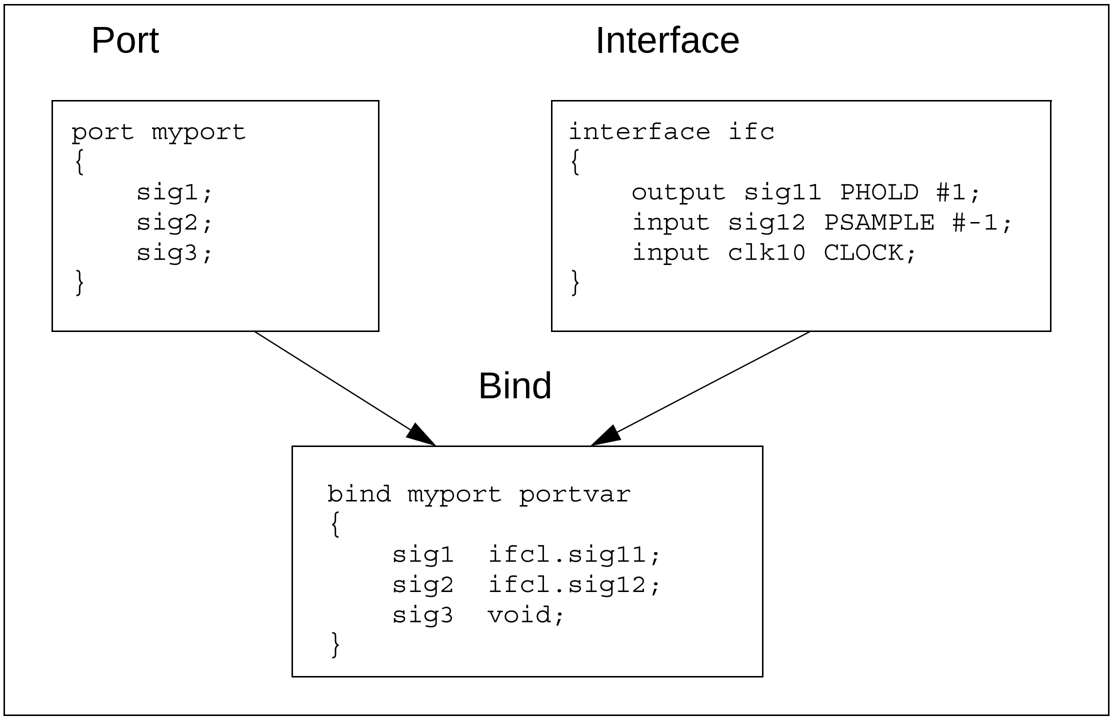
You can use the NTB-OV `bind` construct as a convenient shortcut for associating groups of port signal members with interface signals.

The NTB-OV `bind` is a top-level construct in the testbench program that does several things:

- Creates an instance of the virtual port.
- Initializes the port signal members of the instance with the interface signals given in the bind.
- Defines a global port variable that references the instantiated virtual port. This variable persists throughout the simulation.

Figure 6-4 shows how `bind` declarations work.

Figure 6-4   Creating a bind Declaration



The syntax for declaring a `bind` is:

```
bind virtual_port_name port_variable {
     port_signal_memberN interface_name.signal_name;
}
```

`virtual_port_name`

is a user-defined virtual port name.

`port_variable`

must be a valid identifier.

`port_signal_member`

is a member of the user-defined virtual port.

`interface_name`

is a name of an interface declaration.

`signal_name`

is a valid interface signal. You can specify subfields using *signal_name*[*x:y*].

void Bindings

You can initialize a subset of port signal members, thereby choosing to connect some or all of the signals within a `bind` declaration. When you don't connect a port signal member, you must use a `void` declaration within the bind, as shown in the following example:

```
bind port_name port_variable {
     port_signal_member void;
}
```

Using the `void` construct results in a port signal member that has no interface signal assigned to it.

Concatenations

Within a `bind`, you can initialize a port signal member using a concatenation of several interface signals. The syntax is:

```
port_signal_member {sigspec1,..,sigspecn};
```

Each *sigspec* is of the form:

```
interface.signal
```

-or-

```
interface.signal[upperbound:lower bound]
```

-or-

```
interface.signal[reg]
```

Example 6-4 shows some example concatenations.

Example 6-4   Initializing Port Signal Members with Concatenated Signals

```
port my_port { // port declaration
                i1;
     o1;
}

bind my_port my_portvar { // bind declaration
                i1 {my_ifc.in1[3:0],my_ifc.in2[3:0]};
     o1 {my_ifc.out1[3:0],my_ifc.out2[3:0]};
}
```

Note how the `i1` signal references two different signals from the `my_ifc` interface. These two input signals (`in1[3:0]` and `in2[3:0])` must have the same clock edge and skew.

All signals specified within the concatenation must come from the same interface. The following is illegal because two different interfaces are used:

```
bind my_port bad {
 i1 {interface1.sig1, interface2.sig2};
 o1 void;
}
```

Subfields

When using subfield binding, you can specify further subfields, as shown in the following example:

```
bind myport portvar {
     i1 infcl.sigl1[7:5];
}
```

This example assigns a three-bit subfield to the `i1` port signal. You can specify a subfield within that field as well:

```
portvar.$i1[2:1]
```

This subfield of `i1` corresponds to `infc1.i11[7:6]`.

Connecting Signals

You can also use the `signal_connect()` system function to connect signals. The syntax is:

```
signal_connect(portvar.sig, target.sig);
```

`portvar.sig`

references the signal you want to map. It must be a member of an instantiated port variable.

`target.sig`

references the target signal to which the `portvar.$sig` signal is connected. It can be an interface signal or a port variable, as shown in the following examples:

```
signal_connect(portvar.$sig1, dff.q);
// dff is the interface name
signal_connect(portvar.$sig2, portvar2.$q);
// portvar2 is the port name
```

Creating a New Port Instance

To create a new port instance, declare a port variable and instantiate it using the `port_variable = new` construct. Example 6-5 shows how to declare and instantiate a port variable `portvar` of port type `my_port`:

Example 6-5   Port Variable Declaration and Instantiation

```
task my_task() {
      my_port portvar; // declare variable portvar
      portvar = new();
}
```

The `portvar` port variable is now instantiated, but the port signal members of the new instance do not yet refer to any interface signals. See the following sections for different ways to make that connection.

Assigning an Existing Port Variable

You can assign a port variable to another port variable as long as the variables have the same user-defined port type, as follows:

```
portvar1 = portvar2;
```

After the assignment, both port variables refer to the same port instance. So if you use `signal_connect()` to modify the connections in *portvar1*, those changes are also visible in *portvar2*. For example, if *portvar1.$sig1* is connected to *myifc.sig1*, then *portvar2.$sig1* is also connected to *myifc.sig1*. Figure 6-5 shows how to declare and assign a value to a port variable.

Figure 6-5   Declaring and Assigning a Value to the Port Variable

Virtual Port

```
port myport
{
    sig1;
    sig2;
}
```

Bind

```
bind  myport  p1
{
   sig1 ifcl.sigl1;
  sig2 ifcl.sig12;
}
```

Declare variable of type myport: `myport portvar;`

Assign a bind of type myport: `portvar = p1;`

In Figure 6-5, `p1` is a global port variable that refers to an instance of `myport`. After assigning `p1` to `portvar`, `portvar` references the same instance of `myport`.

Note:

It is legal to assign `null` to a port variable. This resets the variable to its uninitialized state.

Copying an existing port variable

You can also assign a port variable a new copy of another port variable as long as the variables have the same user-defined port type as follows:

```
portvar1 = new portvar2;
```

This creates a new port instance that is a copy of the port instance referred to by *`portvar2`*.

Signal Operations

This section explains the four NTB-OV primitive statements that operate on interface signals: synchronization, drive, sample, and expect. This section also explains synchronous and asynchronous signal operations. These topics are covered in the following sections:

- Synchronization
- Driving a Signal
- Sampling a Signal
- Synchronization Edge Semantics
- The expect Event
- Asynchronous Signal Operations
- Subcycle Delays

Synchronization

Synchronization involves a signal in an NTB-OV interface being synchronized to the interface clock. You use the (`@`) operator to explicitly synchronize a signal. This means that you are synchronizing to the signal changing value. The syntax is:

```
@([specified_edge] interface_signal);
```

`specified_edge`

is the edge at which the synchronization occurs. The value can be `negedge`, which specifies a negative or falling edge of the interface signal, or `posedge`, which specifies a positive or rising edge of the interface signal. If no edge is specified, the synchronization occurs on the next change in the specified signal.

`interface_signal`

is the signal to which the synchronization is linked. It can be any signal of type input or inout in an interface declaration or `CLOCK`. The interface signal can be any subfield of a signal as well. If you specify `CLOCK`, synchronization is tied to the SystemClock.

Note:

Since the interface signal is of type input or inout, NTB-OV always samples the value at the edge of the clock to which the signal is bound. This implies that the signal value changes at the edge of the clock. To synchronize when the signal changes on the design side, use the `async` modifier. (For more information, see async Modifier.)

You can use the `or` keyword to specify multiple interface signals. If you specify more than one signal, the synchronization occurs on the next change of any of the listed signals. Here are some example synchronization statements:

- In this example, the synchronization occurs on the next change of the `ack_l` signal:

```
@(ram_bus.ack_l);
```

- The second example synchronizes to the SystemClock.

```
@(CLOCK);
```

- This next example synchronizes to the positive edge of the interface clock, `ram_bus.clock`.

```
@(posedge ram_bus.clock);
```

- This last example uses the `or` keyword to specify multiple interface signals. The synchronization occurs on either the next positive edge of `intf.sig1` or any edge of `intf.sig2`, whichever changes first.

```
@(posedge intf.sig1 or intf.sig2);
```

Driving a Signal

Use the NTB-OV drive operator (`=` or `<=`) to set the values of output interface signals. The syntax is:

```
[delay] signal_name range drive_operator expression;
```

`delay`

optionally specifies the number of cycles that pass before NTB-OV drives the signal. Specify the `delay` in the form `@`*n*, where *n* is the number of clock edges. When *`delay`* is not specified, the default is `@0`.

Note:

In NTB-OV, you specify drive delays as integers. If *`n`* is 0, the drive operator does not block.

`signal_name`

is the name of the interface signal being driven.

`range`

specifies which regs of the signal are driven. If no range is specified, the entire signal is driven.

`drive_operator`

must be either `=`, which specifies a blocking drive, or `<=`, which specifies a non-blocking drive.

`expression`

can be any valid expression. Here are some NTB-OV drive examples.

```
foo_bus.data[3:0] = 4'h5; // blocking drive
@1 foo_bus.data <= 8'hz; // non-blocking drive
```

Blocking and Non-Blocking Drives

Blocking drives suspend testbench execution until the statement completes. NTB-OV uses the clock edge (NHOLD, PHOLD, or PHOLD NHOLD for DDR signals) that the drive signal is associated with for counting the HDL cycles during suspension. Once the statement completes, testbench execution resumes.

Non-blocking drives schedule the drive at a future cycle and testbench execution continues. When the specified cycle occurs, the drive is executed (see Example 6-6).

Example 6-6   Blocking and Non-blocking Drives

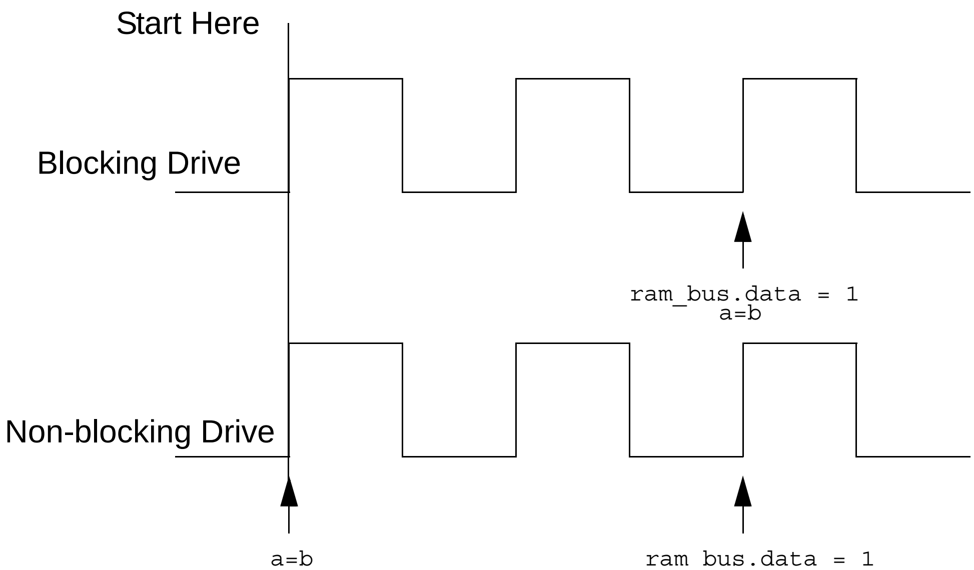
```
// assume sim time is at a clk edge of ram_bus interface
@2 ram_bus.data = 1; // blocking drive
     a = b;
//...
@2 ram_bus.data <= 1; // non-blocking drive
     a = b;
```

Example 6-6 is equivalent to the following:

```
fork
@2 ram_bus.data=1;
join none
a=b;
```

Figure 6-6 illustrates how blocking drives work in NTB-OV.

Figure 6-6 NTB-OV Blocking and Non-blocking Drives



The first block in Example 6-6 is a blocking drive. Two cycles must pass before both lines are executed, as shown in Figure 6-6. The second block is a non-blocking drive. The first line is scheduled to be executed 2 cycles in the future, then the second line is executed.

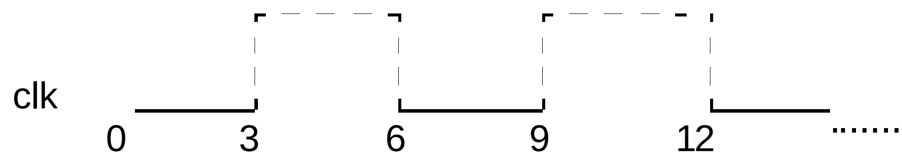
Now, consider interface signal `outp`, which is a DDR signal type:

```
output outp PHOLD NHOLD;
```

Blocking drives wait for the synchronization edge of the interface clock. Synchronization for `outp` is on the positive or negative edge of the interface clock, whichever comes first.

Figure 6-7 shows the `outp` interface clock (clk).

Figure 6-7   Interface clk Waveform



The following example shows a 0-delay blocking drive on `intf.outp` that comes at time 0.

```
program test {
      intf.outp = 1'b1;
}
```

The next synchronization edge is a positive edge of `clk` at time 3 ns. Therefore, the above drive occurs at 3 ns. If the above blocking drive comes at 4 ns:

```
program test{
      delay (4);
      intf.outp = 1'b1;
}
```

then the next synchronization edge occurs at 6 ns (negative clock edge). The drive occurs at 6 ns. Now, consider the following blocking drive that specifies a delay. The drive comes at 0 ns.

```
program test {
      @1 intf.outp = 1'b0;
}
```

Since the drive specifies a delay of 1, NTB-OV skips the first synchronization edge (posedge at 3 ns) and schedules the drive on the next synchronization edge (negative edge) at 6 ns.

Note that the drive is no longer scheduled for 1 cycle, but is scheduled for the next synchronized edge after hitting the appropriate synchronization edge (PHOLD or NHOLD, whichever comes first).

As far as the scheduling and synchronization edge is concerned, non blocking drives are no different from blocking drives for the DDR signal types.

Sampling a Signal

Sampling assigns the value of a signal to a variable. The sampling syntax is:

```
variable = signal_name;
```

The *signal_name* can be an interface signal or port signal, but it must be an `input` or `inout` signal. NTB-OV samples the signal at the next sampling point (as specified in the interface definition) and assigns the value to the variable. You cannot use the delay attribute (`@`) when sampling a signal. Remember that you can sample subfields within the signal by specifying a specific subfield in the signal width.

NTB-OV synchronously samples signals in an assignment when the RHS is of the form `x=signal` (including portvars) or `x=unary-op(signal)`. NTB-OV uses the last synchronously sampled value for all other forms.

You cannot use output interface signals in the RHS of an expression because they cannot be sampled. In particular, you cannot use output interface signals in the RHS of expressions with `sscanf()`, `fprintf()`, `sprintf()`, `psprintf()`, or `printf()`.

Synchronization Edge Semantics

For PSAMPLE, NSAMPLE, PHOLD, and NHOLD signal types, the synchronization edge is either the positive or negative edge of the interface clock. For DDR signal types, the synchronization edge is both the positive and negative edge of the interface clock.

The expect Event

The NTB-OV **expect** event asserts that a given signal has a given value at a given time. There are several forms of the expect primitive:

- Simple expect: `@, operator`
- Full expect: `@@, operator`

The general syntax for an expect statement is:

```
expect_operator [delay], [window] expect_list;
```

`expect_operator`

must be either `@`, which specifies a simple expect, or `@@`, which specifies a full expect. Each form is discussed in detail in subsequent sections.

`delay`

specifies the number of synchronized edges that must pass before the signal is evaluated. Use the form `@`n, where *n* is the number of cycles. For immediate checking, use a `delay` value of 0 cycles (that is, `@0`). In the following example, a `delay` of `@2` in the drive statement causes the simulator to schedule the drive two synchronized edges after the drive statement is encountered. That is, the drive is scheduled to occur on the third synchronization edge after the drive statement is encountered.

```
@2 intf.outp = 1;
```

`window`

specifies how long the check is made. Use the form `@ ,m` where *m* is the number of synchronized edges for which the check is made. When *delay* is specified `(@ n,m)`, the number of edges starts from the `delay`. An expect with a defined `window` is called a floating expect.

`expect_list`

is any number of expressions (separated by the `or` keyword, or commas, which is equivalent to the AND operation) using any binary operator (except for `<=`).

Note:

You cannot mix comma-separated expression lists with `or` lists.

The general form is *`signal_name`* *operator expression*. Signal names must be interface signals, but they can include subfields. If you declare multiple expressions, tie them to the same edge of the same clock; otherwise you get an error.

Note:

The expect primitive is a blocking primitive. It blocks until the expect is satisfied or a simulation error is generated. There is no non-blocking form of the expect primitive.

Simple expect

The simple expect checks that a given signal has a specific value at a given time. The syntax is:

```
@ [delay],[window] expect_list;
```

If the signal value does not match the expression when the check is made, a simulation error is generated. If a subfield within the signal is specified, all other regs in the signal are ignored and only those specified are checked against the expression.

You can define multiple expressions in the `expect_list`. If you separate multiple expressions with commas, the expect is satisfied when all of the conditions are satisfied at the time of the check.

You can also separate expressions using the `or` keyword. With this syntax, the expect is satisfied if any of the conditions are satisfied at the time of the check.

Note:

> You cannot mix comma-separated expression lists with `or` lists.

If you specify a `window`, this makes it a floating expect, meaning that the check is made only for the duration of the specified `window`. If the signal value matches the expression within the `window`, the expect is satisfied. If the signal value does not match the expression within the window, a simulation error is generated.

If you separate multiple expressions with commas, the expect is satisfied when all of the conditions are satisfied simultaneously. If you separate expressions with the `or` keyword, the expect is satisfied as soon as any of the conditions is satisfied.

Here are some expect statement examples:

- This first example expects the signal data to be equal to `0101` after 1 cycle. Notice that a `delay` is specified, whereas `window` is not.

```
@1 bus.data[7:4] == 4'b0101;
```

- The second example expects the signal data to be equal to `0101` within `2` cycles. Notice that a `window` is specified, whereas `delay` is not. The default for an unspecified delay is 0.

```
@,2 bus.data[7:4] == 4'b0101;
```

- The third example expects the signal data to be `0010` and the signal `addr` not to be `0001` after `1` cycle.

```
@1 bus.data == 4'b0010, bus.addr != 4'b0001;
```

- The fourth example expects the signal data to be `0010` and the signal `addr` not to be `0001` within `20` cycles, after a `2` -cycle delay.

```
@2,20 bus.data == 4'b0010, bus.addr != 4'b0001;
```

- This last example expects the signal data to be `0010` or the signal `addr` to be `0001` within `20` cycles, after a `2`-cycle delay.

```
@2,20 bus.data == 4'b0010 or bus.addr == 4'b0001;
```

The form "`@,`" yields a parser error. You must specify either a `delay` or a `window`. The degenerative case for an expect with no timing is:

```
bus.data == value;
```

In this case, NTB-OV makes the comparison immediately, as if you had specified:

```
@0 bus.data == value;
```

Full expect

Full expects check that a signal has a specified value over the entire length of a given interval. The syntax is:

```
@@ [delay],window expect_list;
```

Full expects behave just like simple expects with one difference. With a full expect, the signal value must match the expression over the entire duration of the defined `window`. You can specify multiple expressions using comma-separated lists. If all signals do not match during any part of the interval, the expect is not satisfied and a simulation error is generated. Here, are some examples of full expect statements:

- The first example is satisfied if the signal data is not equal to `0101` and the `addr` signal is equal to `88` over the entire interval, `5` to `105` cycles.

```
@@5,100 bus.data[7:4] != b'0101, bus.addr == 8'h88;
```

- The second example is satisfied if the signal `data` is equal to `0101` and the `addr` signal is equal to `88` over the entire `100` cycles.

```
@@100 bus.data[7:4] == b'0101, bus.addr == 8'h88;
```

- The third example is satisfied if the signal `data` is not equal to `0101` and the `addr` signal is equal to `88` over the entire `100` cycles.

```
@@,100 bus.data[7:4] != b'0101, bus.addr == 8'h88;
```

Note:

> Simple expect and full expect differ in how the `delay` attribute is interpreted. Simple expect interprets `@n` as a `delay`, whereas full expect interprets `@@n` as a `window`. Both simple and full expect interpret the `n` in `@,n` and `@@,n` as a window.

Expect and DDR Signals

Expects on signals of the types PSAMPLE NSAMPLE follow scheduling semantics similar to that of the drives. Consider the interface signal `intf.inp`:

```
interface intf {
input inp PSAMPLE NSAMPLE;
}

program test {
      @0 intf.inp == 1'b1;
}
```

The synchronization edge is a positive edge of clk at time 3 ns, so the above sample happens at 3 ns. (See Figure 6-7).

Consider the following example:

```
program test{
      @1 intf.inp == 1'b0;
}
```

Here, a 1-delay expect on `intf.inp` is encountered at time 0. NTB-OV samples the `intf.inp` signal on the second synchronization edge of the clock (that is, at the negative edge) at 6 ns.

Strong and Soft expect

There are two strengths of expects: strong (default) and soft. The syntax to declare a soft expect is:

```
expect_operator delay window expect_list soft;
```

Soft expects do not generate simulation errors when they are not satisfied. You can use the `soft` keyword with any of the expect primitives.

Asynchronous Signal Operations

By default, NTB-OV drives, samples, and expects are relative to a clock edge specified in the interface specification. However, the HDL side of the simulation may be using very detailed timing constructs. NTB-OV provides the `async` and `delay` constructs to allow detailed timing down to the HDL timestep.

async Modifier

The optional async modifier specifies that the operation happen immediately, without waiting for the edge specified in the interface. You can use this modifier with synchronization operators, drives, samples, and expects. The syntax for the `async` modifier is:

Synchronization:

```
@(signal_name async);
```

Drive:

```
signal_name range drive_operator expression async;
```

Sample:

```
variable = signal_name async;
```

Expect:

```
expect_list async;
```

The synchronization construct allows you to act exactly on the current edge rather than waiting for the corresponding sampling edge. The drive, sample, and expect constructs force the operation immediately instead of waiting for the edge specified in the interface.

Note:

> Drive skews specified for the signal in the interface specification also apply to `async` drives. If you need to drive the signal precisely when the drive is issued, do not specify any skew for that signal in the interface specification.

Here are some example `async` statements:

```
@(posedge main_bus.request async);
memsys.data[3:0] = 4'b1010 async;
data[2:0] = main_bus.data[2:0] async;
main_bus.data[7:4] == 4'b0101 async;
```

Subcycle Delays

NTB-OV provides the `delay()` system task to block the testbench while a specified amount of time elapses on the HDL side of the simulation. The syntax for the `delay()` system task is:

```
task delay(integer time);
```

`time`

> specifies the length of the `delay`. It is the same number of time ticks in terms of the timescale of the program.

Example 6-7 synchronizes to the positive edge of `CLOCK` and then advances the simulation time `5` time ticks. The `function1` executes `5` time ticks after the clock edge.

Example 6-7 delay() System Task

```
@(posedge CLOCK);
delay(5);
function1();
...
```

Force and Release Interface Signals

You use the force and release mechanism to override assignments on registers or nets that are connected to the interface signals of Vera.

Note: This feature is not supported in the VCS-NTB flow.

Retrieving Signal Properties

You can use the following functions to return properties of interface signals or statically bound port signals.

vera_is_bound()

returns 1 if a port signal member is non-void bound, and 0 if void bound. The syntax is:

```
function integer vera_is_bound(signal signal);
```

vera_get_name()

returns the name from the interface definition. The syntax is:

```
function string vera_get_name(signal signal);
```

vera_get_ifc_name()

returns the name of the signal´s interface. The syntax is:

```
function string vera_get_ifc_name(signal signal);
```

vera_get_clk_name()

returns the name of the signal´s clock. The syntax is:

```
function string vera_get_clk_name(signal signal);
```

vera_get_dir()

returns one of the following 0 for an input, 1 for an output, and 2 for an inout. The syntax is:

```
function integer vera_get_dir(signal signal);
```

vera_get_width()

returns the number of bits in the signal. The syntax is:

```
function integer vera_get_width(signal signal);
```

vera_get_in_type()

returns the input type as one of the following:

- 0 for NSAMPLE
- 1 for PSAMPLE
- 2 for CLOCK

- 3 for a DDR input/inout

The syntax is:

```
function integer vera_get_in_type(signal signal);
```

vera_get_in_skew()

returns the input skew. The value is always greater than zero. The syntax is:

```
function integer vera_get_in_skew(signal signal);
```

vera_get_in_depth()

returns the depth of the pipeline for sampled values. The syntax is:

```
function integer vera_get_in_depth(signal signal);
```

vera_get_out_type()

returns the output type as one of the following:

- 0 for NDRIVE
- 1 for PDRIVE
- 2 for NHOLD
- 3 for PHOLD
- 4 for NR0
- 5 for NR1
- 6 for NRZ
- 7 for NRX

- 8 for PR0
- 9 for PR1
- 10 for PRZ
- 11 for PRX

The syntax is:

```
function integer vera_get_out_type(signal signal);
```

vera_get_out_skew()

returns the output skew. The value is always greater than zero. The syntax is:

```
function integer vera_get_out_skew(signal signal);
```

7

Aspect Oriented Extensions

Aspect-Oriented Programming (AOP) methodology complements Object Oriented Programming (OOP) methodology using a construct called aspect. Aspect-oriented extensions (AOEs) can affect the behavior of a class or multiple classes. In AOP methodology, the terms aspect and aspect-oriented extension are synonymous.

Aspect-oriented extensions in OpenVera Native Testbench (NTB-OV) allow you to design test cases more efficiently, using fewer lines of code. AOP addresses issues or concerns that are not addressed in a well-defined manner when using OOP to write constrained-random testbenches.

In OOP, the natural unit of modularity is the class. However, OOP doctrine does not modularize certain functionality concerns, and they cut across multiple classes in an OOP model. These concerns that span multiple classes are termed cross-cutting concerns because they cut across the typical unit of modularity in OOP. These concerns include:

1. **Context-sensitive behavior**: It may be desirable to have a class behave differently depending on the testbench that employs it. This concern to modularize testbench-sensitive behavior is independent of the typical concern in OOP to group functionality into classes.
2. **Adaptability**: It may be desirable to add functionality to a class unit in the future or to have a certain current functionality of a class unit be modified to some other behavior in the future. These concerns are not the primary responsibility of OOP.

AOP is a way of modularizing such cross-cutting concerns. AOP extends the functionality of existing OOP derived classes and uses the notion of aspect as another unit of modularity. You can encapsulate cross-cutting concerns in aspects to form reusable modules. By compartmentalizing code containing aspects, cross-cutting concerns become easier to manage. Aspects of a system can be changed, inserted, or removed at compile time, and are reusable.

It is important to understand that the overall verification environment should be assembled using OOP to retain encapsulation and protection. NTB-OV's aspect-oriented extensions should be used only for constrained-random test specifications with the aim of minimizing code. You should not use NTB-OV's AOE capabilities to:

- Code base classes and class libraries
- Debug, trace, or monitor unknown or inaccessible classes
- Insert new code to fix an existing problem

For information on the creation and refinement of verification testbenches, see the *Reference Verification Methodology User Guide*.

Aspect-Oriented Extensions in NTB-OV

In NTB-OV, AOP is supported by Aspect-Oriented Extensions (AOEs). AOEs in NTB-OV are compiler directives that define the precompilation expansion of the code. That is, AOEs are processed before compilation, yielding an equivalent NTB-OV program that is devoid of aspect extensions. In NTB-OV, you can only define an aspect extension at the top-level scope using a new top-level extends directive.

Note:

The terms aspect and extends directive are used interchangeably in this document.

Normally, you extend a class, but an extends directive defines modifications to a pre-existing class by doing an in-place extension of the class. An in-place extension modifies the definition of a class by adding new member fields and methods, and changing the behavior of earlier defined class methods without creating a new subclass. These changes affect all instances of the original class.

An extends directive for a class defines a scope in NTB-OV. Within this scope exist the items that modify the class definition. These items within an extends directive for a class can be divided into the following categories:

- Introduction

  Declaration of a new property or definition of a new method, constraint, enumerated type, or coverage group within the extends directive scope adds (or introduces) the new symbol into the original class definition as a new member. This declaration definition is called an introduction.

- Advice

  An advice is a construct which specifies code that affects the behavior of a member method of the class by weaving the specified code into the member method definition. The advice item is said to be an advice to the affected member method.

- Hide List

  Some items within an extends directive, such as a virtual method introduction that overrides or hides a symbol with the same name that is visible in the scope of the original class definition, or an advice to a local or protected method, may not be allowed within the extends directive scope depending on the hide permissions where the item is defined. A hide list is a construct whose placement and arguments within the extends directive scope controls the hide permissions. There can be multiple hide lists within an extends directive.

Processing of AOE as a Precompilation Expansion

As a precompilation expansion, AOE code is processed by VCS to modify the class definitions that it extends as per the directives in AOE.

A symbol is a valid identifier in a program. Classes and class methods are symbols that can be affected by AOE. AOE code is processed by adding introductions and weaving in advices in and around the affected symbols. Weaving is performed before actual compilation (and thereby before symbol resolution); therefore, under certain conditions, symbols introduced with the same identifier as an already visible symbols can hide the already visible symbols (see

“hide_list Details” on page 215). The pre-processed input program, now devoid of AOE, is then compiled. The syntax for extends directive is:

```
extends_directive ::=
extends extends_identifier (class_identifier)[dominate_list] {
      extends_item_list
}

dominate_list ::=
      dominates(extends_identifier {,extends_identifier});

extends_item_list ::=
      extends_item {extends_item}

extends_item ::=
      class_item
      | advice
      | hide_list

class_item ::=
      class_property
      | class_method
      | class_constraint
      | class_coverage
      | enum_defn

advice ::= placement procedure_prototype {
      advice_code
}

placement ::=
      before
      | after
      | around

procedure_prototype ::=
      | optional_method_specifiers task
          task_identifier(list_of_task_proto_formals)
      | optional_method_specifiers function function_type
          function_identifier(list_of_function_proto_formals)

      advice_code ::= [stmt] {stmt}
      stmt ::= statement | proceed ;

hide_list ::=
      hide([hide_item {,hide_item}]);
```

```
hide_item ::=
    // Empty
    | virtuals
    | rules
```

The symbols shown in **bold** in the above syntax description are keywords. The user-specified variables have the following definitions:

`extends_identifier`

is the name of the aspect extension.

`class_identifier`

is the name of the class being extended.

`dominate_list`

specifies extensions that are dominated by the current directive. Domination defines the precedence between code woven by multiple extensions into the same scope. One extension can dominate one or more of the other extensions. In such cases, you must use comma-separated lists of extend identifiers.

```
dominates(extends_identifier {,extends_identifier});
```

A dominated extension is assigned lower precedence than an extension that dominates it. Precedence among aspects extensions of a class determine the order in which introductions defined in the aspects are added to the class definition. They also determine the order in which advices defined in the aspects are woven into the class method definitions, thus affecting the behavior of a class method. The precedence rules for aspects are explained in “Precedence” on page 202.

`class_property`

refers to an item that can be parsed as a property of a class.

`class_method`

refers to an item that can be parsed as a class method.

`class_constraint`

refers to an item that can be parsed as a class constraint.

`class_coverage`

refers to an item that can be parsed as a *coverage_group* in a class.

`enum_defn`

definition of an enumerated data type.

`advice_code`

advice code that specifies a block of statements.

`statement`

an NTB-OV statement.

`procedure_prototype`

a full prototype of the target procedure. Prototypes enable the advice code to reference the formal arguments of the procedure.

`opt_method_specifiers`

refers to a combination of protection level specifier (local, public, or protected) and virtual specifier for the method.

`task_identifier`

is the name of the task.

`function_identifier`

is the name of the function.

`function_type`

is the data type of the function's return value.

`list_of_task_proto_formals`

is a list of formal arguments to the task.

`list_of_function_proto_formals`

is a list of formal arguments to the function.

`placement`

specifies the position at which the advice code within the advice is woven into the target method definition. Target method is either the class method, or some other new method that was created as part of the process of weaving, which is a part of pre-compilation expansion of code. For detailed information on the weaving process, see "Precompilation Expansion Details" on page 201.

The placement element can be any of the keywords `before`, `after`, or `around`, and the advices with these placement elements are referred to as before advice, after advice, and around advice, respectively.

`proceed statement`

optionally specifies an NTB-OV statement you can be use within advice code. A `proceed` statement is valid only within an `around` block and only a single `proceed` statement can be used inside the advice code block of an around advice. You cannot use it in a before advice block or an after advice block.

`hide_list`

specifies the permissions for introductions to hide a symbol, and/or permission) for advices to modify local and protected methods. For more information, see "hide_list Details" on page 215.

Weaving an Advice into the Target Method

The target method is either the class method or some other new method that was created as part of the weaving process. Weaving of all advices in the input program comprises several steps of weaving of an advice into the target method, as follows.

A new method is created with the same method prototype as the target method and with the advice code block as the code block of the new method. This method is referred to as the advice method. Table 7-1 shows the rest of the steps involved in weaving of the advice for each type of placement element (`before`, `after`, and `around`).

Table 7-1 Weaving Advice Placement Elements

Element	Description
`before`	Inserts a new method-call statement that calls an advice method. The statement is inserted as the first statement to be executed before any other statements.
`after`	Creates a new method A with the target method prototype, with its first statement being a call to the target method. The second statement with A is a new method call statement that calls the advice method. All the instances in the input program where the target method is called are replaced by newly created method calls to A. A is replaced as the new target method.
`around`	All the instances in the input program where the target method is called are replaced by newly created method calls to the advice method.

Within an extends directive, you can specify only one advice for a given placement element and method. For example, an extends directive may contain a maximum of one `before`, one `after`, and one `around` advice each for a class method `Packet::foo` of a class `Packet`, but it may not contain two `before` advices for the `Packet::foo`. Example 7-1 shows a `before` advice.

Example 7-1 Weaving before Advice

```
Target method:
task myTask() {
          printf("Executing original code\n");
}
Advice:
before task myTask () {
          printf("Before in aoe1\n");
}
```

For Example 7-1, weaving of the advice in the target method yields the following.

```
task myTask() {
          mytask_before();
          printf("Executing original code\n");
}

task mytask_before () {
          printf("Before in aoe1\n");
}
```

Note that NTB-OV does not impose any restrictions on the names of newly created methods such as `mytask_before` during precompilation expansion. Compilers can adopt any naming conventions for methods that are created as a result of the weaving process. Example 7-2 shows an after advice.

Example 7-2 Weaving after Advice

```
Target method:
task myTask() {
          printf("Executing original code\n");
```

```
}

Advice:
after task myTask () {
          printf("Before in aoe1\n");
}
```

For Example 7-2, weaving of the advice in the target method yields the following:

```
task myTask_newTarget() {
     myTask();
     myTask_after();
}

task myTask() {
          printf("Executing original code\n");
}
task myTask_after () {
          printf("After in aoe1\n");
}
```

As a result of weaving, NTB-OV replaces all the method calls to `myTask()` in the input program code with method calls to `myTask_newTarget()`. Also, `myTask_newTarget` replaces `myTask` as the target method for `myTask()`.

Example 7-3 shows an around advice.

Example 7-3   Weaving around Advice

```
Target method:
task myTask() {
          printf("Executing original code\n");
}

Advice:
around task myTask () {
          printf("Around in aoe1\n");
}
```

For Example 7-3, weaving of the advice in the target method yields the following:

```
task myTask_around() {
     printf("Around in aoe1\n");
}

task myTask() {
          printf("Executing original code\n");
}
```

As a result of weaving, NTB-OV replaces all the method calls to `myTask()` in the input program code with method calls to `myTask_around()`. Also, `myTask_around()` replaces `myTask()` as the target method for `myTask()`.

During weaving of an around advice that contains a proceed statement, NTB-OV replaces the proceed statement with a method call to the target method, as shown in Example 7-4.

Example 7-4   Weaving around Advice with proceed Statement

```
Target method:
task myTask() {
          printf("Executing original code\n");
}

Advice:
around task myTask () {
          proceed;
          printf("Around in aoe1\n");
}
```

In this example, weaving of the advice in the target method yields:

```
task myTask_around() {
     myTask();
     printf("Around in aoe1\n");
}

task myTask() {
          printf("Executing original code\n");
}
```

As a result of weaving, NTB-OV replaces all the method calls to `myTask()` in the input program code with method calls to `myTask_around()`. NTB-OV replaces the `proceed` statement in the `around` code with a call to the target method `myTask()`. Also, `myTask_around`() replaces `myTask()` as the target method for `myTask()`.

Precompilation Expansion Details

NTB-OV does the precompilation expansion of a program containing AOE code in the following order:

1. Preprocessing and parsing of all input code.
2. Identification of symbols, such as methods and classes affected by extensions.
3. Identification of the precedence order of aspect extensions (and thereby introductions and advices) for each class.
4. Addition of introductions to their respective classes as class members in their order of precedence. Whether an introduction can override or hide a symbol with the same name that is visible in the scope of the original class definition depends on certain rules related to the `hide_list` parameter. For more information, see “hide_list Details” on page 215.
5. Weaving of all advices in the input program into their respective class methods as per the precedence order.

These steps are described in more detail in the following sections.

Precedence

You specify precedence using a `dominate_list`. There is no default precedence across files; if precedence is not specified, NTB-OV is free to weave code in any order. Within a file, dominance established by `dominate_list` always overrides precedence established by the order in which extends directives are coded. Only when precedence is not established after analyzing the dominate lists of directives, is the order of coding used to define the order of precedence.

Within an extends directive there is an inherent precedence between advices. Advices that are defined later in the directive have higher precedence that those defined earlier.

Precedence does not change the order between adding of introductions and weaving of advices in the code. Precedence defines the order in which introductions to a class are added to the class and the order in which advices to methods belonging to a class are woven into the class methods.

Example 7-5 shows multiple aspect extensions for a class named `packet` defined in a single NTB-OV file.

Example 7-5 Multiple Aspect Extensions

```
program top {
      packet p;
      p = new();
      p.send();
}

class packet {
      ...
      // Other member fields/methods
      ...
      task send() {
            printf("Sending data\n");
```

```
        }
    }

    extends aspect_1(packet) dominates (aspect_2, aspect_3) {
        after task send() { // Advice 1
            printf("Aspect_1: send advice after\n");
        }
    }

    extends aspect_2(packet) {
        after task send() { // Advice 2
            printf("Aspect_2: send advice after\n");
        }
    }

    extends aspect_3(packet) {
        around task send() { // Advice 3
            printf("Aspect_3: Begin send advice around\n");
            proceed;
            printf("Aspect_3: End send advice around\n");
        }
    before task send() { // Advice 4
            printf("Aspect_3: send advice before\n");
        }
    }
```

In Example 7-5, `aspect_1` dominates both `aspect_2` and `aspect_3`. As per the dominating lists of the aspect extensions, there is no precedence order established between `aspect_2` and `aspect_3`, and since `aspect_3` is coded later in the file than `aspect_2`, `aspect_3` has higher precedence than `aspect_2`. Therefore, the precedence of these aspect extensions in decreasing order of precedence is:

{aspect_1, aspect_3, aspect_2}

This means that the advices within `aspect_2` have lower precedence than advices within `aspect_3`, and advices within `aspect_3` have lower precedence than advices within `aspect_1`. Therefore, advice 2 has lower precedence than advice 3 and advice 4. Both advice 3 and advice 4 have lower precedence than advice 1.

Between advice 3 and advice 4, advice 4 has higher precedence because it is defined later than advice 3. That puts the order of advices in the increasing order of precedence as:

{2, 3, 4, 1}

Adding Introductions

Target scope is the scope of the class definition being extended by an aspect. Introductions in an aspect are appended as new members at the end of its target scope. If an extension A has precedence over extension B, the symbols introduced by A are appended first.

Within an aspect extension, symbols introduced by the extension are appended to the target scope in the order that they appear in the extension.

There are certain rules according to which an introduction symbol with the same identifier name as a symbol that is visible in the target scope, may or may not be allowed as an introduction. These rules are discussed later in the chapter.

Weaving Advices

An input program can contain several aspect extensions for any or each of the different class definitions in the program. Weaving of advices needs to be carried out for each class method for which an advice is specified.

Weaving of advices in the input program consists of weaving of advices into each such class method. Weaving of advices into a class method A is unrelated to weaving of advices into a different class method B, and therefore weaving of advices to various class methods can be done in any order of the class methods.

For weaving of advices into a class method, all advices pertaining to the class method are identified and ordered by increasing precedence in a list L. This is the order in which these advices are woven into the class method, thereby affecting the runtime behavior of the method. The advices in list L are woven in the class method as per the following steps. The target method is initialized to the class method.

1. Advice A, which has the lowest precedence in L, is woven into the target method as explained earlier. Note that the target method may either be the class method or some other method newly created during the weaving process.

2. Advice A is deleted from list L.

3. The next advice on list L is woven into the target method. This continues until all the advices on the list are woven into list L.

Before and after advices within an aspect to a target method are unrelated to each other in the sense that their relative precedence to each other does not affect their relative order of execution when a method call to the target method is executed. The before advice code block executes before the target method code block, and the after advice code block executes after the target method code block. When an around advice is used with a before or after advice in the same aspect, code weaving depends upon their precedence with respect to each other. Depending on the precedence of the around advice with respect to other advices in the aspect for the same target method, the around advice can either be woven before all or some of the other advices, or woven after all of the other advices.

As an example, weaving of advices 1, 2, 3, 4 specified in aspect extensions leads to expansion of code as shown in Example 7-6.

Example 7-6   Advices Woven by Precedence

```
program top {
      packet p;
      p = new();
      p.send_Created_a();
}

class packet {
      ...
      // Other member fields/methods
      ...
      task send()     {
            printf("Sending data\n”);
      }

      task send_Created_a() {
            send();
            send_after_Created_b();
      }

      task send_after_Created_b() {
            printf("Aspect_2: send advice after\n");
      }
}

extends aspect_1(packet) dominates (aspect_2, aspect_3) {
      after task send() { // Advice 1
            printf("Aspect_1: send advice after\n");
      }
}

extends aspect_3(packet) {
      around task send() { // Advice 3
            printf("Aspect_3: Begin send advice around\n");
            proceed;
            printf("Aspect_3: End send advice around\n");
      }
```

```
before task send() { // Advice 4
            printf("Aspect_3: send advice before\n");
      }
}
```

Example 7-7 shows what the input program looks like after weaving advice 2 into the class method. Two new methods `send_Created_a` and `send_after_Created_b` are created in the process and the instances of method calls to the target method `packet::send` are modified such that the code block from advice 2 executes after the code block of the target method `packet::send`.

Example 7-7 Weaving Advice 2 into Class Method

```
program top {
      packet p;
      p = new();
      p.send_around_Created_c();
}

class packet {
      ...
      // Other member fields/methods
      ...

      task send() {
            printf("Sending data\n");
      }

      task send_Created_a() {
            send();
            send_after_Created_b();
      }

      task send_after_Created_b() {
            printf("Aspect_2: send advice after\n");
      }

      task send_around_Created_c() {
            printf("Aspect_3: Begin send advice around\n");
            send_Created_a();
            printf("Aspect_3: End send advice around\n");
      }
}
```

```
extends aspect_1(packet) dominates (aspect_2, aspect_3) {
      after task send() { // Advice 1
            printf("Aspect_1: send advice after\n");
      }
}

extends aspect_3(packet) {
                   before task send() { // Advice 4
            printf("Aspect_3: send advice before\n");
      }
}
```

Example 7-8 shows what the input program looks like after weaving advice 3 into the class method. A new method `send_around_Created_c` is created in this step and the instances of method calls to the target method `packet::send_Created_a` are modified so that the code block from advice 3 executes around the code block of method `packet::send_Created_a`. Also note that the proceed statement from the advice code block is replaced by a call to `send_Created_a`. At the end of this step, `send_around_Created_c` becomes the new target method for weaving of further advices to `packet::send`.

Example 7-8 Weaving Advice 3 into Class Method

```
program top {
      packet p;
      p = new();
      p.send_around_Created_c();
}

class packet {
      ...
      // Other member fields/methods
      ...
      task send(){
            printf("Sending data\n");
      }

      task send_Created_a() {
            send();
            send_after_Created_b();
```

```
        }

        task send_after_Created_b() {
              printf("Aspect_2: send advice after\n");
        }

        task send_around_Created_c() {
              send_before_Created_d();
              printf("Aspect_3: Begin send advice around\n");
              send_after_Created_a();
              printf("Aspect_3: End send advice around\n");
        }

        task send_before_Created_d() {
              printf("Aspect_3: send advice before\n");
        }
  }

  extends aspect_1(packet) dominates (aspect_2, aspect_3) {
        after task send() { // Advice 1
              printf("Aspect_1: send advice after\n");
        }
  }
```

Example 7-9 shows the input program after weaving advice 4 into the class method. A new method `send_before_Created_d` is created in this step and a call to it is added as the first statement in the target method `packet::send_around_Created_c`. Note that the outcome would have been different if advice 4 (before advice) was defined earlier than advice 3 (around advice) within `aspect_3`, as that would have affected the order of precedence of advice 3 and advice. In that scenario advice 3 (around advice) would have weaved around the code block from advice 4 (before advice), unlike the current outcome.

Example 7-9 also shows the input program after weaving all four advices {2, 3, 4, 1}. New methods `send_after_Created_e` and `send_Created_f` are created in the last step of weaving and the instances of method call to `packet::send_around_Created_c` were replaced by method call to `packet::send_Created_f`.

Example 7-9 Weaving All Advices into Class Method

```
program top {
      packet p;
      p = new();
      p.send_Created_f();
}

class packet {
      ...
      // Other member fields/methods
      ...
      task send() {
            printf("Sending data\n”);
      }

      task send_Created_a() {
            send();
            send_Created_b();
      }

      task send_after_Created_b() {
            printf("Aspect_2: send advice after\n");
      }

      task send_around_Created_c() {
            send_before_Created_d();
            printf("Aspect_3: Begin send advice around\n");
            send_after_Created_a();
            printf("Aspect_3: End send advice around\n");
      }
            task send_before_Created_d() {
            printf("Aspect_3: send advice before\n");
      }
      task send_after_Created_e() {
            printf("Aspect_1: send advice after\n");
      }

      task send_Created_f() {
            send_around_Created_c();
            send_after_Created_e()
      }
}
```

When executed, the program in Example 7-9 generates the following output:

```
Aspect_3: send advice before
Aspect_3: Begin send advice around
Sending data
Aspect_2: send advice after
Aspect_3: End send advice around
Aspect_1: send advice after
```

Example 7-10 shows some examples of code containing `around` advices.

Example 7-10 Code Containing around Device

```
program top {
      foo f;
      f = new();
      f.myTask();
}
class foo {
      int i;
      task myTask(){
            printf("Executing original code\n");
}
}
extends aoe1 (foo) dominates(aoe2){
      around task myTask() {
            proceed;
            printf("around in aoe1\n");
      }
}
extends aoe2 (foo) {
      around task myTask() {
            proceed;
            printf("around in aoe2\n");
      }
}
```

When `aoe1` dominates `aoe2`, as shown in Example 7-10, the program generates the following output:

```
Executing original code
around in aoe2
around in aoe1
```

But when `aoe2` dominates `aoe1`, as shown in Example 7-11:

Example 7-11 More Code Containing around Device

```
program top {
      foo f;
      f = new();
      f.myTask();
}

class foo {
      int i;
      task myTask() {
            printf("Executing original code\n");
      }
}
extends aoe1 (foo){
      around task myTask() {
            proceed;
            printf("around in aoe1\n");
      }
}
extends aoe2 (foo) dominates (aoe1) {
      around task myTask() {
            proceed;
            printf("around in aoe2\n");
      }
}
```

the program generates the following output:

```
Executing original code
around in aoe1
around in aoe2
```

Symbol Resolution Details

Preprocessing of introductions and advices defined within `extends` directives occurs before final symbol resolution in the compiler. Therefore, it is possible for AOE code to reference symbols that were added to the original class definition using AOEs. Because advices are woven after introductions are added to class definitions, you can specify advices for introduced member methods and reference introduced symbols.

An advice to a class method can access and modify the member fields and methods of the class object to which the class method belongs. An advice to a class function can access and modify the variable that stores the return value of the function.

Furthermore, members of the original class definition can also reference symbols introduced by aspect extensions using extern declarations. You can also use extern declarations to reference symbols introduced by an aspect extension to a class in some other aspect extension code that extends the same class.

You cannot use an introduction that has the same identifier as a symbol that is already defined in the target scope as a member property or member method. Example 7-12 shows how symbol resolution works.

Example 7-12   Symbol Resolution 1

```
program top {
      packet p;
      p = new();
      p.foo();
}

class packet {
      task foo(integer x) //Formal argument is "x"
      {
            printf("x=%0d\n", x);
      }
}

extends myaspect(packet){
      // Make packet::foo always print: "x=99"
      before task foo(integer x)
      {
            x = 99;   //force every call to foo to use x=99
      }
```

```
}
```

The extends directive in Example 7-12 sets the `x` parameter inside the `foo()` task to `99` before the original code inside `foo()` executes. The actual argument to `foo()` is not affected, and is not set unless passed-by-reference using `var`.

An advice to a function can access and modify the variable that stores the return value of the function, as shown in Example 7-13.

Example 7-13 Symbol Resolution 2

```
program top {
      packet p;
      p = new();
      printf("Output is: %d\n", p.bar());
}

class packet {
      function integer bar()
      {
            bar = 5;
            printf("Point 1: Value = %d\n", bar);
      }
}

extends myaspect(packet){
      after function integer bar()
      {
            printf("Point 2: Value = %d\n", bar);
            bar = bar + 1; // Stmt A
            printf("Point 3: Value = %d\n", bar);
      }
}
```

In Example 7-13, a call to `packet::bar` returns 6 instead of 5 because the final return value is set by `Stmt A` in the advice code block. The program generates the following output:

```
Point 1: Value = 5
Point 2: Value = 5
Point 3: Value = 6
```

```
Output is: 6
```

hide_list Details

The `hide_list` item in an `extends_directive` specifies the permissions for introductions to hide symbols, and/or permissions for advices to modify their corresponding local and protected methods. By default, an introduction does not have permission to hide symbols that were previously visible in the target scope, and it is an error for an extension to introduce a symbol that hides a global or super-class symbol.

The `hide_list` option contains a comma-separated list of options such as:

- You can use the `virtuals` option to hide (that is, override) virtual methods defined in a super class. Virtual methods are the only symbols that you can hide. You cannot hide global and file-local tasks and functions. Furthermore, all introduced methods must have the same virtual modifier as their overridden super-class and overriding sub-class methods.
- You can use the `rules` option to make the extension suspend access rules and specify advice that changes protected and local methods. By default, extensions cannot change protected and local methods.
- An empty option list removes all permissions (resets permissions to default.)

In Example 7-14, the print method introduced by the extends directive hides the print method in the super class.

Example 7-14 Hiding Methods

```
class pbase {
      virtual task print() {
            printf("I’m pbase\n");
      }
}

class packet extends pbase {
      task foo() {
            print(); //Call the print task
      }
}

extends myaspect(packet) {
      hide(virtuals); // Allows permissions to
                      // hide pbase::print task

      virtual task print() {
            printf("I’m packet\n”);
      }
}
```

There are two types of hide permissions:

- **Virtuals Permission**. Permission to hide virtual methods defined in a super class (option `virtuals`) is referred to as virtuals-permission. An aspect item is either an introduction, an advice, or a hide list within an aspect. If such permission is granted at an aspect item within an aspect, then the virtuals-permission is said to be on, or the status of virtuals-permission is said to be on at that aspect item and at all the aspect items following that, until a `hide_list` that forfeits the permission is encountered. If virtuals-permission is not on or the status of virtuals-permission is not on at an aspect item, then the virtuals-permission at that item is said to be off, or the status of virtuals-permission at that item is said to be off.

- **Rules Permission**. Permission to suspend access rules and specify advice that changes protected and local methods (option `rules`) is referred to as rules-permission. If such permission is granted within an aspect, at an aspect item, then the rules-permission is said to be on, or the status of rules-permission is said to be on at that aspect item and at all aspect items following that, until a `hide_list` that forfeits the permission is encountered. If rules-permission is not on, or the status of rules-permission is not on at an aspect item, then the rules-permission at that item is said to be off, or the status of rules-permission at that item is said to be off.

Permission for one of the above types of hide permissions does not affect the other. Status of rules-permission and hide-permission varies with the position of an aspect item within the aspect. Multiple `hide_list` options may appear in the extension. In an aspect, whether an introduction or an advice that can be affected by hide permissions is permitted to be defined at a given position within the aspect extension is determined by the status of the relevant hide permission at the position. A `hide_list` at a given position in an aspect can change the status of rules-permission and/or virtuals-permission at that position and all following aspect items until any hide permission status is changed again in that aspect using `hide_list`. Example 7-15 shows how hiding permissions can change permissions at different aspect items within an aspect extension.

Example 7-15 Hiding Permissions

```
class pbase {
      virtual task print1() {
            printf("pbase::print1\n");
      }

      virtual task print2() {
            printf("pbase::print2\n");
      }
}
```

```
class packet extends pbase {
      task foo() {
            print();
      }

      local virtual task rules-test() {
            printf("Rules-permission example\n");
      }
}

extends myaspect(packet) {

      // At this point within the myaspect scope,
      // virtuals-permission and rules-permission
      // are both off.

      hide(virtuals); // Grants virtuals-permission

      // virtuals-permission is on at this point within
      // aspect,and therefore can define print1 method
      // introduction.
      virtual task print1() {
            printf("packet::print1\n”);
      }

      hide(); // virtuals-permission is forfieted

      hide(rules); // Grants rules-permission

      // Following advice permitted as rules-permission is on
      before local virtual task rules-test() {
            printf("Advice to Rules-permission example\n");
      }

      hide(virtuals); // Grants virtuals-permission

// virtuals-permission is on at this point within
      // aspect,and therefore can define print2 method
      // introduction.
      virtual task print2() {
            printf("packet::print2\n”);
      }
}
```

Introducing New Members into a Class

Example 7-16 shows how you can use AOE to introduce new members into a class definition. In this example, `myaspect` adds a new property, constraint, coverage group, and method to the `packet` class.

Example 7-16 Introducing New Members into Class

```
class packet{
      rand bit[31:0]...
      ...
}

extends myaspect(packet) {
      integer sending_port;

      constraint con2 {
            hdr_len == 4;
      }

      coverage_group cov2{
            sample_event = @(posedge CLOCK);
            sample sending_port;
      }

      task print_sender() {
            printf("Sending port = %0d\n", sending_port);
      }
}
```

When you introduce new members into a class, do not use the same names as any symbols already defined in the class scope, as shown in Example 7-17, where the aspect `myaspect` defines `x` as one of the introductions, when the symbol `x` is already defined in class `foo`.

Example 7-17 Incorrect Introduction of New Member

```
class foo {
      rand integer myfield;
      integer x;
      ...
}
```

```
extends myaspect(foo) {
      integer x ;

      constraint con1 {
            myfield == 4;
      }
}
```

Examples of Advice Code

In Example 7-18, the `extends` directive adds advices to the `packet::send` method.

Example 7-18 Adding Advices to Methods

```
program test {
      packet p;
      p = new();
      p.send();
}

class packet {
      task send()
      {
            printf("Sending data\n”);
      }
}

extends myaspect(packet){
      before task send(){
            printf("Before sending packet\n");
      }

      after task send() {
            printf("After sending packet\n");
      }
}
```

This program generates the following output:

```
Before sending packet
```

```
Sending data
After sending packet
```

In Example 7-19, the `extends` directive `myaspect` adds advice to turn off constraint `c1` before each call to the `foo::pre_randomize` method.

Example 7-19 Adding an Advice that Turns off Constraint

```
class foo {
      rand integer myfield;
      constraint c1 {
            myfield == 4;
      }
}

extends myaspect(foo) {
      before task pre_randomize()
      {
            constraint_mode(OFF, "c1")
      }
}
```

In Example 7-20, the `extends` directive `myaspect` adds advice to set a property named `valid` to 0 after each call to the `foo::post_randomize` method.

Example 7-20 Adding an Advice to Set a Property

```
class foo {
      integer valid;
      rand integer myfield;
      constraint c1 {
            myfield == 4;
      }
}

extends myaspect(foo) {
      after task post_randomize(){
            valid = 0;
      }
}
```

Example 7-21 shows an aspect extension that defines an `around` advice for the class method `packet::send`. When this code is compiled and run, the `around` advice code is executed instead of original `packet::send` code.

Example 7-21 Aspect Extension Defining around Advice

```
program test {
      packet p;
      p = new();
      p.setLen(5000);
      p.send();
      p.setLen(10000);
      p.send();
}

class packet {
      integer len;
      task setLen( integer i) {
            len = i;
      }
      task send() {
            printf("Sending data\n”);
      }
}

extends myaspect(packet) {
      around task send()
      {
            if (len < 8000){
                  proceed;
            }
            else {
                  printf("Dropping packet\n");
            }
      }
}
```

Example 7-21 also demonstrates how you can use `around` advice code to reference properties such as `len` in the packet object `p`. The program generates the following output:

```
Sending data
Dropping packet
```

8

Predefined Methods and Procedures

This chapter documents the syntax of the OpenVera Native Testbench (NTB-OV) predefined methods, classes, and procedures in the following major sections:

- Predefined Methods
- Predefined Temporal Assertion Classes
- Predefined Procedures

Predefined Methods

The NTB-OV predefined methods are defined in two separate categories, as follows:

- Class Methods
- String Methods

Class Methods

The NTB-OV class methods include the following:

- new ()
- Randomize Methods
- Object Print
- Deep Object Compare
- Deep Object Copy
- Pack and Unpack by Class Methods

new ()

In NTB-OV, `new()` is a special method that allocates memory for an object and assigns a handle to that object. You can optionally define a `new()` constructor as a task call inside a class definition; in this case its code is executed when `new()` is called. As with other class methods, you can define an argument list in the class constructor, which allows for customization of objects at runtime. The syntax is:

```
class_declaration handle_name = new([parameter_list]);
```

`class_declaration`

is the name of a class.

`handle_name`

is the name of object handle created. It must be a valid identifier.

`parameter_list`

is a comma-separated list of arguments. You can initialize arguments here.

Note:

You can use `new` without parentheses when you are not passing any arguments.

Example 8-1 shows how to use `new()` both as a constructor and an operator.

Example 8-1 new () Method as Constructor and Operator

```
class B {
      task new(){ //constructor
            printf("An object of class B was just created.\n");
      }
}
program test {
      B b = new(); //operator
}
```

In Example 8-1, the line `B b = new` creates an object `b` of class-type `B` (its operator role). When you create this object, NTB-OV calls the constructor `new()` in class `B`. The program produces the following output:

```
An object of class B was just created.
```

Example 8-2 shows how to pass a variable to the `new()` constructor. In this case, the `inCommand` variable is an integer that is initialized at the same time.

Example 8-2 new () Method with Integer Argument

```
class B {
      integer command;
      task new(integer inCommand = 1){
            command = inCommand;
            printf("command is %0d.\n", inCommand);
      }
}
```

```
program test{
      B b = new();
}
```

This program generates the following output:

```
command is 1.
```

Randomize Methods

NTB-OV features several different types of randomize methods, as explained in the following subsections.

randomize ()

Use this method to randomize variables in an object. Note that every NTB-OV class has a built-in `randomize()` virtual method. The syntax is:

```
virtual function integer randomize();
```

Subject to active constraints, `randomize()` generates random values for all the active random variables in the object. This method returns `OK` if it successfully sets all the random variables and objects to valid values; otherwise it returns `FAIL`.

Example 8-3 Class with Random Variables

```
class SimpleSum {
      rand reg[7:0] x, y, z;
      constraint c {z == x + y;}
}
```

This class declares three random variables: `x`, `y`, and `z`. To randomize an instance of class `SimpleSum`, you call the `randomize()` method as shown in Example 8-4.

Example 8-4 Call to randomize () Method

```
program randomize_ex {
      SimpleSum p = new;
      integer success = p.randomize();

      if (success == OK )
            printf("OK \n");
      else
            printf("FAIL \n");
}
```

It is good practice to check results (as shown in this example) even if you know that the constraints can always be satisfied. This is because the actual values of state variables or the addition of constraints in derived classes may render seemingly simple constraints unsatisfiable.

pre_randomize () and post_randomize ()

Every NTB-OV class contains built-in `pre_randomize()` and `post_randomize()` tasks that are automatically called by `randomize()` before and after it computes new random values. Following is the built-in definition for `pre_randomize()`:

```
task pre_randomize() {
if (super) super.pre_randomize();
/* Optional programming before randomization goes here. */
}
```

and the built-in definition for `post_randomize()`:

```
task post_randomize(){
      if (super) super.post_randomize();
      /* Optional programming after randomization goes here. */
}
```

When you invoke `obj.randomize`, the task first invokes `pre_randomize()` on `obj` and enables all of its random object members. The `pre_randomize()` task then recursively calls super.pre_randomize(). After the new random values are computed and assigned, randomize() invokes post_randomize() on obj and enables all of its random object members. The post_randomize() task then recursively calls super.post_randomize().

You can override pre_randomize() in any class to initialize variables and set preconditions before the object is randomized. You can also override post_randomize() in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized.

If you override these methods, you must call their associated superclass methods; otherwise NTB-OV skips their pre- and post-randomization processing steps.

Note:

- Random variables declared as static are shared by all instances of the class in which they are declared. Each time the randomize() method is called, the variable is changed in every class instance.
- If randomize() fails, this means that the constraints are not feasible and the random variables retain their previous values. In this case, NTB-OV does not call the post_randomize() method.
- The randomize() method is implemented using object random stability. To seed an object, use the srandom() system call, specifying the object in the second argument.
- You cannot override the randomize() method.

- Do not use the built-in `pre_randomize()` and `post_randomize()` tasks for blocking operations. This is because their automatic traversal temporarily locks objects from access. If these routines block and another call to `randomize()` attempts to visit the locked object, NTB-OV's behavior is not defined.

randomize () with

You can use this NTB-OV construct to declare inline constraints when you call the `randomize()` class method. These additional constraints are of the same constraint types and forms as would otherwise be declared in the randomized class. NTB-OV applies the inline constraints along with the object constraints. You do not have to change the original constraints in the class definition. The syntax is:

`result` **`=`** *`class_object_name`***`.randomize() with`** *`constraint_block`***`;`**

`class_object_name`

is the name of the instantiated object.

`constraint_block`

is an anonymous constraint block that contains the additional inline constraints to be applied along with the object constraints declared in the class. Example 8-5 shows an how to use `randomize()with`.

Example 8-5   Call to randomize () with

```
class SimpleSum {
      rand reg[7:0] x, y, z;
      constraint c {z == x + y;}
}

task InlineConstraintDemo(SimpleSum p){
      integer success;
```

```
		success = p.randomize() with {x < y;};
		if(!success) error("ERROR\n");
	}

	program randomize_ex {
		SimpleSum p = new;
		InlineConstraintDemo(p) ;
	}
```

In Example 8-5, `randomize()with` introduces an additional constraint block (`x < y`) to the `SimpleSum` example.

You can also use a `randomize()with` constraint block to reference local variables and task and function parameters, eliminating the need for mirroring a local state as member variables in the object class.

The `randomize()with` call uses the object's scope. NTB-OV brings the `randomize()with` class into the object’s scope at the innermost nesting level. Example 8-6 shows how the scoping works. In this example, the `randomize()` with construct is applied to object `foo` of the class `Foo`.

Example 8-6 Scoping Using randomize () with

```
class Foo {
	rand integer x;
}

class Bar {
	integer x;
	integer y;

	task doit(Foo foo, integer x, integer z){
		integer result;
		result = foo.randomize() with {x < y + z; };
	}
}
```

In the `foo.randomize() with` constraint block, `x` is a member of class `Foo` (randomize-with) and hides the `x` in class `Bar`. It also hides the `x` parameter in the `doit()` task; whereas the `y` member of `Bar`. `z` is a local parameter.

If you override these methods you must call their associated superclass methods; otherwise NTB-OV skips their pre- and post-randomization processing steps.

rand_mode ()

You can use the predefined `rand_mode()` method to control whether a random variable is active or inactive. When a random variable is inactive, NTB-OV treats it as if it had not been declared `rand` or `randc`. All random variables are initially active. The syntax is:

```
function integer object_name.rand_mode( ON | OFF | REPORT [, string variable_name
[,integer index]]);
```

`object_name`

is the name of the object in which the random variables are defined.

`ON`, `OFF`, and `REPORT`

specify the action, as shown in Table 8-1.

Table 8-1 Actions

Macro	Action
`ON`	Sets the specified variables to active so that they are randomized on subsequent calls to the `randomize()` method.
`OFF`	Sets the specified variables to inactive so that they are not randomized on subsequent calls to the `randomize()` method.
`REPORT`	Returns the current ON/OFF value for the specified variable.

`variable_name`

is a string name of the variable to be made active or inactive. It can be the name of any variable in the class hierarchy. If it is not specified, NTB-OV applies the action to all variables within the specified object.

`index`

is an optional array index. If the variable is an array, omitting the `index` results in all the elements of the array being affected by the call. NTB-OV treats an `index` value of -1 the same as an index omission.

The `rand_mode()` method returns the new mode value (either OFF, which is 0, or ON, which is 1) if the change is successful. If the specified variable does not exist within the class hierarchy, this method returns a -1. If the specified variable exists but is not declared as `rand` or `randc`, it returns a -1. If the variable is an array, you must specify the array name and index for a REPORT.

If the variable is an object, NTB-OV only changes the mode of the variable. If the variable is an array, NTB-OV changes the mode of each array element. Example 8-7 first disables all the random variables in `packet`, and then enables the `source_value` variable.

Example 8-7   rand_mode () Method Calls

```
class Packet {
      rand integer source_value, dest_value;
      ... other declarations
}

program rand_mode_ex {
      Packet packet_a = new;
      integer ret;

      // Turn all variables off
      ret = packet_a.rand_mode (OFF);
```

```
        // Turn all variables on
        ret = packet_a.rand_mode (ON);

        // ... other code
        // Enable source_value.
    ret = packet_a.rand_mode (ON, "source_value");
    }
```

constraint_mode ()

You can use the predefined `constraint_mode()` method to control whether a constraint is active or inactive. All constraints are initially active. The syntax is:

```
function integer object_name.constraint_mode (ON | OFF | REPORT [, string
constraint_name]);
```

`object_name`

is the name of the object where the constraint block is defined.

`ON`, `OFF`, and `REPORT`

specify the action, as shown in Table 8-2.

Table 8-2 Actions

Macro	Action
`ON`	Sets the specified constraint block to active so that it is enforced on subsequent calls to the `randomize()` method.
`OFF`	Sets the specified constraint block to inactive so that it is not enforced on subsequent calls to the `randomize()` method.
`REPORT`	Returns the current ON/OFF value for the specified variable.

`constraint_name`

is the name of the constraint block you want to make active or inactive. It can be any constraint block in the class hierarchy. If you don't specify a `constraint_name`, NTB-OV applies the switch to all constraints within the specified object. You must specify the constraint name for a `REPORT`.

Example 8-8 first makes the `filter1` constraint inactive (`OFF`) and returns `OFF` to the `ret` variable. Then it makes `filter1` active (ON) and returns ON to `ret`.

Example 8-8 constraint_mode Call

```
class Packet {
      rand integer source_value;
      constraint filter1{
            source_value > 2 * m;
      }
}

Packet packet_a = new;
integer ret = packet_a.constraint_mode (OFF, "filter1");
// ... other code
ret = packet_a.constraint_mode (ON, "filter1");
```

Object Print

You can send the entire object instance hierarchy to stdout or a file using the `object_print()` method. As the deep print routine recursively traverses the object instance hierarchy, it indents object members and array elements as they are printed. NTB-OV displays all the super object members with the same indentation. Because NTB-OV print all elements of arrays and the entire object instance hierarchy, the amount of output can be large. The syntax is:

```
task class_instance_name.object_print([integer fd
[,string "attribute1=value1]...attributeN=valueN]");
```

`class_name`

is the target class to print.

`fd`

is the file descriptor (`fd`). Here are the legal values for `fd`:

- the text macro `stdout` (or `2`), which prints to stdout
- the text macro `stderr` (or `3`), which prints to stderr
- a standard file descriptor

`attributes`

is a string specifying various attributes. If you specify one or more attributes, `fd` is required. Separate attributes by spaces within the string. Any attribute not specified uses its default value. Table 8-3 lists the attributes and their values:

Table 8-3 Attributes for object_print ()

Attribute	Description	Default Value
`depth`	Specifies the number of levels in the instance hierarchy to print. If the `depth` attribute is 0, printing stops if a circular dependency is detected. If there is a circular list and *depth* is not zero, NTB-OV ignores the circularity and stops printing when it reaches the level.	0 (meaning the entire object instance hierarchy)
`indent`	Specifies the number of spaces to indent object members and array elements.	4
`severity`	`low`: NTB-OV ignores errors in `object_print()` and continues with simulation `high`: NTB-OV stops the simulation when an error occurs.	low
`port`	Specify `yes` to display port signals or `no` to *`not`* display them.	yes

Table 8-3 Attributes for object_print () (Continued)

Attribute	Description	Default Value
format	Specifies the format in which NTB-OV prints integers, bit vectors, and signals. Specify `bin` for binary, `dec` for decimal, or `hex` for hexadecimal. NTB-OV displays binary format as chunks of four bits separated by underscores.	-integers in decimal -bit vectors and signals with no x or z bits in hexadecimal -bit vectors and signals with x or z in binary
array_depth	Specifies the maximum number of array elements to print.	20
V	Used to determine what is printed: V = 2 or V = ALL means fully verbose, same as the existing behavior V = 1 or V = NO_CONTEXT means print the object instance name and the content but not the context information V = 0 or V = ONLY_OBJECT means print only the object content	ALL

Following are some usage examples for the `object_print()` method:

```
abc.object_print();
// correct, fd is optional here
       abc.object_print("depth=1 indent=10");
// incorrect, fd is mandatory here
       abc.object_print(4);
// incorrect, invalid file descriptor
       abc.object_print(2);
// correct but, "abc.object_print(stdout)"
// is the recommended usage
       abc.object_print(3);
// correct, output goes to stderr, but,
//"abc.object_print(stderr)" is the recommended usage
```

Here is an example `object_print()` call:

```
make_packet.object_print(stdout, "depth=2 indent=5
severity=high port=no format=hex array_depth=5");
```

Example with an Object as a Member

In Example 8-9, `abc` is a handle to an object of type `simple`. The class `simple` contains a handle to an object of type `embed`.

Example 8-9   object_print() with Object as Member

```
enum colors {red, green, black, white};
class embed {
      integer i;
      colors col;
      reg[3:0] bits_mem;
      string str_mem;
}
class simple {
      integer a,b,c,d; // integers
      colors col; // enum
      reg[3:0] abc; // bit vector
      embed e;
      string str; // string_var
}
program main {
      simple abc = new;
      abc.e = new;
      abc.a = 123;
      abc.b = 111111111;
      abc.d = 12345;
      abc.col = red;
      abc.abc[3:0] = 4'b1100;
      abc.object_print();
}
```

The program shown in Example 8-9 generates the following output:

```
CALLING "abc.object_print": CALL in program main
(a.vr, line 27, cycle 0);
READY in task simple.object_print (a.vr, line 17, cycle 0)
a : dec: 123
b : dec: 111111111
c : X
d : dec: 12345
col : ENUM:red
abc : hex: c
e : OBJECT of CLASS embed
i : X
col : ENUM:X
```

```
bits_mem : bin: xxxx
str_mem : NULL
str : NULL
```

Here, object `e` is a member of object `abc`. So, the members of `e` are indented. The strings `str` and `str_mem` do not have values so NTB-OV displays them as `NULL`. NTB-OV shows other variable types without values as `X`.

Example with Different Uses of Depth Parameter

In Example 8-10 there is a circular dependency:

Example 8-10 Circular Dependency

```
class simple {
      integer a; // integer
      simple s;
}

program main {
      integer fd = fopen("out.txt", "w");
      simple abc = new, next = new, next1 = new;
      abc.s = next;
      next.s = next1;
      next1.s = next;
      abc.a = 123;

      abc.object_print(fd); // depth=0 by default.
      abc.object_print(fd, "depth=4");

      fclose(fd);
}
```

The output file for Example 8-10 (`out.txt)` is:

```
CALLING "abc.object_print": CALL in program main (a.vr,
line 16, cycle 0); READY in task simple.object_print
(a.vr, line 6, cycle 0)
      a : dec: 123
      s : OBJECT of CLASS simple
            a : X
            s : OBJECT of CLASS simple
```

```
          a : X
          s : OBJECT of CLASS simple
               Circularity detected.
Stopped printing this object to avoid infinite
recursion
CALLING "abc.object_print": CALL in program main (a.vr,
line 18, cycle 0); READY in task simple.object_print
(a.vr, line 6, cycle 0)
     a : dec: 123
     s : OBJECT of CLASS simple
          a : X
          s : OBJECT of CLASS simple
               a : X
               s : OBJECT of CLASS simple
                    a : X
                    s : OBJECT of CLASS simple
```

Deep Object Compare

You can use the `object_compare()` predefined function with any NTB-OV class to compare two objects of the same class type. The NTB-OV compiler generates errors for invalid types.

This function compares all members of the two objects, including contained objects. Note that contained objects are compared by contents and not by handles. If the handles are the same, the comparison passes, as with the case where the handles are different but the contents are identical. This function also compares super objects and their contents, if present. In other words, `object_compare()` performs a complete hierarchical comparison.

The `object_compare()` function returns an integer value of 1 if it is successful and an integer value of 0 if the comparison fails.

When you use `object_compare()` on objects that contain embedded coverage groups, it issues a warning message indicating that coverage objects (bin values) are not considered in the comparison. The `object_compare()` syntax is:

```
function integer object1.object_compare(object2);
```

Here, `object1` and `object2` are instances of the same class. Example 8-11 shows an example NTB-OV program that contains an `object_compare()` call.

Example 8-11 object_compare () Call

```
class MyClass {
      // whatever
}

program object_compare_ex {
      MyClass object1, object2;
      object1 = new(); // instantiate an object
      . . .
      //object2 becomes a duplicate of object1.
      object2 = object1.object_copy();

      // if objects not identical
      if(!object1.object_compare(object2))
            error("Object compare failed\n");
      else
            printf("Objects are the same\n");
}
```

This program generates the following output:

```
"Objects are the same"
```

Comparison of Port Variables

When `object1` and `object2` contain port variables, the port variables must refer to the same instantiation of the extended virtual port for `object_compare()` to succeed (that is, they are the same handle). The comparison is not done at the bind or interface level. For information on creating virtual port instances, see “The bind Construct” on page 163.

In Example 8-12, the compare fails because `a.portVar` and `b.portVar` are port variables that reference different instances of the `myPort` virtual port. The fact that the port signal members connect to the same interface signals is not considered within the comparison.

Example 8-12   Failed Compare

```
interface infl {
      input [7:0] sig1  PSAMPLE #-1 depth 5;
}

port myPort{ sig1;} //declare port, myPort

bind myPort port1 {sig1 infl.sig1;}

bind myPort port2 {sig1 infl.sig1;}

class PortClass{ myPort portVar;}

program testPortCompare {
      integer match;
      PortClass a = new();
      PortClass b = new();
      a.portVar = port1;
      b.portVar = port2;
      match = a.object_compare(b); // fails

      b.portVar = port1;
      match = a.object_compare(b); // passes
}
```

Usage Notes

You can override the `object_compare()` method with a user-defined `object_compare()` method if you use the `object_compare()` method prototype:

```
function integer object1.object_compare(object2);
```

When working with contained and super objects, your `object_compare()` method should invoke the existing `object_compare()` methods for super and contained objects.

Example 8-13 shows how to use a `for` loop and the `object_compare()` method to compare a subset of array elements.

Example 8-13   Comparing Subset of Array Elements

```
class a
      integer a[10][20][30];
}
class b {
      reg[7:0] baa[12];
      a a_obj;
      function integer object_compare(b b_obj){
            integer I, result = 1;
// if you want to compare only 5 elements of the array:
// baa[12]

            for (I=0; I<5; I++) {
                  if (baa[I] != b_obj.baa[I]) {
                        result = 0;
                        break;
                  }
            }
            if (result)
            result = a_obj.object_compare(b_obj.a_obj);
            object_compare = result;
      }
}
```

Deep Object Copy

The NTB-OV `object_copy()` virtual function copies the contents of a source object into a destination object. The object copy is deep, replicating the entire data structure, including contained objects and the super object. This is a predefined function that you can use with all NTB-OV classes. When copying contained objects, the object copy does not copy the handle to the contained object; instead it makes a copy of the contained object and all of its contained objects. There is no hierarchical maximum depth limit to the copy. When copying objects that have embedded coverage groups, the current coverage values are copied into the duplicated object. The syntax is:

virtual function *dst_object* = *src_object*.**object_copy();**

`dst_object`

is an object of the same class as `src_object`.

`src_object`

is an instance of the class for which the `object_copy()` is defined.

This function returns the handle of the replicated object when it succeeds and a null handle when it fails. You cannot override the `object_copy()` method. Example 8-14 shows an NTB-OV program that contains an `object_copy()` call.

Example 8-14 object_copy () Call

```
MyClass src_obj, dest_obj;
src_obj = new();
. . .
dest_obj = src_obj.object_copy();// dest_obj and
     // src_object are now duplicates
 if (dest_obj == null) error("Copy failed\n");
```

Copying Contained Objects

The `object_copy()` virtual function copies in the following order: the object and its data members, including contained objects and their data members, followed by super objects, if they exist, as shown in Example 8-15.

Example 8-15 object_copy () Copy Ordering

```
class Foo {
      static reg[31:0] idCount = 0;

      task new(){
            idCount++;
      }
}

class Base {
      string name;
      task new(string s) {this.name = s;}
}

class Extension extends Base {
      string name;
      Foo f;

      task new(string s){
            super.new("extension of Base");
            f = new();
            this.name = s;
      }
}

program DeepExample {
      Base base0, base1;
      Extension ext0, ext1;
      ext0 = new("extension0");
      ext1 = ext0.object_copy();
      ext0.object_print();
      ext1.object_print();
}
```

The program shown in Example 8-15 generates the following results:

```
CALLING ".object_print":

      name : extension of Base
      name : extension0
      f : OBJECT of CLASS Foo
      idCount : hex: 00000001

CALLING ".object_print":

      name : extension of Base
      name : extension0
```

```
f : OBJECT of CLASS Foo
idCount : hex: 00000001
```

Example 8-16 shows that when copying contained objects, `object_copy()` does a complete hierarchical copy. Changes to the copied contained object have no effect on the originating object and its contained objects because this is a hierarchal copy, not a handle copy.

Example 8-16 object_copy () Complete Hierarchical Copying

```
class Data {
      rand reg [31:0] addr;
      rand reg [31:0] data;
      task printData(){
      printf("addr:%h\tdata%h\n",addr, data);
      }
}

class Example {
      rand integer a;
      rand Data d; // variable of class-type "Data"
      task new(){
      d = new(); // "d" is a handle to object
      void = randomize();
      }

      task printData(){
            d.printData();
            printf("a:%0d\n", a);
      }
}

program TestCopy {
      Example e0 , e1;
      e0 = new();
      e0.object_print();
      e1 = e0.object_copy();// duplicate copy - e1 has
                              // its own "d"
      e1.object_print();// objects are exactly the same
      e1.a = 7;
      e1.d.addr = 1;
      e1.d.data = 2;// unique data set to e1
      e0.object_print();// e0 unchanged
      e1.object_print();// e1 reflects changes
}
```

If object `a` has two objects contained within it named `b` and `c`, and `a.b` points to `a.c` (the two have the same handle) then, if you `object_copy()` `a`, NTB-OV also copies these references. So, if you have `d = a.object_copy()`, `d.b` points to `d.c` and not `a.c`, as shown in Example 8-17.

Example 8-17 object_copy () References Copying

```
class A {
      integer data;
      A next;
      task new(integer i )
      {data = i; }
}

program test{
      A f = new(10), s;
      f.next = new(20);
      f.next.next = f.next;
      s = f.object_copy();
      // s.next.next will point to s.next and not
      // f.next
      f.next.data = 307;
      printf("Printing f\n");
      f.object_print();
      printf("Printing s\n");
      s.object_print();
}
```

The program shown in Example 8-17 generates the following output:

```
Printing f

CALLING ".object_print":

data : dec: 10
next : OBJECT of CLASS A
data : dec: 307
next : OBJECT of CLASS A
                      Circularity detected. Stopped
                      printing this object
                      to avoid infinite recursion.
```

```
Printing s

CALLING ".object_print":
data : dec: 10
next : OBJECT of CLASS A
data : dec: 20
next : OBJECT of CLASS A
Circularity detected. Stopped
printing this object to avoid infinite recursion.
```

Copying References

By default, NTB-OV objects passed to methods are passed by reference. Using `object_copy()`, you can also pass objects by value. Example 8-18 shows how to use `object_copy()` to pass an object by value instead of the default (passing by reference).

Example 8-18 object_copy () Pass Object by Value

```
class try {
      integer data[];
      reg[9:0] bmap[string];
      string name;
}

task destroy(try obj){
      integer i, index;
      string s_index;

      i = assoc_index(FIRST, obj.data, index);
      while( i ) {
            void = assoc_index(DELETE, obj.data, index);
            i = assoc_index(NEXT, obj.data, index);
      }
      i = assoc_index(FIRST, obj.bmap, s_index);
      while( i ) {
            void = assoc_index(DELETE, obj.bmap, s_index);
            i = assoc_index(NEXT, obj.bmap, s_index);
      }
      obj.name = "null";

      program test {
      try a = new(); // new try object
```

```
        // populate try object, a
        a.data[12] = 13; a.data[13] = 14

        a.data[14] = 13; a.data[15] = 82;
        a.bmap["twelve"] = 13; a.bmap["thirteen"] = 14;
        a.bmap["fourteen"] = 13; a.bmap["fifteen"] = 82;
        a.name = "Hello";
        printf("\n***** Before calling destroy *****\n");
        a.object_print();

        // Call destroy by value, that is, copy
        destroy(a.object_copy());
        printf("\n***** After calling destroy, BY VALUE *****\n");
        a.object_print();

        // Now, call by reference, original a
        destroy(a);
        printf("\n*** After calling destroy, BY REFERENCE *****\n");
        a.object_print();
}
```

The program shown in Example 8-18 generates the following output:

```
***** Before calling destroy *****

CALLING "a.object_print":

data            : Associative array
[12]            : dec: 13
[13]            : dec: 14
[14]            : dec: 13
[15]            : dec: 82
bmap            : String Indexed Associative array
[fifteen]       : hex: 052
[fourteen]      : hex: 00d
[thirteen]      : hex: 00e
[twelve]        : hex: 00d
name            : Hello

***** After calling destroy, BY VALUE *****

CALLING "a.object_print":

data            : Associative array
[12]            : dec: 13
[13]            : dec: 14
[14]            : dec: 13
[15]            : dec: 82
```

```
bmap            : String Indexed Associative array
[fifteen]       : hex: 052
[fourteen]      : hex: 00d
[thirteen]      : hex: 00e
[twelve]        : hex: 00d
name            : Hello

***** After calling destroy, BY REFERENCE *****

CALLING "a.object_print"

data             : Associative array
bmap             : String Indexed Associative array
name             : null
```

Usage Notes

- The destination object must be a handle to an object of the same type as the source object. The NTB-OV compiler generates errors for invalid types.
- You don't need to `new` the destination object before the copy because NTB-OV performs an implied `new` for objects that are not initialized.
- It is good practice to check the results of the copy for a null value to verify that the operation succeeded.
- When copying to a destination object that has defined data, NTB-OV replaces the original destination data with the copy data.
- You cannot override the `object_copy()` method.

Pack and Unpack by Class Methods

Data packing is integrated into NTB-OV's object-oriented framework. NTB-OV defines several class methods that you can use to pack and unpack data declared within a class. NTB-OV also provides a set of attributes for class members that designate how data is to be packed and unpacked. This section explains packing and unpacking in the following subsections:

- Identifying Data to Pack and Unpack
- Attributes
- Packing Methods
- pre_unpack and post_unpack
- Unpacking Methods

Identifying Data to Pack and Unpack

The first step in using object-oriented packing is to identify the variables to be packed. A variable can be marked for packing by prepending the `packed` keyword when you declare the variable. For example:

```
packed integer n;
```

You can also mark multiple variables for packing by creating a statement block using curly braces preceded by the packed keyword, as shown in Example 8-19.

Example 8-19   Marking Multiple Variables for Packing

```
packed {
      integer n;
      reg[15:0] b;
}
```

Table 8-4 lists the data types you can pack and the number of bits that are packed for each type.

Table 8-4 NTB-OV Data Type Packing

Data Type	Bits Required to Pack
integer	32
reg[*N*-1:0]	*N*
enum	Minimum number of bits needed to represent all the enum values
string	8 x (strlen()+1 for NULL character)

Note that you can pack dynamic, fixed, and associative arrays of the data types shown in Table 8-4, but you cannot pack Smart Queues. All data variables marked as packed qualify for unpacking too.

Packing a null string causes NTB-OV to issue a warning message and pack one character (the NULL character).

Attributes

You control packing behavior using one or more attributes immediately following the `packed` keyword. These attributes include:

- little_endian
- big_endian
- bit_normal
- bit_reverse

You can only assign these attributes to class variables; they cannot be assigned to class methods or constraints. If you don't specify any attributes, the defaults are little_endian and bit_normal.

When you nest variable attributes, the lowest-level attribute overrides the higher-level attributes, as shown in Example 8-20.

Example 8-20 Nesting packed Attributes

```
packed bit_reverse {
      integer i;
      bit_normal string str;
      unpacked   {
            integer k;
      }
      integer n;
}
```

In this example, `i` and `n` are packed `bit_reverse`, `str` is packed `bit_normal`, and `k` is `unpacked`.

When you pack a dynamic array, you also need to specify the size of the array, using the `dynamic_size` attribute. The syntax is:

packed [*attribute*] *data_type array_name*[*] **dynamic_size** *size;*

When you pack an associative array, you also need to specify the size of the array, using the `assoc_size` attribute. The syntax is:

packed [*attribute*] *data_type array_name*[*]
assoc_size *size;*

little_endian

When you specify `little_endian` packing, NTB-OV writes the least significant segment (or piece equal in size to the output array) first. If the segment doesn't fill the entire word in the output array, NTB-OV aligns it towards the LSB, or right side of the word. NTB-OV updates the values of the `right` and `index` arguments to reflect the first bit following the last bit written and resets the value of the `left` argument to 0 when the write crosses a word boundary.

big_endian

When you specify `big_endian` packing, NTB-OV writes the most significant segment (or piece equal in size to the output array width) first. If the segment does not fill the entire word in the output array, NTB-OV aligns it towards the MSB or left side of the word. NTB-OV updates the values of the `left` and `index` arguments to reflect the first bit following the last bit written and resets the value of the `left` argument to 0 when the write crosses a word boundary.

bit_normal

When you specify `bit_normal` packing, NTB-OV writes the data value to the output array unchanged.

bit_reverse

When you specify `bit_reverse` packing, NTB-OV reverses the data value bits before writing to the output array.

Packing Methods

Packing methods are built into every NTB-OV class. You can use them to pack all qualified data variables, including embedded objects and inherited data, and can override these methods by defining your own packing methods. The packing methods are:

- pack ()
- pre_pack ()
- post_pack ()

Input for these consists of a class object containing variables of type reg, integer, enum, and string (event variables cannot be packed). Output is to an associative array organized as *n* words of *m* bits. The arguments to pack specify the object to be packed, the `index` into the array, and two values left and right which specify how many bits in the output word pointed to by `index` are used starting from the left and right of the word respectively.

NTB-OV writes data one segment at a time with ordering and word alignment according to the `little_endian` or `big_endian` specification. NTB-OV fills the output array starting from the specified `index` and leftmost unused bit for `little_endian`, or rightmost unused bit for `big_endian`. If all bits in the current word are filled, then left is zeroed for `big_endian`, or right for `little_endian`. At the end of the pack operation the `index` points to the last word written to. For `big_endian`, data `left` is updated to specify how many bits were written to the left side of the word. For `little_endian`, data `right` is updated to specify how many bits were written to the right side of the word. The syntax for `pack()` is:

```
function integer object_name.pack(var reg[N-1:0] array[], var integer index, var
integer left, var integer right);
```

`object_name`

is the name of the object whose data is to be packed.

`array`

specifies the array into which data is to be packed. The array can only be an associative array of bit width *N*.

`index`

specifies the array index at which to start packing.

`left/right`

specify the number of bits on the left and right to leave unchanged in `array[index]`. Normally, you initialize them to 0 before calling `pack()`.

pack ()

The `pack()` method returns the number of bits packed. It also updates `index`, `left`, and `right` so that they can be used as arguments to subsequent `pack()` calls when packing multiple objects into a single stream (see Example 8-21).

Example 8-21 pack () Call

```
integer nbits;
integer offset = 0;
integer left = 0;
integer right = 0;
reg [7:0] stream[];

nbits = packet_head.pack(stream, offset, left, right);
nbits+ = packet_body.pack(stream, offset, left, right);
```

When you call the `pack()` method, it begins by packing the first member it encounters. If an object´s handle is encountered, NTB-OV invokes the `pack()` method of that class. Packing an object handle that is null produces a warning, and nothing is packed for that nested object; that object handle should also be null when the containing object is unpacked. If a nested object´s handle is not null when it is packed, the handle should also be non-null before it is unpacked; unpacking does not call `new()` to create objects.

NTB-OV automatically calls the `pre_pack()` and `post_pack()` methods before and after `pack()`.

pre_pack ()

`pre_pack()` calls its super class methods by invoking the `pre_pack()` method of its parent class. This method must be public.

post_pack ()

`post_pack()` calls its super class methods by invoking the `post_pack()` method of its parent class.This method must be public.

NTB-OV allows you to override these methods with your own functionality. If you override the default `pre_pack()` and `post_pack()` methods in a particular class, then invoking the `pack()` method on this class causes the overridden methods to be invoked. Unless the overridden `pre_pack()` explicitly calls `pre_pack()`, the parent's `pre_pack()` is not executed. If you override the predefined method, you must specify a bit width, which is inherited by all the descendants of that class.

Unpacking Methods

The NTB-OV unpacking methods are analogous to the packing methods, and include:

- unpack ()
- pre_unpack ()
- post_unpack ()

unpack ()

The syntax for `unpack()` is:

```
function integer object_name.unpack(reg[N-1:0], array[], var integer index, var
integer left, var integer right);
```

The `unpack()` parameters have the same definitions as the `pack()` parameters. Set the `array` parameter to the array where data was packed. You normally initialize the `index`, `left`, and `right` parameters to 0. NTB-OV then update them with each call to unpack(). This allows you to unpack multiple objects from a single array. You should generally unpack multiple objects from one array in the same order in which they were packed (see Example 8-22).

Example 8-22 unpack () Call

```
x.pack(...);
y.pack(...);
z.pack(...);
x.unpack(...);
y.unpack(...);
z.unpack(...);
```

Example 8-23 shows how to use the pack() and unpack() methods.

Example 8-23 pack () and unpack () Calls

```
class Serial_Data_Type {
      static integer total_inst_count = 0;
      packed {
            rand reg [19:0] bit_data;
            string comment;
      }

      task new() {
            integer status;
            status = this.randomize();
            if ( !status )
            error ("Randomize failed!\n");
            comment  = psprintf("comment_%0d", total_inst_count) ;
            printf("inst = %9d , data = %25b comment =
            %0s\n",total_inst_count, bit_data, comment );
            total_inst_count++ ;
      }   // new
}
program packed_test {
      Serial_Data_Type sdata_arr[5];
      reg data_stream[]; // need not be byte stream
      integer i, offset, left, right;

      printf ("\n\nPacking data ...........\n");
      offset = 0; left = 0; right = 0;

      for ( i = 0; i < 5; i++ ) {
            sdata_arr[i] = new();
```

```
            void = sdata_arr[i].pack (data_stream, offset, left,
            right );
        } // for

        printf ("\n\nUnpacking data in order .....\n");

        offset = 0; left = 0; right = 0;
        for ( i = 0; i < 5; i++ ) {
            void =  sdata_arr[i].unpack ( data_stream, offset,
                left, right );
            printf("inst = %9d , data = %25b comment = %0s\n", i
            , sdata_arr[i].bit_data
            , sdata_arr[i].comment );
        } // for
    }  // packed_test
```

pre_unpack and post_unpack

NTB-OV automatically calls the `pre_unpack()` and `post_unpack()` methods before and after `unpack()`.

pre_unpack ()

`pre_unpack()` calls its super class methods by invoking the `pre_pack()` method of its parent class. This method must be public.

post_unpack ()

`post_unpack()` calls its super class methods by invoking the `post_pack()` method of its parent class.This method must be public.

NTB-OV allows you to override these methods with your own functionality. If you override the default `pre_unpack()` and `post_unpack()` methods in a particular class, then invoking the `unpack()` method on this class causes NTB-OV to invoke the overridden `pre_unpack()` and `post_unpack()` methods. Unless the overridden `pre_unpack()` explicitly calls `pre_unpack()`, NTB-OV does not execute the parent's `pre_unpack()`. If you override the predefined method, you must specify a particular bit width, which is inherited by all descendants of the class.

Details of Pack and Unpack

The equivalent pseudocode for the `pack()` method is shown in Example 8-24:

Example 8-24 pack () Pseudocode

```
virtual function integer MyClass::pack(var reg[N-1:0] array[],
      var integer offset, var integer left,var integer right,
      integer flag = 1) {
      integer nbits = 0;

      if (flag == 1)
            this.pre_pack();

      nbits = super.pack (array, offset, left, right, 0);

//code to pack the member variables of "this" that are
// marked packed -- in order, including nested objects
// -- adding to nbits

      if (flag == 1)
            this.post_pack();

      pack = nbits;
}

task MyClass::pre_pack() {
      super.pre_pack();
}

task MyClass::post_pack() {
      super.post_pack();
}
```

In Example 8-24, notice that the `pack()` method calls the `pack()` of its superclass before packing member variables of this class. Thus, the packing operation ultimately begins with the base class and works its way down the hierarchy, so that data associated with the base class is packed before data associated with its derived classes. Notice also that the flag parameter of `super.pack` is 0, so that it does not call `pre_pack()` and `post_pack()`.

String Methods

NTB-OV supports the following methods for the string data type.

len ()

The `len()` function returns the number of characters in the specified string, excluding the terminating null character. It returns 0 if `string_variable` is null or empty. The syntax is:

```
function integer string_variable.len();
```

Example 8-25 shows a `len()` call:

Example 8-25 len () Call

```
string str; // declare str as a string type
str = "This is a string"; // assign string literal to str
printf("String length = %0d\n", str.len());//obtain length of str
```

Example 8-25 generates the following output:

```
String length = 16
```

putc ()

The `putc()` task assigns a given character to a specified location. The syntax is:

```
task string_variable.putc(integer i, string char);
```

`i`

is the first location in the string. If `0<=i <string_variable.len()` is not satisfied, NTB-OV returns an empty string. Valid values for *`i`* range from 0 to the length of the string -1.

`char`

is any expression that can be assigned to a string.

The `putc()` string member task puts the first character of string *`char`* in position *`i`* of *`string_variable`*. If `i` is greater than the length of the string, *`char`* is ignored and the string remains unchanged (see Example 8-26).

Example 8-26 putc () Call

```
string str = "X23456789";
str.putc(0, "15");
```

Example 8-26 sets string `str` to “123456789”.

getc ()

The `getc()` function returns the ASCII code of the *i*-th character in the specified string. The syntax is:

```
function integer string_variable.getc(integer i);
```

`i`

The *`i`* formal argument (an integer) is the first location in the string. If `0<=i< string_variable.len()` is not satisfied, NTB-OV returns an empty string. Valid values for *`i`* range from 0 to the length of the string -1.

The `getc()` function returns the character specified by *`i`* in ASCII code form. If the formal argument is larger than the given string, NTB-OV returns a 0. The first character of the string is indexed by 0 (see Example 8-27).

Example 8-27   getc () Call

```
string str = "This is a string";
printf ("The fourth character is %c\n", str.getc(3));
```

Example 8-27 prints the fourth character, `s`, in the string `str`.

toupper ()

The `toupper()` function changes lower-case characters in a string to upper-case, and returns the new string. The syntax is:

```
function string string_variable.toupper();
```

NTB-OV does not change the *`string_variable`* object.

tolower ()

The `tolower()` function changes upper-case characters in a string to lower-case, and returns the new string. The syntax is:

```
function string string_variable.tolower();
```

NTB-OV does not change the *`string_variable`* object.

compare ()

The `compare()` function compares two strings to determine if they are identical. The comparison is case-sensitive. If the strings are identical, this function returns a 0. If the string pointed to by `t` is shorter than the string, NTB-OV returns >0. If the strings have a different number of characters, the first different character determines whether NTB-OV returns <0 or >0. The syntax is:

function integer *string_variable*.**compare(***t***);**

t

can be either a string variable or string literal (see Example 8-28).

Example 8-28 compare () Call

```
integer rc;
string s1, s2, s3, s4;

s1 = "abcd";
s2 = "abcd";
s3 = "abce";
s4 = "abc";

rc = s1.compare(s2); // rc == 0
rc=  s1.compare(s3); // rc < 0
rc = s3.compare(s1); // rc is > 0
rc = s1.compare(s4); // rc is > 0
```

icompare ()

The `icompare()` function works just like `compare()` except that it ignores case (see “compare ()” on page 262). The syntax is:

function integer *string_variable*.**icompare(***t***);**

t

is a string variable.

Example 8-29 shows how to use the `icompare()` function.

Example 8-29 icompare () Call

```
integer rc;
string s1, s2, s3, s4;
```

```
s1 = "Abcd";
s2 = "abCd";
s3 = "abce";
s4 = "aBc";

rc = s1.icompare(s2);// rc == 0
rc=  s1.icompare(s3);// rc < 0
rc = s3.icompare(s1);// rc is > 0
rc = s1.icompare(s4);// rc is > 0
```

substr ()

The `substr()` function returns the substring of characters between two specified locations. The syntax is:

function string *string_variable*.**substr(***i*,*j***);**

`i`

The *i* formal argument (an integer) is the first location in the string. If `i <= j < string_variable.len()` is not satisfied, NTB-OV returns an empty string.

`j`

The *j* formal argument (an integer) is the second location in the string.

The `substr()` function prints the characters between the locations, inclusively (see Example 8-30).

Example 8-30 substr () Call

```
string str, str1;
str = "ABCDEF";
str1 = str.substr(2,4);
```

Example 8-30 assigns string `str1` the value “CDE”.

String Class Methods for Type Conversion

NTB-OV provides several class methods that you can use for type conversion.

atoi ()

The `atoi()` function returns the integer corresponding to the ASCII decimal representation of a string. The syntax is:

```
function integer string_variable.atoi();
```

If the ASCII string is not a number representation, the function returns 0. Underscores are ignored. Spaces and all characters other than digits are invalid (see Example 8-31).

Example 8-31 atoi () Call

```
integer i;
string str;
str = "123";
i = str.atoi();
```

Example 8-31 converts the ASCII text `"123"` and assigns the value `123` to integer `i`.

itoa ()

The `itoa()` task converts an integer to its decimal representation in a string. The syntax is:

```
task string_variable.itoa(i);
```

`i`

must be an integer. Underscores are ignored. All other characters are invalid (see Example 8-32).

Example 8-32 itoa () C all

```
string str;
str.itoa(456);
```

Example 8-32 converts the numeric string value of `456` to ASCII text and assigns the value “`456`” to string `str`.

atohex ()

The `atohex()` function returns the integer corresponding to the ASCII hexadecimal representation of a string. The string is converted to the first non-digit. The syntax is:

`function integer` *`string_variable`***`.atohex();`**

If the ASCII string is not a hexadecimal number representation, the function returns 0 (see Example 8-33).

Example 8-33 atohex () Call

```
integer i;
string str;
str = "12";
i = str.atohex(); // assigns 'h12 to integer i.
printf("%d \n", i);  // prints 18
```

Example 8-33 converts the ASCII text value “‘`h12`” to the hexadecimal number `'h12` and assigns it to integer `i`.

atooct ()

The `atooct()` function handles a string as an ASCII octal number and converts it to an integer value. The syntax is:

`function integer` *`string_variable`***`.atooct();`**

If the ASCII string is not an octal number representation, the function returns 0. The string is converted to the first non-digit (see Example 8-34).

Example 8-34 atooct () Call

```
integer i;
string str;
str = "12";
i = str.atooct(); // assigns 'o12 to integer i
printf("%d \n", i); //prints 10
```

Example 8-34 converts the ASCII text value "'`o12`" to the octal number `'o12` and assigns it to integer `i`.

atobin ()

The `atobin()` function handles a string as an ASCII binary number and converts it to an integer value. The syntax is:

`function integer` *`string_variable`*`.`**`atobin();`**

If the ASCII string is not a binary number representation, the function returns 0. The string is converted to the first non-digit (see Example 8-35).

Example 8-35 atobin () Call

```
integer i;
string str;
str = "10";
i = str.atobin();// assigns 'b10 to i
printf("%d \n", i); //prints 2
```

Example 8-35 converts the ASCII text value "`b10`" to the binary `number 'b10` and assigns it to reg/integer `i`.

bittostr ()

The `bittostr()` task converts a bit representation to a string of ASCII characters (see Example 8-36). The syntax is:

```
task string_variable.bittostr([msb :0] bit_variable);
```

Example 8-36 bittostr () Call

```
reg [63:0] b;
string str;
b = "Hello";
str.bittostr(b);
printf("%h\n", b);
printf("%s\n", str);
```

Example 8-36 converts the bit string “`Hello`” to a string and assigns the value “`Hello`” to string `str`:

```
00000048656c6c6f
Hello
```

String Class Methods for Matching Patterns

NTB-OV provides several class methods that you can use to match patterns within strings.

search ()

The `search()` function searches for a pattern in the string and returns the integer index to the beginning of the pattern. The syntax is:

```
function integer string_variable.search(string pattern);
```

`pattern`

must be a string.

`return value`

returns the index value at the start of the string pattern. If the pattern is not found, returns -1.

Example 8-37 assigns the index 8 to integer `i` and prints out 8.

Example 8-37   search () Call

```
integer i;
string str = "This is a test";
i = str.search("a test");
printf("%d \n", i);
```

match ()

The `match()` function processes a regular expression pattern match. The syntax is:

```
function integer string_variable.match(string pattern);
```

`pattern`

must be a Perl regular expression.

`return value`

returns a 1 if the expression is found, or a 0 if the expression is not found.

If there is a syntax error in the regular expression, the function returns a 0 and sets the status to STR_ERR_REGEXP_SYNTAX.

Example 8-38 assigns the value 1 to integer i because the pattern `"is"` exists within string `str`.

Example 8-38 match () Call

```
integer i;
string str;
str = "1234 is a number.";
i = str.match("is");
```

You can use the following functions to access the match strings.

prematch ()

The `prematch()` function returns the string before a match, based on the result of the last `match()` function call. The syntax is:

```
function string string_variable.prematch();
```

Example 8-39 assigns the value `"1234 "` to string `str1`.

Example 8-39 prematch () Call

```
integer i;
string str, str1;
str = "1234 is a number.";
i = str.match("is");
str1 = str.prematch();
```

postmatch ()

The `postmatch()` function returns the string after a match, based on the result of the last `match()` function call. The syntax is:

```
function string string_variable.postmatch();
```

Example 8-40 assigns the value `" a number."` to string `str1`.

Example 8-40 postmatch () Call

```
integer i;
string str, str1;
str = "1234 is a number.";
i = str.match("is");
```

```
str1 = str.postmatch();
```

thismatch ()

The `thismatch()` function returns the matched string, based on the result of the last `match()` function call. The syntax is:

```
function string string_variable.thismatch();
```

Example 8-41 assigns the value `"is"` to string `str1`.

Example 8-41 thismatch () Call

```
integer i;
string str, str1;
str = "1234 is a number.
i = str.match("is");
str1 = str.thismatch();
```

backref ()

The `backref()` function returns matched patterns, based on the last `match()` function call. The syntax is:

```
function string string_variable.backref(integer index);
```

`index`

is the integer number of the Perl expression being matched. Indexing starts at 0.

Note:

Generally, Perl regular expression indexing starts at 1, but the starting value with this command is 0 so that OpenVera Native Testbench can stay in sync with the starting index value in Vera, which is also 0. This eases the transition from Vera to OpenVera Native Testbench.

This function matches a string with Perl expressions specified in a second string (see Example 8-42).

Example 8-42 backref () Call

```
integer i;
string str, patt, str1, str2;
str = "1234 is a number."
patt = "([0-9]+) ([a-zA-Z .]+)";
i = str.match(patt);
str1 = str.backref(0);
str2 = str.backref(1);
```

Example 8-42 checks the Perl expressions given by the `patt` string with the `str` string. It assigns the value “`1234`” to string `str1` because of the match to the expression “`[0-9]+`”. It assigns the value “`is a number`.” to string `str2` because of the match to the expression “`[a-zA-Z .]+`”. You can list any number of additional Perl expressions in the `patt` definition and call them using sequential index numbers with the `backref()` function.

Smart Queue Methods

NTB-OV provides a set of built-in methods for analyzing and manipulating Smart Queue elements. The syntax for calling the predefined methods is:

```
Smartqueue.method();
```

For example:

```
SQPacket.sort();
```

Where `SQPacket` is a queue containing objects of type `Packet`.

The Smart Queue methods include:

- Add/Delete
  - delete ()
  - insert ()
- Order
  - reverse ()
  - rsort ()
  - sort ()
  - sum ()
- Push/Pop
  - pop_back ()
  - pop_front ()
  - push_back ()
  - push_front ()
- Random
  - pick ()
  - pick_index ()
  - unique ()
  - unique_index ()
- Search
  - find ()
  - find_index ()

- first ()
- first_index ()
- last ()
- last_index ()
- max ()
- max_index ()
- min ()
- min_index ()

• Size
  - capacity ()
  - empty ()
  - reserve ()
  - size ()

These methods are explained here in alphabetical order.

capacity ()

The `capacity()` method returns the total number of elements a queue can accommodate without requiring automatic or manual reallocation. You use it to override the internal reallocation mechanism (advance users only). The syntax is:

```
function integer SmartQueue.capacity();
```

For example:

```
value = SQint.capacity();
```

delete ()

The `delete()` method deletes the element at the specified `index`. NTB-OV issue an error message if the `index` is invalid. If you don't specify an `index`, NTB-OV deletes all elements of the queue. The syntax is:

```
task SmartQueue.delete([integer index]);
```

For example:

```
queue.delete(5); // Delete the element at index 5
queue.delete(); // Delete all the elements
```

You can use the `delete()` method to delete all existing elements from a dynamic array, making its size zero (see Example 8-43).

Example 8-43 delete () Call

```
program size_ex {
      integer packet_size[*];
      integer array_size;
      packet_size = new[100];
      array_size = packet_size.size();
      printf("packet_size is %0d\n",array_size);
      packet_size.delete();
      array_size = packet_size.size();
      printf("packet_size is %0d\n", array_size);
}
```

Example 8-43 prints:

```
packet_size is 100
packet_size is 0
```

find ()

The `find()` method finds all elements that satisfy the expression using the `with` construct, which is required. If the match fails or the queue is empty, NTB-OV returns an empty queue. The syntax is:

```
function data_type[$] SmartQueue.find() with index : (expression);
```

In the first example shown in Example 8-44, NTB-OV returns all elements that are greater than 5 and assigns them in the same order to `queue2`. In the second example, NTB-OV assigns all elements for which the product of member variables `x` and `y` is greater than 5 to `queue4` in the same order.

Example 8-44   find () Call

```
queue2 = queue1.find() with index :
                 (queue1[index] > 5);
queue4 = queue3.find() with index :
                 (queue3[index].x  * queue3[index].y > 5);
```

find_index ()

The `find_index()` method finds all indices satisfying the specified `expression` using the `with` construct, which is required. If the match fails or the queue is empty, NTB-OV returns an empty queue. The syntax is:

```
function integer[$]SmartQueue.find_index() with index: (expression);
```

In the first example shown in Example 8-45, NTB-OV returns all indices of elements greater than `5` and assigns them to `indices1`. In the second example, NTB-OV assigns to `indices2` all indices of elements for which the product of member variables `x` and `y` is greater than `5`.

Example 8-45 find_index () Call

```
indices1 = queue1.find_index() with index: (queue1[index] > 5);
indices2 = queue2.find_index() with index:
      (queue2[index].x  * queue2[index].y > 5);
```

first ()

The `first()` method finds the first element satisfying the specified `expression` using `with`. If the queue is empty, NTB-OV issues a warning message and returns a type-dependent default value. With this method, usage of the `with` expression is optional. The syntax is:

```
function data_type SmartQueue.first() [with index : (expression)];
```

In the first example shown in Example 8-46, NTB-OV returns the element at the 0th index. In the second example, NTB-OV returns the first element which is greater than `5`. In the third example, NTB-OV returns the first element with member variable `x` greater than `5`.

Example 8-46 first () Call

```
value = queue1.first();
value = queue1.first() with index : queue1[index]  > 5);
object = queue2.first() with index : (queue2[index].x  > 5);
```

first_index ()

The `first_index()` method finds the first index satisfying the specified `expression` using `with`. If the queue is empty or no match is found with the expression, NTB-OV returns –1. With this method, usage of the `with` expression is optional. The syntax is:

```
function integer SmartQueue.first_index() [with index: (expression)];
```

In the first example shown in Example 8-47, NTB-OV returns 0 if the queue is not empty. In the second example, NTB-OV returns the index of the first element which is greater than `5`. In the third example, NTB-OV returns the index of the first element with member variable `x` greater than `5`.

Example 8-47 first_index () Call

```
index = queue1.first_index();
index = queue1.first _index() with index : (queue1[index]  > 5);
index = queue2.first _index() with index : (queue2[index].x > 5);
```

empty ()

The `empty()` method returns 1 if the queue is empty; otherwise it returns 0. The syntax is:

```
function integer SmartQueue.empty();
```

Example 8-48 shows an example.

Example 8-48 empty () Call

```
is_empty = queue.empty();
```

insert ()

The `insert()` method inserts an element at the specified `index`. NTB-OV issues an error message if you specify an invalid index. The element can be an individual element or another queue of the same type. NTB-OV does strict type checking between the element and queue type. The syntax is:

```
task SmartQueue.insert(integer index, data_type element);
task SmartQueue.insert(integer index, data_type SQ1[$]);
```

Example 8-49 shows some examples.

Example 8-49 insert () Call

```
// Insert 10 at index 5
queue1.insert(5, 10);

// Insert queue1 at index 8
queue2.insert (8, queue1);

// Insert queue2 at the back of queue3
queue3.insert(queue3.size(), queue2);

// Insert queue2 in front of queue3
queue4.insert(0, queue2);
```

last ()

The `last()` method finds the last element satisfying the specified `expression`. If the queue is empty, NTB-OV issues a warning message and returns a type-dependent default value. With this method, the `with` expression is optional. The syntax is:

function *data_type SmartQueue*.**last()**[**with index :** (*expression*)]**;**

In the first example shown in Example 8-50, NTB-OV returns the element at index size – 1. In the second example, NTB-OV returns the last element which is greater than `5`. In the last example, NTB-OV returns the last element with member variable `x` greater than `5`.

Example 8-50 last () Call

```
value = queue1.last();
value = queue1.last() with index : (queue1[index] > 5);
object = queue2.last() with index : (queue2[index].x > 5);
```

last_index ()

The `last_index()` method finds the last index satisfying the specified `expression`. If the queue is empty or no match is found, NTB-OV returns –1. With this method, use of the `with` expression is optional. The syntax is:

```
function integer SmartQueue.last_index()[with index : (expression)];
```

In the first example shown in Example 8-51, NTB-OV returns the last index. In the second example, NTB-OV returns the index of the last element which is greater than `5`. In the last example, NTB-OV returns the index of the last element with member variable `x` greater than `5`.

Example 8-51 last_index () Call

```
value = queue1.last_index();
value = queue1. last_index() with index :(queue1[index] > 5);
value = queue2. last_index() with index : (queue2[index].x > 5);
```

max ()

The `max()` method finds the maximum value element using the specified `expression`. If the queue is empty, NTB-OV issues a warning message and returns a type-dependent default value. The `with` expression is required for class and port types. The syntax is:

```
function data_type SmartQueue.max() [with index : (expression)];
```

In the first example shown in Example 8-52, NTB-OV returns the maximum value element. In the second example, NTB-OV returns the element for member variable `x` that has the maximum value. In the third example, NTB-OV reruns the element for the product of member variable `x` and `y` that has the maximum value.

Example 8-52 max () Call

```
value = queue1.max();
object = queue2.max() with index : (queue2[index].x);
object = queue1.max() with index : (queue1[index].x *
      queue1[index].y);
```

max_index ()

The `max_index()` method finds the index of the maximum value element. If the queue is empty or no match is found, NTB-OV returns –1. The syntax is:

function integer *SmartQueue***.max_index();**

In Example 8-53, NTB-OV returns the index of the maximum value element.

Example 8-53 max_index () Call

```
value = queue1.max_index();
```

min ()

The `min()` method finds the minimum value element using the specified `expression`. If the queue is empty, NTB-OV issues a warning message and returns a type-dependent default value. The `with` expression is required for class and port types. The syntax is:

function *data_type SmartQueue***.min()**[**with index :** (*expression*)]**;**

The first example shown in Example 8-54 returns the minimum value element. In the second example, NTB-OV returns the element for the member of variable `x` that has the minimum value. In the third example, NTB-OV returns the element for the product of member variables `x` and `y` that has the minimum value.

Example 8-54   min () Call

```
value = queue1.min();
// Queue1 is of type integer or reg
object = queue2.min() with index : (queue2[index].x);
      object = queue1.min() with index :
      (queue1[index].x * queue1[index].y);
```

min_index ()

The `min_index()` method finds the index of the minimum value element. If the queue is empty or no match is found, NTB-OV returns –1. The syntax is:

function integer *SmartQueue*.**min_index();**

The example shown in Example 8-55 returns the index of the minimum value element.

Example 8-55   min_index () Call

```
value = queue1.min_index();
```

pick ()

The `pick()` method picks an element from a random index. If the queue is empty, NTB-OV issues a warning message and returns a type-dependent default value. The syntax is:

function *data_type SmartQueue*.**pick();**

Example shows an example `pick()` call.

pick () Call

```
value = queue1.pick();
// Returns the element at a random index
```

pick_index ()

The `pick_index()` method picks a random index. If the queue is empty or no match is found NTB-OV returns –1. The syntax is:

```
function integer SmartQueue.pick_index();
```

Example 8-56 shows an example `pick_index()` call.

Example 8-56 pick_index () Call

```
// Returns a random index
index = queue1.pick_index();
```

pop_back ()

The `pop_back()` method removes and returns the last element in the queue. NTB-OV issues an error message if the queue is empty. The syntax is:

```
function data_type SmartQueue.pop_back();
```

Example 8-57 shows an example `pop_back()` call.

Example 8-57 pop_back () Call

```
element1 = queue1.pop_back();
```

pop_front ()

The `pop_front()` method removes and returns the first element in the queue. NTB-OV issues an error message if the queue is empty. The syntax is:

```
function data_type SmartQueue.pop_front();
```

push_back ()

The `push_back()` method inserts a new element at the end of the queue. The syntax is:

```
task SmartQueue.push_back(data_type element);
```

Example 8-58 shows a `push_back()` call.

Example 8-58 push_back () Call

```
SQint.push_back(9); // Last element will be 9.
```

push_front ()

The `push_front()` method inserts the specified `element` at the beginning of the queue. The syntax is:

```
task SmartQueue.push_front(data_type element);
```

Example 8-59 push_front () Call

```
SQint.push_front(5); // First element will be 5.
```

reserve ()

The `reserve()` method allocates memory for the number of elements specified by `value`. If `value` is smaller than the current size, NTB-OV does not reserve any extra memory. Because you use this method to override the internal reallocation mechanism, it is for advanced users only. If you know the capacity to which the queue must eventually grow, it is usually more efficient to allocate that memory all at once rather than relying on the built-in automatic reallocation scheme. The syntax is:

```
task SmartQueue.reserve(integer value);
```

Example 8-60 shows an example `reserve()` call.

Example 8-60 reserve () Call

```
integer SQint[$];
SQint.capacity();     // Suppose it is 500 elements
SQint.reserve(490);   // No extra memory allocated
SQint.reserve(510);  // 10 extra memory locations
                     // allocated.
SQint.reserve(SQint.capacity() + 5);  // 5 extra
                    // memory locations allocated.
```

reverse ()

The `reverse()` method puts all elements in reverse order. The syntax is:

```
task SmartQueue.reverse();
```

Example 8-61 shows an example `reverse()` call.

Example 8-61 reverse () Call

```
integer SQint[$] = {1, 2, 3, 4};
// SQint contains 1, 2, 3, 4
SQint.reverse();
// now SQint contains 4, 3, 2, 1
```

rsort ()

The `rsort()` method sorts the elements in the queue in descending order using the specified `expression`. NTB-OV ignores elements that have a null value. NTB-OV also ignores X or Z bits and moves them to the end the queue. The `with` expression is required for class and port types. The syntax is:

```
task SmartQueue.rsort()[with index : (expression)];
```

In Example 8-62, NTB-OV sorts the elements in the queue in descending order. In the second example, NTB-OV sorts the elements in descending order using member variable `x`. In the third example, NTB-OV sorts the elements in descending order using the product of member variables `x` and `y`.

Example 8-62 rsort () Call

```
queue1.rsort();
queue2.rsort() with index : (queue2[index].x);
queue2.rsort() with index : (queue2[index].x *
queue2[index].y);
```

size ()

The `size()` method returns the size of the Smart Queue. The syntax is:

function integer *SmartQueue*.**size();**

Example 8-63 shows an example `size()` call.

Example 8-63 size () Call

```
value = queue.size(); //  Returns size of queue.
```

You can also the `size()` method to retrieve the size of a dynamic array, as shown in Example 8-64.

Example 8-64 Retrieving Size of Dynamic Array

```
program size_ex {
      integer packet_size[*];
      integer array_size;
      packet_size = new[100];
      array_size = packet_size.size();
            printf("packet_size is %0d\n",array_size);
}
```

This example prints:

```
packet_size is 100
```

sort ()

The `sort()` method sorts the elements in the queue in ascending order using the specified `expression`. NTB-OV ignores elements that have a null value. NTB-OV also ignores X or Z bits and moves them to the front of the queue. The `with` expression is required for class and port types. The syntax is:

```
task SmartQueue.sort() [with index : (expression)];
```

In Example 8-65, the first example sorts the queue in ascending order by element values. In the second example, NTB-OV sorts the elements in the queue in ascending order by the value of member variable `x`. In the third example, NTB-OV sorts the elements in the queue using the product of member variables `x` and `y`.

Example 8-65 sort () Call

```
queue1.sort();
queue2.sort() with index : (queue2[index].x);
queue2.sort() with index : (queue2[index].x *
      queue2[index].y);
```

sum ()

The `sum()` method computes the sum of all elements specified using the `with` construct and `expression`. If the array is of type integer or bit vector, and the `with` construct is not used, `sum()` calculates the sum of all elements. For all other types of queues the `with` construct is required. The NTB-OV `sum()` method returns an integer or bit, depending on the data type of the queue. NTB-OV issues a warning if the queue is empty and the returned value is unknown. The syntax is:

```
function integer SmartQueue.sum()[with index : (expression)];
```

Example 8-66 shows an example `sum()` call.

Example 8-66 sum () Call

```
value = SQint.sum();
// Calculates sum of all elements
value = SQPacket.sum() with ptr:
      (SQPacket[ptr].pkt_size);
// Computes sum of member pkt_size from all objects.
```

unique ()

The `unique()` method finds all unique elements using the specified `expression`. If the queue is empty or no match is found, NTB-OV returns an empty queue. With this method, the `with` expression is optional. The syntax is:

```
function data_type[$] SmartQueue.unique()[with index : (expression)];
```

In Example 8-67, NTB-OV only returns unique elements from the queue. In the second example, NTB-OV returns unique elements using member variable `x`.

Example 8-67 unique () Call

```
queue2 = queue1.unique();
queue4= queue3. unique() with index :
 (queue3[index].x);
```

unique_index ()

The `unique_index()` method finds all unique indices using the specified `expression`. If the queue is empty or no match is found, NTB-OV returns an empty queue. With this method, the `with` expression is optional. The syntax is:

```
function integer[$] unique_index()[with index : (expression)];
```

In Example 8-68, the first example only returns indices of the unique elements. In the second example, NTB-OV returns the indices of unique elements using member variable x.

Example 8-68 unique_index () Call

```
queue2 = queue1.unique_index();
queue4= queue3. unique_index() with index :
(queue3[index].x);
```

Functional Coverage Group Methods

NTB-OV provides numerous methods that you can use with functional coverage groups.

Predefined Coverage Group Tasks and Functions

You can invoke the NTB-OV predefined coverage group methods on an instance of a coverage group; they use the same syntax as invoking class functions and tasks on an object.

- Although you invoke predefined methods on a coverage group instance, some of these methods are related to the cumulative coverage information that NTB-OV maintains for the coverage group as a whole. For example, the `query()` function returns coverage information related to the coverage group as a whole. On the other hand, the `inst_query()` function returns coverage information for the coverage group instance on which it is invoked. If you don't set the cumulative attribute of a coverage group to `OFF`, functions that return instance-based information return a -1.

All predefined coverage group methods whose names are prefixed by `inst_` are related to instance-based coverage information. For more information on cumulative and instance-based coverage information, see "Querying Cumulative and Instance-Based Information" on page 310.

Predefined Functions for the coverage_group Construct

You can invoke predefined functions on an instance of a coverage group. For more detailed information on the coverage group syntax, see "Coverage Group" on page 478). The syntax is:

```
function return_type coverage_group_instance_name.function_name
                                                  ([arguments]);
```

`coverage_group_instance_name`

For stand-alone coverage groups, *`coverage_group_instance_name`* is the name of the coverage group instance.

For class-embedded coverage groups, *`coverage_group_instance_name`* has the following form:

```
object_name.coverage_group_name
```

where *`object_name`* is the name of the class instance, and *`coverage_group_name`* is the name of the coverage group construct embedded in the class.

`function_name`

is the predefined coverage group function.

`arguments`

some of the predefined coverage group functions accept arguments.

`return_type`

all predefined coverage group functions have an integer return type, except `query_str()`, which has a string return type.

Predefined Coverage Group Functions

Following are descriptions of all the NTB-OV predefined coverage group functions.

`get_at_least()`

Returns the value of the cumulative `at_least` attribute for the coverage group definition. Does not require arguments.

`get_auto_bin_max()`

Returns the value of the cumulative `auto_bin_max` attribute for the coverage group definition. Does not require arguments.

`get_cov_weight()`

Returns the value of the cumulative `cov_weight` attribute for the coverage group definition. Does not require arguments.

`get_coverage_goal()`

Returns the value of the cumulative `coverage_goal` attribute for the coverage group definition. Does not require arguments.

`inst_get_at_least()`

Returns the value of the `at_least` attribute for the coverage group instance. Does not require arguments.

`inst_get_auto_bin_max()`

Returns the value of the `auto_bin_max` attribute for the coverage group instance. Does not require arguments.

`inst_get_collect()`

Returns the value of the `collect` attribute for the coverage group instance. Does not require arguments.

`inst_get_cov_weight()`

Returns the value of the `cov_weight` attribute for the coverage group instance. Does not require arguments.

`inst_get_coverage_goal()`

Returns the value of the `coverage_goal` attribute for the coverage group instance. Does not require arguments.

`inst_query()`

Queries for coverage information related to the coverage group instance. Requires arguments.

`inst_set_bin_activation()`

Returns the number of bins affected by the function. Requires arguments.

`set_bin_activation()`

Returns the number of bins affected by the function. Requires arguments.

`query()`

Queries for coverage information related to the coverage group as a whole (cumulative). Requires arguments.

`query_str()`

Queries for the name of the current sample or cross bin in a `query(FIRST)`/`query(NEXT)` or `inst_query(FIRST)`/ `inst_query(NEXT)`) sequence. Requires arguments.

Predefined Tasks for the coverage_group Construct

You can invoke predefined tasks can on an instance of a coverage group. The syntax is:

```
task coverage_group_instance_name.task_name (argument);
```

`coverage_group_instance_name`

For stand-alone coverage groups, *`coverage_group_instance_name`* is the name of the coverage group instance.

For class embedded coverage groups, *`coverage_group_instance_name`* has the following form:

```
object_name.coverage_group_name
```

where *`object_name`* is the name of the class instance, and *`coverage_group_name`* is the name of the coverage group construct embedded in the class.

`task_name`

is the predefined coverage group task.

`argument`

all predefined coverage group tasks require an argument.

Predefined Coverage Group Tasks

Following are descriptions of all the NTB-OV predefined coverage group tasks.

`set_at_least()`

Sets the cumulative `at_least` attribute, which is used for computing cumulative coverage statistics. Argument type is integer.

`set_auto_bin_max()`

Sets the cumulative `auto_bin_max` attribute, which is used for controlling cumulative coverage information. Argument type is integer.

`set_cov_weight()`

Sets the cumulative `cov_weight` attribute, which is used for computing cumulative coverage statistics. Argument type is integer.

`set_coverage_goal()`

Sets the cumulative `coverage_goal` attribute, which is used for computing cumulative coverage statistics. Argument type is integer.

`inst_set_at_least()`

Sets the `at_least` attribute for the coverage group instance. Argument type is integer.

`inst_set_auto_bin_max()`

Sets the `auto_bin_max` attribute for the coverage group instance. Argument type is integer.

`inst_set_collect()`

Sets the `collect` attribute for the coverage group instance (turning off coverage data collection for that instance). Argument type is integer.

`inst_set_cov_weight()`

Sets the `cov_weight` attribute for the coverage group instance. Argument type is integer.

`inst_set_coverage_goal()`

Sets the `coverage_goal` attribute for the coverage group instance. Argument type is integer.

`load()`

Loads data for the coverage group instance from the coverage group database. Argument type is string.

`set_name()`

Sets the name of the coverage group instance. Argument type is string.

Predefined Coverage Group Tasks and Functions for Sample and Cross Constructs

NTB-OV also provides predefined tasks and functions for the samples and crosses of a coverage group. You can invoke these methods on a sample or cross of a coverage group instance. When invoked on a sample or cross, predefined coverage group functions have the following syntax:

```
function integer coverage_instance_name.member_name.method_name
([arguments]);
```

When invoked on a sample or cross, predefined coverage group tasks have the following syntax:

```
task coverage_instance_name.member_name.method_name (argument);
```

`coverage_instance_name`

For stand-alone coverage groups, *coverage_instance_name* is the name the coverage group instance.

For class embedded coverage groups, *coverage_instance_name* has the following form:

```
object_name.coverage_group_name
```

where *object_name* is the name of the class instance, and *coverage_group_name* is the name of the coverage group construct embedded in the class.

`member_name`

is the name of the coverage group sample or cross construct.

`argument`

all predefined coverage group tasks require an argument.

`method_name`

is the name of a predefined task or function that can be invoked on a sample or cross of the coverage group construct.

The methods listed in the next three subsections are a subset of predefined methods for the coverage group construct.

Predefined Coverage Group Methods that can be Invoked on a Sample or Cross

Following are descriptions of all the NTB-OV predefined coverage group methods that you can invoke on a sample or cross.

`get_at_least()`

Returns the value of the cumulative `at_least` attribute for a sample or cross.

`get_cov_weight()`

Returns the value of the cumulative `cov_weight` attribute for a sample or cross.

`get_coverage_goal()`

Returns the value of the cumulative `coverage_goal` attribute for a sample or cross.

`inst_get_at_least()`

Returns the value of the `at_least` attribute for a sample or cross of a coverage group instance.

`inst_get_cov_weight()`

Returns the value of the `cov_weight` attribute for a sample or cross of a coverage group instance.

`inst_get_coverage_goal()`

Returns the value of the `coverage_goal` attribute for a sample or cross of a coverage group instance.

`inst_query()`

Queries for coverage information related to a sample or cross of a coverage group instance.

`query()`

Queries for cumulative coverage information related to a sample or cross of a coverage group.

`query_str()`

Queries for the name of the current sample or cross bin in a `query(FIRST)/query(NEXT)` or `inst_query(FIRST)/inst_query(NEXT)` sequence.

`set_at_least()`

Sets the cumulative `at_least` attribute (used for computing cumulative coverage statistics).

`set_cov_weight()`

Sets the cumulative `cov_weight` attribute (used for computing cumulative coverage statistics).

`set_coverage_goal()`

Sets the cumulative `coverage_goal` attribute (used for computing cumulative coverage statistics).

`inst_set_at_least()`

Sets the `at_least` attribute for the coverage group instance.

`inst_set_bin_activation()`

Activates/deactivates a user-defined state/transition bin for a coverage group instance.

`inst_set_cov_weight()`

Sets the `cov_weight` attribute for the coverage group instance.

`inst_set_coverage_goal()`

Sets the `coverage_goal` attribute for the coverage group instance.

`set_bin_activation()`

Deactivates/activates a user-defined state/transition bin for a coverage group definition and the instance on which the function was invoked.

Predefined Coverage Group Methods that can only be Invoked on a Sample of the coverage_group Construct

Following are descriptions of all the NTB-OV predefined coverage group methods that you can invoke only on a sample of the coverage group construct.

`get_auto_bin_max()`

Returns the value of the cumulative `auto_bin_max` attribute for a sample.

`inst_get_auto_bin_max()`

Returns the value of the `auto_bin_max` attribute for a sample of coverage group instance.

`set_auto_bin_max()`

Sets the cumulative `auto_bin_max` attribute that is used to control cumulative coverage information on a sample.

`inst_set_auto_bin_max()`

Sets the `auto_bin_max` attribute for a sample of a coverage group instance.

Reporting and Querying Coverage Numbers

NTB-OV reports testbench coverage data in coverage HTML and text reports. The reports include detailed information for each coverage group along with the samples and crosses of each group.

You can also query for the testbench coverage during a simulation run. This allows you to react to the coverage statistics dynamically (for example, stop the simulation run when the testbench achieves a particular coverage).

The following system function returns the cumulative coverage (an integer between 0 and 100) for the testbench:

```
function integer get_coverage();
```

The following system function returns the instance-based coverage (an integer between -1 and 100) for the testbench:

```
function integer get_inst_coverage();
```

Note:

The `get_inst_coverage()` system function returns -1 when there is no instance-based coverage information (that is, the cumulative attribute of the coverage group has not been set to 0).

For details on how to query for the coverage of individual sample and crosses of each coverage group, see “Runtime Access to Coverage Results” on page 305.

Loading Coverage Data

You can load both cumulative and instance-specific coverage data. When you load coverage data from a previous simulation run, this implies that the bin hits from the previous run are to be added to this run.

Loading Cumulative Coverage Data

To load cumulative coverage data for all coverage groups, use the following syntax:

```
coverage_load("database_file");
```

This command directs NTB-OV to find the cumulative coverage data for all coverage groups found in the `database_file` and load this data if there is a coverage group with the appropriate name and definition in this simulation run.

Note:

> That the tool maps the `database_file` internally to an appropriate unified coverage directory and a corresponding database test name.

For example: `coverage_load (a/b/c/foo.db)` internally maps to top-level unified coverage directory: `a/b/c/simv.vdb` and test name: `foo.`

To load the cumulative coverage data for just a single coverage group, use the following syntax:

```
coverage_load("database_file","coverage_group_name");
```

In Example 8-69, there is a NTB-OV class `MyClass` with an embedded coverage object `covType`. NTB-OV finds the cumulative coverage data for the coverage group `MyClass:covType` in the database file `Run1.db` and loads it into the `covType` embedded coverage group in `MyClass`.

Example 8-69 Loading Cumulative Coverage Information

```
MyClass {
      integer m_e;
      coverage_group covType {
            sample_event = wait_var(m_e);
            sample m_e;
      }
}
...
coverage_load("Run1.db", "MyClass::covType");
```

Loading Instance Coverage Data

To load coverage data for a stand-alone coverage instance, use the following syntax:

```
coverage_instance.load("database_file");
```

To load the coverage data for an embedded coverage instance, use the following syntax:

```
class_object.cov_group_name.load("database_file");
```

This command directs NTB-OV to find the coverage data for the specified instance name in the database, and load it into the coverage instance.

In Example 8-70, there is an NTB-OV class named `MyClass` with an embedded coverage object named `covType`. Two objects (`obj1` and `obj2`) are instantiated, each with the embedded coverage group `covType`. NTB-OV finds the coverage information for the coverage instance `obj1:covType` in the `Run1.db` database file and loads this coverage data into the newly instantiated `obj1` object. Note that object `obj2` is not be affected as part of this load operation.

Example 8-70 Loading Instance Coverage Data

```
MyClass {
      integer m_e;
      coverage_group covType {
            sample_event = wait_var(m_e);
            sample m_e;
      }
}
...
MyClass obj1 = new;
obj1.load("Run1.db");
MyClass obj2 = new;
```

File Control

All testbench functional coverage data is stored in a coverage directory named simv.vdb. This is different from previous versions of VCS, where coverage database files were stored in the current working directory or the path specified by the `coverage_database_filename`. VCS assigns a default test name to the coverage data generated during simulation. You can override this name using the `coverage_set_database_file_name` task. The syntax is:

```
task coverage_set_database_file_name (file_name);
```

VCS avoids overwriting existing database file names by generating unique test names for consecutive tests.

For example, if the coverage data is to be saved to a test name called pci_test, and a database with that test name already exists in the coverage directory (simv.vdb), VCS automatically generates the new name pci_test_gen1 for the next simulation run. The following table explains the name generation details:

Test Name	Database
pci_test	Database for the first testbench run.
pci_test_gen_1	Database for the second testbench run.
pci_test_gen_2	Database for the 3rd testbench run.
pci_test_gen_n	Database for the *n*th testbench run.

You can disable this method of ensuring database backup and force VCS to always overwrite an existing coverage database using the following system task:

```
task coverage_backup_database_test (flag );
```

The value of `flag` can be `OFF` for disabling database backup or `ON` for enabling database backup. If you don't want coverage_backup_database_test to save coverage data to a database file (for example, if there is a verification error), use the following system task:

```
task coverage_save_database (flag);
```

The value of `flag` can be `OFF` for disabling database backup or `ON` for enabling database backup.

Runtime Access to Coverage Results

Use the `query()` function to monitor functional coverage group statistics dynamically during simulation, and react to the coverage results. You can invoke the `query()` function on a coverage group instance or a sample or cross of a coverage group instance. You can use the `query()` function to:

- obtain the current coverage number (percentage) of a coverage group or a sample or cross of a coverage group
- query whether the coverage goal of a group (or a sample or cross of a coverage group) is met
- count the number of hits for a particular sample or cross bin
- count the number of bins of a particular type (state or transition) of a sample or cross
- iterate through the bins of a coverage group, or a sample or cross of a coverage group

Note that the `query()` function retrieves cumulative information for the entire coverage group, not a particular instance of a coverage group. If you set the cumulative attribute of a coverage group to `OFF`, you can retrieve both cumulative and instance-based coverage information. In that case, you can use the `inst_query()` function to retrieve instance-specific information and the `query()` function to retrieve cumulative coverage information. The syntax for `inst_query()` is the same as `query()`. The `query()` functions are in these forms:

```
function integer query(command);
function integer query(command, bin_type);
function integer query(command, bin_type,bin_pattern);
function integer query(command, bin_type,bin_pattern,
                            operand, hit);
```

The basic form of `query()` allows you to specify one of several commands: `COVERAGE_DENOMINATOR`, `COVERAGE`, `NUM_BIN`, `SUM`, `FIRST`, `NEXT`, `GOAL`, and `SAMPLE`. Each command returns a different value based on the `bin_type`, as shown in Table 8-5.

Table 8-5 query () Commands

Command	Function
`COVERAGE_DENOMINATOR`	If you invoke `query(COVERAGE_DENOMINATOR)` on the same coverage point, it returns the a total number of possible bins (the denominator used in coverage number computation). This function works the same way on a cross construct.
`COVERAGE`	Returns the current coverage number (percentage). Do not pass any other arguments when using this command.
`GOAL`	Returns a 1 if the coverage goal is met (0 otherwise). Do not pass any other arguments when using this command.
`NUM_BIN`	Counts the number of bins of type *`bin_type`*.
`SUM`	Sums the counters for all bins of type *`bin_type`*.
`FIRST`	Returns the count of the first bin of type *`bin_type`* (or -1 for failure). Starts an iteration sequence through bins of type *`bin_type`*.
`NEXT`	Returns the count of the next bin of type *`bin_type`* in the sequence started with `FIRST` (or -1 if the sequence is complete). Do not pass any other arguments when using this command.
`SAMPLE`	Returns the sampled value of the indicated coverage point (sample construct). This command requires an additional integer argument that specifies the depth of the sampled value.

The single-argument form of the command applies to all bins. You can the additional arguments to narrow the bin selection.

`bin_type`

Valid bin types are:

- STATE
- BAD_STATE

- TRANS
- BAD_TRANS

You can specify multiple bin types using the or operator (|).

`bin_pattern`

NTB-OV matches the `bin_pattern` against the bin names of the specified type. The pattern can be any Perl regular expression. Only bins whose names contain the `bin_pattern` are included in the query. For example, if `bin_pattern` is `bus`, all bins with `bus` in their names are included. If you want to select all bins whose names begin with a specific string, insert a caret (^) before the string (such as `^bus`). To select all bins of the specified type regardless of name, use `.*` as the `bin_pattern`.

`operand`

The operand must be one of:

- GT (greater than)
- GE (greater than or equal to)
- LT (less than)
- LE (less than or equal to)
- EQ (equal to)
- NE (not equal to)

`hit`

specifies the number of counter hits to which the query is compared using the operand. It can be any non-negative integer.

Outside of a coverage declaration you need to qualify `query()` to get the appropriate method. For example, to get the query method for the cp0 coverage object, you use `cp0.query()`.

Querying Coverage Objects, Samples and Crosses (Examples)

You can invoke the `query()` method on:

- coverage group instances (embedded and stand-alone)
- samples and crosses of coverage group instances

When you invoke the `query()` method on a coverage instance, the method operates on all samples and cross constructs within the coverage instance. The coverage numbers and bin counts returned are the aggregation across all samples and constructs. Example 8-71 shows an example `query()` call.

Example 8-71 query () Call

```
class MyClass {
      integer m_x, m_y;
      coverage_group covType {
            sample_event = wait_var(m_e);
            sample m_x;
            sample m_y {
                  m_state (0:10);
                  m_trans (0:10->0:10);
            }
            cross cc1 (m_x, m_y);
      }
}
...
MyClass obj1 = new;
...
```

With Example 8-71, the following query returns the coverage number (coverage percentage) for the embedded coverage group `covType` of class `MyClass`:

```
numCrossBins = obj1.covType.query(COVERAGE);
```

For Example 8-71, the following query returns the total hit count of all sample and cross bins in embedded `covType` coverage group for object `obj1`.

```
numBins = obj1.covType.query(SUM);
```

For Example 8-71, the following query returns 1 if the coverage goal associated with the `m_x` sample in the embedded coverage group is met.

```
done = obj1.covType.m_x.query(GOAL);
```

For Example 8-71, the following query returns the number of bins created for the `cc1` cross in the embedded coverage group `covType` of object `obj1`.

```
numCrossBins = obj1.covType.cc1.query(NUM_BIN);
```

The following statement computes the sum of the bin counts for state or transition bins with counts greater than `10`:

```
numBins = obj1.covType.query(SUM, STATE|TRANS, ".*", GT, 10);
```

The following example loops until at least 10 bins of sample `m_y` have `10` or more hits:

```
while (obj1.covType.m_y.query(NUM_BIN,
STATE|TRANS, ".*", GT, 10) < 11) {
...
}
```

You can also set up loops that continue processing until the coverage goal is met, as shown in this next example:

```
while (!obj1.covType.query(GOAL)){...}
```

Querying Cumulative and Instance-Based Information

If you set the cumulative attribute of a coverage group to `OFF`, you can retrieve both cumulative and instance-based coverage information. In that case, you can use the `inst_query()` function to retrieve instance-specific information and the `query()` function to retrieve cumulative coverage information. The syntax for the `inst_query()` function is the same as the `query()` function, as shown in Example 8-72:

Example 8-72   query () and inst_query () Calls

```
class MyClass {
      integer m_var;
      coverage_group MyCov {
            sample m_var;
            sample_event = @(posedge CLOCK);
            cumulative = OFF;
      }
}

program simple {
      MyClass obj1 = new();
      MyClass obj2 = new();
      @(posedge CLOCK);
      obj1.m_var = 100;
      obj2.m_var = 200;
      @(posedge CLOCK);
      obj1.m_var = 300;
      @(posedge CLOCK);

      printf("MyCov.m_var bins: %0d\n",
            obj1.MyCov.m_var.query(NUM_BIN));
      printf("MyCov.m_var bins: %0d\n",
            obj2.MyCov.m_var.query(NUM_BIN));
      printf("obj1.MyCov.m_var bins: %0d\n",
            obj1.MyCov.m_var.inst_query(NUM_BIN));
      printf("obj2.MyCov.m_var bins: %0d\n",
            obj2.MyCov.m_var.inst_query(NUM_BIN));
}
```

Example 8-72 produces the following output:

```
MyCov.m_var bins: 3
```

```
MyCov.m_var bins: 3
obj1.MyCov.m_var bins: 2
obj2.MyCov.m_var bins: 1
```

In Example 8-72, the cumulative attribute of embedded coverage group `MyCov` is set to `OFF`. There are two instances of class `MyClass` and therefore two instances of coverage group `MyCov`.

When the query function is invoked on `obj1.MyCov.m_var`, NTB-OV returns the number of bins created for sample `m_var` of coverage group `MyCov` as a whole. This is the same number that is returned when the `query()` function is invoked on `obj2.MyCov.m_var`. On the other, hand the `inst_query()` function calls return the number of bins created for the `m_var` sample of each `MyCov` instance. This number differs for `obj1.MyCov.m_var` and `obj2.MyCov.m_var`.

If you left the cumulative attribute at its default value of ON, the print statements in Example 8-72 would produce the output shown in Example 8-73:

Example 8-73 Output with Cumulative Attribute ON

```
MyCov.m_var bins: 3
MyCov.m_var bins: 3
obj1.MyCov.m_var bins: -1
obj2.MyCov.m_var bins: -1
```

When the cumulative attribute is `ON`, VCS-OV only maintains cumulative information for the coverage group as whole. In that case VCS-OV does not maintain instance-specific information.

Querying The Name of a Bin

You can also query the name of a monitor bin using the predefined `query_str()` function of the coverage group construct. You can invoke this function on a coverage group instance or the sample or cross of a coverage group instance. The syntax is:

```
function string query_str(NAME);
```

This function returns the monitor bin name in a FIRST/NEXT series (or NOT_FOUND if the series is complete).

Querying for Coverage Denominator

To get the value of the denominator used for coverage calculations, use the predefined `query()/inst_query()` functions. The syntax is:

```
function integer query(COVERAGE_DENOMINATOR);
function integer inst_query(COVERAGE_DENOMINATOR);
```

Given a coverage point, NTB-OV computes the coverage number for this coverage point as the number of bins with at_least number of hits divided by the total number of possible bins (denominator) for that coverage point.

If you invoke `query(COVERAGE_DENOMINATOR)` on the same coverage point, it returns the same number of possible bins (the denominator used in coverage number computation). This function works the same way on cross constructs.

If you invoke this function on a coverage group, the query returns the sum of coverage denominators for all the coverage points and crosses within the coverage group.

Temporal Coverage

To return past values of a coverage group´s coverage point (that is, sample construct of a coverage_group construct) you can use the predefined `query()` function of the coverage group. You can only retrieve previous values if the sample construct associated with that coverage point defines transition bins (using the `trans` construct). The syntax is:

```
function data_type query(SAMPLE, integer depth);
```

`data_type`

can be a reg, bit vector, integer, or enumerated type.

`depth`

is an unsigned integer, which specifies the previously sampled value that is returned. If the sample does not define any transition sequences, depth must be less than 2. If the sample defines transition sequences, `depth` must be less than the length of any of the transition sequences. Otherwise, a simulation error occurs.

If the command to the `query()` function is `SAMPLE`, the function must be invoked on the sample of the coverage group instance.

When you call the `query()` function of a coverage group with the `SAMPLE` keyword, NTB-OV returns the value of the sampled coverage point associated with the `depth`. If you specify a `depth` of 0, NTB-OV returns the last sampled value of the coverage point. This is the value of the coverage point that was sampled the last time that sampling event triggered. If you specify a `depth` of 1, the function returns the sampled value of the coverage point two sampling event triggers prior to the call. If you specify a `depth` that has not yet occurred, NTB-OV returns an unknown value (X).

The following example returns the sampled value of sampled variable `gVar` of coverage group instance `cov1`, three sampling event triggers prior to the call:

```
integer i = c1.query(SAMPLE, 2);
```

Controlling Coverage Collection Globally

You can use the `coverage_control()` system task to enable or disable data collection for one or more coverage groups at the program level. When you combine this task with the -`cg_coverage_control` runtime argument, you get a single-point mechanism for enabling/disabling coverage collection for all coverage groups or a particular coverage group. The syntax is:

```
task coverage_control(integer COV_STOP|COV_START [, string
cov_grp_name]);
```

`COV_STOP`

disables coverage collection for all coverage groups or all instances of the specified coverage group until the program executes a call to `coverage_control()` with the `COV_START` argument.

`COV_START`

enables coverage collection. The `COV_STOP` and `COV_START` macros are defined in the vera_defines.vrh file that NTB-OV includes automatically.

`cov_grp_name`

is the name of coverage group. If you don’t specify this argument the task acts on all coverage groups.

Note that coverage for all coverage groups in a program is enabled by default. For a given instance, coverage information is collected only if you enable `coverage_control()` for the group and the collect attribute of the instance, as shown in the following table.

coverage_control()	**collect**	**Collection Enabled?**
OFF	OFF	No
ON	OFF	No
OFF	ON	No
ON	ON	Yes

The `collect` attribute for an instance is always enabled by default until explicitly set to `OFF` using the `inst_set_collect()` task.

Enabling or disabling coverage on the command line is equivalent to specifying `coverage_control()` at the start of the program. Subsequent `coverage_control()` commands override previous settings. Example 8-74 illustrates the usage.

Example 8-74   Coverage Control

```
coverage_group Cov {
      sample_event = @(posedge CLOCK);
      sample x {
            state x1(10);
            state x2(20);
            state x3(30);
            state x4(40);
            state x5(50);
            state x6(60);
            state x7(70);
       }
}
coverage_group Another {
      sample_event = @(posedge CLOCK);
      sample x {
            state x1(10);
            state x2(20);
            state x3(30);
```

```
            state x4(40);
            state x5(50);
            state x6(60);
            state x7(70);    }
}

task query_and_print(string str) {
      printf("Coverage is %d:%s\n",c.query(COVERAGE), str);
}

program test {
      integer x = 0;
      Cov c = new;
      Another c1 = new;
      @(posedge CLOCK);

      coverage_control(0);
      x = 10;
      @(posedge CLOCK);
      x = 30;
      @(posedge CLOCK);

      coverage_control(1);
      x = 40;
      @(posedge CLOCK);
      x = 50;
      @(posedge CLOCK);

      coverage_control(0, "Cov");
      x = 60;
      @(posedge CLOCK);
      coverage_control(1, "Cov");
      coverage_control(0, "Another");
      x = 70;
      @(posedge CLOCK);
}
```

Predefined Temporal Assertion Classes

NTB-OV provides the following temporal assertion classes:

- AssertEngine Class
- Assertion Class

- AssertEvent Class

You use these classes to interact with OpenVera or SystemVerilog assertions. An assertion is an executable piece of code that specifies how a design should behave. This section explains the syntax and semantics of the public methods for these temporal assertion classes.

AssertEngine Class

Only one object of the AssertEngine class is allowed in an NTB-OV program. This object:

- globally monitors and controls the behavior of both OpenVera and SystemVerilog assertions
- controls the reporting of assertion results
- provides methods to obtain handles to individual assertions and expressions. These handles are used to control the assertions or expressions.

The methods for the AssertEngine class are:

- new ()
- Configure ()
- DoAction ()
- EnableTrigger ()
- DisableTrigger ()
- GetFirstAssert ()
- GetNextAssert ()

- GetAssert ()

new ()

Call the `new()` task to create an object of the AssertEngine class. The syntax is:

```
task new();
```

Configure ()

Call the `Configure()` task to change the reporting configuration. By default, line information in assertion messages is not shown, runtime assertion messages are not printed, and the assertion report at the end of a simulation is not printed. The syntax is:

```
task Configure(integer operation, integer value);
```

`operation`

valid values for *`operation`* are:

Table 8-6 Operations

0peration	Definition
ASSERT_INFO	Show line information in messages
ASSERT_QUIET	Don’t print messages at runtime
ASSERT_REPORT	Print report at the end of simulation

`value`

enables or disables the specified *`operation`*. Valid values are:

- `ASSERT_FALSE`, which disables *`operation`*
- `ASSERT_TRUE`, which enables *`operation`*

DoAction ()

The `DoAction()` task globally resets or terminates all assertion attempts. The syntax is:

```
task DoAction(integer action);
```

`action`

The valid values for *`action`* are:

Table 8-7 Actions

Action	Definition
ASSERT_RESET	Resets all assertions and expressions. All attempts to match assertions and expressions end immediately without reporting results. New attempts start normally with the next clock cycle.
ASSERT_TERMINATE	Terminates all attempts to match assertions and expressions. New attempts do not start. Recovers the memory space of the assertion objects. Once this action is taken, you cannot restart assertion activities.

EnableTrigger ()

Use the `EnableTrigger()` member task to associate an AssertEvent object with the AssertEngine object. The syntax is:

```
task EnableTrigger(AssertEvent ev);
```

`ev`

is the AssertEvent object to associate with the AssertEngine object.

DisableTrigger ()

Use the `DisableTrigger()` member task to disassociate the specified AssertEvent object from the AssertEngine object. You should use this method to recover the memory space allocated by the `EnableTrigger()` member task. The syntax is:

```
task DisableTrigger(AssertEvent ev);
```

`ev`

is the AssertEvent object to disassociate from the AssertEngine object.

GetFirstAssert ()

The `GetFirstAssert()` function returns a handle to the first assertion or expression in the compiled list. This may not be the first item seen in the assertions file. The syntax is:

```
function Assertion GetFirstAssert();
```

GetNextAssert ()

The `GetNextAssert()` function returns a handle to the next assertion or expression in the compiled list. This function returns null if there are no more assertions or expressions. If you want to go back to the beginning of the list, use `GetFirstAssert()`. The syntax is:

```
function Assertion GetNextAssert();
```

GetAssert ()

The `GetAssert()` function returns an Assertion object, which is a handle to an assertion or expression. The syntax is:

```
function Assertion GetAssert(string name);
```

`name`

is the full hierarchical name of the assertion or expression to access.

Assertion Class

The following Assertion method objects allow you to monitor and control individual assertions and expressions:

- GetName ()
- EnableCount ()
- GetCount ()
- EnableTrigger ()
- DisableTrigger ()
- DoAction ()

GetName ()

The `GetName()` method returns the name of the assertion or expression. The syntax is:

```
function string GetName();
```

EnableCount ()

The `EnableCount()` method enables automatic counting of success or failure evaluation attempts. No AssertEvent object is needed to run `EnableCount()`. To obtain the current value of the count, use the `GetCount()` function. The syntax is:

```
task EnableCount(integer event_type);
```

`event_type`

can be either `ASSERT_SUCCESS` or `ASSERT_FAILURE`.

GetCount ()

The `GetCount()` method returns the current count of the specified `event_type`. Counting starts after you call the `EnableCount()` task. The syntax is:

```
function integer GetCount(integer event_type);
```

`event_type`

can be either `ASSERT_SUCCESS` or `ASSERT_FAILURE`.

EnableTrigger ()

The `EnableTrigger()` task associates an AssertEvent object with the Assertion object. You can enable multiple AssertEvent objects of the same type for the same Assertion object, but each AssertEvent object can only be associated with one Assertion object. The syntax is:

```
task EnableTrigger(AssertEvent event_name);
```

`event_name`

is the AssertEvent object to be associated with the Assertion object.

DisableTrigger ()

The `DisableTrigger()` member task disassociates the specified AssertEvent object from the Assertion object. You can use this method to recover the memory space allocated by the `EnableTrigger()` member task. The syntax is:

```
task DisableTrigger(AssertEvent event_name);
```

`event_name`

is the AssertEvent object to disassociate from the Assertion object.

DoAction ()

By default, Assertion objects are enabled for assertions and disabled for expressions. Use the `DoAction()` member task to change the state of the assertion or expression referred to by the Assertion object. The syntax is:

```
task DoAction(integer action);
```

action

can be ASSERT_RESET, ASSERT_DISABLE or ASSERT_ENABLE (see Table 8-8 for definitions).

Table 8-8 Assertion Actions

Action	Definition
ASSERT_RESET	Resets the assertion or expression. All attempts to match end immediately without results. New attempts start normally with the next clock cycle.
ASSERT_DISABLE	Disables the assertion or expression. Immediately ends all attempts to match. New attempts do not start.
ASSERT_ENABLE	Enables the assertion or expression. New attempts start normally with the next clock cycle. Cancels the effect of the OVA_DISABLE action.

AssertEvent Class

You can use objects of the AssertEvent class to synchronize the testbench with the associated objects of the AssertEngine and Assertion classes. Use the AssertEngine or Assertion's EnableTrigger () and DisableTrigger () methods to enable and disable triggering of the AssertEvent objects. The AssertEvent class members are:

- new ()
- Wait ()
- Event
- GetNextEvent ()

new ()

The syntax is:

```
task new(integer event_type);
```

`event_type`

If you are using the AssertEvent in the context of an AssertEngine object, see Table 8-9 for possible *event_type* values. These events occur as a result of a DoAction () method call from the AssertEngine object.

If you are using the AssertEvent in the context of an Assertion object, see Table 8-10 for possible values of *event_type*.

Table 8-9 Types for use in AssertEngine Context

Event Type	Definition
ASSERT_RESET	Reset sequence completed
ASSERT_TERMINATE	Termination sequence completed

Table 8-10 Types for use in Assertion Context

Event Type	Definition
ASSERT_RESET	All evaluation attempts in progress terminated
ASSERT_FAILURE	Evaluation attempt failed
ASSERT_SUCCESS	Evaluation attempt succeeded
ASSERT_DISABLE	Generation of new evaluation attempts disabled
ASSERT_ENABLE	Generation of new evaluation attempts enabled
ASSERT_ALL	All of the above

Wait ()

Use `Wait()` to suspend the current thread until the event occurs. The syntax is:

```
task Wait();
```

Event

Event is a variable of the type event, which is a basic NTB-OV data type. You can use the Event variable with the `sync()` system task. The syntax is:

```
event Event;
```

GetNextEvent ()

Use `GetNextEvent()` to return the assertion events that unblocked the thread. If more than one assertion event type caused the trigger, you can call the function again until it returns `OVA_NULL`. If you call `Wait()` multiple times, each call to `GetNextEvent()` returns the most recent event since the last call to `Wait()`. The syntax is:

```
function integer GetNextEvent();
```

Predefined Procedures

NTB-OV provides numerous predefined procedures that you can use in your testbench. This section explains the syntax for all of these procedures, which are organized by type.

Formatted Input and Output

The NTB-OV formatted input and output procedures include the following.

printf ()

NTB-OV supports a C-style `printf()` system task that sends information to stdout during simulation. The

syntax is:

```
task printf(string format, argument_list);
```

`format`

is a C-style format string.

`argument_list`

consists of the arguments to be printed.

For `%h, %d, %o, %b`, NTB-OV sizes the values automatically to the maximum possible space needed for the given expression. You can minimize the size by inserting a zero between the `%` character and the radix. Example 8-75 shows a `printf()` call.

Example 8-75 printf () Call

```
printf("Data = %0h Addr = %0h\n", data, addr);
```

The prototype to print a string is:

```
task printf(string format, argument_list);
```

NTB-OV uses several predefined format specifiers (see Table 8-11).

Table 8-11 Format Specifiers

Specifier	Format	Argument Type
`%d` or `%D`	decimal	integer, bit vector, enum
`%h` or `%H` or `%x` or `%X`	hexadecimal	integer, bit vector, enum
`%o` or `%O`	octal	integer, bit vector, enum
`%b or %B`	binary	integer, bit vector, enum
`%u or %U`	unsigned integer	integer, bit vector, enum
`%c or %C`	first character	string
`%s or %S`	a string	string or enum variable
`%m or %M or %v`	hierarchical trace, starting from top-level module to context of the `printf()` statement*	none
`%%`	%	none

NTB-OV handles binary specifiers (`%b`) like hex and octal specifiers. Leading zeroes are added if the width specifier is larger than the value needs, and all of the digits are shown using the minimum space required if the width specifier is smaller than the value needs.

Note that the format specifier, not the variable type, determines how the value is printed. If the variable `b` was declared as an integer in Example 8-76, the same value would be printed.

Example 8-76 More printf () Calls

```
reg[31:0] b = 32'h1234_5678;
```

```
printf(" ‘%d’ ...’%0d’...’%4d’...’%12d’\n", b,b,b,b,);
// gives:
‘305419896’...’305419896’...’305419896’...’   305419896’

printf(" ‘%h’ ...’%0h’...’%4h’...’%12h’\n", b,b,b,b,);
// gives:
’12345678’...’12345678’...’12345678’...’   12345678’
```

The format specifier `%u` prints the argument as an unsigned integer. If you use `%d` or `%i` to print the integer value, the highest positive number that can be printed is 2147483647 (2^{31} -1). NTB-OV treats any integer value higher than 2147483647 as a negative number since the MSB is set. If you use `%u`, NTB-OV treats the whole number as positive, and the printed value can go up to 4294967295 (2^{32} -1). The `%m` format specifier displays the hierarchical trace from the NTB-OV main program to the current context of the `printf()` task (see Example 8-77).

Example 8-77 Hierarchical Trace

```
program inv_test {
          printf("%m\n");
          task_a();
          task_b();
     } // end program

     task task_a() {
          printf("%m\n");
          /* ... user wants to debug here ...*/
     } // end task_a

     task task_b() {
          task_a();
     } // end task_b
```

In Example 8-77 the `inv_test` program calls two tasks: `task_a()` and `task_b()`. The `task_b()` task, in turn, calls `task_a()`. When `task_a()` is called, the `%m` format specifier shows whether `task_a()` is called from the NTB-OV main program or from `task_b()`, and provides the entire hierarchical trace of the context. Example 8-77 generates the following output:

```
inv_test.inv_test
inv_test.task_a
inv_test.task_b
```

You can also use escape characters for formatting (see Table 8-12).

Table 8-12 NTB-OV Escape Characters

Escape String	Character Produced
\n	new line
\t	tab
\\	\
\"	"
\ddd	a character specified in 1 to 3 octal digits

NTB-OV provides a set of operators that you can use to manipulate combinations of string variables and string constants (see Table 1-3 on page 1).

String Manipulation Methods

The NTB-OV string manipulation functions/tasks include the following.

sprintf ()

The `sprintf()` system task sends output to a string variable. The syntax is:

```
task sprintf(string string_name, string format, argument_list);
```

`string_name`

is a string variable.

`format`

is a C-style format string. The legal format specifiers are listed in Table 8-11. NTB-OV assigns the output specified by *`format`* to *`string_name`*.

`argument_list`

the arguments to write into *`string_name`*.

Example 8-78 assigns the string "`S0 is S0 string`" to `str`.

Example 8-78 sprintf () Call

```
string S0;
string str;
S0 = "S0 string";
sprintf(str, "S0 is %s\n", S0);
```

psprintf ()

The `psprintf()` system function is just like the sprintf () system task, except that the printed string is the function's return value, not its first argument. The syntax is:

```
function string psprintf(string format, argument_list);
```

`format`

is a C-like format specifier string (see printf ()).

`argument_list`

consists of the arguments to be printed. For example:

Example 8-79 psprintf () Call

```
if (error_flag)
my_log = (psprintf("Error at time %0d", get_time(LO));
```

sscanf ()

The `sscanf()` system task reads input from a string. The syntax is:

task sscanf(**string** *string_name*, **string** *format argument_list*)**;**

`string_name`

is a string variable.

`format`

is a C-style format string. The legal format specifiers are listed in Table 8-11.

`argument_list`

consists of the arguments to be printed. NTB-OV assigns the input specified by *`format`* to *`argument_list`*.

Example 8-80 assigns the values `123`, `16`, and `45` to the variables `i1`, `i2`, and `i3`, respectively.

Example 8-80 sscanf () Call

```
string s1;
integer i1, i2, i3;
s1 = " 123 'h10 4567";
sscanf(s1, " %d %h %2d", i1, i2, i3);
```

File Access

The NTB-OV file access functions include the following.

fopen ()

The `fopen()` system function opens the specified `filename` and returns an integer that is the 32-bit file descriptor if it succeeds or 0 if it fails. The syntax is:

```
function integer fopen(string filename, string access_mode);
```

`filename`

the file to be opened.

`access_mode`

Table 8-13 defines the access modes.

Table 8-13 access_mode

access_mode	Value
"r" or "rb"	open for reading
"w" or "wb"	truncate or create for writing
"a" or "ab"	append or create for writing
"r+", "r+b", "rb+"	open for update (reading and writing)
"w+", "w+b", or "wb+"	truncate or create for update
"a+", "a+b", or "ab+"	append; open or create for update at end-of-file

fclose ()

The `fclose()` system task closes the specified `file`. The syntax is:

```
task fclose(integer file_descriptor);
```

There are three predefined file descriptors in the vera_defines.vrh file that NTB-OV includes automatically:

- stdin terrminal input
- stdout terminal output
- stderr terminal error output

fprintf ()

The `fprintf()` system task writes the output to the specified `file_descriptor`. The syntax is:

```
task fprintf(integer file_descriptor, string format,
                                      argument_list);
```

`file_descriptor`

indicates the file to write to.

`format`

is the same as the `printf()` system task (see Table 8-11).

`argument_list`

the arguments to print.

freadb ()

The `freadb()` function reads binary formatted data from a specified text `file_descriptor`, line by line, and returns the data as a bit vector (reg). Lines with only white spaces and comments are ignored. The syntax is:

```
function reg freadb(integer file_descriptor);
```

`file_descriptor`

indicates the file to read. Each line of the file must be in the following format:

```
white_space* binary comment*
```

`white_space`

can be any number of spaces.

`binary`

must be a combination of ‘`0`’, ‘`1`’, ‘`z`’, ‘`x`’, ‘`Z`’, ‘`X`’ and ‘`_`’.

`comment`

must be an NTB-OV comment starting with “`//`”.

When the end of the file is reached, the `freadb()` function sets the error flag.

freadh ()

The `freadh()` system function reads hexadecimal data from a specified `file_descriptor`, line by line, and returns it as a bit vector (reg). NTB-OV ignores lines with only white spaces and comments. The syntax is:

```
function reg freadh(integer file_descriptor
                                 [VERBOSE | SILENT]);
```

`file_descriptor`

indicates the file to read. Each line of the file must be in the following format:

```
white_space* hex comment*
```

`mode`

The mode can be either SILENT or VERBOSE. Use the SILENT option to prevent a WARNING and error flag generation when freadh() reads the end of the file. VERBOSE is the default setting, which sets the error flag when the end of the file is reached.

`hex`

must be a combination of '`0`' to '`9`', '`a`' to '`f`', '`A`' to '`F`', '`x`', '`z`', '`X`', '`Z`' and '`_`'.

`comment`

must be an NTB-OV comment starting with "`//`".

When the end of the file is reached, the `freadh()` function sets the error flag.

freadstr ()

The `freadstr()` function returns a string containing a line of text from a specified `file_descriptor`. The returned string does not contain line-feed characters. The syntax is:

```
function string freadstr(integer file_descriptor, [VERBOSE | SILENT |
    RAWIN]);
```

`file_descriptor`

variable indicates the file to read.

Note:

`VERBOSE`, `SILENT`, and `RAWIN` specify the report mode. The default is `VERBOSE`.

`VERBOSE`

When the end of the file is reached, NTB-OV prints a warning and returns a null string. This is the default.

`SILENT`

When the end of the file is reached, NTB-OV returns a null string.

`RAWIN`

When the end of the file is reached, NTB-OV returns a null string, but does not filter out comments and blank lines.

NTB-OV ignores comments and blank lines in the input `file_descriptor` unless you specify `RAWIN` mode. When the end of the file is reached, the `freadstr()` function sets the error flag.

fflush ()

The `fflush()` system task writes buffered data to a specified `file`. The syntax is:

```
task fflush(integer file_descriptor);
```

`file_descriptor`

variable indicates the file to write to.

NTB-OV writes information in the write buffer to a file when:

- the buffer is full
- the file is closed using the `fclose()` system function
- the write buffer is flushed using the `fflush()` system function

feof ()

The `feof()` function returns a non-zero integer when END OF FILE is encountered. The syntax is:

```
function integer feof(integer file_handle);
```

ferror ()

The `ferror()` function returns a non-zero integer when an error occurs in the file stream. The syntax is:

```
function integer ferror(integer file_handle);
```

rewind ()

The `rewind()` system task moves the file access pointer to the beginning of the file. The syntax is:

```
task rewind(integer file_descriptor);
```

`file_descriptor`

indicates the file where the file access pointer is being changed.

lock_file ()

To lock a file, use the `lock_file()` system function. The syntax is:

```
function integer lock_file(string filename, integer timeout);
```

`filename`

must be a valid identifier specifying the file (including the path from the working directory) to be locked.

`timeout`

must be an integer specifying the `timeout` length, in seconds. A 0 value indicates that the function call never times out.

The `lock_file()` function returns a 1 if it is successful, or a 0 if it is unsuccessful.

Simulation Control

The NTB-OV simulation control functions include the following.

exit ()

The `exit()` system task causes the program to terminate. The normal exit status is zero. A non-zero exit status usually indicates an error. In the non-zero case, NTB-OV displays an exit status statement indicating the value specified in the `exit()` call at the end of the simulation summary. The syntax is:

```
task exit(integer status);
```

`status`

must be an integer constant or variable.

stop ()

The `stop()` system task stops the simulation. The syntax is:

```
task stop();
```

This task is equivalent to the Verilog `$stop` task. If you are running a Verilog simulation, the simulation stops, reports that a stop was encountered, and exits to a Verilog prompt. You can issue normal Verilog commands at the command line. To continue the simulation, enter a period (`.`) on the command line.

System Interaction

The NTB-OV system interaction functions include the following.

get_cycle ()

The `get_cycle()` system function returns the current simulation cycle count. The syntax is:

```
function integer get_cycle([signal_name]);
```

`signal_name`

can be any valid interface signal reference (including clocks). The default is SystemClock.

NTB-OV counts the internal simulation cycles at the positive edge of the specified clock signal. If the signal is an interface signal or signal reference, NTB-OV counts the internal simulation cycles at the positive edge of the clock signal defined in the same interface as the specified signal. If there is no specified signal, `get_cycle()` returns the number of SystemClock cycles (see Example 8-81).

Example 8-81 get_cycle Call

```
printf("Current Cycle: %d \n", get_cycle() );
```

get_time ()

The `get_time()` system function returns the current 64-bit simulation time as two 32-bit values. NTB-OV uses the timescale in the Verilog design file. The syntax is:

```
function reg[31:0] get_time(LO | HI);
```

`LO`

returns the lower 32-bit value.

`HI`

returns the higher 32-bit value. Only `LO` is currently implemented.

Example 8-82 shows an example.

Example 8-82 get_time() Call

```
printf("Current Time: %d \n", get_time(LO) );
```

os_command ()

The `os_command()` system function issues commands to the OS shell. The syntax is:

```
function integer os_command(string command);
```

`command`

is the exact string command to issue to the operating system.

The `os_command()` system function uses the UNIX `system()` call to issue commands, and is therefore OS-dependent. This function returns the value returned by the `system()` call. If a `command` is not specified, a runtime error occurs (you cannot pass null strings).

Note:

Using the `os_command()` system function may degrade performance because the `system()` call uses OS forks.

Bit-type Procedures

The NTB-OV bit-type procedures include the following.

vera_bit_reverse ()

The syntax for the task is:

```
task vera_bit_reverse(var reg[M-1:0] dst_bit_vector, reg[N-1:0]
src_bit_vector);
```

The syntax for the function is:

```
function reg[M-1:0] vera_bit_reverse(reg[N-1:0] src_bit_vector);
```

`dst_bit_vector`

is the destination of the bits copied from `src_bit_vector`.

`src_bit_vector`

NTB-OV reverses the bits inside `src_bit_vector` and copies them into *`dst_bit_vector`*. The `src_bit_vector` can be an integer (enumerated), reg, or bit vector.

Note that the width of destination and source bit-vectors (`M` and `N`) do not need to be the same. If `M` is greater than `N`, NTB-OV fills the upper bits of the reversed vector with zeros. If `M` is less than `N`, NTB-OV truncates the upper bits of the reversed vector. This way, the result is always equal to size `M`.

Random Number Generation

The NTB-OV bit-type random number generation functions include the following. For more information, see “Random Number Generation” on page 384.

random ()

The `random()` function returns a 31-bit pseudorandom positive integer value.

urandom ()

The `urandom()` function returns an unsigned 32-bit random number.

urandom_range ()

The `urandom_range()` function returns an unsigned value in the range *`maxval`*..*`minval`*.

rand48 ()

The `rand48()` function generates an random number (integer) based on the lrand48 algorithm.

urand48 ()

The `urand48()` function generates an unsigned 32-bit random number based on the mrand48 algorithm.

Seeding for Randomization

The NTB-OV bit-type random number generation functions include the following. For more information, see “Seeding for Randomization” on page 380.

The `srandom()` task initializes the current RNG array using the value of the seed.

initstate ()

The `initstate()` task initializes a user-defined state array pointed to by the `state` argument.

setstate ()

The `setstate()` task attaches the state array to the current thread or specified object. The state array changes whenever you call `random()` or `urandom()` on the current randomizer. You can use the optional `object` argument to seed an object other than the current thread (context).

getstate ()

The `getstate()` call returns a `copy` of the state values for the current thread or specified object.

Associative Array Manipulation

The NTB-OV associative array manipulation functions include the following.

assoc_index ()

You can use the `assoc_index()` system function to manipulate or analyze associative arrays. The syntax is:

```
function integer assoc_index (CHECK | DELETE | FIRST | NEXT,
assoc_array_name [, var reg[63:0] index]);
```

Note:

Use CHECK, DELETE, FIRST, or NEXT to specify the action of to take with this command (see Table 8-14). Note that the maximum index size is 64’hffffffff_ffffffe, or 2^64-2.

Table 8-14 assoc_index () Predefined Macros

Option	Description
CHECK	Checks if an element exists at the specified index within the array. If it does, this function returns a 1; else it returns a 0. If the index is omitted, the function returns the number of allocated elements in the array.
DELETE	Deletes the element at the specific index. If it is successful, this function returns a 1; else it returns a 0. If the index is omitted, NTB-OV deletes all elements in the array. Only the array elements are deleted, and not the array itself.
FIRST	Returns the element associated with the first valid index. The index is assigned the value of the first valid element in the array. This function returns a 0 if it fails and a 1 if an element is returned.
NEXT	Searches for the first valid array element with an index greater than the specified parameter index. If an element is found, this function returns 1 and assigns the new index to the parameter index. If none exists, the function leaves the value of index unchanged, and returns a 0.

`assoc_array_name`

is the name of the associative array being analyzed. It must be a valid array reference.

`index`

is the numerical or string index of the element being analyzed.

The function `assoc_index()` returns a 1 is returned if successful, or a 0 if unsuccessful. With this function, you don't need to assign the return value to a variable. Example 8-83 shows an example NTM program with `assoc_index()` calls.

Example 8-83 assoc_index () Calls

```
class A {
       reg [5:0] num;
}

program main {
      integer i, j, k;
      integer assoc_arr[];
      integer dup_assoc_arr[];
      A a;
      a = new() ;
// Fill an associative array using a random index.
      for(i=0; i<20; i++) {
            a.num = random();
            j = a.num;
            assoc_arr[j] = i;
            printf("Putting %0d at index %0d\n",i,j);
      }

// Find total number of elements in the associative array.
      k = assoc_index(CHECK, assoc_arr);
      printf("Total number of elements = %0d\n", k);

// Find whether element exists at the index 55 (passed
// by variable i).
      i = 55;
      k = assoc_index(CHECK, assoc_arr, i);
      if(k)
            printf("Element exists at index location %0d and its \
            value is = %0d\n", i, assoc_arr[i]);
      else
            printf("No element exits at %0d\n", i);

// Find the first element in the associative array.
      void = assoc_index(FIRST, assoc_arr, j);
      printf("First element at index %0d\n",j);
```

```
// Print all the indexes in the associative array
// starting from the first.
      while(assoc_index(NEXT, assoc_arr, j)){
            printf("Next element at index %0d\n", j);
      }

// Copy one associative array to another.
      void = assoc_index(FIRST, assoc_arr, i);
      for(j=0; j<assoc_index(CHECK,assoc_arr); j++) {
            dup_assoc_arr[i] = assoc_arr[i];
            assoc_index(NEXT, assoc_arr,i);
      }

// Find the first element and delete it
      void = assoc_index(FIRST, assoc_arr, i);
      printf("There are %0d elements in assoc_arr\n",
            assoc_index(CHECK,assoc_arr));
      assoc_index(DELETE, assoc_arr, i);

// Delete the remaining elements
      while(assoc_index(NEXT, assoc_arr, i)){
            printf("There are %0d elements in assoc_arr\n",
                  assoc_index(CHECK,assoc_arr));
            assoc_index(DELETE, assoc_arr, i);
      }
      printf("After DELETE assoc_arr has %0d elements\n",
            assoc_index(CHECK,assoc_arr));

// Deleting an associative array
      printf("dup_assoc_index has %0d elements\n",
            assoc_index(CHECK, dup_assoc_arr));
      assoc_index(DELETE, dup_assoc_arr);
      printf("After DELETE dup_assoc_index has %0d elements\n",
      assoc_index(CHECK, dup_assoc_arr));
} // end program
```

Pack and Unpack

You use the `vera_pack()` and `vera_unpack()` system functions to create a data stream from global variables and variables in multiple objects. These system functions can pack/unpack a data stream into fixed-size arrays, dynamic arrays, associative arrays, or bit vectors. By default, NTB-OV packs/unpacks the data stream in the little-endian format. NTB-OV provides two additional system functions, `vera_pack_big_endian()` and `vera_unpack_big_endian()`, for packing and unpacking in big-endian format.

- In little-endian, the least significant segment, or piece equal in size to the width of the data stream (or storage), is written first. If the segment does not fill the entire word in the storage it is aligned towards the LSB or right side of the word.
- In big-endian, the most significant segment, or piece equal in size to the width of the data stream (or storage), is written first. If the segment does not fill the entire word in the storage it is aligned towards the MSB or left side of the word.

vera_pack ()

The `vera_pack()` system function returns the number of bits packed. NTB-OV supports packing from multidimensional fixed arrays, dynamic arrays, and associative arrays. If a `vera_pack()` system call fails, NTB-OV generates an error and stops the simulation. The syntax is:

```
function integer vera_pack(reg[M:0] storage, var integer bit_offset,
argument_list);
```

`storage`

specifies the variable into which the data is to be packed. Can be a single-bit vector, a fixed array of bit vectors, a dynamic array of bit vectors, or an associative array of bit vectors (if the packing size is unknown). It cannot be an integer or an array of integers.

`bit_offset`

is the total number of bits packed in *storage* for the current call. NTB-OV updates the *bit_offset* value during the call.

`argument_list`

is a list of variables to be packed of type reg, integer, enum, or string.

Packing from fixed and dynamic arrays starts at the 0 index (independent of big/little-endian). Packing from associative arrays starts at the entry with the smallest index (independent of big/little-endian). Only entries that exist in the associative array are packed. If the associative array has no entries, nothing is packed.

Example 8-84 shows an NTB-OV program with some `vera_pack()` calls.

Example 8-84 vera_pack () Calls

```
program Test {
      reg[7:0] Storage[];
      reg[31:0] MyVector;
      reg MyBit1;
      reg MyBit2;
      integer PackedSize;
      integer BitOffset = 0;
      integer i;
      MyVector =
            32´b_11101100_00011001_11011100_10101001;
      MyBit1 = 1;
      MyBit2 = 0;

      PackedSize = vera_pack(Storage, BitOffset, MyVector);

      printf("Current number of bits packed = %0d\n",PackedSize);
      printf("Total number of bits packed = %0d\n\n",BitOffset);

      for (i = 0; i < 6; i++){
            printf("Storage[%0d] = %b;\n", i, Storage[i]);
      }

      PackedSize = vera_pack(Storage, BitOffset, MyBit1, MyBit2);
```

```
        printf("\nCurrent number of bits packed = %0d\n",PackedSize);

        printf("Total number of bits packed = %0d\n\n",BitOffset);

        for (i = 0; i < 6; i++){
             printf("Storage[%0d] = %b;\n", i, Storage[i]);
        }
   }
```

Example 8-84 generates the following output:

```
Current number of bits packed = 32
Total number of bits packed = 32

Storage[0] = 10101001;
Storage[1] = 11011100;
Storage[2] = 00011001;
Storage[3] = 11101100;
Storage[4] = xxxxxxxx;
Storage[5] = xxxxxxxx;

Current number of bits packed = 2
Total number of bits packed = 34

Storage[0] = 10101001;
Storage[1] = 11011100;
Storage[2] = 00011001;
Storage[3] = 11101100;
Storage[4] = xxxxxx01;
Storage[5] = xxxxxxxx;
```

vera_unpack ()

The `vera_unpack()` system function returns the number of bits unpacked. NTB-OV supports unpacking into multidimensional fixed arrays, dynamic arrays, or associative arrays. If a `vera_unpack()` system call fails, NTB-OV generates an error and stops the simulation. The syntax is:

```
function integer vera_unpack(reg[M:0] storage, var
     integer bit_offset, var argument_list);
```

`storage`

specifies the variable from which data is unpacked. It can be a single-bit vector, a fixed array of bit vectors, a dynamic array of bit vectors, or an associative array of bit vectors (if the packing size is unknown). It cannot be an integer or an array of integers.

`bit_offset`

is the total number of bits unpacked from *`storage`* for the current call. NTB-OV updates the *`bit_offset`* value during the call.

`argument_list`

The data unpacked from *`storage`* is assigned to the variables in the *`argument_list`*. NTB-OV supports variables of type reg, integer, enum, arrays, and string. When an argument in the *`argument_list`* is an associative array an *`additional`* argument following each associative array in the list is required. This argument tells `vera_unpack()` how many indices to fill.

Example 8-85 shows an NTB-OV program with some `vera_unpack()` calls.

Example 8-85 vera_unpack () Calls

```
n_unpacked = vera_unpack(stream, n_offset, assoc_array,
n_entries);
```

-or-

```
n_unpacked = vera_unpack(stream, n_offset, assoc_array1,
n_entries1, assoc_array2, n_entries2);
```

In Example 8-85, NTB-OV attempts to fill all entries starting with index 0 (independent of big/little-endian). Additional entries are indexed by adding 1. If there are entries in the associative array beyond the `assoc_size`, NTB-OV removes them without generating a warning message. If there aren't enough bits in the stream to fill all the entries, `vera_unpack()` issues a warning message and sets the remaining entries to 0.

For unpacking into fixed and dynamic arrays, `vera_unpack()` attempts to fill the entire array starting with the 0 index (independent of big/little-endian). If there are not enough bits in the stream to fill the entire array, `vera_unpack()` issues an error and stops the simulation.

vera_pack_big_endian ()

The `vera_pack_big_endian()` system function returns the number of bits packed. If a `vera_pack_big_endian()` system call fails, NTB-OV generates an error and stops the simulation. The syntax is:

```
function integer vera_pack_big_endian(reg[M:0] storage, var integer
bit_offset, argument_list);
```

`storage`

specifies the variable into which the data is to be packed. *`storage`* can be a single-bit vector, a fixed array of bit vectors, a dynamic array of bit vectors, or an associative array of bit vectors (if the packing size is unknown). It cannot be an integer or an array of integers.

`bit_offset`

is the total number of bits packed into *`storage`* for the current call. NTB-OV updates the *`bit_offset`* value after the call.

`argument_list`

is a list of variables of type reg, integer, enum, or string to be packed in big-endian format.

Example 8-86 shows an NTB-OV program that contains some `vera_pack_big_endian` calls.

Example 8-86   vera_pack_big_endian () Calls

```
program Test {
      reg[7:0] Storage[];
      reg[31:0] MyVector;
      reg MyBit1;
      reg MyBit2;
      integer PackedSize;
      integer BitOffset = 0;
      integer i;
      MyVector =
            32´b_11101100_00011001_11011100_10101001;
      MyBit1 = 1;
      MyBit2 = 0;

      PackedSize = vera_pack_big_endian(Storage, BitOffset,
            MyVector);
      printf("Current number of bits packed = %0d\n",PackedSize);
      printf("Total number of bits packed = %0d\n\n",BitOffset);

      for (i = 0; i < 6; i++){
            printf("Storage[%0d] = %b;\n", i,Storage[i]);
      }

      PackedSize = vera_pack_big_endian(Storage, BitOffset,
            MyBit1, MyBit2);

      printf("\nCurrent number of bits packed =
            %0d\n",PackedSize);
      printf("Total number of bits packed = %0d\n\n",BitOffset);

      for (i = 0; i < 6; i++){
            printf("Storage[%0d] = %b;\n", i,Storage[i]);
      }
}
```

Example 8-86 generates the following output:

```
Current number of bits packed = 32
Total number of bits packed = 32
Storage[0] = 11101100;
Storage[1] = 00011001;
Storage[2] = 11011100;
Storage[3] = 10101001;
Storage[4] = xxxxxxxx;
Storage[5] = xxxxxxxx;
```

```
Current number of bits packed = 2
Total number of bits packed = 34
Storage[0] = 11101100;
Storage[1] = 00011001;
Storage[2] = 11011100;
Storage[3] = 10101001;
Storage[4] = 10xxxxxx;
Storage[5] = xxxxxxxx;
```

vera_unpack_big_endian ()

The `vera_unpack_big_endian()` system function returns the number of bits unpacked. If the system call fails, NTB-OV generates an error and stops the simulation. The syntax is:

`function integer vera_unpack_big_endian``(`**`reg`**`[M:0]` *`storage`*`,` **`var integer`** *`bit_offset`*`,` **`var`** *`argument_list`*`);`

`storage`

specifies the variable from which data is unpacked in big-endian format. It can be a single-bit vector, a fixed array of bit vectors, a dynamic array of bit vectors, or an associative array of bit vectors (if the packing size is unknown). It cannot be an integer or an array of integers.

`bit_offset`

is the total number of bits packed into *`storage`* for the current `vera_unpack_big_endian()` call. NTB-OV updates the *`bit_offset`* value after the call.

`argument_list`

NTB-OV assigns the data unpacked from *`storage`* to the variables in the *`argument_list`*. NTB-OV supports variables of type reg, integer, enum, and string.

Simulation Errors

NTB-OV provides the following tasks for simulation errors.

error ()

The `error()` system task generates a testbench simulation error. The syntax is:

```
task error(string format argument_list,...);
```

`format`

is a C-style format string. The valid format specifiers are listed in Table 8-11.

`argument_list`

consists of the arguments to be printed.

The `error()` system task prints the message specified with *`format`* and generates the error. Use this function to intentionally generate a simulation error condition.

Example 8-87 shows an `error()` call.

Example 8-87 error () Call

```
if( length > 256 )
      error("Too long length %d specified\n",
      length);
```

flag ()

The `flag()` system function sets and clears error flags. The syntax is:

```
function integer flag([ON | OFF]);
```

`ON`

If the argument is `ON`, NTB-OV sets the error flag.

`OFF`

If the argument is `OFF`, NTB-OV clears the error flag.

Use the `flag()` function to set and clear error flags raised when non-fatal simulation errors occur (for example, soft-expects). This function returns the value of the flag before setting or clearing the flag. If you don't specify an argument in the call, the function just returns the state of the error flag.

If the error flag is set in a child process (in a fork/join block), NTB-OV transfers the error flag to its parent only if the parent is waiting for the child. Returning from a task or procedure with the flag raised triggers a simulation error (default setup), and NTB-OV clears the flag. Example 8-88 shows an example `flag()` call.

Example 8-88 flag () Call

```
foo_bus.data =?= 8'h54 soft;  // soft expect
      if( flag () )   {
            printf("Warning: data is not 8'h54\n" );
            flag( OFF );
            ...
      }
```

Sub-Cycle Delays

NTB-OV provides the following sub-cycle delay function.

delay ()

The syntax is:

```
task delay(integer time);
```

`time`

specifies the length of the delay. NTB-OV determines the time unit of the delay using the timescale of the Verilog design.

Example 8-89 shows an example `delay()` call.

Example 8-89 delay () Call

```
@(posedge CLOCK);
delay(5);
task1();
...
```

Example 8-89 synchronizes to the positive edge of `CLOCK` and then waits for the simulation time to advance 5 time units. NTB-OV executes `task1` 5 time units after the `CLOCK` edge.

Plusargs String Matching

NTB-OV provides the following plusargs string matching functions.

test_plusargs ()

The `test_plusargs()` system function searches the list of plusargs for a user-specified plusarg string. The syntax is:

```
function integer test_plusargs(string string_name);
```

`string_name`

is a string variable.

You can specify the `string_name` to `test_plusargs()` as either a string or a reg (which NTB-OV interprets as a string). If NTB-OV finds the string, it converts the remainder of the string to the type specified in the *string_name* and stores the resulting value the variable provided. When this function finds a string it returns a non-zero integer. If no matching string is found, the function returns a 0 and the variable provided is not altered.

value_plusargs ()

The `value_plusargs()` system function searches the list of plusargs for the specified string. The syntax is:

```
function integer value_plusargs(string string_name, data_type
variable_name);
```

`string_name`

is a quoted string that contains a format specifier.

`data_type`

depends on the format specifier in *string_name*. The supported specifiers are `%d`, `%s,` `%h`, `%o`, and `%b`.

`variable_name`

The data type of *variable_name* depends on the format specifier (for example, reg, integer, or string).

NTB-OV searches the specified plusargs in the order provided. If the prefix of one of the supplied plusargs matches all characters in the provided string, NTB-OV returns a non-zero integer. If none of the specified plusargs matches the string provided, NTB-OV returns the integer value 0. Example 8-90 shows an example `value_plusargs()` call.

Example 8-90   value_plusargs () Call

```
program test {
     string s;
```

```
        if (value_plusargs("my_str=%s", s))
        {
             printf("got my_str with value = %s\n", s);
        }
   }
```

Concurrency Control

NTB-OV provides the following tasks for concurrency control:

trigger ()

Use the `trigger()` task to change the state of an event. Triggering an event unblocks waiting syncs, or blocks subsequent syncs. All events are `OFF` by default. The syntax is:

```
task  trigger([ONE_SHOT | ONE_BLAST| HAND_SHAKE | ON |
     OFF,] event event_name1 , ... ,event event_nameN);
```

`event_name`

is the event variable name on which the sync is activated.

`ONE_SHOT`

is the default trigger type; any process waiting for a trigger receives it. If there are no processes waiting for the trigger, NTB-OV discards the trigger.

Note:

You must call the sync before the trigger is executed when using `ONE_SHOT` triggers. If the sync is called after the trigger is executed, the process waits indefinitely.

`ONE_BLAST`

triggers work just as `ONE_SHOT` triggers do except that they trigger any sync called within the simulation time, regardless of whether it was called before the trigger was executed.

`HAND_SHAKE`

triggers unblock only one sync, even if multiple syncs are waiting for triggers. This setting causes the function to trigger the most recent pending sync, or queues requests. If the order of triggering the unblocking of a sync is important, use `semaphore_get()` and `semaphore_put()` around the sync to maintain order.

If a sync was already called and is waiting for a trigger, the `HAND_SHAKE` trigger unblocks the sync.

If no sync was called when the trigger occurs, the `HAND_SHAKE` trigger is stored. When a sync is called, the sync is immediately unblocked and the trigger is removed.

`ON`

Use the `ON` trigger to turn on an event. When an event is turned on, all syncs waiting for that event immediately trigger. Also, all future sync operations on that event immediately trigger until there is a trigger (`OFF`) call.

`OFF`

Use the `OFF` trigger to turn off an event. You cannot use an event to synchronize concurrent processes if it is turned off.

sync ()

Use `sync()` to synchronize statement execution to one or more triggers. You can use `sync()` as either a task or a function. For detailed syntax and an example, see "sync () Task or Function" on page 130.

Controlling fork/join Blocks

NTB-OV provides the following task for controlling fork/join blocks.

wait_child ()

Use the `wait_child()` system task to halt execution of the current process until all descendant processes are executed. For detailed syntax and an example, see "wait_child ()" on page 125.

Type Casting

NTB-OV provides the following function for type casting.

cast_assign ()

Use the `cast_assign()` system function to assign values to variables that might not ordinarily be valid because of differing data types. For detailed syntax and an example, see "Type Casting" on page 116.

Semaphores

NTB-OV provides the following system functions for semaphores.

alloc ()

Use the `alloc()` system function enables to allocate a semaphore. For detailed syntax and an example, see “Allocating Semaphores” on page 138.

semaphore_get ()

Use `semaphore_get()` to obtain keys from a semaphore. You can use `semaphore_get()` as either a task or a function. For detailed syntax and an example, see “Obtaining Semaphore Keys” on page 138.

semaphore_put ()

Use the `semaphore_put()` system task to return keys to a semaphore. For detailed syntax and an example, see “Returning Semaphore Keys” on page 140.

Mailboxes

NTB-OV provides the following system functions for mailboxes.

mailbox_put ()

Use the `mailbox_put()` system task to send data to a mailbox. For detailed syntax and an example, see “Sending Data to a Mailbox” on page 143.

mailbox_get ()

Use `mailbox_get()` to retrieve data from a mailbox. The mailbox waiting queue is similar to the semaphore waiting queue as far as relative ordering of requests is concerned. You can use `mailbox_get()` as either a task or a function. For detailed syntax and an example, see “Retrieving Data from a Mailbox” on page 144.

Connecting Signals

NTB-OV provides the following system function for connecting signals.

signal_connect ()

Use the `signal_connect()` system function to connect interface signals to virtual port signal members at runtime. Example 8-91 shows an example program that contains `signal_connect()` calls.

Example 8-91 signal_connect () Call

```
interface intf_a {
     input reg regA PSAMPLE #-1;
}
port myport {
     port_regA;
}
program test {
     myport p1;
     reg sampleA;
     p1 = new;
     signal_connect(p1.$port_regA, intf_a.regA);
// port connected to interface
     sampleA = p1.$port_regA;
// This reads the interface value
}
```

Cyclic Redundancy Check (CRC) Function

NTB-OV provides an efficient and flexible feature for the calculation of a cyclic redundancy check (CRC) value. The default algorithm used in the `vera_crc()` system function operates on 8-bit segments of an input data stream. If the input data stream is not 8-bit aligned, NTB-OV packs the empty bits with zeros. CRC calculation starts with the LSB of each segment. The `vera_crc()` system function has a set of predefined CRC algorithms. Each algorithm uses a default polynomial for the CRC calculation (see Table 8-15). You can override the default polynomial with your own polynomial.

Table 8-15 CRC Algorithm Polynomials

CRC Algorithms	Default Polynomial
CRC-8	0x07
CRC-16	0x8005
CRC-32	0x04C11DB7
CRC-64	0xE543279765927881

vera_crc ()

The syntax for the `vera_crc()` system function using a bit vector for the input data stream is:

```
function reg[63:0] vera_crc(integer N, reg[M-1:0] stream,
reg[63:0] index1, reg[63:0] index2 [, reg[N-1:0]init_crc ]);
```

The syntax for the `vera_crc()` system function using an array of bit vectors for the input data stream is:

```
function reg[63:0] vera_crc(integer N, reg[N-1:0] stream_array,
reg[63:0] index1, reg[63:0] index2 [, reg[N-1:0]init_crc ]);
```

The `vera_crc()` system function returns a 64-bit CRC value. All but the *N* least significant bits can be ignored.

`N`

is the order of the CRC algorithm. Valid values are 8, 16, 32, and 64.

`stream`

is the bit vector used for the CRC calculation. There is no limit to the number of bits passed to the CRC algorithm via *`stream`*. NTB-OV generates a fatal error or any X or Z bit values detected by the CRC algorithm.

`stream_array`

is an array of bit vectors used for the CRC calculation. The valid array types are: single-dimensional array, multidimensional array, dynamic array, and associative array. There is no limit to the number of bits passed to the CRC algorithm via *`stream_array`*. NTB-OV generates a fatal error for any X or Z bit values detected by the CRC algorithm.

`index1, index2`

specify the range of a bit vector defined in *`stream`* or *`stream_array`* used for the CRC calculation.

`init_crc`

specifies an initial value for the CRC calculation (that is, before *`stream`* or *`stream_array`* is processed).

`index1` and *`index2`* specify the range of a the bit vector defined in *`stream`* or *`stream_array`* used for the CRC calculation. The CRC calculation starts with *`index2`* and ends with *`index1`*. Therefore, if *`index2`* is greater than *`index1`*, the bit vector used for the CRC

calculation is effectively reversed. For example, given `stream_array[N,M]`, if *index2* < *index1*, the CRC calculation starts with *stream_array*`[`*index2*`,0]` and ends with *stream_array*`[`*index1*`,M-1]`. If *`index2`* > *`index1`*, the CRC calculation starts with *`stream_array`*`[`*`index2`*`,M-1]` and ends with *stream_array*`[`*index1*`,0]`.

If *`stream_array`* is a single-dimensional fixed array, a dynamic array, or an associative array, *`index1`* and *`index2`* apply to the corresponding array entries. The CRC calculation starts with *`stream_array`*`[`*`index2`*`]` and ends with *stream_array*`[`*index1*`]`.

If *`stream_array`* is a multidimensional array, *`index1`* and *`index2`* apply to the left-most subscript (the subscript that varies the slowest).

If *`index1`* and *`index2`* have the same value, then only one bit of *`stream`* or one entry of a single-dimensional *`stream_array`* is used for the CRC calculation. If the value of either *`index1`* or *`index2`* is greater than the MSB value of *`stream`*, the value defaults to the MSB value. If the value of either *`index1`* or *`index2`* is greater than the array size of *`stream_array`*, the value defaults to the array size. Example 8-92 shows an example that contains `vera_crc()` calls.

Example 8-92 vera_crc () Call

```
reg[7:0]array[10];
integer i = 5;
reg [31:0] bit32a, bit32b, bit32c;
bit32a = vera_crc(32,array,64,i);
bit32b = vera_crc(32,array,i,32);
bit32c = vera_crc(32,array,i,i);
```

In Example 8-92, in the first call to `vera_crc()`, *`index1`* is set to 64, which is greater than the array size of `array[]`. Therefore, NTB-OV calculates the CRC value for `bit32a` using the array elements `array[5]` to `array[9]`.

In the second call to `vera_crc()`, *`index1`* is set to 32, which is greater than the array size of `array[]`. Therefore, NTB-OV calculates the CRC value for `bit32b` using the array elements `array[9]` to `array[5]`.

In the third call to `vera_crc()`, *`index1`* and *`index2`* have the same value. Therefore, NTB-OV calculates the CRC value for `bit32c` using just the array element `array[5]`.

The `vera_crc()` system function provides an optional parameter that you can use to specify a polynomial to use instead of the default polynomial for the CRC calculation. The syntax for using a bit vector for the input data stream when specifying a polynomial is:

```
function reg[63:0] vera_crc(integer N, reg[high:0] stream,
integer index1, integer index2 , reg[N-1:0]init_crc , reg[N-1:0]
polynomial [, reg[N-1:0] xor_out [, integer reflect_out
[,integer reflect_in]]]);
```

The syntax for using an array of bit vectors for the input data stream when specifying a polynomial is:

```
function reg[63:0] vera_crc(integer N, reg[high:0] stream_array[],
integer index1, integer index2 , reg[N-1:0]init_crc , reg[N-1:0]
polynomial [, reg[N-1:0] xor_out [, integer reflect_out
[, integer reflect_in]]]);
```

`N`

is the order of the CRC algorithm. The valid values are 8, 16, 32, and 64.

`stream`

is the bit vector used for the CRC calculation. There is no limit to the number of bits passed to the CRC algorithm via *stream*. NTB-OV generates a fatal error for any X or Z bit values detected by the CRC algorithm.

`stream_array`

is an array of bit vectors used for the CRC calculation. The valid array types are: single-dimensional array, multidimensional array, dynamic array and associative array. There is no limit to the number of bits passed to the CRC algorithm via *stream_array*. NTB-OV generates a fatal error message for any X or Z bit values detected by the CRC algorithm.

`index1, index2`

specify the range of a bit vector defined in the *stream* or *stream_array* used for the CRC calculation.

`init_crc`

specify an initial value for the CRC calculation (that is, before *stream* or *stream_array* is processed). The MSB value of *init_crc* is *N* -1.

`polynomial`

use this parameter to override the default polynomial used for the CRC calculation. If you use *polynomial*, you must also specify the *init_crc* parameter (which is optional in other cases).

`xor_out`

If you specify *xor_out*, the function call returns the bit-wise XOR'ed value of the CRC calculation result and *xor_out*. If you use this parameter, you must also specify the *init_crc* and *polynomial* parameters.

`reflect_out`

If you specify *reflect_out*, the function returns the CRC calculation result in reversed bit order. If you use this parameter, you must also specify the *init_crc* and *polynomialxor_out* parameters.

`reflect_in`

If you specify *reflect_in*, NTB-OV reverses the bit order of each byte in a *stream_array* element or *stream* before using it in the CRC calculation. If you use this parameter, you must also specify the *init_crc*, *polynomial*, *xor_out,* and *reflect_out* parameters.

Debug

NTB-OV provides the following statement for debugging.

breakpoint

Use the breakpoint statement to stop the simulation and bring up the command-line debugger. The syntax is:

```
breakpoint;
```

Passing Data at Runtime

NTB-OV provides the following function for passing data at runtime.

get_plus_arg ()

Use the `get_plus_arg()` system function to read HDL plus arguments. NTB-OV includes the VHDL equivalent to plus arguments in the .ini file. The syntax is:

```
function reg get_plus_arg(CHECK | HNUM | NUM, string
plus_arg);
```

Table 8-16 shows the valid values for the plus argument.

Table 8-16 get_plus_arg () Macros

Request	Action
CHECK	Returns 1 if the specified plus argument is present
HNUM	Returns a hexadecimal number attached to the specified plus argument
NUM	Returns an integer attached to the specified plus argument

`plus_arg`

is the plus argument you want to evaluate.

The `get_plus_arg()` system function returns a value based on the request type and `plus_arg` value. This function can capture only 32-bits of a number from the command line. Example 8-93 shows an example that includes `get_plus_arg()` calls.

Example 8-93 get_plus_arg () Call

```
#define DEF_RP_TIMES  10
#define DEF_RSEED  32'habcd_ef01
program test {
      integer repeat_times = DEF_RP_TIMES;
      reg [31:0] random_seed = DEF_RSEED;

// get repeat times if any
           if ( get_plus_arg ( CHECK, "set_repeat_times=" ) ) {
```

```
                    repeat_times = get_plus_arg ( NUM,
                         "set_repeat_times=");
               }

// get random seed if any
     if ( get_plus_arg ( CHECK, "set_random_seed=" ) {
          random_seed = get_plus_arg (HNUM, "set_random_seed=" );
     }

     printf (" repeat times is %0d\n", repeat_times ;
     printf (" random seed is %h\n", random_seed );
}
```

To compile and execute Example 8-93, use the following commands:

```
% vcs -ntb file_name.vr
% simv +set_repeat_times=7 +set_random_seed=3
```

System Interaction

NTB-OV provides the following function for system interaction.

get_systime ()

The `get_systime()` system function returns the number of seconds since 00:00: UTC, January 1, 1970. The syntax is:

```
function reg[31:0] get_systime();
```

This function corresponds to the UNIX `time()` library function. For more information, see the UNIX man page. Example 8-94 shows an example `get_systime()` call.

Example 8-94 get_systime () Call

```
     program adder_test{ // start of top block
          integer time1;
          reg[31:0] time2;

          time1 = get_systime();
```

```
    get_systime();

    repeat (200000) // wait for some time
    @(posedge CLOCK);

    void = get_systime();
    time2 = get_systime();

    printf("From NTB-OV: time2 = %d\n", time2);
}  // end of program adder_test
// define tasks/classes/functions here if necessary
```

9

Randomization in NTB-OV

This chapter explains how randomization works in OpenVera Native Testbench (NTB-OV) in the following major sections:

Random Stability

In NTB-OV, Random Number Generation (RNG) is localized to threads and objects. Because the stream of random values returned by a thread or object is independent of the RNG in other threads or objects, this feature is called random stability. Random stability applies to:

- the system randomization calls: `random()` (see “random ()” on page 382), `urandom()` (see “urandom ()” on page 383), and `srandom()` (see “srandom ()” on page 378)
- the `randomize()` object randomization method
- `randcase` (see “randcase Statements” on page 381)

With random stability, your testbench programs show more stable RNG behavior when you make small changes to your code. And you can choose to exercise precise control over the generation of random values by manually seeding threads and objects. Random stability encompasses the following properties:

- Program and Thread Stability
- Object Stability
- Manual Seeding

Program and Thread Stability

Each thread has an independent RNG source for all randomization system calls invoked from that thread. When a new thread is created, NTB-OV seeds its RNG with the next random value from its parent thread. This property is called hierarchical seeding. Program and thread repeatability is guaranteed as long as you create threads and generate random numbers in the same order as before.

NTB-OV returns random values from system calls (for example, `random()`, `randcase`, and `urandom()`) independent of thread execution order. Example 9-1 illustrates how thread locality and hierarchical seeding work in NTB-OV.

Example 9-1 NTB-OV Thread Locality and Hierarchical Seeding

```
integer x, y, z;
fork //set a seed at the start of a thread
     {srandom(100); x = random();} //of srandom()
     //set a seed during a thread
     {y = random(); srandom(200);}
     //randcase draws a value from the thread RNG
     {z = random(); randcase(){ … }}
join all
```

In Example 9-1 the values returned for `x`, `y`, and `z` are independent of the thread execution order. This thread locality is important because it allows you to develop subsystems that are independent, controllable, and predictable.

When you create a thread, NTB-OV initializes its random state using the next random value from the parent thread as a seed. The three forked threads in Example 9-1 are all seeded from the parent thread. NTB-OV seeds each thread with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This hierarchical seeding allows you to move subtrees, while preserving their behavior by manually seeding their root thread.

Object Stability

Each class instance (object) has an independent RNG source for the randomization method in the class. When you create an object using `new()`, NTB-OV seed its RNG with the next random value from the thread that created the object. Object repeatability/stability is guaranteed as long as you create objects and threads and generate random numbers in the same order as before. New objects, threads, and random numbers should be created after existing objects are created.

The `randomize()` method built into every NTB-OV class exhibits object stability, whereby calls to `randomize()` in one instance are independent of calls to `randomize()` in other instances, and independent of calls to the system randomize functions. Example 9-2 illustrates how this works.

Example 9-2 Object Stability

```
class Foo { rand integer x; }
class Bar { rand integer y; }

program main {
      Foo foo = new();
      Bar bar = new();
      integer z;

      void = foo.randomize();
      // z = random();
      void = bar.randomize();
}
```

In Example 9-2, the values returned for `foo.x` and `bar.y` are independent of each other, and the calls to `randomize()` are independent of the random system calls. If you uncomment the line `z = random()` in this example, there is no change in the values assigned to `foo` and `bar`.

Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when you create the instance. You can seed instances at any time using the `srandom()` (see “srandom ()” on page 378), `initstate()` (see “initstate ()” on page 378), and `setstate()` (see See "setstate ()" on page 381) system calls with an optional object argument, as shown in Example 9-3.

Example 9-3   Instance with Unique Random Values

```
class Foo {
      task new (integer seed) {
      //set a new seed for this instance
      srandom(seed, this);
      }
}
```

Note that once an object is created there is no guarantee that the creating thread can change the object’s random state before another thread accesses the object. Therefore, it is good practice to let objects self-seed within their `new()` methods rather than externally. Unlike threads, an object’s seed can be set from any thread.

Manual Seeding

You can manually seed all RNG sources. Combined with hierarchical seeding, this allows you to define the operation of a subsystem (hierarchy subtree) completely with a single seed at the root thread of the system.

You can also seed threads using `srandom()`, `initstate()`, and `setstate()`. There is no way for a thread to set the random state of another thread. The `srandom()` system call in Example 9-3 initializes the state for the thread’s RNG state using the seed.

Reseeding and Random Dynamic Arrays

Stability applies to all NTB-OV language constructs and functions that generate random values. A thread may set a new seed at any time during execution.

You can randomize dynamic arrays, but you must specify the size of the array at runtime, as shown in the following example:

```
rand type array_name [*] dynamic_size num_elements;
```

`num_elements`

specifies the number of elements in the array to be randomized. Note that the array size itself can be a random number.

Example 9-4 uses `randomize()` to create an array of variable size.

Example 9-4 randomize () Call

```
#define FACTOR 2
#define ORDINARY_SIZE 3
#define MAX_SMALL_ARRAY_SIZE 7

class SmallClass  { rand reg[2:0] small_var; }
class ArrayClass {
// All these variables are random.
      rand {
            integer m, n;
            integer ordinary [ORDINARY_SIZE]; // ordinary array
            reg [7:0] b_array [*] dynamic_size FACTOR * m;
            // dynamic array
            SmallClass small_array [*] dynamic_size n;
            // array of objects
      }
      /* Constraint blocks are class members, just as the properties
      (variables)and methods (tasks and functions) in a class are class
      members. Constraints must follow the variables, but come before
      the tasks and functions, of a class.*/

      constraint size_cons {
            m >= 0; m <= 3;
            n >= 1; n < MAX_SMALL_ARRAY_SIZE;
```

```
    }
    task new() {
        integer i;
        small_array[i] = new[MAX_SMALL_ARRAY_SIZE];

        for (i = 0; i < small_array.size(); ++i)
        small_array[i] = new();
    }

    task print_array() {
        integer i;
        printf ("m is %0d, n is %0d\n", m, n);
        for (i = 0; i < ORDINARY_SIZE; ++i)
            printf ("ordinary[%0d] is %0d\n", i, ordinary[i]);

        for (i = 0; i < FACTOR * m; ++i)
            printf ("b_array[%0d] is %h\n", i, b_array[i]);

        for (i = 0; i < n; ++i)
            printf ("small_array[%0d].small_var is %b\n",i,
                small_array[i].small_var);
    }
}

task DoArray(){
    integer success;
    ArrayClass a;

    a = new();
    success = a.randomize();

    if (success != OK)
        printf("randomize failed in DoArray\n");
        printf ("In DoArray, ArrayClass vals after
            randomize() call:\n");
        a.print_array();
        printf("\n");
}
```

You can specify constraints on the individual elements of the dynamic array, but a constraint with an out-of-bounds index results in a fatal error, even when the constraint is switched off.

Seeding for Randomization

NTB-OV provides the following tasks for seeding for randomization.

srandom ()

The `srandom()` task initializes the current RNG array using the value of the `seed`. The syntax is:

```
task srandom(integer seed,[object obj]);
```

Note that the `seed` value cannot be 0. You can use the optional `object` argument to seed an object other than the current thread (context). NTB-OV initializes the top-level randomizer state with `srandom(1)` prior to any randomization calls, including those that occur during initialization.

initstate ()

The `initstate()` task allows you to initialize a user-defined state array pointed to by the `state` argument.The syntax is:

```
task initstate(integer seed, var VeraRandomState state,[object obj]);
```

The `seed` value cannot be 0. This task initializes the state array using the `seed` and attaches the `state` array to the current randomizer using setstate () semantics. You can use the optional `object` argument to seed an object other than the current thread (context). You must declare the `state` array argument as type `VeraRandomState` (this macro is defined in <vera_defines.vrh>, which NTB-OV includes automatically).

setstate ()

The `setstate()` task attaches the `state` array to the current thread or specified object. The `state` array changes whenever `random()` or `urandom()` is called on the current randomizer. You can use the optional `object` argument to seed an object other than the current thread (context). The syntax is:

```
task setstate(var VeraRandomState state, [object obj]);
```

You must declare the `state` array argument as type `VeraRandomState` (this macro is defined in <vera_defines.vrh>, which NTB-OV includes automatically).

When you use this task, NTB-OV garbage collects any previously attached state. The system maintains a reference to the `state` array until `initstate()` or `setstate()` is called again, or until the thread or object is garbage collected.

getstate ()

This `getstate()` task returns a copy of the state values for the current thread or specified object.

```
task getstate(var VeraRandomState state,[object obj]);
```

You must declare the `state` array argument as type `VeraRandomState` (this macro is defined in <vera_defines.vrh>, which NTB-OV includes automatically).

Manually Seeding Randomize

In NTB-OV, each object maintains its own internal random number generator, which is used exclusively by its `randomize()` method. This allows you to randomize objects independent of each other and calls to the system random functions. When an object is created, its random number generator (RNG) is seeded using the next value from the RNG of the thread that created the object. This process is called hierarchical object seeding.

Sometimes it is desirable to manually seed an object's RNG using the `srandom()` system call. You can do this either in a class method or external to the class definition (see Example 9-5).

Example 9-5 srandom () External Call

```
class Packet {
      rand bit[15:0] header;
      ...
      task new (integer seed){
            srandom(seed, this);
            ...
       }
}
```

or externally:

```
Packet p = new(200); // Create p with seed 200.
srandom(300, p);    // Re-seed p with seed 300
```

You can also control object RNGs using the `initstate()` and `setstate()` system calls.

When used in the `new()` task, it is good practice to call `srandom()` first and initialize and all the class variables following the call. This ensures that the object's RNG is set with the new seed before any class member values are initialized to their default values or any contained objects are created.

randcase Statements

The `randcase` statement specifies a block of statements, one of which is executed randomly. The syntax is:

```
randcase {
      weight1 : statement1
      weight2 : statement2
      ...
      weightN : statementN
}
```

`weight`

can be any valid expression, including a constant. NTB-OV evaluates the *`weight`*expression every time `randcase` is executed. If *`weight`* is zero, that branch is not used.

`statement`

can be any valid statement or block of statements. If a code block is used, the entire block is executed.

When you use the `randcase` statement, NTB-OV randomly selects a *`statement`* from the `randcase` block. You can use different weights to change the probability that any given *`statement`* is selected. The probability that any single statement is selected is determined by weight/total_weight. Note that you can nest `randcase` statements. Example 9-6 shows an example `randcase` block.

Example 9-6 randcase Block

```
randcase {
      10: i=1;
      20: i=2;
      50: i=3;
}
```

Example 9-6 defines a `randcase` block with specified weights. There is a .125 probability that the first statement is executed, a .25 probability that the second statement is executed, and a .625 probability that the third statement is executed.

Random Number Generation

NTB-OV provides the following random number generation functions.

random ()

The `random()` function returns a 31-bit pseudorandom positive integer value from the current thread RNG. The syntax is:

```
function integer random([integer seed]);
```

`seed`

is an optional argument that determines which random number sequence is generated. The `seed` can be any valid expression, including variable expressions. The random number generator generates the same number sequence every time the same seed is used. A seed value of 0 is a special case in which the random number sequence is the same as the one generated with a seed value of 1.

The `random()` function takes an optional `seed` argument that is used to call `srandom()` before computing the random value. In this case the system behaves as if `srandom(seed)` had been called before `random()`.

The NTB-OV random number generator is deterministic. Each time the program executes, it cycles through the same random sequence. You can make this sequence non-deterministic by seeding the `random()` function with an extrinsic random variable, such as the time of day. Because `random()` always returns a positive number, the sign bit is 0. Only the 31 least significant bits are random.

Typically, to generate random numbers, you first call the `random()` system function with an integer `seed` value. Then call the `random()` function without a seed when you need a new random number (see Example 9-7).

Example 9-7   random () Call

```
random( 184984 ); // Initialize the generator
addr = {random(),random()};
ph_number = random() >> 5;
```

urandom ()

The `urandom()` function returns a 32-bit pseudorandom unsigned bit vector (reg) value from the current thread RNG. The syntax is:

```
function bit[31:0] urandom([integer seed]);
```

The `urandom()` function takes an optional `seed` argument that is used to call `srandom()` before computing the random value. Not that the seed value cannot be 0. In this case the system behaves as if `srandom(seed)` had been called before `urandom()`.

rand48 ()

The `rand48()` function generates a random integer based on the lrand48 algorithm. The syntax is:

```
function integer rand48([integer seed]);
```

Because `rand48()` always returns a positive number, the sign bit is 0. Only the 31 least significant bits are random. For more information on the `lrand48()` algorithm, see the UNIX man pages (at the UNIX shell prompt, type `man lrand48`).

Note:

> The `random()` function is preferred over `rand48()` because it yields a better distribution of random values. Also, `rand48()` does not adhere to random stability.

urand48 ()

The `urand48()` function generates an unsigned 32-bit random number based on the mrand48 algorithm. The syntax is:

```
function bit [31:0] urand48([integer seed]);
```

Note that `urand48()` does not adhere to random stability. For more information on the `mrand48()` algorithm, see the UNIX man pages (at the UNIX shell prompt, type `man mrand48`).

urandom_range ()

The `urandom_range()` function returns an unsigned value in the range *`maxval`*..*`minval`*. The syntax is:

```
function bit[31:0] urandom_range(bit[31:0] maxval[,
bit[31:0] minval=0]);
```

For example:

```
Example: val=urandom_range(7,0);
```

If the `minval` argument is omitted, the function returns a value in the range maxval..0.

```
Example: val=urandom_range(7);
```

If `maxval` is less than `minval`, the arguments are automatically reversed so that the first argument is larger than the second argument. This swap is done rather than issuing a runtime exception.

```
Example: val=urandom_range(0,7);
```

All of these examples produce values in the range of 0 to 7.

Constraint-Based Randomization

Constraint-based test generation enables you to automatically generate tests for functional verification. Random testing can be more effective than directed testing. By specifying constraints, you can create tests that find hard-to-reach, corner-case bugs. NTB-OV enables you to specify constraints in a compact, declarative way. NTB-OV processes the constraints using a solver that generates random values which honor the constraints.

Random constraints are built on top of an object-oriented data abstraction that models the data to be randomized. The data to be randomized include objects that contain random variables and user-defined constraints. Your constraints define the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

This section introduces the basic concepts and uses for generating constrained random stimulus within objects. NTB-OV uses an object-oriented method for assigning random values to the member variables of an object, subject to the constraints that you define (see Example 9-8).

Example 9-8 Constraint

```
class Bus {
      rand bit[15:0] addr;
      rand bit[31:0] data;

      constraint word_align {addr[1:0] == '2b0;}
}
```

In Example 9-8, the `Bus` class models a simplified bus with two random variables: `addr` and `data`, representing the address and data values on a bus. The `word_align` constraint declares that the random values for `addr` must be such that `addr` is word-aligned (the low-order 2 bits are 0).

Generating Random Values

Example 9-9 shows how to use the `randomize()` method to generate new random values for a `bus` object. (See "randomize ()" on page 4 for the `randomize()` syntax.)

Example 9-9 randomize () Call

```
program test {
      Bus bus = new;
```

```
        repeat (50){
            integer result = bus.randomize();

            if (result == OK)
                printf("addr = %16h data = %32h\n",
                    bus.addr, bus.data);
            else
                printf("Randomization failed.\n");
        }
    }
```

When you call `randomize()`, NTB-OV selects new values for all random variables in an object such that all of the constraints are satisfied. In Example 9-9, a bus object is created and then randomized `50` times. NTB-OV checks the result of each randomization for success. If the randomization succeeds, the new random values for `addr` and `data` are printed. If the randomization fails, an error message is printed. In this example, only the `addr` value is constrained (the data value is unconstrained). NTB-OV assigns unconstrained variables any value in their declared range.

Constraint programming is a powerful approach that lets you build generic, reusable objects that you can later extend or constrain to perform specific functions. This approach differs from traditional procedural and object-oriented programming, as illustrated Example 9-10, which extends the `Bus` class.

Example 9-10 Extending Bus Class

```
    class MyBus extends Bus {
        enum AddrType = {low, mid, high};
        rand AddrType type;

        constraint addr_range {
            (type == low ) => addr in {  0 :  15};
            (type == mid ) => addr in { 16 : 127};
            (type == high) => addr in {128 : 255};
        }
    }
```

Here, the `MyBus` class inherits all of the random variables and constraints of the `Bus` class, and adds a random variable called `type`, which is used to control the address range using another constraint. The `addr_range` constraint uses implication to select one of three range constraints depending on the random value of `type`. When a `MyBus` object is randomized, values for `addr`, `data`, and `type` are computed such that all of the constraints are satisfied. Using inheritance to build layered constraint systems allows you to develop general-purpose models that can later be constrained to perform application-specific functions.

You can further constrain objects using the `randomize()with` construct, which declares additional constraints inline with the call to `randomize()`, as shown in Example 9-11. See page 8 for the `randomize()with` syntax.

Example 9-11 randomize () with Call

```
task exercise_bus (MyBus bus){
      integer res;
// EXAMPLE 1: restrict to small addresses
res = bus.randomize() with {type == low;};
...
// EXAMPLE 2: restrict to address between 10 and 20
res = bus.randomize() with
      {10 <= addr && addr <= 20;};
...
// EXAMPLE 3: restrict data values to powers-of-two
res = bus.randomize() with
      {data & (data - 1) == 0;};
...
}
```

Example 9-11 illustrates several important properties of constraints:

- Constraints can be any expression with variables and constants of type bit, integer, or enumerated type.

- Constraint expressions follow Verilog syntax and semantics, including precedence, associativity, sign extension, truncation, and wraparound.

- The NTB-OV constraint solver is very robust and can handle a wide spectrum of seemingly hard problems, including algebraic factoring, complex Boolean expressions, and mixed integer and bit expressions. In the example above, the power-of-two constraint is expressed arithmetically, but it could also have been expressed using a shift operator (for example, `1 << n`, where `n` is a 5-bit random variable). If a solution exists, the solver finds it. The solver only fails when the problem is over-constrained and there is no combination of random values that satisfy the constraints.

- Constraints interact bidirectionally. In Example 9-11, the value chosen for `addr` depends on type and how it is constrained, and the value chosen for `type` depends on `addr` and how it is constrained. It is important to understand that NTB-OV treats all expression operators bidirectionally, including the implication operator (`=>`).

- Sometimes it is desirable to disable constraints on random variables. For example, consider the case where we want to deliberately generate an illegal address (non-word aligned), as shown in Example 9-12:

Example 9-12   Disabling Constraints on Random Variables

```
task exercise_illegal(MyBus bus, integer cycles){
     integer res;

     // Disable word alignment constraint.
     res = bus.constraint_mode(OFF, "word_align");

     if (res != OK) printf("constraint_mode(OFF) failed\n");

     repeat (cycles) {
     // CASE 1: restrict to small addresses.
```

```
        res = bus.randomize() with {addr[0] || addr[1];};
        ...
        }

        // Re-enable word alignment constraint.
        res = bus.constraint_mode(ON, "word_align");
        if (res != OK) printf("constraint_mode(ON) failed\n");
    }
```

You can use the `constraint_mode()` method to enable or disable any named constraint block in an object. In Example 9-12, the word-alignment constraint is disabled, and the object is then randomized with additional constraints, forcing the low-order address bits to be non-zero (and thus not aligned).

It is good practice to design your constraint hierarchy such that the lowest-level constraints represent physical limits. These limits should be grouped by common properties into well-named constraint blocks that can be independently enabled or disabled.

Similarly, you can use the `rand_mode()` method to enable or disable any random variable. When a random variable is disabled, it behaves in exactly the same way as other non-random variables.

NTB-OV provides two built-in methods you can use to perform operations immediately before or after randomization: `pre_randomize()` and `post_randomize()` . NTB-OV automatically calls these methods before and after randomization. You can be overload them with the desired functionality, as shown in Example 9-13.

Example 9-13 pre_randomize () and post_randomize () Calls

```
class XYPair {
    rand integer x, y;
}

class MyYXPair extends XYPair {
```

```
        task pre_randomize()
              super.pre_randomize();
              printf("Before randomize x=%0d, y=%0d\n", x, y);
        }

        task post_randomize(){
              super.post_randomize();
              printf("After randomize x=%0d, y=%0d\n", x, y);
        }
  }
```

By default, `pre_randomize()` and `post_randomize()` call their overloaded superclass methods. If you overload `pre_randomize()` or `post_randomize()`, you should invoke the superclass methods, except when the class is a base class (has no superclass).

With NTB-OV's object-oriented, constraint-based verification methodology, you can rapidly develop tests that cover complex functionality and better assure design correctness.

Random Variables

You can declare class variables as random using the `rand` and `randc` type-modifier keywords. The syntax is:

```
rand variable;
randc variable;
```

NTB-OV can randomize scalar variables of type integer, reg, and enumerated type. Reg variables can be any size. The `rand` and `randc` variables can be static. You can declare arrays `rand` or `randc`, in which case NTB-OV treats all of their member elements as `rand` or `randc`. You can also declare associative arrays `rand` or randc; however, only the elements in the numeric key range of 0 to n-1 are randomized, where *n* is declared using the optional assoc_size keyword, as shown in Example 9-14.

Example 9-14 Randomizing Associative Array

```
rand bit[7:0] len;
rand integer data[] assoc_size len;
```

In Example 9-14, the len variable is declared to be 8 bits wide. The randomizer computes a random value for the len variable in the 8-bit range of 0 to 255, and then randomizes the first len elements of the data array. If an element does not exist, it is created. If assoc_size is not specified, then randomize() randomizes all values in the associative array, but it does not create any new values in the array.

You can declare dynamic arrays rand or randc, and constrain the length using the optional dynamic_size keyword. The behavior here is very similar to that of associative arrays. The difference is that since dynamic arrays have to represent a set of contiguous indices, the array is resized to the new size, if the new size if more than the current size. When NTB-OV resizes an array, it copies over the old values stored in relevant indices to the new array.

It's best to avoid the old-style use of assoc_size to create a variable-length vector, because it's inefficient. Instead, use NTB-OV dynamic arrays to model variable-length vectors.

When you declare an object variable rand, all of that object's variables and constraints are solved concurrently with the other class variables and constraints. You cannot declare objects randc. For associative or dynamic arrays, if there is no element in the size range, NTB-OV creates and stores null object references for the element.

rand Modifier

Variables declared with the `rand` keyword are standard random variables. Their values are uniformly distributed over their range (see Example 9-15).

Example 9-15 rand Variables

```
rand bit[7:0] x;
```

Example 9-15 declares `x` as an 8-bit unsigned integer with a range of 0 to 255. Left unconstrained, `x` is assigned any value in the range 0 to 255 with equal probability. In this example, the probability of the same value repeating on successive calls to `randomize()` is 1/256.

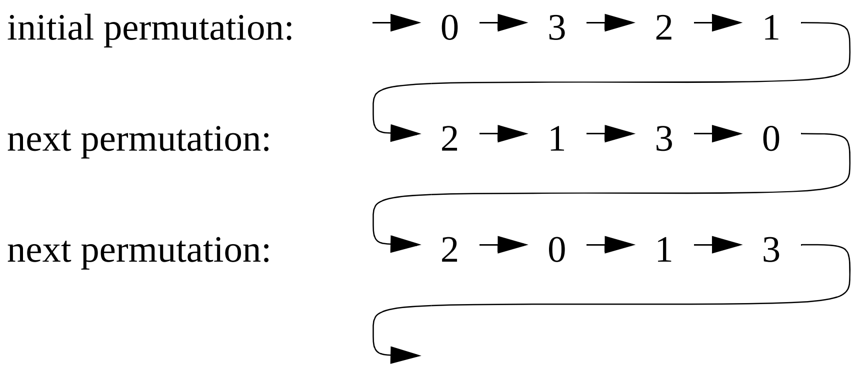
randc Modifier

Variables declared with the `randc` keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range. Random-cyclic variables can only be bit or enumerated types, and are limited to a maximum size of 16 bits, so the maximum range for any `randc` variable is 0 to 255. To understand `randc`, consider a 2-bit random variable `y`:

```
randc bit[1:0] y;
```

which can take on the values 0, 1, 2, and 3 (range 0 to 3). Randomize computes an initial random permutation of the range values of `y`, and then returns those values in order on successive calls. After it returns the last element of a permutation, randomize repeats the process by computing a new random permutation (see Figure 9-1).

Figure 9-1 randc Permutations



The basic idea is that `randc` randomly iterates over all values in the range and does not repeat any value within an iteration. When the iteration is finished, a new iteration starts automatically.

NTB-OV recomputes the permutation sequence for a `randc` variable whenever the constraints change on that variable, or when none of the remaining values in the permutation can satisfy the constraints.

NTB-OV solves for `randc` variables first, using only constraints that involve one `randc` variable and other non-random variables. The values obtained for the `randc` variables are then used as constants while solving for the remaining random variables (see Example 9-16).

Example 9-16 randc and Constraints

```
randc bit[3:0] x,y;
rand integer z;

constraint C {
      x < 10;
      y > 5;
      z == x+y;
}
```

In Example 9-16, `x` gets a value between 0 and 9 and `y` gets a value between 6 and 15. These values are chosen over successive calls to `randomize()`. In this example, the values obtained are used to solve for `z`.

Constraint Blocks

NTB-OV determines the values of random variables using constraint expressions that you declare in constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. You must define them after the variable declarations in a class, and before the task and function declarations in a class. Constraint block names must be unique within a class. The syntax to declare a constraint block is:

`constraint` *`constraint_name`* {*`contraint_expressions`*}

`constraint_name`

is the name of the constraint block. You use this name to enable or disable a constraint using the built-in `constraint_mode()` method.

`constraint_expression`

is a list of expression statements that restrict the range of a variable or define relations between variables. A constraint expression can be any expression, or can use the constraint-specific set operators: `in`, `!in`, and `dist`.

The declarative nature of constraints requires the following restrictions on constraint expressions:

- You cannot call a task or function.
- Operators with side effects (such as `++` and `--`) are not allowed.

- `Dist` expressions cannot appear in other expressions (unlike `in` and `!in`); they can only be used as top-level expressions.
- Port variables are not allowed because the width of port-bind variables is not known at compile time.

External Constraint Blocks

NTB-OV allows you to declare external constraint block bodies the same way you declare external task and function bodies (see Example 9-17).

Example 9-17 External Constraint Declaration

```
// class declaration
     class XYPair {
          rand integer x, y;
          constraint c;
     }
     // external constraint body declaration
     constraint XYPair::c { x < y; }
```

Unlike external task and function bodies, external constraint bodies can be declared in any file; they don’t have to be in the same file as the class definition.

You cannot define external constraint blocks more than once; a loader error results when NTB-OV finds a constraint block declared in more than one file.

Inheritance

Constraints follow the same general rules for inheritance as class variables, tasks, and functions. When you declare a constraint block in an extended class using the same name as a constraint in the base class, the extended class definition overrides the base class definition (see Example 9-18).

Example 9-18 Extended Class Definition Override

```
class A {
      rand integer x;
      constraint c { x < 0; }
}

class B extends A {
      constraint c { x > 0; }
}
```

In Example 9-18, an instance of class `A` constrains `x` to be less than `0`, whereas an instance of class `B` constrains `x` to be greater than `0`. The extended class `B` overrides the definition of constraint `c`. Note that NTB-OV treats constraints the same as virtual functions, so casting an instance of `B` to an `A` does not change the constraint set.

The built-in `randomize()` task is virtual, so it treats the class constraints in a virtual manner. When a named constraint is overloaded, the previous definition is overridden.

Set Membership

Constraints support integer value sets and set membership operators. The syntax is:

```
expression set_operator {value_range_list};
```

`expression`

can be any expression.

`set_operator`

can be `in` or `!in`. The `in` operator returns true if the *expression* is contained in the set; otherwise it returns false. The `!in` operator returns the negated value of the `in` operator. For example, `x !in {}` is the same as `!(x in {})`.

In the absence of any other constraints, all values (either single values or values within ranges) have an equal probability of being chosen by the `in` or `!in` operators.

`value_range_list`

is a comma-separated list of integers, enumerated types, bit expressions, and ranges. Ranges are defined by specifying a low and high bound, separated by a colon (`low_bound` : `high_bound`). Ranges include all of the integer elements between the bounds. The bound to the left of the colon must be less than or equal to the bound to the right; otherwise, the range is `NULL` (contains no values).

Consider Example 9-19.

Example 9-19 Set Membership

```
rand integer x, y, z;
constraint c1 {x in {3, 5, 9:15, 24:32, y:2*y, z};}

rand integer a, b, c;
constraint c2 {a in {b, c};}
```

Set values and ranges can be any expression. You can repeat values, and values and ranges can overlap. It is important to note that the `in` and `!in` operators are bidirectional, so the second example is equivalent to `a==b||a==c`.

Distributions

In addition to set membership, constraints support sets of weighted values called distributions. Distributions have two properties: they are a relational test for set membership, and they specify a statistical distribution function for the results. The syntax to define a distribution expression is:

```
expression dist_operator {value_range_ratio_list};
```

`expression`

can be any expression that refers to at least one variable declared as `rand`.

`dist_operator`

is `dist`. The `dist` operator returns true if the expression is contained in the set; otherwise, it returns false. In the absence any other constraints, the probability that the expression matches any value is proportional to its specified weight.

`value_range_ratio_list`

is a comma-separated list of integers, enumerated types, bit expressions, and ranges (the same as the *`value_range_list`* for set membership). Optionally, each term in the list has a weight, which you specify using the `:=` or `:/` operators. If no weight is specified, the default weight is 1. The weights can be any expression.

The `:=` operator assigns the specified weight to the item, or if the item is a range, to every value in the range.

The `:/` operator assigns the specified weight to the item, or if the item is a range, to the range as a whole. If there are `n` values in the range, the weight of each value is *`range_weight/n`*. For example:

```
x dist {100 := 1, 200 := 2, 300 := 5}
```

means `x` is equal to 100, 200, or 300 with a weighted ratio of 1-2-5. If you add an additional constraint that `x` cannot be 200:

```
x != 200;
x dist {100 := 1, 200 := 2, 300 := 5}
```

then `x` is equal to 100 or 300 with a weighted ratio of 1-5.

It is easier to think about mixing ratios (like 1-2-5) than the actual probabilities, because mixing ratios do not have to be normalized to 100%. It is also easy to convert probabilities to mixing ratios.

When you apply weights to ranges, they can be applied to each value in the range or to the range as a whole. For example:

```
x dist {100:102 := 1, 200 := 2, 300 := 5}
```

means `x` is equal to 100, 101, 102, 200, or 300, with a weighted ratio of 1-1-1-2-5.

```
x dist {100:102 :/ 1, 200 := 2, 300 := 5}
```

means `x` is equal to one of 100, 101, 102, 200, or 300, with a weighted ratio of 1/3-1/3-1/3-2-5.

In general, distributions guarantee two properties: set membership and monotonic weighting, which means that increasing a weight increase the likelihood of choosing those values.

Condition Constraints

NTB-OV provides two constructs for declaring condition (predicated) constraints:

- Implication
- if-else Constraints

Implication

You can use the implication operator (=>) to declare an expression that implies a constraint. The syntax to define an implication constraint is:

```
expression => constraint;
expression => constraint_set;
```

`expression`

can be any expression.

`implication operator (=>)`

returns true if the `expression` is false or the constraint is satisfied; otherwise, it returns false.

`constraint`

is any valid constraint.

`constraint_set`

is any valid constraint or unnamed constraint block. If the expression is true, all constraints in the `constraint_set` must be satisfied. A `constraint_set` looks like the following:

```
{   constraint1;
     constraint2;
     constarint3;
     constraintN;
}
```

In Example 9-20, the value of `mode` implies that the value of `len` is less than 10 or greater than 100. If `mode` is neither small nor large, the value of `len` is unconstrained.

Example 9-20 Implication Operator Usage

```
mode == small => len < 10;
```

```
mode == large => {len > 100; len < 200;}
```

The boolean equivalent of `(a => b)` is `(!a||b)`. Implication is a bidirectional operator. Consider Example 9-21:

Example 9-21   Implication Operator and Constraint

```
bit[3:0] a, b;
constraint c {(a == 0) => (b == 1);}
```

In Example 9-21, both `a` and `b` are 4 bits, so there are 256 combinations of `a` and `b`. The constraint `c` says that `a == 0`implies `b == 1`, so 15 combinations have to be eliminated: {0,0}, {0,2}, … {0,15}. Therefore, the probability that `a == 0` is 1/(256-15) or 1/241.

It is important to understand that NTB-OV is designed to cover the whole random value space with uniform probability. This allows randomization to better explore the whole design space.

if-else Constraints

NTB-OV also supports if-else style constraint declarations. The syntax is:

```
if (expression) constraint_or_constraint_set;
[else constraint_or_constraint_set;]
```

`expression`

can be any expression.

`constraint`

can be any valid constraint. If the `expression` is true, the first constraint must be satisfied; otherwise the optional else-constraint must be satisfied.

`constraint_set`

is a set of semicolon separated constraints. If the `expression` is true, all constraints in the first constraint set must be satisfied; otherwise, all constraints in the optional else-constraint-block must be satisfied.

If-else style constraint declarations are equivalent to implications (see Example 9-22).

Example 9-22 if-else Constraint

```
if (mode == small)
      len < 10;
else
      if (mode == large)
      { len > 100; len < 200; }
```

is equivalent to:

```
mode == small => len < 10;
mode == large => { len > 100; len < 200; }
```

In Example 9-22, the value of `mode` implies that the value of `len` is less than `10`, greater than `100`, or unconstrained. Like implication constraints, if-else style constraints are bidirectional.

Hierarchical Constraints

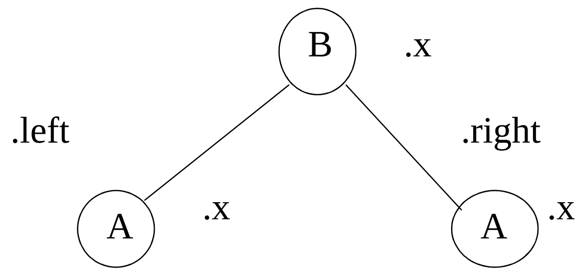
When you declare an object member of class `rand`, NTB-OV simultaneously randomizes all of its constraints and random variables along with the other class variables and constraints. Constraint expressions involving random variables from other objects are called hierarchical constraints (see Example 9-23).

Example 9-23 Hierarchical Constraints

```
class A  {
      rand bit[7:0] x;
}
class B  {
      rand A left;
```

```
    rand A right;
    rand bit[7:0] x;

    constraint C {left.x <= x; x <= right.x;}
}
```



In Example 9-23, hierarchical constraints are used to define the legal values of an ordered binary tree called a heap. Class `A` represents a leaf-node with an 8-bit value `x`. Class `B` extends class `A` and represents a heap node with value `x`, a left subtree, and a right subtree. Both subtrees are declared as `rand` in order to randomize them at the same time the other class variables are randomized. Constraint block `C` has two hierarchical constraints that relate the left and right subtree values to the heap node value. When an instance of class `B` is randomized, NTB-OV simultaneously solves for `B` and its left and right children, which in turn may be leaf nodes or more heap nodes. In Example 9-23, NTB-OV solves five heap nodes simultaneously, ensuring that each heap node meets the left-right ordering requirement.

In general, NTB-OV determines which objects, variables, and constraints are to be randomized as follows:

1. First, determines the set of objects that are to be randomized. Starting with the object that invoked the `randomize()` method, NTB-OV adds all the objects that are contained within it, are declared `rand`, and are active (see rand_mode ()). This definition is recursive and includes all active random objects that can be reached from the starting object. The objects selected in this step are referred to as the active random objects.
2. Next, global randomization selects all of the active constraints from the set of active random objects. These are the constraints that are applied to the problem.
3. Finally, global randomization selects all of the active random variables from the set of active random objects. These are the variables that NTB-OV randomizes. All other variable references are treated as state variables, whose current value is used as a constant.

Variable Ordering

NTB-OV ensures that random values are given a uniform value distribution over the legal value space (that is, all combinations of values have the same probability of being chosen). This important feature guarantees that all value combinations are equally probable.

However, you can also force certain combinations to occur more frequently. Consider the case where a 1-bit control variables constrains a 32-bit data value (`d`), as shown in Example 9-24.

Example 9-24 Variable Ordering

```
class B {
      rand bit s;
      rand bit[31:0] d;
      constraint c { s => d == 0; }
}
```

In Example 9-24, constraint `c` says: `s` implies `d` equals `0`. Although this reads as if `s` determines `d`, NTB-OV actually determines `s` and `d` together. There are 2**32 valid combinations of `{s,d}`, but `s` is only true for {1,0}. Thus, the probability that `s` is true is 1/2**32, which is almost never. NTB-OV provides a mechanism you can use for ordering variables so that `s` is chosen independent of `d`. This mechanism defines a partial ordering on the evaluation of variables. You specify the partial ordering using the `solve-before` keywords, as shown in Example 9-25.

Example 9-25 solve-before

```
class B {
      rand bit s;
      rand bit[31:0] d;

      constraint c { s => d == 0; }
      constraint order { solve s before d; }
}
```

In Example 9-25, the `order` constraint instructs NTB-OV to solve for `s` before solving for `d`. The effect is that `s` is now chosen true with 50% probability, and then `d` is chosen subject to the value of `s`. Accordingly, `d == 0` occurs 50% of the time, and `d != 0` occurs for the other 50%.

You can use variable ordering to force selected corner cases to occur more frequently than they otherwise would. The syntax to define variable order in a constraint block is:

```
solve variable_list before variable_list;
```

`variable_list`

is a comma-separated list of integral scalar variables or array elements.

The following restrictions apply to variable ordering:

- The variables can only be those declared as `rand`. Variables declared as `randc` are not allowed.
- The variables must be integral scalar values of type integer, reg, or enumerated type.
- A named constraint block member of a class can contain both regular value constraints and ordering constraints.
- A `randomize() with` constraint block in a task or function can contain both value constraints and ordering constraints.
- There cannot be any circular dependencies in the ordering, such as solve a before b combined with solve b before a.
- Variables that are not explicitly ordered are solved with the last set of ordered variables. These values are deferred until as late as possible to ensure a good distribution of value.
- Variables can be solved for in an order that is not consistent with the ordering constraints. An example where this might occur is shown in Example 9-26.

Example 9-26 Variable Ordering not Consistent with Constraints

```
x == 0;
x < y;
solve y before x;
```

In Example 9-26, because `x` has only one possible assignment (`0`), `x` can be solved for before `y`. The constraint solver uses this flexibility to speed up the solving process.

Array Constraints

You can use array constraints to specify templetized constraints for the NTB-OV constraint solver. NTB-OV currently supports array constraints that use `foreach` loops and array aggregate/ containment operators.

foreach loops

Use the `foreach` construct to apply constraints to each member of an array or Smart Queue. In a `constraint_set`, NTB-OV support all array types, including fixed size, multidimensional, associative, and dynamic arrays. The foreach construct applies the contained constraints to every indexable member in the array. The syntax is:

```
foreach (name, [loop_variable  | loop_varible_list]) {
...constraint_set...
}
```

`name`

is the name of the array or Smart Queue.

`loop_variable`

is the name of the automatically generated index variable, which acts as an indexing member. Declare the `loop_variable` in the argument list of the foreach block (it is scoped in that block).

`loop_variable_list`

is a comma-separated list of loop variables. You can use an asterisk (`*`) as a placeholder for an index in a multidimensional array. That index is ignored in the foreach loop.

`constraint_set`

can be any valid code for constraints.

Example 9-27 shows a `foreach` loop that acts on a Smart Queue. This example constrains every member of the queue to be equal to zero.

Example 9-27 Simple foreach Loop on a Smart Queue

```
class D {
      rand integer my_q[$];
      constraint init {
            foreach(my_q,i){
                  my_q[i] == 0;
            }
        }
}
```

Example 9-28 shows a foreach loop that acts on a two-dimensional array.

Example 9-28 Foreach Loop on a Two-dimensional Array

```
class A {
      rand integer d[2][2];
      foreach(d, *, index) {
            (d[0][index] > 22) && (d[0][index] < 33);
      }
}
```

Example 9-29 shows how to use guards to constrain a two-dimensional array. In this example, the implication operator is used to apply constraints to each member of the array. The first member of the array is unconstrained.

Example 9-29 Using Guards to Constrain Two-dimensional Array

```
class C {
      rand integer size;
      rand integer arr[*];

      constraint cc{
            size > 0; size < 5;
```

```
                foreach (arr, index) {
                      (index > 0 ) => (arr[index] == 0 );
                }
          }
    }
```

Example 9-30 shows how to apply constraints to specific ranges in an array.

Example 9-30 Applying Constraints to Specific Ranges in Array

```
class D {
      rand integer arr[*] dynamic_size 30;
      constraint cc {
            arr.size() == 30;
            foreach (arr, index) {
                  (index < 10 ) =>
                  { arr[index] > 0;
                        arr[index] < 10;}
                  (index > 9 && index < 20 ) =>
                        {arr[index]> 20; arr[index] < 30;}
                  (index > 19 && index < 30) =>
                        {arr[index] > 30; arr[index] < 40;}
      }
      }
}
```

Example 9-31 shows how to use two foreach loops to apply constraints.

Example 9-31 Two foreach Loops

```
class E {
      rand  integer x[];
      rand  integer y[*];

      constraint size_cons {
            x.size() == 10;
            y.size() == 10;

            foreach (x, index) {
                  x[index] > 0;
                  x[index] < y[index];
            }

            foreach (y, index) {
```

```
                y[index] > 0;
                y[index] < 100;
            }
        }
}
```

Example 9-32 shows how to use nested foreach loops to constrain elements of an integer array so that all elements of the array are unique. A further constraint is applied limiting the values of each element between `0` and `1000`.

Example 9-32 Nested foreach Loops

```
class MyClass {
      rand integer a[$];
      constraint c0 {
            a.size() >= 10;
            a.size() != 15;
            a.size() <= 20;
            foreach(a , i) {
                  a[i] in {0:1000};
                  foreach(a , j)  {
                        (i  != j ) =>   a[i] != a[j];
                  }
            }
      }
}

program ConstraintTest {
      MyClass c0 = new();
      void = c0.randomize();
      foreach(c0.a, index) {
            printf("a[%0d]: %0d\n",index, c0.a[index]);
      }
}
```

Example 9-32 generates the following output:

```
a[0]: 833
a[1]: 425
a[2]: 788
a[3]: 336
a[4]: 304
a[5]: 352
a[6]: 509
```

```
a[7]: 434
a[8]: 779
a[9]: 339
```

Elaboration Guards Inside foreach Loops

You can optionally guard constraints residing within the scope of a foreach block using a top-level implication operator (`=>`). If you specify a guard for a foreach constraint and that guard comprises of only state variables or random variables whose rand mode is `OFF`, the NTB-OV constraint solver evaluates the guard before any error checking is performed. If such a guard evaluates to `FALSE`, the NTB-OV constraint solver does not generate a verification error for the following conditions:

- `x` or `z` values on state variables
- Null pointer errors
- Out of bounds array index errors

Consider Example 9-33, which does not guard the constraint.

Example 9-33   Constraint Out of Bounds without Guarded

```
class problem {
      rand integer buf[5];

      constraint out_of_bounds {
// When i = 4, buf[i+1] = buf[5] is
// an out of bounds array index
            foreach(buf,i){
                  buf[i] == buf[i+1];
            }
      }
}
program run_time_error{
      problem p = new;
      void = p.randomize();
}
```

For Example 9-33, the NTB-OV constraint solver generates the following error message:

```
Constraint solver failed - Constrained entry in rand array is outside

of the size.
ERROR: Vera Runtime : Fatal Error
      Location: CALL in program run_time_error (bounds.vr, line 17,
      cycle 0);
READY in function problem.randomize (bounds.vr, line 12, cycle 0)
```

Now, consider a minor variation of the same example, where the foreach constraint is guarded by an appropriate expression, as shown in Example 9-34.

Example 9-34   Constraint Out of Bounds with Guarded

```
class problem {
      rand integer buf[5];

      constraint guarded {
      // When i<4, buf[i+1] are valid array indices
      foreach(buf,i){
            (i < 4) => buf[i] == buf[i+1];
        }
    }
}

program no_run_time_error {
      problem p = new;
      void = p.randomize();
}
```

In Example 9-34, the NTB-OV constraint solver does not generate a verification error because as `buf[i+1]` is appropriately guarded.

Specifying Guards for solve-before Constraints inside foreach Loops

You can also specify elaboration guards for `solve-before` constructs that occur inside foreach loops (see Example 9-35).

Example 9-35 Using Guards with solve-before Constructs

```
rand integer x[];

constraint b1 {
      x.size() in{2,3};
      foreach(x,i){
            i> 0 => solve x[i-1] before x[i] hard;
      }
}
```

If you specify a guard expression for any `solve-before` construct inside a `foreach` loop, NTB-OV evaluates that guard expression for every applicable value of the corresponding index variable. The constraint solver elaborate the relevant `solve-before` construct only for those values of the corresponding index variable that result in a TRUE value for the guard expression (see Example 9-36).

Example 9-36 Guard with solve-before in foreach Loop

```
rand integer x[];
constraint b1 {
      x.size() == 3; foreach(x,i) {
            i > 0 => solve x[i-1] before x[i] hard;
}
```

For Example 9-36, the constraint solver expands the specified `solve-before` construct into `solve x[0]before x[1]hard` and `solve x[1]before x[2]hard`.

Array Aggregates in Constraints

NTB-OV includes a set of aggregate operators you can use to declare complex constraints for arrays and Smart Queues in a compact, flexible format. Following is the syntax for specifying constraints using the NTB-OV aggregate operators:

```
array_name.aggregate_operator() [with loop_variable
     |loop_variable_list: (loop_expression)];
```

`array_name`

is the name of the array or Smart Queue.

`aggregate_operator`

is any one of the following: `sum`, `product`, `logical_and`, `logical_or`, `bit_and`, `bit_or`, `bit_xor`, or `bit_xnor`.

`loop_variable`

is the variable used to loop through the array or Smart Queue.

`loop_variable_list`

is a list of loop variables. This is only used with multidimensional arrays. The syntax for a `loop_variable_list` is:

```
* | loop_variable, * | loop_variable...
```

asterisk (*)

is the placeholder for an index of one of the dimensions. You can specify the actual index in the `loop_expression`.

`loop_variable_list`

- cannot contain only an asterisk (*), which is a placeholder
- the number of `loop_variables` and asterisks together must match the number of dimensions in the multidimensional array.
- `loop_variables` and stars can occur in any order. For example:

```
loop_var1, *, *, loop_var2 // four-dimensional array
-or-
*, *, loop_var1, loop_var2
```

`loop_expression`

is any valid NTB-OV expression.

The array aggregate expression is a valid part of a constraint expression you can use any place that a variable can be used, with the exception of `solve-before` constraints (see Example 9-37).

Example 9-37 Array Aggregates in Constraints

```
x+(arr.sum() with i:(arr[i]+1)) == 10;
```

contains

You can specify constraints using the `contains` aggregate operator. The syntax is:

```
array_name.contains(contains_expression) [with loop_variable
|loop_variable_list: (loop_expression)]
```

`contains_expression`

is any valid NTB-OV expression.

For information on the other syntactic components, see “Array Aggregates in Constraints” on page 414. Example 9-38 and Example 9-39 show examples that use the contains aggregate operator.

Example 9-38 Contains

```
x+arr.contains(1) == 10;
```

Example 9-39 Another Contains

```
(arr.contains(x+y) with i:(arr[i]*2)) || (p < q);
```

Generating Expressions from Aggregate Operator Expressions

When you use an aggregate operator, NTB-OV generates an expression and expands it into a constraint that it then solves for. The generated expression is a result of applying the `except contains` operator to each element of the array or Smart Queue (see Example 9-40).

Example 9-40 Expressions from Aggregate Operator

```
arr[0] operator arr[1] operator arr[2] ...
```

In Example 9-40, if `operator` is `sum`, and the size of the array or Smart Queue is three, the generated expression is:

```
arr[0] + arr[1] + arr[2]
```

When you specify the `loop_expression` using `with`, the generated expression is the result of applying the operator to a list of expressions created by applying each array or Smart Queue element to the `loop_expression`. For this next example:

```
arr.sum() with i: (arr[i] + i)
```

the generated expression is:

```
( (arr[0] + 0) + (arr[1] + 1) + (arr[2] + 2) )
```

The `contains` aggregate operator generates a set membership expression and expands it into a constraint, which is then solved for. The set membership expression is a result of using the `contains_expression` as the set expression with each element of the array or Smart Queue as the `value_range_list`. For example:

```
contains_expression in { arr[0], arr[1], arr[2], ... }
```

When you specify the optional `loop_expression` using `with`, the set membership expression is a result of using the `contains_expression` as the set expression and applying each element of the array or Smart Queue to the `loop_expression` as the `value_range_list`. For example:

```
contains_expression in { loop_expression(arr[0]),
loop_expression(arr[1]), loop_expression(arr[2]), ... }
```

Table 9-1 shows a list of all the NTB-OV aggregate operators.

Table 9-1 NTB-OV Aggregate Operators

Operator	Description
`sum`	performs addition
`product`	performs multiplication
`logical_and`	performs logical AND operation
`logical_or`	performs logical OR operation
`bitwise_and`	performs bitwise AND operation
`bitwise_or`	performs bitwise OR operation
`bitwise_xor`	performs bitwise exclusive OR operation
`bitwise_xnor`	performs bitwise exclusive NOR operation
`contains`	creates a set membership

Example 9-41 shows how to use the `sum` aggregate operator.

Example 9-41 sum Operator

```
class C {
rand integer size;
rand integer arr[*] dynamic_size size;
constraint cc {
size == 3;
arr.sum() < 100;
    }
}
```

For Example 9-41, NTB-OV generates the following constraint expression:

```
arr[0] + arr[1] + arr[2] < 100;
```

Example 9-42 shows how to use the `sum` aggregate operator using `with`.

Example 9-42 sum Operator using with

```
class C {
      rand integer size;
      rand integer arr[*] dynamic_size size;
      constraint cc {
            size == 3;
            arr.sum() with i: (arr[i]+i) < 100;
      }
}
```

For Example 9-42, NTB-OV generates the following constraint expression:

```
((arr[0] + 0) + (arr[1] + 1) + (arr[2] + 2) )  < 100;
```

Example 9-43 shows how to use the `sum` aggregate operator using `with` on a multidimensional array.

Example 9-43 Multidimensional Arrays

```
class C {
      rand integer arr[2][2];
      constraint cc {
            arr.sum() with i, j: (arr[i][j]) < 100;
      }
}
```

For Example 9-43, NTB-OV generates the following constraint expression:

```
( arr[0][0] + arr[0][1] + arr[0][2] + arr[1][0] + arr[1][1] +
arr[1][2] + arr[2][0] + arr[2][1] + arr[2][2] ) < 100
```

Example 9-44 shows how to use nested aggregates.

Example 9-44   Nested Aggregates

```
class D {
      rand integer x[*];
}
class C {
      rand D arr[*];
      constraint cc {
            foreach(arr, i) {
                  arr[i].x.sum() with j: (arr[i].x[j]) <
                  100;
            }
      }
}
```

Example 9-45 shows how to use the `contains` operator.

Example 9-45   Contains Operator

```
class C {
      rand integer x;
      rand integer arr[3];
      constraint cc {
            arr.contains(x) with i: (arr[i]);
      }
}
```

For Example 9-45, NTB-OV generates the following constraint expression:

```
x in { arr[0],arr[1],arr[2]};
```

Example 9-46 shows how to use a multidimensional array.

Example 9-46   Multidimensional Array

```
class C    {
      rand bit[15:0] x[3][3][3];
      constraint aggr
      {
            x.contains(253) with i,j,*: (x[j][i][0]);
```

```
        }
    }
```

For Example 9-46, NTB-OV generates the following constraint expression:

```
253 in {x[0][0][0], x[0][1][0], x[0][2][0], x[1][0][0], x[1][1][0],
x[1][2][0], x[2][0][0], x[2][1][0], x[2][2][0]}
```

Note:

When you specify `*` as a placeholder for a particular dimension in the context of array aggregates/containment, NTB-OV skips that dimension elaboration.

Aggregate Operation Usage Notes

Empty Array. If you define an aggregate operation on an array of size 0, NTB-OV does the following:

- `array.sum()`: Substitutes 0 for the expression.
- `array.product()`: Substitutes 1 for the expression.
- `array.contains()`: Substitutes 0 for the expression.

For all other aggregate operations, NTB-OV issues a runtime error if it finds a reference to the aggregate operation in an ON constraint block.

The array aggregate operators and the `foreach` loop support fixed-size, dynamic, and associative arrays, and Smart Queue types of arrays in every context.

Using Elaboration guards Inside Array Aggregates/ Containment Operators

NTB-OV applies aggregate/containment operators over expressions specified using the associated `with` construct. You can optionally guard such expressions using a ternary operator. If you use a ternary operator and the predicate expression of that ternary operator contains only state variables or random variables whose `rand` mode is `OFF`, the constraint solver evaluates the predicate before any error checking is performed. If the predicate evaluates to true (false), the solver chooses expressions specified in the true (false) branch, and applies the specified aggregate/containment operator over the same. For the expression that was not chosen, the solver does not generate any verification error for the following conditions:

- X or Z values on state variables
- Null pointer errors
- Out of bounds array index errors

Consider Example 9-47.

Example 9-47   Unguarded Aggregate Expression

```
class problem {
      rand integer buf[5];
      constraint out_of_bounds {
/* When i = 4, buf[i+1] = buf[5] is an out of bounds array index*/
            foreach(buf,i){
                  buf.sum() with i:(buf[i+1]) == 10;
            }
      }
}
program run_time_error {
      problem p = new;
      void = p.randomize();
}
```

For Example 9-46, the constraint solver generates the following error message:

```
Constraint solver failed - Constrained entry in rand array is outside
of the size.
ERROR: Vera Runtime : Fatal Error
Location: CALL in program run_time_error (bounds.vr, line 17, cycle 0);
READY in function problem.randomize (bounds.vr, line 12, cycle 0)
```

Now, if you make minor variation to this example, where the aggregate expression is guarded using an appropriate ternary operator, the solver does not generate a verification error, because `buf[i+1]` is appropriately guarded (see Example 9-48).

Example 9-48   Aggregate Expression Guarded by Ternary Operator

```
class problem {
      rand integer buf[5];
      constraint guarded {
/* When i < 4, buf[i+1]are valid array indices */
            foreach(buf,i){
                  buf.sum() with i: ( i<4 ? buf[i+1]:0) == 10;
            }
      }
}

program no_run_time_error {
      problem p = new;
      void = p.randomize();
}
```

Specifying Default Constraints

You can specify default constraints by placing the `default` keyword ahead of a constraint block definition. The syntax is:

```
[default] constraint constraint_name {constraint_expressions}
```

A default constraint for a particular variable applies if no other non-default constraints are used for that variable. The constraint solver attempts to satisfy the applicable default constraints for all variables; if they cannot be satisfied, the solver generates an error. Example 9-49 shows how to specify default constraints.

Example 9-49 Default Constraint

```
default constraint foo {
      x > 0;
      x < 5;
}
```

If no non-default constraints apply to variable `x`, the constraint solver satisfies the specified `default` constraints.

Properties of Default Constraints

Default constraints have the following properties:

- All constraint expressions in a default constraint block are considered to be default constraints.
- You can specify multiple default constraints (possibly in multiple constraint blocks) for multiple random variables; these constraints are solved together.
- You can query the status of a default constraint block and turn it `ON` or `OFF` using `constraint_mode()`.
- You cannot define unnamed constraint blocks (that is, `randomize()with`) as defaults; the NTB-OV compiler generates an error for this.
- A default constraint block can refer to any variable visible in the scope where it is declared.

- A default constraint block can be defined externally in a different file than one in which the constraint block is declared.
- Ordering constraints are allowed in default blocks, but these constraints are treated as non-default constraints.

Overriding Default Constraints

The default variables of a default constraint are defined as those variables that are `rand` and are `rand mode ON` and part of the default constraint. The constraint solver applies default constraints when they are not overridden by any non-default constraints and contain at least one default variable. Default constraints that do not contain any default variables are ignored. To override a default constraint, the non-default constraint must satisfy the following properties:

- The constraint mode for the constraint block that contains the non-default constraint is `ON`, or is inside a `randomize-with` constraint block.
- Non-default constraints should constrain all the variables of the default constraint or constraint expression.
- If the default variable is on the right-hand side of a guarded constraint, the guard has to be true, regardless of whether the guard has random variables.
- The non-default constraint should not be an ordering constraint.

If the default variable is in the guard of the non-default constraint, the non-default constraint does not override the default constraint for that variable.

A default constraint does not override other default constraints under any condition. Thus, if all constraint blocks are declared as defaults, none of them are overridden (see Example 9-50).

Example 9-50 Constraint Override

```
class C {
          rand reg[3:0] x;
          default constraint c1 {x == 0;}
}

class D extends C {
          constraint d1 {x > 5;}
}

program P {
     D d = new();
     void = d.randomize();
     void = d.constraint_mode(OFF, "d1");
     void = d.randomize();
}
```

In Example 9-50, the default constraint in block `c1` is overridden by the non-default constraint in block `d1`. The first call to **`randomize()`** picks a value between 6 and 15 for `x`, since the non-default constraint applies. Here, the default constraint is ignored. The second call switches off the non-default constraint, and a value of 0 is picked for `x` in the call to **`randomize()`**. Here, the default constraint is applied.

Example 9-51 Another Constraint Override

```
class C {
     rand reg[3:0] x;
     default constraint c1 {x >= 3; x <= 5;}
}

class D {
     C c;
     reg y;
     constraint d1 {y >= c.x > 5;}
}
```

```
program P {
      D d = new();
      d.c = new();
      d.y = 10;
      void = d.randomize();
      d.y = 0;
      void = d.randomize();
}
```

Example 9-51 shows another constraint override example. In this example, the first call to **`randomize()`** picks a value between 5 and 10 for x, because the non-default constraint applies, given that the guard evaluates to TRUE. Here, the default constraint is ignored. In the second call to **`randomize()`**, the guard for the non-default constraint evaluates to FALSE, so the default constraint is applied and the solver picks a value between 3 and 5 for x.

Example 9-52 shows another example. With this example the output is `x = 10` because only the second constraint in the default constraint block is overridden.

Example 9-52 Second Constraint Overridden

```
class A {
      rand reg[3:0] x;
      rand reg[3:0] y;

      default constraint T {
            x == 10 ; y == 10;
      }
      constraint E {
            y == 0;
      }
      task post_randomize() {
            printf("x = %0d y = %0d
      }
}

program test {
      integer ret;
      A a = new;
      ret = a.randomize();
}
```

Example 9-52 generates the following output:

```
x = 10 y = 0
```

Example 9-53 shows cases where the guard expression is true and then false.

Example 9-53 Guard Expression True and False with Constraint

```
class A    {
      rand reg[3:0] x;
      rand reg[3:0] y;
      rand reg g;
      default constraint T {
            x + y == 10 ;
      }
      constraint E {
            g => y == 1;
      }
      task post_randomize(){
            printf("x = %0d y = %0d g = %0d
      }
}
program test {
      integer ret;
      A a = new;
      printf("First call to randomize,
            the guard g evaluates to FALSE\n");
      ret = a.randomize() with {a.g == 0;};
      printf("Second call to randomize, the
            guard g evaluates to TRUE\n");
      ret = a.randomize() with {a.g == 1;};
      printf("Third call to randomize, constraint mode of
            constraint block E is OFF \n");
      ret = a.constraint_mode(OFF, "E");
      ret = a.rand_mode(OFF, "g");
      ret = a.randomize();
}
```

Example 9-53 generates the following output:

```
First call to randomize,the guard g evaluates to FALSE
x = 9 y = 1 g = 0
Second call to randomize,the guard g evaluates to TRUE
```

```
x = 3 y = 1 g = 1
Third call to randomize, constraint mode of constraint block E is OFF
x = 1 y = 9 g = 1
```

In Example 9-53, in the first call to `randomize()`, the guard evaluates to FALSE. Therefore, the constraint does not override the default constraint and `x + y == 10`. In the second call to `randomize()`, the guard evaluated to TRUE.Therefore, since the non-default constraint is valid it overrides the default constraint. In the third call to `randomize()`, the constraint block is turned `OFF`. Therefore, because the non-default constraint is turned off, the default constraint is not overridden and `x + y == 10`.

Example 9-54 shows one more example of how default and non-default constraints work in NTB-OV.

Example 9-54   More Guard Expressions with Constraints

```
class A {
      rand reg[3:0] x;
      rand reg[3:0] y;
      default constraint T {
            x + y == 10 ;
      }
      task post_randomize(){
            printf("x = %0d y = %0d
      }
}
program test {
      integer ret;
      A a = new;
      printf("First call to randomize, rand mode of x and y are ON
      ret = a.randomize();
      printf("Second call to randomize, rand mode of x and y are OFF
      ret = a.rand_mode(OFF , "x");
      ret = a.rand_mode(OFF , "y");
      a.x = 0; a.y = 0;
      ret = a.randomize();
      printf("Return status of randomize is %0d
}
```

Example 9-54 generates the following output:

```
First call to randomize, rand mode of x and y are ON
x = 3 y = 7
Second call to randomize, rand mode of x and y are OFF
x = 0 y = 0
Return status of randomize is 1
```

In the first call to `randomize()` in Example 9-54, the default is not overridden; therefore, `x + y == 10`. In the second call to `randomize()`, the rand mode of `x` and `y` is turned `OFF` and both are set to 0. Here, the default constraints are ignored because they don't contain any default variables. If the default constraint had not been ignored, the return status would have been 0 because `randomize()` would have failed.

Using Unidirectional Constraints

The current extension to the constraint language allows you to specify partition points in the constraint set using the `void()` unary function or the `hard` keyword in conjunction with `solve-before`. These techniques improve the constraint solver's performance and capacity.

- void ()
- solve-before hard

void ()

You can use any valid constraint `expression` as a parameter to the `void()` function. The syntax is:

```
function return_type void(expression);
```

`return_type`

is the return type of the specified `expression`.

`expression`

can be any valid constraint expression.

Example 9-55 shows an example that uses the `void()` function.

Example 9-55 void () Call

```
constraint b1 {
     y in {2,3};
     x % void(y) == 0;
}
```

The `void()` function imposes an ordering among the variables. NTB-OV solves all parameters to `void()` function calls in a constraint before it solves the constraint.

NTB-OV considers the set of variables that participate in a constraint in two sets: those that are `void()` parameters, and those that are not. The NTB-OV constraint solver:

1. solves function parameters before solving the constraint.
2. solves a constraint when any of the non-function parameters incident on it are solved.
3. solves a constraint when at least one non-function parameter incident on it is solved, unless the constraint has no non-function parameters incident on it. In that case, the constraint acts as a checker.
4. solves previously unsolved non-function parameters when the constraint is solved.
5. ignores ordering directives derived from the use of `void()` in `OFF` constraint blocks; they do not affect partitioning in any way.

Consider the constraint block shown in Example 9-56.

Example 9-56 void () Constraint Blocks

```
constraint b1 {
      y in{2,3};
      x % void(y) == 0;
      x != (y+r);
      z == void(x*y)
      r == 5;
}
```

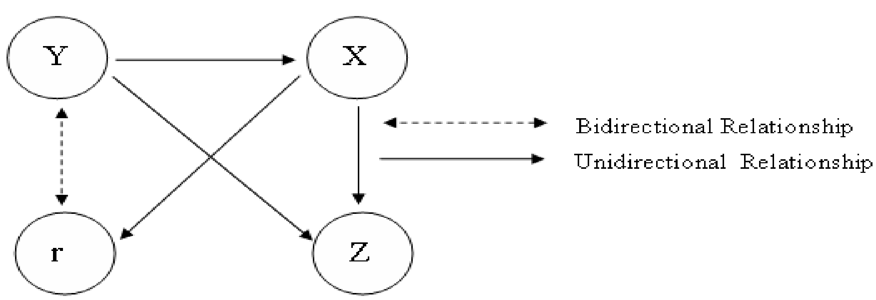
Table 9-2 lists the constraints shown in Example 9-56 and tells which ones are function and non-function parameters.

Table 9-2 Function and Non-function Parameters

Constraint	Function Parameter	Non-Function Parameter
`y in {2,3};`	`---`	`y`
`x%void(y)==0;`	`y`	`x`
`x!=(y+r);`	`---`	`x,y.r`
`z==void(x*y);`	`x,y`	`z`
`r == 5;`	`---`	`r`

Figure 9-2 shows the dependencies between the variables used in Example 9-56. Here, `y` must be solved before `x` (see `x% void(y) ==0;`). And `y` and `x` must be solved before `z` (see `z==void(x*y);`). Therefore, the solve order is `y`, `x`, and then `z`.

Figure 9-2 Variable Dependencies



Now, one ordering question remains: when can `r` be solved for?

- Can `r` be solved at the same time as `y`?

Consider the constraint `x !=(y+r)`. If no solve order is specified by unidirectional constraints, `r` can be solved at the same time as `y` and `x`. However, because it says `x% void(y)==0` in the same constraint block, NTB-OV imposes an ordering whereby `y`, which is a function parameter in `x%void(y)==0`, must be solved before `x`. This means that the constraint solver cannot solve for the three variables at the same time in `x!=(y+r)`. Here, NTB-OV solves for `y` in `y in{2,3}` before `x` and `r`.

- Can `r` be solved at the same time as `x`?

  Consider `x !=(y+r)` again. Figure 9-2 illustrates that the relationship between `r` and `x` is bidirectional, meaning there is no unidirectional constraint imposed between these two variables, and thus no ordering dependency between `x` and `r`. Therefore, `x` and `r` can be solved at the same time.

So the solve order is `y`, `x` and `r`, and then `z`.

The constraints used for solving `x`, `y`, `z`, and `r` are as follows:

1. `y` is solved using the constraint `y in {2,3}`.
2. `x` and `r` are solved using the constraints `x%y == 0`, `x!=y+r`, and `r==5`.
3. `z` is solved using the constraint `z==x*y`.

NTB-OV issues a warning message when it detects the following uses of `void()`:

- when the entire constraint expression is the parameter to `void()`. For example:

```
constraint b1 {void(x<y);}
```

At runtime, NTB-OV ignores this `void()` specification.

- when it finds expressions purely over constants and state variables, including loop variables in the case of array constraints, used as parameters to `void()`. For example:

```
constraint b1{x==void(5);}
```

At runtime, NTB-OV ignores this `void()` specification.

- when it finds a `void()` function applied to a subexpression of the parameter of another `void()` function. For example:

```
constraint b1{x==void(y + void(z));}
```

At runtime, NTB-OV ignores the nested `void()` specifications (same as `x==void(y+z)`).

solve-before hard

The `solve-before` construct supports an optional `hard` modifier (see Example 9-57).

Example 9-57 solve-before hard

```
constraint b1 {
      y in{2,3};
      x%y == 0;
      solve y before x hard;
}
```

Using `solve-before hard` imposes an ordering between the variables. You can use this feature to assign a priority order to each variable. A higher priority indicates that the variable should be solved earlier.

NTB-OV splits the set of variables for a constraint into two sets, those at the lowest priority and those not at the lowest priority. Then the following semantic rules apply:

1. Higher priority variables are solved before the constraint is solved.
2. The constraint is solved when the lowest priority variables on it are solved.
3. NTB-OV ignores `solve-before hard` constraints in `OFF` constraint blocks.

Example 9-58 shows a constraint block that contains some `solve-before hard` constraints.

Example 9-58 solve-before hard Constraints

```
constraint b1 {
      y in {2,3};
      x % y == 0;
      x != y;
      z == x * y;
      solve y before x hard;
      solve x before z hard;
}
```

Table 9-3 Early and Late Constraints

Constraint	Early	Late
y in{2,3};	--	y
x%y == 0;	y	x
x !=y;	y	x
z==x*y;	x,y	z

In Example 9-58, the constraints used for solving `x`, `y`, and `z` are:

1. y is solved using the constraint: `y in(2,3};`

2. x is solved using the constraints: `x%y == 0; x!=y;`

3. z is solved using the constraint: `z ==x*y;`

The solve order is the same as shown in the previous section.

solve-before hard and void()

Every randomized variable inside `void()` or `solve-before hard` is solved first and set to static values prior to randomizing any other variables in the constraint.

The difference is that you can use the `void()` construct as part of an expression, whereas `solve-before hard` is on a separate line, by itself.

Effect of rand_mode on Unidirectional Constraints

Using `rand_mode()` to turn a variable `OFF` has the same effect as declaring the variable non-random. Thus, NTB-OV drops ordering directives on it. This has the side effect of dropping some transitive ordering relationships as well. For example, if `x` were turned `OFF` in Example 9-59, there would be no ordering relationship left between `y` and `z` either.

Example 9-59 rand_mode with Unidirectional Constraints

```
constraint b1 {
      y in {2,3};
      x % void(y) == 0;
      x != y; z == void(x);
}
```

The introduction of unidirectional constructs has resulted in extensions to the semantics of numerous pre-existing constraint constructs.

Semantics of solve-before Construct Without the hard Modifier

The `solve-before` constraints that are not modified by the `hard` directive are used together in a partition, so they don't affect the partitioning, but do affect the order of variable assignment within a partition, very much like they affect the order of variable assigning on the complete problem prior to this release (see Example 9-60).

Example 9-60   solve-before with no hard

```
constraint b1 {
      y in {2,3};
      x % y == 0;
      x != y;
      z == void(x * y);
      solve y, z before x;
}
```

For Example 9-60, NTB-OV solves the constraints are follows:

1. `x` and `y` are solved using the constraints `y in{2,3}; x%y == 0;` and `solve y before x;`.
2. `z` is solved using the constraint `z == x*y;` Note that because `x` is already solved at this point, the constraint `solve z before x` is irrelevant and hence dropped.

Semantics for Array Constraints (Solving Array Sizes)

You can use either of the two constructs defined above to determine the partition to be solved for the size of an array. Example 9-61 illustrates the semantics.

Example 9-61   Array Constraint Semantics

```
class B {
      rand integer x[$]; rand bit[1:0] k;
      constraint b1 {
            x.size == k;
            k != 2;
            foreach(x, i) {
```

```
                void(k-1) == i => x[i] == 0;
                }
        }
}
```

In Example 9-61, NTB-OV solves for the constraints `x.size() = k; k != 2;` because the `void()` function call makes `k` unconstrained in the constraint inside the `foreach` loop. The use of `void()` in this example specifies that `k` is solved for before any of the members of the `x` array.

Example 9-62 illustrates the other important semantic.

Example 9-62   Array Constraint Implicit Ordering

```
class B {
        rand integer x[$];
        rand bit[1:0] k;

        constraint b1 {
                x.size() == void(k);
                k != 2;
                foreach(x, i) {
                        k-1 == i => x[i] == 0;
                }
        }
}
```

In Example 9-62, `k` has to be solved before `x.size()`, and `x.size()` needs to be solved before the `foreach` loop is expanded. Therefore, there is an implicit ordering between `x.size()` and `x[i]`, such that `x.size()` gets solved first. So the constraints are solved as follows:

1. `k` is solved for using the constraint `k !=2`.
2. `x.size()` is solved for using the constraint `x.size()== k`.
3. The loop is expanded and all the members of the `x` array are solved for.

The semantics of array constraints in the presence of `void` calls are as follows:

4. Implicit ordering relations are introduced between array sizes and members of the array, only for those with variable expressions in the index.
5. Array sizes are solved before the constraints inside loops are solved or constraints involving array aggregate methods calls are solved.

Semantics of Default Constraints

With partitions, it is possible for some random variables over which constraints are defined to get solved before the default constraint itself. In such cases, the solver considers previously solved random variables like state variables. When the default constraint is solved, it is disabled only if any of the currently being solved random variables incident on it are overridden. This is consistent with the semantics of default constraints, when considered in the context of the partition being solved. Consider Example 9-63.

Example 9-63   Default Constraints Semantics

```
default constraint b1 {
      x <= y;
}
constraint b1 {
      z == void(y);
      z => x == 0;
      y in {0, 1};
}
```

For Example 9-63, NTB-OV solves the constraints in the following order:

1. `y` is solved using the constraint `y in{0,1}`.

2. `z` and `x` are solved using the default constraint `x<=y` and the non-default constraints `z==y;` and `z=>x ==0;`. The default constraint applies only if `z` is 0 (that is, if and only if `y` was chosen as 0 in the previous partition.).

Semantics of randc

NTB-OV solves `randc` variables in partitions that contain a single `randc` variable, after which they behave like state variables when solving for `rand` variables (see Example 9-64).

Example 9-64 rand and randc Variables

```
randc bit[3:0]x;
rand bit[3:0] y,z;

constraint b1 {
      y in{0,1};
      x<12;
      if(y){
            x<5;
      } else {
            x<10;
      }
      z>x;
}
```

For Example 9-64, NTB-OV solves the constraints in the following order:

1. `x` is solved for using the constraint `x<12`.
2. `y` and `z` are solved for using the remaining constraints. Note that this could result in a failure, depending on the solution picked in the previous step. For example, if `x` were picked as 11 in the previous step, then there would be no solution for `y` in this step.

In summary, `randc` variables are solved before `rand` variables. However, when unidirectional constraints are used in the presence of `randc` variables, it is possible that `rand` variables may be solved before `randc` variables, as shown in Example 9-65.

Example 9-65 rand Before randc

```
randc bit[3:0]x;
rand bit[3:0] y,z;

constraint b1 {
      y in{0,1};
      x<12;
      if(void(y)){
            x<5;
      } else {
            x<10;
      }
      z>x;
}
```

In Example 9-65, NTB-OV solves the constraints as follows:

1. solves `y` using the constraint `y in{0,1}`.
2. solves `x` using the constraints `if (y) x<5 else x<10; x<12`.
3. solves `z` using the constraint `z>x`.

Static Constraint Blocks and Random Variables

You can define a static constraint block by including the `static` keyword in the definition. The syntax is:

`static constraint` *`constraint_name`* `{`*`constraint_expression`*`}`

When you declare a constraint block as `static`, calls to `constraint_mode()` affect all instances of the specified constraint in all objects. So if you set a static constraint to `OFF`, it is `OFF` for all instances.

Randomize Methods

You randomize variables in an object using the `randomize()` class method. Every class has a built-in `randomize()` virtual method. The syntax is:

```
virtual function integer randomize([integer options=0]);
```

`options`

The `options` parameter can be one or more of the macros shown in Table 9-4, all of which are defined in the vera_defines.vrh header file that NTB-OV includes automatically.

Table 9-4 randomize() options

Macro Argument	Equivalent Runtime Option and Setting
VERA_ENABLE_SOLVER_TRACE_0	+vera_enable_solver_trace=0
VERA_ENABLE_SOLVER_TRACE_1	+vera_enable_solver_trace=1
VERA_ENABLE_SOLVER_TRACE_2	+vera_enable_solver_trace=2
VERA_ENABLE_SOLVER_TRACE_ON_FAILURE_0	+vera_enable_solver_trace_on_failure=0
VERA_ENABLE_SOLVER_TRACE_ON_FAILURE_1	+vera_enable_solver_trace_on_failure=1
VERA_ENABLE_SOLVER_TRACE_ON_FAILURE_2	+vera_enable_solver_trace_on_failure=2
VERA_ENABLE_SOLVER_TRACE_ON_FAILURE_3	+vera_enable_solver_trace_on_failure=3
VERA_ENABLE_CHECKER_TRACE_0	+vera_enable_checker_trace=0
VERA_ENABLE_CHECKER_TRACE_1	+vera_enable_checker_trace=1
VERA_ENABLE_CHECKER_TRACE_2	+vera_enable_checker_trace=2
VERA_ENABLE_CHECKER_TRACE_ON_FAILURE_0	+vera_enable_checker_trace_on_failure=0
VERA_ENABLE_CHECKER_TRACE_ON_FAILURE_1	+vera_enable_checker_trace_on_failure=1

Table 9-4 randomize() options (Continued)

Macro Argument	**Equivalent Runtime Option and Setting**
`VERA_ENABLE_CHECKER_TRACE_ON_FAILURE_2`	+vera_enable_checker_trace_on_failure=2
`VERA_ENABLE_CHECKER_TRACE_ON_FAILURE_3`	+vera_enable_checker_trace_on_failure=3
`VERA_SOLVER_MODE_1`	+vera_solver_mode=1
`VERA_SOLVER_MODE_2`	+vera_solver_mode=2

If you specify incompatible options, you get compile-time or runtime errors. For example:

- Specifying both `VERA_CHECK_MODE` and `VERA_ENABLE_SOLVER_TRACE` or `VERA_ENABLE_SOLVER_TRACE_ON_FAILURE` results in an error.
- Specifying both `VERA_SOLVE_MODE` and `VERA_ENABLE_CHECKER_TRACE` or `VERA_ENABLE_CHECKER_TRACE_ON_FAILURE` results in an error.
- Specifying multiple `VERA_ENABLE_SOLVER_TRACE` or `VERA_ENABLE_SOLVER_TRACE_ON_FAILURE` modes or multiple `VERA_ENABLE_CHECKER_TRACE` or `VERA_ENABLE_CHECKER_TRACE_ON_FAILURE` results in an error.

If you don't specify `options` in the randomize call, NTB-OV uses the values specified on the command line. If no options are specified on the command line, NTB-OV uses default values. NTB-OV does not check for incompatible `options` specified on the command line. NTB-OV looks for options in the following order:

- `randomize()` per-call options
- command-line plus arguments

- proj file
- .ini file
- shell environment variable

For example, per-call options override options specified on the command line. For example:

```
ret = a.randomize(VERA_SOLVER_MODE_1 | VERA_ENABLE_SOLVER_TRACE_2);
```

If this was the only call to `randomize()` in the testbench, it would be equivalent to running the simulation with the options:

```
+vera_solver_mode=1 +vera_enable_solver_trace=2
```

Since neither the `VERA_SOLVE_MODE` nor the `VERA_CHECK_MODE` mode are specified in this example, NTB-OV uses the default value of `VERA_SOLVE_MODE`. Next, consider:

```
ret = a.randomize(VERA_ENABLE_SOLVER_TRACE_1);
```

If the options on the command line or in the .ini file are:

```
+vera_enable_solver_trace=2
+vera_solver_mode=1
```

then `VERA_ENABLE_SOLVER_TRACE_1` overrides the command-line argument `+vera_enable_solver_trace=2`, but the `+vera_solver_mode=1` is not affected.

The following examples result in compile-time errors:

```
ret = a.randomize(VERA_ENABLE_SOLVER_TRACE_1 |
VERA_ENABLE_SOLVER_TRACE_2);
```

--or--

```
ret = a.randomize(VERA_ENABLE_SOLVER_TRACE_1 | VERA_CHECK_MODE);
```

This next example results in a runtime error:

```
val = VERA_ENABLE_SOLVER_TRACE_1 | VERA_ENABLE_SOLVER_TRACE_2;
ret = a.randomize(val);
```

Here is another, more extensive example:

```
ret_val = obj.randomize(VERA_SOLVE_MODE | VERA_ENABLE_SOLVER_TRACE_0 |
VERA_ENABLE_SOLVER_TRACE_ON_FAILURE_0);
```

These options have the following effects:

1. Inconsistent constraints result in a return value of 0, and NTB-OV does not print messages in the log file. (This is a non-backwards compatible modification to the current behavior for `+vera_enable_solver_trace_on_failure=0`, which issues a single-line error message to the log file).

2. For fatal errors (null pointers, array out of bounds, and X or Z values on state variables) and solver time-outs, NTB-OV prints messages in the log file. This is the same as the default behavior.

3. If `error_mode(ON, EC_RANDOMIZEFAILURE`) is called before the call to `randomize()`, you get inconsistent constraints and solver timeouts that result in a verification failure. To prevent this, call `error_mode(OFF, EC_RANDOMIZEFAILURE)` before the call to `randomize()`. This is the same as the default behavior.

VERA_SOLVE_MODE and VERA_CHECK_MODE

In `VERA_SOLVE_MODE` mode, NTB-OV selects values satisfying all active constraints and assigns them to `rand` and `randc` variables.

In VERA_SOLVE_MODE mode, the randomize() virtual method generates random values for all active random variables in the object, subject to the active constraints. The randomize() method returns OK if it successfully sets all the random variables and objects to valid values; otherwise, it returns FAIL.

In VERA_CHECK_MODE mode, if all active constraints are satisfied given the current values of all variables (rand, randc, and non-rand), a call to randomize(VERA_CHECK_MODE) returns OK. If the constraints fail, the call returns FAIL. The values of random variables do not change as a result of a call to randomize(VERA_CHECK_MODE).

In all calls to randomize(VERA_CHECK_MODE), NTB-OV treats random variables as if rand_mode() were set to OFF. However, all calls to randomize([VERA_SOLVE_MODE]) are subject to the actual rand_mode() specification.

If you don't specify VERA_SOLVE_MODE or VERA_CHECK_MODE, the default is VERA_SOLVE_MODE.

The class definition shown in Example 9-66 declares three random variables: x, y, and z.

Example 9-66   Class Definition

```
class SimpleSum {
      rand reg[7:0] x, y, z;
      constraint c {z == x + y;}
}
```

To randomize an instance of class SimpleSum, call the randomize() method, as shown in Example 9-67:

Example 9-67   randomize (VERA_SOLVE_MODE)

```
SimpleSum p = new;
```

```
integer success = p.randomize(VERA_SOLVE_MODE);
if (success == OK ) ...
```

Assuming the class definition in Example 9-66, this next example shows how to call `randomize(VERA_CHECK_MODE)`.

Example 9-68   randomize (VERA_CHECK_MODE)

```
SimpleSum p = new;
integer success;
p.z = 5;
p.x = 3;
p.y = 2;

success = p.randomize(VERA_CHECK_MODE);
if (success == OK ) ...
```

It is good practice to verify the status of the `randomize()` call even if you know that the constraints can always be satisfied. This is because the actual values of state variables or the addition of constraints in derived classes may render seemingly simple constraints unsatisfiable.

Inline Constraints Using randomize() with

You can use the `randomize() with` construct to declare inline constraints. NTB-OV applies these additional constraints along with the object constraints. The syntax is:

```
result = class_object_name.randomize([integer mode]) with
constraint_block;
```

`class_object_name`

is the name of the instantiated object.

`mode`

can be `VERA_CHECK_MODE` or `VERA_SOLVE_MODE`. The default is `VERA_SOLVE_MODE`.

`constraint_block`

is an anonymous constraint block. The *`constraint_block`* contains the additional inline constraints to be applied along with the object constraints declared in the class.

Example 9-69 shows the same `simpleSum` example, but here `randomize()with` is used to introduce an additional constraint (`x < y`).

Example 9-69 Adding Constraint Using randomize () with

```
class SimpleSum {
      rand reg[7:0] x, y, z;
      constraint c {z == x + y;}
}

task InlineConstraintDemo(SimpleSum p){
      integer success;
      success = p.randomize() with {x < y;};
}
```

You can use the `randomize()with` construct anywhere an NTB-OV expression can appear. The constraint block following `with` can define all of the same constraint types and forms that you would otherwise declare in a class.

You can also use the `randomize()with` constraint block to reference local variables and task and function parameters, eliminating the need for mirroring a local state as member variables in the object class. The scope for variable names in a constraint block, from inner to outer, is: `randomize()with` object class, automatic and local variables, task and function parameters, class variables, and global variables. NTB-OV brings the randomize() with class into scope at the innermost nesting level. For example, see the scoping in Example 9-70, where the randomize() with class is Foo.

Example 9-70 randomize () with Scoping

```
class Foo {
      rand integer x;
}

class Bar {
      integer x;
      integer y;

      task doit(Foo foo, integer x, integer z {
            integer result;
            result = foo.randomize() with {x < y + z; };
      }
}
```

In the foo.randomize() with constraint block, x is a member of Foo (randomize-with) and hides Bar::x. It also hides the doit() task parameter x. y is a member of Bar, and z is a local parameter.

pre_randomize() and post_randomize()

Every NTB-OV class contains built-in pre_randomize() and post_randomize() tasks that randomize() calls automatically before and after it computes new random values.

The built-in definition for pre_randomize() is:

```
task pre_randomize([integer options]){
      if (super) super.pre_randomize([options]);
      /* Optional programming before randomization goes here. */
}
```

The built-in definition for post_randomize() is:

```
task post_randomize([integer options]){
```

```
        if (super) super.post_randomize([options]);
        /* Optional programming after randomization goes here. */
}
```

When you invoke `obj.randomize()`, it first invokes `pre_randomize()` on `obj` and all of its random object members that are enabled. The `pre_randomize()` task then recursively calls `super.pre_randomize()`. After NTB-OV solves or checks the constraints, `randomize()` invokes `post_randomize()` on *`obj`* and all of its random object members that are enabled. The `post_randomize()` task then recursively calls `super.post_randomize()`.

You can overload `pre_randomize()` in any class to initialize and set preconditions before the object is randomized. You can also overload `post_randomize()` in any class to clean up, print diagnostics, and check post-conditions after the object is randomized. If you overload these methods you must call their associated superclass methods; otherwise, NTB-OV skips their pre- and post-randomization processing steps.

The `pre_randomize()` and `post_randomize()` tasks also have an optional integer argument that takes the same value as the argument passed by `randomize()`. You control the actions of `pre_randomize()` and `post_randomize()` using the specified mode, as shown in Example 9-71.

Example 9-71 pre-randomize () Call

```
class C {
        rand bit x;
        bit y;
        constraint R {y => x == 1;}
        task pre_randomize(integer options) {
                if (options & VERA_CHECK_MODE)
                        return;
                y = 1;
        }
```

```
}
program P {
      C c = new();
      integer ret;
      void = c.randomize(VERA_SOLVE_MODE); // equivalent to
                                         // c.randomize();
      void = c.randomize(VERA_CHECK_MODE);
}
```

In Example 9-71:

- Random variables declared as static are shared by all instances of the class in which they are declared. Each time the `randomize()` method is called, NTB-OV changes the variable in every class instance.
- If `randomize()` fails, the constraints are infeasible and the random variables retain their previous values. In this event, NTB-OV does not call the `post_randomize()` method.
- The `randomize()` method is implemented using object random stability. To seed an object, use the `srandom()` system call, specifying the object in the second argument.
- You cannot overload the `randomize()` method.
- Don't use the built-in `pre_randomize()` and `post_randomize()` tasks for blocking operations because their automatic traversal temporarily locks objects from access. If these routines block, and another call to randomize attempts to visit the locked object, NTB-OV's behavior is not defined.

Disabling Random Variables

You can use the predefined `rand_mode()` method to control whether a random variable is active or inactive. When a random variable is inactive, NTB-OV treats it the same as if it had not been declared `rand` or `randc`. All random variables are initially active. The syntax is:

```
function integer object_name.rand_mode(
          ON | OFF | REPORT [, string variable_name [,integer index] ]);
```

`object_name`

is the name of the object in which the random variables are defined.

Predefined Macros:

Use `ON`, `OFF`, or `REPORT` to specify the action, as shown in the following table:

Table 9-5 Actions

Macro	Action
`ON`	Sets the specified variables to active so that they are randomized on subsequent calls to `randomize()`.
`OFF`	Sets the specified variables to inactive so that they are not randomized on subsequent calls to `randomize()`.
`REPORT`	Returns the current `ON`/`OFF` value for the specified variable.

`variable_name`

is a string with the name of the variable to be made active or inactive. The *variable_name* can be the name of any variable in the class hierarchy. If no variable_name is specified, NTB-OV applies the action to all variables within the specified object.

index

is an optional array index. If the variable is an array, omitting the index results in all the elements of the array being affected by the call. NTB-OV treats an index value of -1 the same as index omission.

The rand_mode() method returns the new mode value (either OFF, which is 0, or ON, which is 1) if the change is successful. If the specified variable does not exist within the class hierarchy, the method returns a -1. If the specified variable exists but is not declared as rand or randc, the function returns a -1. If the variable is an array, you must specify the array name and index for a REPORT.

If the variable is an object, only the mode of the variable is changed. If the variable is an array, then the mode of each array element is changed.

Example 9-72 first disables all random variables in packet, and then enables the *source_value* variable.

Example 9-72 Disabling Random Values

```
class Packet {
      rand integer source_value, dest_value;
      ... other declarations
}

Packet packet_a = new;
integer ret;
// Turn all variables off.
ret = packet_a.rand_mode (OFF);
// ... other code
// Enable source_value.
ret = packet_a.rand_mode (ON, "source_value");
```

The `rand_mode()` specification can only apply to random variables not defined as local or protected. If you use this method to make a local or protected variable active or inactive, NTB-OV issues a fatal runtime error.

If you don't specify any of the optional arguments with `rand_mode()`, NTB-OV turns on (off) all random variables globally. If an object contains a local or protected random variable, NTB-OV issues the following one-time warning message:

```
runtime: caller context does not have access permission to all variables.
Warning will not be issued again.
```

In this situation, `rand_mode()` does not apply to local or protected random variables, and the simulation continues.

Disabling Constraints

NTB-OV provides the predefined `constraint_mode()` method to control whether a constraint is active or inactive. All constraints are initially active. The syntax is:

```
function integer object_name.constraint_mode(
     ON | OFF | REPORT [, string constraint_name]);
```

`object_name`

is the name of the object in which the constraint block is defined.

Predefined Macros:

Use `ON`, `OFF`, or `REPORT` to specify the action, as shown in the following table:

Table 9-6 Actions

Macro	Action
`ON`	Sets the specified variables to active so that they are randomized on subsequent calls to `randomize()`.
`OFF`	Sets the specified variables to inactive so that they are not randomized on subsequent calls to `randomize()`.
`REPORT`	Returns the current `ON`/`OFF` value for the specified variable.

`constraint_name`

is the name of the constraint block to be made active or inactive. The `constraint_name` can be the name of any constraint block in the class hierarchy. If no constraint name is specified, NTB-OV applies the switch to all constraints within the specified object.

The `constraint_mode()` method returns the value of switch (either `OFF` or `ON`) if the change is successful. If the specified constraint block does not exist within the class hierarchy, the method returns a -1. You must specify the `constraint_name` for a `REPORT`.

Example 9-73 first makes constraint `filter1` inactive (`OFF`) and returns `OFF` to the variable `ret`. Then it makes constraint `filter1` active (`ON`) and returns `ON` to the variable `ret`.

Example 9-73 Making Constraint Mode OFF

```
class Packet {
      rand integer source_value;
      constraint filter1 {
            source_value > 2 * m;
      }
```

```
    }

    Packet packet_a = new;
    integer ret = packet_a.constraint_mode (OFF, "filter1");
    // ... other code
    ret = packet_a.constraint_mode (ON, "filter1");
```

You can only apply the `constraint_mode()` specification to random variables not defined as local or protected. If you use this method to make a constraint active or inactive, NTB-OV issues a fatal runtime error.

If you don't specify any optional arguments when calling constraint_mode(), NTB-OV turns all constraint blocks on (off) globally. If an object contains a local or protected constraint, NTB-OV issues a one-time warning message:

```
runtime: constraint_mode caller does not have access permission to all
constraint blocks. Warning will not be issued again.
```

In this case, the `constraint_mode()` specification does not apply to local or protected random variables, and the simulation continues. Example 9-73 shows the `Packet` class definition used to build the `DerivedPacket` class shown in Example 9-74.

Example 9-74   DerivedPacket Definition

```
class DerivedPacket extends Packet {
    local constraint filter1 {
        source_value < 2 * m;
     }
}
program test {
    DerivedPacket d = new;
    Packet packet_a = d;
    integer ret = packet_a.constraint_mode(OFF , "filter1");
}
```

The `constraint_mode()` method is virtual. Both the base class `Packet` and the derived class `DerivedPacket` have a constraint named `filter1`. When the constraint mode of `filter1` is switched off using the `packet_a` handle that points to an instance of `DerivedPacket`, `constraint_mode()` returns an error because it does not have access permission to `filter1` in `DerivedPacket` because `filter1` in the derived class is local.

Dynamic Constraint Modification

There are several ways to dynamically modify constraints on randomization:

- Use implication and if-else style constraints to declare predicated constraints.
- Make a constraint block active or inactive using the `constraint_mode()` function. Initially, all constraint blocks are active. Inactive constraints are ignored by the `randomize()` function
- Make random variables active or inactive using the `rand_mode()` function. Initially, all `rand` and `randc` variables are active. Inactive variables are ignored by the `randomize()` function.
- Change the weights in a `dist` constraint, thus affecting the probability that particular values in the set are chosen.

Efficient Use of Constraints

It is important to keep the number of random bits small for any given randomization problem because memory use and CPU time grow quickly for large problems. To this end it is good practice to:

- Declare exact-size bit vectors (regs) instead of integers, which are internally represented as 32-bit signed values.

- Use enumerated types instead of a list of integers or bit-vectors for cases where you want to represent constant values. This allows the compiler to infer the sizes of identifiers automatically. This also allows for writing more maintainable code.

For example, Example 9-75:

Example 9-75 Randomization with Integer

```
rand integer x;
integer pkt_type_one = 0;
integer pkt_type_two = 1;
constraint C {
x dist {pkt_type_one :/ 1, pkt_type_two :/2};
}
```

can be rewritten as shown in Example 9-76:

Example 9-76 Randomization with Enumerated Type

```
enum pkt_type_enum{PKT_TYPE_ONE=0,PKT_TYPE_TWO = 1};
rand integer x
constraint C {
     x dist {PKT_TYPE_ONE :/ 1, PKT_TYPE_TWO :/ 2};
}
```

Use exact-size constants in constraints; for example, `8'h00` instead of `0`. Using decimal constants causes non-integer values to be promoted to integer values, which consume more time and memory.

10

Random Sequence Generation

OpenVera Native Testbench (NTB-OV) has a random sequence generation feature that allows you to specify the syntax of valid sequences using BNF-like notation. Random sequences are a neat way to generate streams of instructions because it's easier to specify the syntax of valid streams than the constraints on specific values. This chapter explains how to use random sequence generation in the following major sections:

- VSG Overview
- Production Declaration
- Production Controls
- Passing Values Between Productions

VSG Overview

The syntax for programming languages is often expressed in Backus Naur Form (BNF) or a similar derivative. Parsers use BNF to define the language to be parsed. However, it is possible to reverse the process. Instead of using BNF to check that existing code fits the correct syntax, BNF can be used to assemble code fragments into syntactically correct code. The result is the generation of pseudorandom sequences of text, ranging from sequences of characters to syntactically and semantically correct assembly language programs. This is essentially how NTB-OV's stream generator, (called VSG) works. VSG is defined by a set of rules and productions encapsulated in a `randseq` block. The general syntax to define a VSG code block is:

```
randseq (production_name) {
      production_definition1;
      production_definition2;
      ...
      production_definitionN;
}
```

When a `randseq` block is executed, NTB-OV selects random production definitions and puts them together to generate a random stream. How these definitions are generated is determined by the base elements included in the block.

Any VSG code block is comprised of production definitions. NTB-OV also provides weights, production controls, and production system functions to enhance production usage. Each of these VSG components is discussed in detail in subsequent sections.

Production Declaration

A language is defined in BNF by a set of production definitions. The syntax to define a production is:

```
production_name : production_list;
```

`production_name`

is the reference name of the production definition.

`production_list`

is one or more production items.

Production items are made of terminals and non-terminals. A terminal is an indivisible code element. It needs no further definition beyond the code block associated with it. Code blocks should be encapsulated in braces ({ }). A non-terminal is an intermediate variable defined in terms of other terminals and non-terminals.

If a production item is defined using non-terminals, those non-terminals must then be defined in terms of other non-terminals and terminals using the production definition construct. Ultimately, every non-terminal has to be broken down into its base terminal elements.

Multiple production items specified in a production list can be separated by white space or by the **`or`** operator (|). Production items separated by white space indicate that the items are streamed together in sequence. Production items separated by the | operator force a random choice, which is made every time the production is called.

Example 10-1 illustrates the use of production items.

Example 10-1 Production Items

```
main  : top middle bottom;
top :  add | dec;
middle : popf | pushf;
bottom : mov;
```

In this example, `main` is defined in terms of three non-terminals: `top`, `middle`, and `bottom`. When a call is made to this random sequence, NTB-OV evaluates `top`, `middle`, and `bottom` and streams their definitions together.

The `top`, `middle`, and `bottom` production definitions are defined using terminals. The `|` operator forces a choice between the `add` and `dec` terminals and between the `popf` and `pushf` terminals. This sequence block leads to these possible outcomes:

```
add popf mov
add pushf mov
dec popf mov
dec pushf mov
```

Example 10-2 shows a full set of production definitions.

Example 10-2 Production Definitions

```
assembly_program: text_section data_section ;
text_section: {printf(".text");} my_code ;
data_section: {printf(".data");} data ;
data : initialized_data | uninitialized_data ;
my_code: { /* NTB-OV code */}
initialized_data: { /* NTB-OV code */ }
uninitialized_data: { /* NTB-OV code */}
```

Example 10-2 defines the production `assembly_program` in terms of `text_section` and `data_section`. The production `text_section` is then broken down to include the string `.text` and the non-terminal `my_code`. The production `data_section` is defined in terms of either the `initialized_data` and `uninitialized_data` terminals, the selection of which occurs when the `randseq` block is called. The resulting output is:

```
.text
my_code output
.data
initialized_data output OR uninitialized_data output
```

The `my_code`, `initialized_data`, and `uninitialized_data` outputs are determined by the code blocks associated with those productions.

Production Controls

NTB-OV provides several mechanisms you can use to control productions: weights for randomization, if-else statements, case statements, and repeat loops, as explained in the following subsections:

- Weights for Randomization
- if-else Statements
- case Statements
- repeat Loops
- break Statement
- continue Statement

Weights for Randomization

You can assign weights to production items to change the probability that they are selected when the `randseq` block is called. The syntax is:

```
production_name : weight production_item;
```

`weight`

> must be in the form `&(expression)` where `expression` can be any valid NTB-OV expression that returns a non-negative integer. You can make function calls within the `expression`, but the `expression` must return a numeric value; if it doesn't you get a simulation error.

Assigning weights to a production item affects the probability that it is selected when the `randseq` block is called. You should only assign weight when a selection is forced with the `|` operator. NTB-OV evaluates the weight for each production item when it executes its production definition. This allows you to change the weights dynamically using a sequence of calls to the same production.

Example 10-3 defines the `integer_instruction` production in terms of the `add_instruction` and `sub_instruction` weighted production items.

Example 10-3 Weighted Production Definition

```
integer_instruction: &(3) add_instruction|&(i*2) sub_instruction;
```

If `integer_instruction` is 1 when the definition is executed, there is a 60% (3/5) chance that `add_instruction` is selected, and a 40% (2/5) chance that `sub_instruction` is selected.

if-else Statements

You can conditionally reference a production using an `if-else` statement. The syntax is:

```
production_name :<if (condition) production_name else
      production_name>;
```

`condition`

can be any valid NTB-OV expression

If the `condition` evaluates to true, NTB-OV selects the first production item; else it selects the second production item. You can legally omit the `else` statement; if so NTB-OV ignores the entire `if` statement when `condition` evaluates to false.

Example 10-4 defines the `assembly_block` production.

Example 10-4 Production Definition With if-else Statement

```
assembly_block : <if (nestingLevel > 10) seq_block else any_block>;
```

In Example 10-4, if the `nestingLevel` variable is greater than `10`, NTB-OV selects the `seq_block` production item. If `nestingLevel` is less than or equal to `10`, NTB-OV executes `any_block`.

case Statements

The NTB-OV case statement provides a general selection mechanism. The syntax to declare a case statement within a production definition is:

```
production_name : <case(primary_expression)
case1_expression : production_name
case2_expression : production_name
...
```

```
caseN_expression : production_name
[default : production_name > ;]
```

`primary_expression`

NTB-OV successively checks the value of *primary_expression* against each *case_expression*. When it finds an exact match, it executes the production corresponding to the matching `case` and passes control to the production definition for the matching case item. If other matches exist, they are not executed. If no case item value matches the evaluated primary expression and there is no default case, nothing happens.

case_expression

can be any valid NTB-OV expression or comma-separated list of expressions. Expressions separated by commas allow multiple expressions to share the same statement block

A `case` statement must have at least one case item aside from the default case, which is optional. The default case must be the last item in a `case` statement. Example 10-5 shows an example of a production definition using a `case` statement:

Example 10-5 Case Statement

```
assembly_block : <case(i*3)
      0 : seq_block
      3: loop_block
      default : any_block> ;
```

Example 10-5 defines the `assembly_block` production with a `case` statement. NTB-OV evaluates the `i*3` `primary_expression` and checks it against the `case` expressions before executing the production item for the matching case.

repeat Loops

Use the `repeat` loop to loop over a production a specified number of times. The syntax to declare a `repeat` loop within a production definition is:

```
production_name : <repeat (expression) production_name>;
```

`expression`

can be any valid NTB-OV expression that evaluates to a non-negative integer, including functions that return a numeric value.

NTB-OV evaluates the `expression` when it executes the production definition. The value derived from the `expression` specifies how many times NTB-OV executes the corresponding production item. Example 10-6 defines the `seq_block` production, which repeats the *`integer_instruction`* production item a random number of times, depending on the value returned by the `random()` system function.

Example 10-6 Production Definition using a Repeat Loop

```
seq_block : <repeat (random() ) integer_instruction>;
```

break Statement

Use the `break` statement to terminate a `randseq` block. The syntax is:

```
break;
```

You can put a `break` statement in a code block for a production definition. When NTB-OV executes a `break` statement, the `randseq` block terminates immediately and control is passed to the first line of code after the `randseq` block. For Example 10-7, NTB-OV executes the break if the conditional expression is satisfied. When the `break` is executed, control passes to the first line of code after the `randseq` block.

Example 10-7 Production Definition using a Break Statement

```
SETUP_COUNTER:
{
      integer regis = regFile.getRegister();
      integer value = ADDI;
      nestingLevel++;
      if (nestingLevel == MAX_NESTING) break;
      ...
} ;
```

continue Statement

Use the `continue` statement to interrupt execution of the current production and continue on the next item in the production. The syntax is:

```
continue;
```

Example 10-8 shows a production definition that uses a `continue` statement:

Example 10-8 Production Definition using a continue Statement

```
LOOP_BLOCK: SETUP_COUNTER GEN_LABEL;
SETUP_COUNTER:
{
      integer regis = regFile.getRegister();
      integer value = ADDI;
      nestingLevel++;
      if (nestingLevel == MAX_NESTING) continue;
      ...
} ;
```

For Example 10-8, NTB-OV first executes the `SETUP_COUNTER` production definition. When the `continue` is executed, NTB-OV ignores the code after the `continue` and control passes to the `GEN_LABEL` production definition.

Passing Values Between Productions

You can pass values within `randseq` blocks to associate a data type between production definitions. There are two components of value passing within `randseq` blocks: the value declaration and value passing functions.

Value Declaration

To associate a data type and value with a non-terminal production, declare the production using the `prod` declaration. The syntax is:

`prod` *`data_type production_name`*`;`

`data_type`

can be integer, reg, string, enum, or any object.

`production_name`

is the name of the production passing the value.

You can declare multiple productions in a single declaration statement. NTB-OV performs strict type checking on values you pass.

Value Passing Functions

NTB-OV provides two functions you can use to pass values in `randseq` blocks: prodset () and prodget (). You call these system functions from NTB-OV code blocks within production definitions.

prodset ()

Use the `prodset()` system task to set a value associated with a non-terminal. The syntax is:

```
task prodset (reg|integer|string|enum|obj value [, string
production_name [, integer occurrence_number] ]);
```

`value`

is the value you want to pass. It must be the same type as the `prod` declaration. It can be an integer, reg, string, enumerated type, or object.

`production_name`

optionally specifies the name of the non-terminal production you are assigning the value to. If `production_name` is omitted, NTB-OV uses the production to the left of the colon (`:`).

`occurrence_number`

optionally specifies which occurrence of the same `production_name` receives the value. If the same production is referred to multiple times in the same definition, the first is 1, and the others are numbered sequentially. If `occurrence_number` is omitted, NTB-OV uses 1.

prodget ()

Use the `prodget()` system function to retrieve values assigned to non-terminal productions. The syntax is:

```
function reg|integer|string|enum|obj prodget([string
production_name [, integer occurrence_number] ]);
```

`production_name`

> specifies the name of the non-terminal production the value was assigned to. If it is omitted, NTB-OV assumes that the production the task is to the left of the colon (`:`).

`occurrence_number`

> optionally specifies which occurrence of the same production name receives the value. If the same production is referred to multiple times in the same definition, the first is 1, and the others are numbered sequentially. If it is omitted, it is assumed to be 1.

The `prodget()` system function returns the value assigned to the specified non-terminal production.

Production values can only be associated with production names to the left of the code block where `prodset()` is called. If you assign a value to the production name to the left of the colon (`:`), NTB-OV passes the value up to the calling production, where it can be retrieved. Calling `prodget()` on a production entry that has not been assigned returns an undefined value. You cannot set a value for a production that has not yet been executed, as shown in Example 10-9.

Example 10-9   proget() Valid and Invalid Calls

```
main : prod_a {prodset(5,prod_b,1); } prod_b /* invalid! */
main : prod_a {prodset(prod_a,2); } prod_b prod_a /* invalid! */
main : prod_a {integer tmp = prodget(prod_a);
```

```
prodset(tmp+5,prod_a);} prod_b   /* valid! */
```

Example 10-10 defines the list object `vsgList`.

Example 10-10   Passing Values Within a randseq Block

```
program GenList {
      list vsgList;
      randseq(){
            prod list_node ELEMENT;

            TOP :{ vsgList = new(); } LIST END;
            LIST : &(10) LIST ELEMENT {
                        list_node lnode =  prodget ( ELEMENT, 1 );
                        vsgList.insert ( lnode );
                        printf ("last insert %0d\n", lnode.data);
                        ELEMENT {
                        list_node lnode =  prodget ( ELEMENT, 1 );
                        vsgList.insert ( lnode );
                  printf ("inserting %0d \n", lnode.data);
                  }
                  ;
                  ELEMENT : {
                              list_node lnode = new();
                              prodset ( lnode );
                              printf ("new node is created  %0d\n",
                                    lnode.data);
                        }
                        ;
                  END : { vsgList.printAll(); }
                        ;
                  }
            }
```

In Example 10-10, when NTB-OV enters the `randseq` block, it executes the `TOP` production. First, a list object is instantiated. Then the `LIST` production is executed. The `LIST` production consists of a weighted `LIST ELEMENT` production and an `ELEMENT` production. If the `LIST ELEMENT` production is selected, the `LIST` production is called recursively and the `ELEMENT` production is postponed. The original selection between `ELEMENT` and `LIST ELEMENT` is then made again. The cycle continues until the `ELEMENT` production is selected. At that time, `lnode` is assigned a value via the prodset call in the `ELEMENT` production. That value is inserted into `vsgList` via the `insert()` call in the code block. Finally, the previously unexecuted `ELEMENT` calls that had been postponed when `LIST ELEMENT` was selected are executed. Control then passes back to the `TOP` production, which executes the `END` production.

Note that NTB-OV does not assign each production a single value. Instead, it assigns a stack of values to a production, which it retrieves in order through the dynamic execution of the production set.

11

Functional Coverage Groups

This chapter explains the syntax and language constructs you use to define coverage models in OpenVera Native Testbench (NTB-OV), and explains how to use coverage models for functional coverage group analysis. For an introduction to testbench functional coverage analysis and a discussion on how to use it, see the *VCS OpenVera Native Testbench User Guide.*

Coverage Overview

NTB-OV supports a functional coverage group system you can use to monitor all states and state transitions, as well as changes to variables and expressions. NTB-OV tracks simulation activity by setting up a number of monitor bins that correspond to states, transitions, and expression changes.

Each time a user-specified sampling event occurs, NTB-OV increments a counter associated with the bin. By establishing a bin for each state, state transition, and variable change that you want monitored, you can check the bin counter after the simulation to see how many activities occurred. This way, you can check the degree of completeness of the testbench and simulation.

Coverage Group

The `coverage_group` construct specifies the coverage model. You define it either at the top level (referred to as stand-alone) or in a class (referred to as embedded). The syntax for declaring a `coverage_group` is the same for both:

```
coverage_group definition_name [(argument_list)] {
      sample_event_definition;
      coverage_point_definition[s]
      [cross_definitions]
      [attribute_definitions]
}
```

`definition_name`

is the name of the `coverage_group`.

`argument_list`

are arguments passed at instantiation. They have the same conventions as subroutine arguments and can have their default values set within the declaration (for more information about passing arguments to coverage groups, see “Passing Arguments to Coverage Groups” on page 491).

`sample_event_definition`

defines when the coverage objects are sampled.

Coverage objects can be sampled on clock or signal edges. When the specified edge occurs, the object is sampled. The syntax is:

```
sample_event = @([specified_edge] interface_signal | CLOCK);
```

specified_edge

posedge | negedge (no edge detects any signal change.

interface_signal

is *interface.signalName*

You can sample coverage objects when variables change value using the `wait_var()` system task. When the value of a variable changes, NTB-OV samples the object. The syntax is:

```
sample_event = wait_var(NTB-OV_variable);
```

You can sample coverage objects on NTB-OV sync events. When the sync event is unblocked, the object is sampled. The syntax is:

```
sample_event = sync(ALL | ANY, some_OpenVera_event);
```

You can also sample coverage objects on assertion events. When the assertion event occurs in the design, the `Wait()` member task on the assertion object unblocks (for more information, see “Wait ()” on page 325). Example 11-1 shows an example `Wait()` call.

Example 11-1 Wait () Call

```
sample_event = ova_object.Wait();
```

where `ova_object` is an instance of an AssertEvent (see “AssertEvent Class” on page 324).

In the most general case, the sampling event expression can be any task call (see "General Task Calls" on page 502). You can use this when you need more elaborate event sequences to trigger sampling of coverage objects. For more information on `sample_event`, see "Sample Event Definitions" on page 499.

`coverage_point_definitions`

`cross_definitions` define the coverage points sampled by the coverage group, which includes the `cross_definition`. You declare them using the `sample` construct of a coverage group. You must include at least one `coverage_point_definition` in the coverage group. The simplest form is:

```
sample coverage_point [, coverage_point] [conditional];
```

In this case, NTB-OV automatically generates bins (see Example 11-2).

Example 11-2 Coverage Group

```
class MyClass {
      bit [0:7] m_x, m_y;

      coverage_group cov1 {
            sample_event = @(posedge CLOCK);
            sample m_x, m_y;
      }
}
```

When specifying state and transition bin definitions for a *coverage_point*, the `sample` construct has the following form:

```
sample coverage_point [, coverage_point] {
      [state_bin_definitions]
      [transition_bin_definitions]
      [attribute_definitions]
}
```

coverage_point

is either a `coverage_point_expression` or a `coverage_point_variable`.

Table 11-1 Coverage Points

coverage_point	**Description**	**Syntax**
`coverage_point_expression`	The NTV-OV expression to sample.	`coverage_point_name (expression)` where `sample_name` is a name associated with the NTB-OV `expression`
`coverage_point_variable`	A variable visible in the scope of the coverage group (Note: a string is not a legal coverage point.)	

`state_bin_definitions`

associate named bins with ranges of values of sampled coverage points. The syntax is:

```
[wildcard] state_construct state_bin_name (state_specification_list) \
[conditional];
transition_bin_definitions
```

Table 11-2 State Bin Definitions

state_bin_definitions	**Description**
`wildcard`	Treats X value as a wildcard in state declarations (see "Wildcard State and Transition Definitions" on page 515).
`state_construct`	Can be: `state`, `m_state` (see "m_state" on page 520), `bad_state`, or `m_bad_state` (see "Bad States and Transitions" on page 516).

Table 11-2 State Bin Definitions

state_bin_definitions	Description
`state_bin_name`	The name of the bin.
`state_specification_list`	A list of value ranges (separated by commas) that NTB-OV matches against the current value of the `coverage_point`.
`conditional`	Must be of the form `if(NTB-OV_expression)`.

`transition_bins`

record the transition between values. The syntax is:

```
[wildcard] trans_construct trans_bin_name
(trans_bin_sequence_list) [conditional];
```

Table 11-3 Transition Bin Definitions

transition_bin_definitions	Description
`wildcard`	Treats X values as wildcards in transition declarations (see "Wildcard State and Transition Definitions" on page 515).
`trans_construct`	Can be `state`, `m_trans`, `bad_trans`, `m_bad_trans` (see "Defining Transition Sequences" on page 527).
`trans_bin_name`	The name of the transition.
`trans_bin_sequence_list`	A comma-separated list of transition sequences (see "Defining Transition Sequences" on page 527).
`conditional`	Must be of the form `if(NTB-OV_expression)`.

`attribute_definitions`

Attributes are directives you use to control various aspects of the coverage model. The syntax is:

```
attribute_name = value_expression;
```

For a full list of attributes available at the sample level, see Table 11-14.

cross_definitions

You can specify crosses of coverage points for a coverage group using the `cross` construct. In its simplest form, the `cross` construct has the following syntax:

```
cross cross_name (coverage_point_name, coverage_point_name
[,coverage_point_name])
```

Table 11-4 Cross Definitions

cross_definitions	Description
`cross_name`	The name of the cross.
`coverage_point_name`	The name of a `coverage_point_label` or a `coverage_point_variable`.
`attribute_definitions`	Directives that control various aspects of cross coverage.

In the above case, NTB-OV automatically generates cross bins.

When defining bins, use this `cross` construct syntax:

```
cross cross_name (coverage_point, coverage_point,[,coverage_point_name]) {
[cross_bin_definitions]
[attribute_definitions]
}
```

cross_bin_definition

Defines a cross coverage bin that combines several cross products into single named bin or equivalence class. Use the following syntax:

```
state state_name((select_expression)[logical_operator (select_expression)]) [
if (expression)];
```

Table 11-5 Cross Bin Definitions

cross_bin_definitions	Description
state	can be either state, ignored or bad_state
state_name	user-specified name of the cross bin
select_expression	Subset of bins_expression
logical_operator	&& or ||

`select_expression`

Subset of `bins_expression`. The syntax is:

```
([!]binsof(bins_expression) [intersect {value_range_list}])
```

Table 11-6 Select Expressions

select_expression	Description
bins_expression	Expression that identifies the bins to be specified for a cover point identifier.
value_range_list	List of value ranges

`bins_expression`

The expression that identifies the bins to be specified for a cover point identifier. The syntax is:

```
cover_point_identifier [.bins_identifier]
```

Table 11-7 Bins Expressions

bins_expression	Description
cover_point_identifier	The name of the cover point.
bins_identifier	The name of the bin

`attribute_definition`

Attributes are directives that control various aspects of the coverage model. The syntax is:

```
attribute_name = value_expression;
```

"Attribute Definitions" on page 552 details the attributes that you can specify at the coverage group level, and their default values.

Embedded Coverage Group

When you define a `coverage_group` construct in a class it is referred to as an embedded group. The syntax for defining an embedded coverage group is:

```
class class_name {// class properties and coverage
      coverage_group definition_name [(argument_list)] {
            sample_definitions;  [cross_definitions;]
            sample_event_definition; [attribute_definitions;]
      }
      // constraints and methods
}
```

A class can have more than one embedded coverage group, as shown in Example 11-3.

Example 11-3   Embedded Coverage Groups

```
class MyClass {
      reg [3:0] m_x;
      local reg m_z;
      coverage_group cov1 {
            sample_event = @(posedge CLOCK);
            sample m_x {
                  state s0(0:7) if (m_z > 0);
                  state s1(8:15);
            }
      }
      coverage_group cov2{
            sample_event = wait_var(m_z);
            sample m_z;
      }
```

```
}
```

Instantiating a Coverage Group

To collect coverage data, you instantiate the declared coverage group. Similar to classes in NTB-OV, coverage groups serve as templates you can instantiate wherever you need them. You can define the construct as a top-level construct (file scope) or contain it inside a class (for syntax information, see "Embedded Coverage Group" on page 485).

Stand-alone Coverage Group

Once you define it, you must explicitly instantiate a stand-alone coverage group using the `new()` system call for coverage to be triggered, as shown in Example 11-4.

Example 11-4 Instantiated Stand-alone Coverage Group

```
interface ifc {
      input clk CLOCK;
      input sig1 PSAMPLE #-1;
}

coverage_group CovGroup {
      sample_event = @ (posedge CLOCK);
      sample var1, ifc.sig1;
      sample s_exp(var1 + var2);
}

program covTest {
      integer var1, var2;
      CovGroup cg = new(); //explicit instantiation
}
```

In Example 11-4, `CovGroup` defines the coverage model for the global variable `var1`, the signal `ifc.sig1`, and the expression composed of global variables `var1` and `var2`. Here, `s_exp` is the name used to refer to the sampled expression. NTB-OV samples the two variables and the expression on every positive edge of the system clock.

Instantiating Embedded Coverage Groups

You can instantiate an embedded coverage group either explicitly or implicitly.

Explicit Instantiation

To explicitly instantiate an embedded coverage group, use the new assignment statement within the body of the `new()` task for the embedding class. If you need to pass arguments into an embedded coverage group, you must explicitly instantiate it in the `new()` task of the embedding class. NTB-OV issues an error when it sees an embedded coverage group which needs arguments and is not explicitly instantiated. The syntax for explicitly instantiating an embedded coverage group is:

```
coverage_group_name = new ([arguments]);
```

`coverage_group_name`

is the name of the embedded coverage group.

`arguments`

are passed to the coverage group.

Example 11-5 instantiates an embedded coverage group named `MyCov` of class `MyClass`. Here, a passed-in argument is used for setting the `at_least` attribute of the coverage group.

Example 11-5 Instantiated Embedded Coverage Group

```
class MyClass {
```

```
        bit [7:0] m_x;
        coverage_group MyCov (integer param1){
              sample_event = @(posedge CLOCK);
              at_least = param1;
              sample m_x;
        }

        task new(integer p1) {
              m_x = 0;
              // Instantiate embedded coverage group MyCov
              MyCov = new(p1);
        }
  }

  program simple {
        MyClass obj = new(4);
        @(posedge CLOCK);
        // ...
  }
```

You can also disable an embedded coverage group (with or without arguments) using the `null` assignment statement. When you disable an embedded coverage group, collection and reporting of coverage data for the coverage group is turned off. The syntax for disabling an embedded coverage group is:

```
coverage_group_name = null;
```

`coverage_group_name`

is the name of the embedded `coverage_group`.

Note that you can only disable an embedded coverage group using the above syntax from within the body of the `new()` task of the enclosing class. You can instantiate or disable an embedded coverage group from inside nested blocks (like an if-then-else construct), so different instances of the class may have the embedded coverage groups enabled or disabled, as shown in Example 11-6.

Example 11-6 Enabled and Disabled Embedded Coverage Groups

```
class A {
      integer x;
      coverage_group cov1 {
            sample x;
            sample_event = @(posedge CLOCK);
      }
      task new(integer flag){
            x = 0;
            // Coverage group can be instantiated
            // inside a conditional block
            if (flag) {
                  // Instantiation using new, cov1 will
                  // be instantiated.
                  cov1 = new;
            } else {
                  // Instantiation using null, cov1 will
                  // not be active (not instantiated).
                  //must be inside the constructor, i.e.
                  // task new()
                  cov1 = null;
            }
      }
}

program test {
      A obj1;
      A obj2;
      // Embedded cov_grp cov1 will be active for
      // this instance of A.
      obj1 = new(1);
      // Embedded cov_grp cov1 will not be active
      // for this instance of A
      obj2 = new(0);
      ...
}
```

Implicit Instantiation

Implicit instantiation of embedded coverage groups occurs when:

- There is no user-defined `new()` task for the class, or the embedded coverage group is not initialized (disabled or instantiated) in the user-defined `new()` task for the class.
- The embedded coverage group requires no arguments.

Implicit instantiation means that NTB-OV automatically instantiates the embedded coverage group as the first statement in the `new()` task for the enclosing class. Be careful when using implicit instantiation; there can be cases where implicit instantiation leads to runtime errors. Example 11-7 shows an embedded coverage group that does not have any passed-in parameters, yet requires explicit instantiation in the `new()` task of the embedding class.

Example 11-7 Embedded Coverage Group Needing Explicit Instantiation

```
class Helper {
      event m_ev;
      // ...
}

class MyClass {
      Helper m_obj;
      integer m_a;
      coverage_group Cov {
            sample m_a;
            sample_event = sync(ALL, m_obj.m_ev);
      }

      task new(){
            m_obj = new;
            /* Instantiate embedded coverage_group here, after
             instantiating m_obj */
            Cov = new;
      }
}
```

In Example 11-7, `Cov` is embedded under `MyClass`, which also declares an instance of the `Helper` class, called `m_obj`. The `sample_event` for the `Cov` embedded coverage group refers to data member `m_ev` of `m_obj`. Therefore, the embedded coverage group should not try to access `m_obj.m_ev` until `m_obj` is instantiated. Otherwise, a null object access occurs at runtime. To avoid this, the embedded coverage group `Cov` is instantiated after instantiating `m_obj` in the task `new()`.

Passing Arguments to Coverage Groups

What if the coverage model cuts across a class abstraction and all elements in the coverage model don't reside in the same class? For example, you may want to cross variables from different classes or cross a data member from one instance with a data member from another instance of the same class.

You can pass arguments to a coverage group to address this need. The coverage group construct optionally allows for the declaration of formal parameters. You pass actual arguments in to the coverage group instance as parameters to the `new()` task call. You can define three kinds of arguments in the coverage group definition:

- Sampled Arguments
- Non-Sampled Arguments Passed by Value
- Non-Sampled Arguments Passed by Reference

Sampled Arguments

Precede sampled arguments with the `sample` keyword in the argument list of the coverage group definition. NTB-OV handles a sampled argument like a constant `var` argument passed to a task.

In Example 11-8, `MyCov` defines a sampled argument `paramVar` that NTB-OV samples at every positive edge of the system clock.

Example 11-8   Sampled Arguments in Coverage Group

```
coverage_group MyCov(sample bit[3:0] paramVar) {
      sample_event = @(posedge CLOCK); // Sample event
      sample paramVar;    // Passed-in sample argument
}
...
program Example {
      bit [3:0] gVar;
      MyCov cov1;
      ...
      cov1 = new(gVar);
      ...
}
```

In Example 11-8, variable `gVar` is passed as an argument to instance `cov1` of the coverage group when the `new()` task is called. The coverage group instance therefore samples variable `gVar` at every positive edge of the system clock.

Non-Sampled Arguments Passed by Value

NTB-OV considers an argument that is not preceded by a keyword in the formal argument list of a coverage group definition to be passed-by-value (having a constant value). When a pass-by-value argument is passed to a coverage group instance, the value of the passed argument at the time the `new()` task is called is used by the coverage group instance.

You can use such arguments in state and transition bin specifications to define ranges of values associated with a bin, or in state and transition bin conditionals. You can also use them to define states associated with the cross of coverage points (for example, illegal and ignored bin specifications of crosses).

In Example 11-9, `MyCov` defines two arguments that are passed by value (`param` and `param2`). It also samples a global variable (`gVar`) at the positive edge of the system clock.

Example 11-9   Passed by Value Arguments in Coverage Group

```
coverage_group MyCov(integer param, integer param2)
//no sample keyword, so passed by value {
      sample_event = @(posedge CLOCK);
      sample gVar
      {
            state s1 (0:param);
            state s2 (param+1 : 15);
            state condState (16 : 31) if (param2 > 4);
      }
}
...
program Example {
      bit [4:0] gVar;
      integer gP1, gP2;
      MyCov cov1;
      ...
      gP1 = 7;
      gP2 = 3;
      cov1 = new(gP1, gP2);
      ...
}
```

In Example 11-9, instance `cov1` of `MyCov` is instantiated with actual arguments `gP1` (parameter `param`) and `gP2` (parameter `param2`) having the values 7 and 3, respectively when `cov1` is instantiated. The values of `gP1` and `gP2` when `cov1` is instantiated imply the following:

- State `s1` is associated with 0 <= gVar1 <= 7.
- State `s2` is associated with 8 <= gVar1 <= 15.

In Example 11-9, `state condState` is not hit because the value of the second parameter of the coverage group instance `cov1` (argument `gP2`) is 3 at the time of instantiation, causing the conditional to evaluate to false at every sample event.

Non-Sampled Arguments Passed by Reference

Parameters preceded by the `var` keyword in the argument list of the coverage group definition are passed-by-reference. Passed-by-reference arguments are considered constant references because NTB-OV does not attempt to modify their value.

If you use a pass-by-reference parameter in a conditional of a state or transition bin specification, NTB-OV uses the current value of the actual argument passed to the instance of the coverage group every time the instance samples.

If you use a pass-by-reference parameter to define the range of values associated with a state or transition bin, NTB-OV uses the value of the actual argument when the coverage group is instantiated. This is because NTB-OV requires that the range of values associated with a state or transition bin be defined when the coverage group is instantiated and remain constant for the life of the coverage group instance. The range of values associated with each state or transition bin cannot change every time a sample event triggers.

In Example 11-10 `MyCov` samples global variable `gVar` at every positive edge of the system clock. It defines a passed-by-value parameter `param` and a passed-by-reference parameter `param2`. Parameter `param` is used in the definition of state bins `s1` and `s2`. Parameter `param2` is used in the conditional associated with state bin `condState`.

Example 11-10 Pass by Reference and Pass by Value Parameters

```
coverage_group MyCov(integer param, var integer
param2){
      sample_event = @(posedge CLOCK);
      sample gVar {
            state s1 (0:param), state s2 (param+1 : 15);
            state condState (16 : 31) if (param2 > 4);
```

```
        }
    }
    ...
    program Example {
        bit [4:0] gVar;
        integer gP1, gP2;
        MyCov cov1;
        ...
        gP1 = 7;
        cov1 = new(gP1, gP2);
    // instantiate cov1 and pass arguments
        ...
        gP2 = 3;
        @(posedge CLOCK);
    // 1st sampling event after instantiation
        gP2 = 5;
        @(posedge CLOCK);
     // 2nd sampling event after instantiation
        ...
    }
```

In Example 11-10, when coverage group instance `cov1` is instantiated, arguments `gP1` and `gP2` are passed for parameters `param` and `param2`, respectively. When `gP1` is instantiated it has a value of 7. This implies that state `s1` of coverage group instance `cov1` is associated with 0 <= gVar1 <= 7. Similarly, state `s2` of coverage group instance `cov1` is associated with 8 <= gVar1 <= 15.

When the first sampling event occurs at the first at positive edge of `CLOCK`, the value of `gP2` is 3. This implies that state `condState` of coverage group instance `cov1` is not hit in the first sampling of sampled variable `gVar`. This is because the conditional of that bin evaluates to false. In contrast, the bin is hit in the second sampling because at the time of the second sampling event (second `(@posedge CLOCK)`), the value of `gP2` is 5.

Expressions in Coverage Group Definitions

You can use expressions within coverage group definitions to control user-defined bins as follows:

- Specify a coverage point to be sampled. NTB-OV samples the result of the expression when it samples the coverage group.
- Define a value in the range of values that defines a state, m_state, bad_state, trans, m_trans, and bad_trans of a coverage point. NTB-OV evaluates these expressions when the coverage group is instantiated.
- Define a state, ignored, or bad_state bin of a cross (see the "Valid Expressions for Defining Cross Coverage Bins" on page 549 for specific restrictions related to such expressions). NTB-OV evaluates these expressions when the coverage group is instantiated.
- Conditional of a user-defined bin in a coverage point or cross. NTB-OV evaluates these expressions every time the coverage group is sampled.
- Values of coverage group, coverage point, or cross attributes. NTB-OV evaluates these expressions when the coverage group is instantiated.

Coverage group expressions are a subset of the general NTB-OV expressions. Expressions must evaluate to an integer, bit vector or enum value. Operators with side effects, such as `++` and `--` are not allowed. The only functions allowed are `get_cycle()` and `get_time()`. The `query()` method of a coverage group is allowed inside expressions used for bin conditionals. The `query()` and `inst_query()` methods of a coverage group are allowed inside expressions used to set values of coverage group, coverage point, or cross attributes.

You cannot initialize the following coverage group attributes using `query()` or `inst_query()` function calls:

- bin_activation
- cumulative
- cov_comment
- overlap_state
- overlap_trans

These attributes affect the way a coverage group instance is initialized, and therefore their value must be determined before NTB-OV instantiates the coverage group. If you initialize the above attributes with the `query()` or `inst_query()` functions, the compiler issues an error.

Interface signals and port variables declared via the bind construct are allowed in expressions in coverage group definitions. Expressions within a coverage group can reference any variable or signal visible in the scope of the coverage group, including arguments passed to the coverage group. For example, an embedded coverage group can include expressions that refer to the data members of the class. (See “Embedded Coverage Group” on page 485.)

Variables and Scope

You can specify coverage information for any NTB-OV variable or interface signal in the scope of a coverage group's domain. This means that not all variables being monitored need to be passed in as arguments to the coverage group. A coverage group can directly refer:

- Global variables
- Interface signals
- Static binds
- Global virtual ports

In addition, an embedded coverage group can refer to:

- Class data members
- Base class data members
- Public data members of member objects

Example 11-11 shows an embedded coverage group that samples global variables, class data members, and an interface signal.

Example 11-11 Embedded Coverage Group

```
#include "MyInterface.vrh"  // Interface ifc1
extern integer g1;
extern MemberClass { extern integer m_y; }
extern MyBaseClass { extern integer m_z; }

class MyClass extends MyBaseClass {
      integer m_x;
      MemberClass m_obj;

      coverage_group cov1() {
            sample_event = @(posedge CLOCK);
            sample m_x ;  // Sample member of this class
```

```
            sample g1;    // Sample global variable
            sample ifc1.sig; // Sample interface signal
            sample m_z;    // Base class member
            sample m_obj.m_y; // Member of member object
            sample exp1(m_x + g1); // expression
        }
    }
```

Sample Event Definitions

You must specify a sampling event expression in the coverage group definition. NTB-OV uses this sampling event for all instantiations of a coverage definition.

The sampling event expression defines when to update the bins. Coverage objects can be triggered on clock edges, signal edges, variable changes, sync events, assertion events, and completions of user-defined tasks.

NTB-OV samples coverage objects at the end of the simulation cycle after all processing that can change variables is complete, and it samples coverage objects no more than once per simulation cycle even if the sampling event triggers more than once, unless you use the `async` attribute for the sample event.

Clock and Signal Edges

You can sample coverage objects on clock or signal edges as per the synchronization command. When the specified edge occurs, NTB-OV samples the object. The syntax is:

```
sample_event = @([specified_edge] interface_signal | CLOCK);
```

In Example 11-12, NTB-OV samples all instances of the `cov1` coverage group at the positive edge of the system clock.

Example 11-12 Coverage Group Sampling

```
coverage_group cov1 {
      sample_event = @(posedge CLOCK);
      sample g_var;
}
```

Variable Changes

You can sample coverage objects when variables change value using the `wait_var()` system task. When the value of a variable change occurs, NTB-OV samples the object. The syntax is:

```
sample_event = wait_var(variable_list);
```

In Example 11-13, NTB-OV samples the embedded `cov1` in instances of `MyClass` when there is a value change in the `xref` data member.

Example 11-13 Embedded Coverage Group Sampling

```
class MyClass {
      integer xref;
      coverage_group cov1 {
            sample_event = wait_var(xref);
            sample xref;
      }
}
```

Sync Events

You can sample coverage objects on `sync` events. When the `sync` is unblocked, the object is triggered. The syntax is:

```
sample_event = sync(ALL|ANY, some_OpenVera_event);
```

In Example 11-14, NTB-OV samples `cov1` in all instances of `MyClass` when `ev1` is triggered.

Example 11-14 Sync Event Sampling

```
extern Event ev1;
class MyClass {
      integer xref;
      coverage_group cov1 {
            sample_event = sync(ALL, ev1);
            sample xref;
      }
}
```

If the sample event is a `sync` the event needs to be triggered. If the event is in the `ON` state because an NTB-OV thread performed a `trigger(ON)` statement, sampling does not occur.

Coverage objects waiting on a `sync` event do not count towards the count for `HAND_SHAKE` triggers. If the `sync` event is triggered in `HAND_SHAKE` mode, all coverage objects waiting on that event are triggered. These do not count towards the `HAND_SHAKE`. An additional explicit thread waiting on that `sync` event also triggers.

Assertion Events

You can sample coverage objects on assertion events. When the assertion event occurs in the design under test, the `Wait()` member task on the object unblocks (see “Wait ()” on page 325). Example 11-15 shows an example.

Example 11-15 Sample Coverage Object on Assertion Event

```
sample_event = ova_object.Wait();
```

In this example, `ova_object` is an instance of an AssertEvent (see “AssertEvent Class” on page 324).

In Example 11-16, MyClass has a member variable which is an AssertEvent class object. NTB-OV samples the embedded coverage group cov1 when the assertion event attached to the AssertEvent class object fires.

Example 11-16 Embedded Coverage Group Sampling

```
class MyClass {
      integer port_num;
      AssertEvent portEvent;
      coverage_group cov1 {
            sample_event = portEvent.Wait()
            sample port_number;
      }
}
```

General Task Calls

If a coverage group needs an elaborate triggering mechanism, you can use a task call for the sampling event. In Example 11-17, NTB-OV samples the coverage object a certain number of clock cycles after the assertion event triggers. You specify the number of clock cycles using a parameter. CoverageTrigger1 is called by the sample_event in cov1. The sample_event passes in the value for numCycles to cov1 when the coverage group was instantiated with new().

Example 11-17 Task Call as Sampling Event

```
task CoverageTrigger1(integer numCycles) {
      integer i;
      globalAssertEvent.Wait();
      for (i=0;i<numCycles;i++){
            @(posedge CLOCK);
      }
}
coverage_group cov1 (integer numCyclesAfterAssertEvent){
sample_event = CoverageTrigger1 (numCyclesAfterAssertEvent);
      sample xref;
}
```

Note:

To avoid unpredictable behavior, the task should not have any side effects such as changing the value of variables. When you use a task as a `sample_event`, the task must block.

Optional Async Behavior of Sampling Event

You can optionally use the `async` attribute with the `sample_event` specification to gather coverage information immediately. If the `sample_event` is a task, you cannot use the `async` modifier.

When you use the `async` attribute, NTB-OV samples the coverage object immediately when the sampling event triggers. If the sampling event fires more than once in the simulation cycle, NTB-OV samples the coverage object more than once too (once per trigger).

When you use the `async` attribute with a sampling event that is a `wait_var` or `sync` event, NTB-OV samples the coverage object immediately when the event triggers.

When you use the `async` attribute with a sample event that synchronizes on a signal, clock edge, or assertion event, NTB-OV samples the coverage object immediately instead of at the end of the simulation cycle. In Example 11-18, NTB-OV samples the embedded coverage group `cov1` in all instances of `MyClass` immediately when `global_event` triggers.

Example 11-18 Async Sampling Event

```
class MyClass {
      integer xref;
      coverage_group cov1 {
            sample_event = sync(ALL, global_event) async;
            sample xref;
      }
}
```

Sample Event Global Variables and Signals

You can use sample event expressions to reference global variables and signals. In this case, all instantiations of the coverage group definition have the same sampling event. In Example 11-19, NTB-OV samples all instances of the coverage group `covType` on the posedge of the `ifc.clk` signal.

Example 11-19   Sample Event on Signal

```
coverage_group covType () {
      sample_event = @(posedge ifc.clk);
      sample g_var;
}
```

Sample Event Class Member Variables

You can use a sample event expression of an embedded coverage group to reference member variables of the nesting class. Because an embedded coverage group is within the scope of the containing class, it has access to the object's members (for example, `this.variables`). In Example 11-20, NTB-OV samples coverage object `obj1.covType` when `obj1.m_e` changes value, and samples coverage object `obj2.covType` when `obj2.m_e` changes value.

Example 11-20   Sample Event on Class Member Variable

```
MyClass
      integer m_e;
      coverage_group covType {
            sample_event = wait_var(m_e);
            sample m_e;
      }
}
...
...
MyClass obj1 = new();
MyClass obj2 = new();
```

Using Passed-in Arguments

You can use the sample event expression to reference arguments passed in to the coverage group. This is handy for setting up a different sampling event for each instance of a coverage group.

In Example 11-21, NTB-OV samples coverage object `cov1` every time the `gWait` variable changes and samples coverage object `cov2` every time the `gAcc` variable changes. Note that the argument is passed as a `var` argument when tracking all changes to its value.

Example 11-21   Passed-in Arguments Sampling

```
coverage_group covType (var integer waitInt) {
      sample_event = wait_var(waitInt);
      sample g_var;
}

program myTest {
      integer gWait, gAcc, gvar;
      covType cov1 = new(gWait);
      covType cov2 = new(gAcc);
      ...
}
```

Defining State and Transition Bins for Coverage Points

If you don't define any state or transition bins for a coverage point, NTB-OV automatically creates state bins for you. You can also explicitly define named state or transition bins for each coverage point. Each named bin groups a set of values (state) or a set of value transitions (trans) associated with a coverage point.

Revised OpenVera Syntax for Transition Bin Specifications

This section outlines the changes in OpenVera syntax that implement this feature.

```
trans trans_bin_name(state_bin_name
[repeated_transition_values] -> state_bin_name
[repeated_transition_values] -> …) [conditional];
```

trans_bin_name

Represents the name of the transition bin.

state_bin_name

Represents the name of a state bin defined in the same sample construct. This name must be a user-defined state bin name. It cannot be the name of `bad_state` bin, `m_state` bin, `m_bad_state` bin or "all" state bin.

`repeated_transition_values` (optional)

Specifies the number of times a particular value is to be repeated in a transition sequence. Use the following constructs:

`[*`*constant*`]` | `[*`*min_constant*`:`*max_constant*`]`

The *`constant`* construct must represent an unsigned integer constant. The state preceding the constant must be repeated a fixed number of times.

The *`min_constant:max_constant`* construct must represent unsigned integer constants. The state preceding the constants must be repeated at least *`min_constant`* times and not more than *`max_constant`* times.

The following example illustrates the use of the state bin name syntax:

```
coverage_group COV1
{
    sample x {
        state s1(1);
        state s2(2);
        trans t0sbn(s1->s2);
    }
    sample_event = wait_var(x) async;
}
```

There is no dependency on the order of specification of state bins and transition bins where the state bin names are used. State bin names can be specified after their names are used in a transition bin specification.

Conditionals for Sample Level Definition

You can add conditional statements at the end of the sample level definitions in the form `if(NTB-OV_expression)`. You cannot call functions other than `get_cycle()` and `get_time()` in the conditional. When you attach a conditional to the sample level definition, NTB-OV considers all the bins for that sample only if the conditional evaluates to true (see Example 11-22).

Example 11-22 Conditionals for Sample Levels

```
sample var1 if (open_vera_expression);
// sample with autobinning

sample var1 if (open_vera_expression),
 var2 if (open_vera_expression);
// multiple samples in the same statement
// with different guard conditions

sample var1 if (NTB-OV_expression) {
// sample with user-defined bins
```

```
    state s0(0);
    state s1(1);
    ...
}
```

If you use a global variable or class data member in a condition, NTB-OV uses the current value of the variable at the sampling time. There are no restrictions on the use of coverage point names in a conditional associated with that coverage point or any other coverage point of the coverage group.

If you use a non-sampled, passed-by-value argument in a conditional statement, NTB-OV uses the value of the argument when the coverage group is instantiated in the evaluation of the conditional.

You can reference a `sample_name` in a guard (conditional) statement associated with a state or transition bin of a sample or cross. This is equivalent to using the corresponding sample expression in place of the `sample_name` (see Example 11-23).

Example 11-23   Sample Name Conditional

```
extern integer x, y;
coverage_group Cov {
      sample sam2 (x * y); sample sam1 (x + y) {
            state s1 (0: 50) if (sam2 > 30); // same as if (x*y > 30)
      }
      sample_event = @(posedge CLOCK);
}
```

Auto Bin Creation

NTB-OV follows the SystemVerilog style of auto bin creation to automatically create state bins for a coverage point. Use the following shorthand syntax to define the coverage point:

```
sample coverage_point [, coverage_point,...];
```

You can refer to multiple coverage points in a single sample statement. In Example 11-24, NTB-OV automatically creates state bins for coverage points `m_x` and `m_y`.

Example 11-24 Automatic Coverage Bins

```
class MyClass {
      bit [0:7] m_x, m_y;

      coverage_group cov1 {
            sample_event = @(posedge CLOCK);
            sample m_x, m_y;
      }
}
```

By default, NTB-OV creates automatic state bins covering the entire range of values of the sample.

You can control the number of automatic state bins created by using the `auto_bin_max` attribute. The entire range of values of the sample is divided into *N* equal bins, where *N* is the value you specify with the `auto_bin_max` attribute. You can specify this attribute inside of a sample construct (applying to that coverage point only) or at the coverage group level for all coverage points of the group.

For example, consider a sample that is a 4-bit variable and with `auto_bin_max` set to 2. The possible range of values is 0 to 15. NTB-OV creates two bins, with the associated ranges [0:7] and [8:15]. For 100 percent coverage, the simulation should result in values of the sample uniformly distributed across the entire range. Example 11-25 shows NTB-OV code snippets that illustrate the effects of auto bin specification.

Example 11-25 Auto Bin Specification

```
sample a {
      auto_bin_max = 7;
}
```

-or-

```
// affects all samples
coverage_group cg {
      auto_bin_max = 20;
      sample a, b;
      sample e (a + b){
      }
}
```

-or-

```
...
sample a, b {
       auto_bin_max = 7;
}
// affects both a and b
```

For enumerated types, `auto_bin_max` has no effect.

Example 11-26 shows a fragment of NTB-OV code that defines a coverage group with automatically created bins for global sampled variables `m_x` and `m_y`. The `auto_bin_max` is used to limit the number of automatically created state bins to two.

Example 11-26 Coverage Group with Automatically Created Bibs

```
bit [3:0] m_x, m_j;
coverage_group cov1 {
      sample_event = @(posedge CLOCK);
      sample m_x, m_y;
      auto_bin_max = 2; // max of 2 auto-created bins
}
      ...
      m_x = 12;
      m_y = 3;
      @(posedge CLOCK);
      m_x = 1;
      m_y = 8;
      @(posedge CLOCK);
      m_x = 5;
      m_y = 4;
      @(posedge CLOCK);
```

Table 11-8 and Table 11-9 show the automatically created bins when the coverage group is first instantiated.

Table 11-8 Bins for Sample m_x

Bin Name	Hit Count
auto[0:7]	0
auto[8:15]	0

Table 11-9 Bins for Sample m_y

Bin Name	Hit Count
auto[0:7]	0
auto[8:15]	0

Table 11-10 shows the automatically created bins and their hit counts after the first `@(posedge CLOCK)`.

Table 11-10 Hit Count After First CLOCK

Sampled Variable	Bin Name	Hit Count
`m_x`	auto[0:7]	0
	auto[8:15]	1
`m_y`	auto[0:7]	1
	auto[8:15]	0

Table 11-11 shows the automatically created bins and their hit counts after the second `@(posedge CLOCK)`.

Table 11-11 Hit Count After Second CLOCK

Sampled Variable	Bin Name	Hit Count
`m_x`	auto[0:7]	1

Table 11-11 Hit Count After Second CLOCK

Sampled Variable	Bin Name	Hit Count
	auto[8:15]	1
`m_y`	auto[0:7]	1
	auto[8:15]	1

Table 11-12 shows the automatically created bins and their hit counts after third and final `@(posedge CLOCK)`.

Table 11-12 Hit Count After Third CLOCK

Sampled Variable	Bin Name	Hit Count
`m_x`	auto[0:7]	2
	auto[8:15]	1
`m_y`	auto[0:7]	2
	auto[8:15]	1

Automatically Generated Bins for Enumerated Types

When a coverage point is an enumerated type, NTB-OV automatically creates state bins for all the enumerated values of the coverage point. In this case, the `auto_bin_max` attribute does not affect the number of automatically created bins. A coverage point that is an expression is considered to be an enumerated type if all the operands are of the same enumerated type. Otherwise, the type of the coverage point expression depends on the type and width of the sampled expression. In Example 11-27, the `covType` coverage group samples the `sParam` global variable, which is a `cellType` enumerated type.

Example 11-27 Coverage Group Samples Enumerated Type

```
enum cellType = blue,red,green;
extern cellType sParam;
coverage_group covType {
     sample_event = wait_var(sParam);
```

```
        sample sParam;
        auto_bin_max = 2;
    // Does not affect the number of auto-bins for sParam
    }
```

In Example 11-27, NTB-OV automatically creates three state bins even though the auto_bin_max attribute is set to 2. Example 11-28 shows a shorthand way to explicitly type the following coverage group .

Example 11-28 Explicit Typing for Coverage Group

```
coverage_group covType {
    sample_event = wait_var(sParam);
    sample sParam {
        state s_blue(blue);  // sParam == blue
        state s_red(red);    // sParam == red
        state s_green(green); // sParam == green
    }
}
```

Overview of User Defined Bins for Coverage Points

You can explicitly define coverage bins for each coverage point. This way you can create named bins for states or value transitions of a coverage point. A state bin using the state or m_state construct defines a named bin for a set of values of a coverage point. A transition bin using trans or m_trans constructs defines a named bin for one or more sequence of values associated with a coverage point.

The syntax for defining explicit state or transition bins uses ranges of values to define states and transitions. Example 11-29 shows how to define two state bins and a transition bin for the port_number sampled variable of the MyCov coverage group. The port_number variable is assumed to be global; NTB-OV samples it every time it changes.

Example 11-29 User-defined Bins for Coverage Points

```
coverage_group MyCov () {
      sample_event = wait_var(port_number)async;
            // Sample every time  port_number changes
      sample port_number {
            state s0(0:7);
            state s1(8:15);
            trans t1("s0"->"s1");
      }
}
```

In Example 11-29, state bin `s0` is associated with values of the sampled variable `port_number` between `0` and `7`. State `s1` is associated with values of `port_number` between `8` and `15`. Transition `bin t1` is associated with any single value transition of sampled variable `port_number` from 0-7 to 8-15. For example, when `port_number` changes from 0 to 8 or 0 to14 transition bin, `t1`'s NTB-OV increments the hit count.

Multi-State and Multi-Transition Specification

You can use the `m_state` (short for multistate) and `m_trans` constructs to create multiple bins for a range of values. NTB-OV automatically creates a bin for each value in the range. The following `m_state` creates eight bins. Each bin corresponds to one of the values in the range `0` to `7`.

```
m_state ms(0:7);
```

NTB-OV prefixes each of the automatically created bins with the name used in the `m_state` construct. In the example above, NTB-OV creates state bins `ms_0`, `ms_1`, `ms_2`, `ms_3`, `ms_4`, `ms_5`, `ms_6`, and `ms_7`. State bin `ms_0` is associated with the coverage point having value 0.

In the following example, the `m_trans` construct is used to automatically create four transition bins associated with the transitions 0->3, 0->4, 1->3, and 1->4 of a coverage point.

```
m_trans ms(0:1->3:4);
```

In this example, transition bin `ms:0->3` is associated with the transition of the coverage point from value 0 to value 3.

Multistate (`m_state`) and multitransition (`m_trans`) specifications are useful for explicitly controlling the range of values that you are interested in binning, and creating individual bins for each of the values in the range. In the above `m_state` example the user is only interested in coverage of the coverage point when its value falls within the range 0 to 7, and wants to know how many times the coverage point takes on the values of interest. An alternative is to avoid explicit definition of the `m_state` and use NTB-OV's auto-binning capabilities. However, the initial values of the coverage points may be outside of the range of interest, causing NTB-OV to create bins that are not interesting.

In general, explicit state and transition specifications help you define and refine the shape of your coverage space. However, you should be wary of defining `m_state` and `m_trans` specifications for a large range of values, since this can affect performance. The `m_state` and `m_trans` bin specifications do not expand beyond 4,096 bins.

Wildcard State and Transition Definitions

By default, NTB-OV only matches a state or transition bin definition that uses an `x` if the coverage point has an `x` in that bit position (the semantics are similar to the `===` operator). You can use the wildcard state and wildcard trans bin definitions to make NTB-OV consider an `x` in the bin definition as a wildcard. In the following example, the state bin `s5` increments when the coverage point is either 1100, 1101, 1110, or 1111.

```
wildcard state s5(4'b11xx);
```

Bad States and Transitions

You can define illegal states (values) of a coverage point using a `bad_state` statement as follows:

```
bad_state state_name (range_definition) [conditional];
```

You can also define illegal value transitions of a coverage point using a `bad_trans` statement as follows:

```
bad_trans trans_name (transition_specification)[conditional];
```

Illegal or bad transitions are value transitions of the coverage point that result in a verification error.

All States and Not State

A special case of a state bin definition uses the `all` specification:

```
state state_bin_name (all);
```

This statement indicates that NTB-OV creates a bin for each sampled value of the coverage point (or increments the bin hit count). Similarly, a special case of a transition bin definition uses the `all` specification:

```
trans transition_bin_name (all);
```

This statement creates a transition bin for every 2-state transition of the coverage point.

Note that the `all` state or transition specifications are intended for debugging coverage information only. The bin hit counts associated with the individual bins created for `all` state or transitions do not affect coverage numbers. More importantly, if you use `state(all)` or `trans(all)`, a significant amount of memory may be used if the number of unique values of the coverage point is large. Finally, NTB-OV creates individual bins for `state(all)` and `trans(all)` and increments and their hit counts even if the sampled values match other user-defined bins.

Another special case of a state definition uses the `not state` specification:

```
state state_bin_name (not state);
```

With this construct, all values not recorded by other states increment the bin-hit counter for `state_bin_name`.

Using Conditionals in Bin Definitions

You can add a conditional statement at the end of any state or transition bin specification as long as they are user-defined bins. Conditional statements must be in the form `if(NTB-OV_expression)`. You cannot use functions other than `get_cycle()` and `get_time()` in the conditional. When you attach a conditional to a bin definition, NTB-OV only increments the hit count of the bin if the conditional evaluates to true and the value of the coverage point matches the bin's specification.

In the following example, NTB-OV increments the hit count for `state s0` if the coverage point's value is between `0` and `7` and value of variable `g_var` is greater than 0.

```
state s0(0:7) if (g_var > 0);
```

In this example, `g_var` could be any variable visible to the coverage group based on NTB-OV's normal scoping rules (involving global variables, class data members, etc.).

If you use a global variable or class data member in a condition, NTB-OV uses the current value of the variable at the sampling time. There are no restrictions on the use of coverage point names in a conditional associated with that coverage point or any other coverage point of the coverage group.

If you use a non-sampled, passed-by-value argument in a state or transition bin's conditional, NTB-OV uses the value of the argument when the coverage group is instantiation to evaluate the conditional.

User-defined States for Coverage Points

You use state definitions inside a sample construct to associate a named bin with a range of values of a coverage point. The syntax is:

```
state state_bin_name (state_specification) [conditional];
```

`state_specification`

is a list of elements (separated by commas) that NTB-OV matches against the current value of the coverage point. In a state declaration, a single state or multiple states are associated with a monitor bin by means of a `state_specification`. For the current cycle, any matches increment the bin counter by one. Each element of the `state_specification` is a value range specification in one of the following formats:

`expression`

a counter is added to the bin when the state of the coverage point matches the expression exactly. `x` or `z` must match exactly.

`low:high`

a counter is added to the bin when the state of the coverage point matches any value in the range from low to high.

`low:high:step:repeat`

creates multiple ranges. The first block ranges from low to high. The second block ranges from (high+step) to (2*high-low+step). New blocks are generated repeat times. For example:

2:5:10:3 produces states {2, 3, 4, 5,15, 16, 17, 18, 28, 29, 30, 31}

(2, 3, 4, 5) --> (15, 16, 17, 18) -->(28, 29, 30, 31)

You can generate complex state specifications by separating multiple formats with commas as shown in Example 11-30.

Example 11-30 Complex State Specifications

```
state bin1 (8'b0000_01XX, 8:10, 15:17:7:2);
```

This state specification increments the bin counter if any of the specifications matches the state of the coverage point. In this example, a counter is added to the bin if:

- The state of the coverage point matches `8'b0000_01XX` exactly.
- The state of the coverage point falls in the range of 8 to 10.

- The state of the coverage point falls in the range of 15 to 17 and 24 to 26.

Note:

x and z are not allowed in repeated range statements.

It is important to note that NTB-OV evaluates state specifications only once, when the coverage object is instantiated. While the state of the coverage point is being monitored, the state specification remains constant.

m_state

Use the m_state state declaration to declare multiple state bins up to a maximum of 4096 bins. The syntax is:

m_state *state_bin_name* (*exp1*:*exp2*)**;**

state_bin_name

is the base name of the state bins being created.

exp

can be any valid coverage expression. You cannot call functions in the expressions, but the expressions can include variables.

When you use the m_state declaration, NTB-OV creates multiple state bins covering all the values in the range and evaluates expressions when the coverage object is instantiated. For example:

```
m_state s1 (2:4);
```

This example creates the following bins and state values:

State Bin Name State Value

s1_2 — 2

s1_3 — 3

s1_4 — 4

If you don’t specify a bin name, the same example yields the following bin names and values:

State Bin Name State Value

s_2 — 2

s_3 — 3

s_4 — 4

All States and Not State

A special case of a state declaration uses the `all` state specification. You should only use the `all` state specification to debug coverage information. It does not contribute to the coverage statistics for the enclosing sample and coverage group. Consider this next example:

```
state state_bin_name (all);
```

This statement indicates that NTB-OV creates a unique state bin for each sampled value of the coverage point. The generated state bins are named:

```
s_value
```

where *`value`* is the value of the coverage point.

Note:

> If you use `state(all)`, NTB-OV uses a significant amount of memory if the number of sampled and unique states is large.

Another special case of a state declaration uses the `not state` specification:

```
state state_bin_name (not state);
```

When you use this construct, NTB-OV increments the counter for *`state_bin_name`* for all values not recorded by other states.

State and Transition Bin Names

You can assign bins explicit names, or let NTB-OV generate implicit names based on the state specification. The general format is always prefaced by `s_`. NTB-OV converts the following characters to underscores (_) when it generates bin names:

- comma (`,`), colon (`:`), slash (`/`), dash (-), plus (+), asterisk (*), and period (.)

NTB-OV ignores the following characters when it generates bin names:

- double quote (`"`), brackets (`[]`), parentheses (`()`), dollar sign (`$`), caret (^), back slash (\), and braces ({ })

Table 11-13 shows some examples.

Table 11-13 State Expressions and Explicit Bin Names

State Expression	Implicit Bin Name
`(5,7,9,11)`	s_5_7_9_11
`(15:37)`	s_15_37
`(base*10+3)`	s_base_10_3
(`all`)	s_*value* (see above)

Conditionals in State Definitions

You can add conditional statements at the end of any state or transition declaration, but you cannot call functions other than `get_cycle()` and `get_time()` in the conditional. If a conditional statement is attached to a state or transition declaration, the bin counter increments only if the condition is true and the state of the coverage point matches the state specification at the same time.

You don't need to pass variables used in coverage conditionals as arguments to the coverage object if the variables are visible in the scope where the coverage group is defined.

If you use a passed-by-value parameter of the coverage group in a conditional, NTB-OV uses the value of the parameter when the coverage group is instantiated. If you use a passed-by-reference parameter of the coverage group in a conditional, NTB-OV uses the value of the parameter when the conditional is evaluated (at every sample point). You can also use a sample name associated with a sampled expression in the conditional expression. Example 11-31 shows an example.

Example 11-31 Parameter in Conditional of Coverage Group

```
state jmp_ins (8'b0000_01XX, 8:10, 15:17:7:2) if (test == ON);
```

In Example 11-31, the `state` declaration creates the `jmp_ins` bin. NTB-OV increments the bin counter when the state of the coverage point matches the specification and the conditional is true.

Wildcard State Declarations

By default, a bit with a value of `x` must match a sampled value of `x` for NTB-OV to increment a state bin counter. You can use the `wildcard` keyword to make NTB-OV treat `x` values as wildcards in state declarations. The syntax is:

```
wildcard state state_bin_name(state_specification);
```

Example 11-32 shows an example.

Example 11-32 Wildcarded State Declaration

```
wildcard state sw(4'b11xx);
```

In Example 11-32, NTB-OV increments the state bin for 1100,1101,1110, and 1111.

Multiple State Bin Declarations

You can use a state declaration to declare multiple bins on a single line as follows:

```
state b0 (0), b1 (1), b2 (2);
```

In this example, `b0`, `b1`, and `b2` are separate state bins, each with their own state specifications. You can also specify multiple state declarations, and associate the same value with multiple state bins, as shown in Example 11-33.

Example 11-33 Multiple State Declarations

```
state b0 (0:10);
state b1 (10:20);
```

In Example 11-33, value `10` is associated with bin `b0` and `b1`.

User-defined Illegal States for Coverage Points

You can use the sample construct to define illegal state definitions that associate illegal states with a coverage point being sampled by a coverage group. The syntax is:

```
bad_state error_bin_name (state_specification) [conditional];
```

Bad or illegal states are states in the design that result in verification errors.

The state specification can be any expression or combination of expressions (as in state declarations). However, it is often useful to define every state that is not in the state declarations as a `bad_state`. To use that definition of bad states, use the `not state` specification. The syntax is:

```
bad_state error_bin_name (not state);
```

For this statement, NTB-OV increments the specified bin counter every time the state of the coverage point matches a value not defined in the state declarations, and issues a runtime verification error. If you do not specify an error bin name, the implicit name is `s_not_state`.

To specify multiple bad states, you can use the `m_bad_state` declaration. The syntax is:

```
m_bad_state error_bin_name (exp1:exp2);
```

When you use the `m_bad_state` declaration, NTB-OV creates a bin for each value in the range. If you don't specify a bin name, NTB-OV uses the same implicit naming conventions as with `m_state`.

Ignored States for Samples

The ignored keyword is used to define a bin that should not be considered for coverage calculations. If the expression for the bin evaluates to true, then the bin is skipped and hit count is not updated.

To define state that should not be considered for coverage calculations, use the ignored keyword.

Syntax

```
ignored state_name ((select_expression)[logical_operator
(select_expression)]) [ if (expression) ];
```

Example 11-34 Ignored States for Samples

```
coverage_group Cov {
sample addr;
sample mode {
state s_read(READ);
ignored s_write(WRITE);
}
sample_event = @(posedge CLOCK);
}
```

If the expression for the ignored sample coverage bin evaluates to true, then the enclosing state is skipped and the bin is not updated.

User-defined Transitions for Coverage Points

Transition definitions associate value transitions with a coverage point. You define transitions inside the sample construct of a coverage group. The syntax is:

```
trans trans_bin_name (transition_sequence_list) [conditional];
```

`trans_bin_name`

is the name of the transition bin.

`transition_sequence_list`

is a comma-separated list of transition sequences.

Defining Transition Sequences

Each transition sequence defines value transitions of the coverage point. A transition sequence has the following form:

```
value_range_set -> value_range_set -> ... -> value_range_set
```

A value range set is made up of one or more value ranges. A value range can be one of:

- any of the value range specifiers used for defining a state.
- a state bin name specified in a state declaration and enclosed in double quotes. (When specifying a state bin name, NTB-OV uses the value ranges specified for that state to match the transition sequence and ignores the guard condition for that state bin.)
- a Perl regular expression matching a state bin name, enclosed in double quotes.

If a value range set has more than one value range, the set must be enclosed by brackets ([]), and the members separated by commas, as shown in Example 11-35.

Example 11-35   Transition Sequence Definition

```
trans trans_bin_name (["jms_ins", br:xor, "jmp[0-9]+"] -> [15:20:2:100]);
```

Example 11-35 defines a complex value transition. With this example, NTB-OV increments the specified bin counter when any of the following occurs:

- The state of the coverage point changes from a value defined by state `jms_ins` to a value that falls in the repeated range defined by `[15:20:2:100]`.
- The state of the coverage point changes from a value in the range from `br` to `xor` to a value that falls in the repeated range defined by `[15:20:2:100]`.
- The state of the coverage point changes from a value defined in any state bin that begins with `jmp` and ends with at least one digit to a value that falls in the repeated range defined by `[15:20:2:100]`.

If NTB-OV matches all value range sets in a transition sequence in order, it increments the transition monitor bin. For a transition sequence with more than two value range sets, if only one of the transitions from one range set to another is observed, the monitor bin is not incremented.

m_trans

Use the `m_trans` transition declaration to declare multiple transition bins up to a maximum of 4096 bins. The syntax is:

```
m_trans trans_bin_name (exp1:exp2 -> exp3:exp4);
```

`trans_bin_name`

is the base name of the transition bins being created.

`exp`

can be any valid coverage expression. You cannot call functions in the expressions, but they can include variables that are visible in the scope of the coverage group.

When you use an `m_trans` declaration, NTB-OV creates multiple transition bins that cover all transitions in the specified ranges. Each set of expressions specifies a range. NTB-OV creates a bin for each permutation of valid values. Example 11-36 shows an example:

Example 11-36   m_trans Declaration

```
m_trans t1 (2:3 -> 4:5);
```

Example 11-36 creates the following bins and transitions:

Transition Bin	Name Transition
t1:2->4	2 to 4
t1:2->5	2 to 5
t1:3->4	3 to 4
t1:3->5	3 to 5

If you don't specify a bin name, the same example yields these bin names and values:

Transition Bin   Name Transition

t_s_2_3_s_4_5:2->4 2 to 4
t_s_2_3_s_4_5:2->5 2 to 5
t_s_2_3_s_4_5:3->4 3 to 4
t_s_2_3_s_4_5:3->5   3 to 5

All Trans and Not Trans

The `all` transition argument is a special case for transition bins that causes NTB-OV to create and track a transition bin for every two-state value transition of the coverage point. The syntax is:

```
trans trans_bin_name (all);
```

The automatically created bins associated with the `all` transition specification do not contribute to the coverage statistics of the enclosing sample and coverage group constructs. Note that when you use `trans(all)`, a significant amount of memory is consumed if the number of sampled and unique transitions is large.

The `not` trans specification is another special case for transition bins that causes NTB-OV to record undefined transitions without print a verification error. The syntax is:

```
trans trans_bin_name (not trans);
```

NTB-OV names transition bins as explained in "State and Transition Bin Names" on page 522.

Conditional Statements

You can add conditional statements at the end of any transition declaration, but you cannot call functions other than `get_cycle()` and `get_time()` in the conditional. If there is a conditional statement attached to the transition declaration, NTB-OV increments the bin counter only if the condition is true and the state of the coverage point makes the specified transition at the same time.

You don't need to pass variables used in coverage conditionals as arguments to the coverage object. However, you must declare those variables using the `extern` construct within the main program.

If you use a passed-by-value parameter of the coverage group in a conditional, NTB-OV uses the value of the parameter when the coverage group is instantiated. If you use a passed-by-reference parameter of the coverage group in a conditional, NTB-OV uses the value of the parameter when the conditional is evaluated (at every sample point). Example 11-37 shows a transition declaration.

Example 11-37 Transition Declaration

```
trans jmp_ins (8:10 -> 15:17:7:2) if (test == ON);
```

In Example 11-37, the transition declaration creates the `jmp_ins` bin. NTB-OV increments the bin counter when the state of the coverage point makes the specified transition and the conditional is true. If you specify a sequence of transitions, NTB-OV only evaluates the conditional during the final transition.

Wildcard Transition Declarations

By default, a bit with a value of `x` must match a sampled value of `x` for NTB-OV to increment a transition bin counter. You can use the `wildcard` keyword to make NTB-OV treat the `x` value as a wildcard in transition declarations. The syntax is:

```
wildcard trans trans_bin_name(value_transitions);
```

Example 11-38 shows an example.

Example 11-38 Wildcard Transition Declaration

```
wildcard trans tw(2'b0x -> 2'b1x);
```

In Example 11-38, NTB-OV increments the transition bin for transitions from:

```
00 -> 10
00 -> 11
01 -> 10
01 -> 11
```

Defining Multiple Transitions in One Statement

You can use a transition declaration to declare multiple bins on a single line:

```
trans b0 (0 -> 1), b1 (1 -> 2), b2 (2 -> 3);
```

In this example, `b0`, `b1`, and `b2` are separate transition bins, each with its own transition specification.

Repeated Transition Values

You can specify the number of times a particular value is to be repeated in a transition sequence using these constructs:

```
[.constant.]
[.min_constant:max_constant.]
```

`constant`

must be an unsigned integer constant. It cannot be an expression that evaluates to a constant.

Using the first construct, the transition value preceding the constant must be repeated a fixed number of times. For example, these transition declarations are equivalent:

```
trans t1(1 -> 2[.3.] -> 3);
trans t1 (1 -> 2 -> 2 -> 2 -> 3);
```

Using the second construct, the transition value preceding the constant must be repeated at least *min_constant* times and not more than *max_constant* times. For example:

```
trans t1(1 -> 2[.1:3.] -> 3);
```

This transition declaration creates a single monitor bin that is equivalent to these transition declarations:

```
trans t1(1 -> 2 -> 3);
trans t2(1 -> 2 -> 2 -> 3);
trans t3(1 ->2 -> 2 -> 2 -> 3);
```

Note that the first example uses a single monitor bin to count any of the valid transitions, whereas the second example uses three separate bins to monitor the same transitions.

User-defined Illegal Transitions for Samples

Illegal transition definitions associate an illegal transition with a coverage point. You define illegal transitions inside the sample construct of a *coverage group*. The syntax is:

```
bad_trans trans_bin_name (transition_sequence_list) conditional;
```

"Defining Transition Sequences" on page 527 explains the syntax for defining transition sequence lists. It is sometimes useful to monitor all transitions that have not been defined in other transition bins. For such cases, you can use the `not trans` argument. The syntax is:

```
trans trans_bin_name (not trans);
```

NTB-OV increments the counter associated with the specified bin every time a transition occurs that is not explicitly defined in any of the transition declarations for the coverage point. If you don't specify a bin name, the implicit name is `t_not_trans`.

The `not trans` modifier applies only to single transitions between two values or two value range sets. It does not apply to larger transition sequences. So, if you define a bad transaction using `not trans`, make sure all valid single transitions are covered in one of the transition bins. To specify multiple bad transitions, use the `m_bad_trans` declaration. The syntax is:

```
m_bad_trans error_bin_name (exp1:exp2 -> exp3:exp4);
```

When you use an `m_bad_trans` declaration, NTB-OV creates a bin for each transition. If you don't specify a bin name, NTB-OV uses the same naming conventions used for `m_trans`.

Cross Coverage Definitions

Integrating cross coverage into the coverage model specification ensures tighter and easier-to-understand semantics. You specify crosses up front in the coverage group specification and refer to the coverage points in the same coverage group. NTB-OV samples all crosses (cross constructs) and coverage points (sample constructs) using the same `sample_event` of the coverage group.

You specify coverage point crosses for a coverage group using the `cross` construct. You can use NTB-OV expressions to concisely specify cross coverage bins. The syntax is:

```
cross cross_name (coverage_point [, coverage_point,...]){
```

```
        [cross_bin_definition]
        [attribute_definitions]
    }
```

You must specify at least two `coverage_points`.

`cross_name`

is the name for the cross. You can use this name to query or set information about the cross in the testbench.

`cross_bin_definitions`

you can use NTB-OV to define named cross coverage bins. For more information, see “User-Defined Cross Coverage Bins” on page 539.

`attribute_definitions`

you can use attributes to control various aspects of a *cross*. For more information, see “Attribute Definitions” on page 552. You specify an attribute’s value as follows:

```
attribute_name = value_expression;
```

where *`attribute_name`* is the name of the attribute, and *`value_expression`* is a NTB-OV expression. For more information, see “Expressions in Coverage Group Definitions” on page 496.

Auto-Bin Creation of Cross Coverage Bins

In the following example, NTB-OV automatically creates bins for the cross product of state or transition bins covering the complete range of values for the sample or coverage point:

```
cross cross_name (coverage_point, coverage _point, ...);
```

In this next example (Example 11-39), the `cov1` embedded coverage group samples two data members of the embedding class `MyClass` (`m_x` and `m_y`), and a global variable `g_var1`.

Example 11-39 Cross Coverage Bins

```
extern bit [3:0] g_var1;
class MyClass {
      bit [3:0] m_x, m_y;
      bit       m_z;
      coverage_group cov1 {
            sample_event = @(posedge CLOCK);
            sample m_x, m_y, g_var1;
            cross cc1(m_y, g_var1);
            auto_bin_max = 8;
      }
      task new() {
            m_x = 4'b0000;
            m_y = 4'b0000;
            m_z = 1'b0;
      }
}
program test {
      bit [3:0] g_var1 = 4'b0000;
      MyClass Obj1 = new;
      @(posedge CLOCK);
      Obj1.m_y = 2;
      g_var1 = 1;
      @(posedge CLOCK);
      Obj.m_y = 0;
      @(posedge CLOCK);
}
```

In Example 11-39, the coverage group defines a cross named `cc1` for crossing sampled variables `m_y` and `g_var1`. Since the example does not define any state or transitions for the sampled variables, NTB-OVB automatically creates state bins covering the complete range of values for the sampled variables. Cross `cc1` includes bins associated with the cross product of the automatically generated bins for the `m_y` and `g_var1` sampled variables. Example 11-39 generates the report shown in Example 11-40.

Example 11-40 Cross Coverage Report

```
Summary for variable m_x
                     Expected Covered Percent
Automatically Generated Bins 8        1       12.50

Automatically Generated Bins for m_x
Uncovered bins
name                      count at least
[auto[2:3] - auto[14:15]] --    1        (7 bins)

Covered bins
name      count at least
auto[0:1] 2     1
----------------------------------------------------
Summary for variable m_y
                             Expected Covered Percent
Automatically Generated Bins 8        2       25.00
Automatically Generated Bins for m_y
Uncovered bins
name                      count at least
[auto[4:5] - auto[14:15]] --    1        (6 bins)

Covered bins
name      count at least
auto[0:1] 1     1
auto[2:3] 1     1
                ----------------------------------------------------
Summary for variable g_var1
                             Expected Covered Percent
Automatically Generated Bins 8        1       12.50

Automatically Generated Bins for g_var1
Uncovered bins
name                      count at least
[auto[2:3] - auto[14:15]] --    1        (7 bins)

Covered bins
name      count at least
auto[0:1] 2     1
---------------------------------------------------
Summary for cross cc1
Samples crossed: m_y g_var1
                                   Expected Covered Percent Missing
Automatically Generated Cross Bins 64       2       3.12    62

Automatically Generated Cross Bins for cc1
Uncovered bins
m_y                    g_var1                 count at least
```

```
[auto[0:1] - auto[2:3]]  [auto[2:3] - auto[14:15]] --   1     (14 bins)
[auto[4:5] - auto[14:15]] [auto[0:1] - auto[14:15]] --   1     (48 bins)

Covered bins
m_y       g_var1    count at least
auto[2:3] auto[0:1] 1     1
auto[0:1] auto[0:1] 1     1
```

NTB-OV creates automatic state and cross bins as shown in Example 11-40. After the first sampling event, NTB-OV updates the hit count for the automatically generated state bins auto[0:1] of state `m_y` (since `m_y` has a value of 0) and auto[0:1] of state `g_var1` (since `g_var1` has a value of 0). In turn, the corresponding cross bin (auto[0:1], auto[0:1]) of the `cc1` cross is hit. Each of these three bins has a hit count of 1.

After the second sampling event, NTB-OV updates the hit count for the automatic state bin auto[2:3] for sampled variable `m_y` (since `m_y` has a value of 2) with a hit count of 1, and increments the hit count of bin auto[0:1] of sampled variable `g_var1` with a hit count of 2. In turn, the corresponding cross bin (auto[2:3], auto[0:1]) of the `cc1` cross is also updated with hit count of 1.

Conditionals for Cross Level Definition

You can use conditional statements at the end of the cross level definitions in the form `if(NTB-OV_expression)`. You cannot call functions other than `get_cycle()` and `get_time()` in the conditional. When you attach a conditional to a cross level definition, NTB-OV considers all the bins for that cross only if the conditional evaluates to true (see Example 11-41).

Example 11-41   Cross Definitions

```
cross cc1(cp1, cp2) if (NTB-OV_expression);
   // cross with no user defined bins
```

```
cross cc1(cp1, cp2) if (NTB-OV_expression) {
   // cross with user defined bins
         state cc1_s1(binsof(cp1) intersect {2});
         bad_state cc1_b1(binsof(cp1) intersect {4});
}
```

If you use a global variable or class data member in a condition, NTB-OV uses the current value of the variable at the sampling time. There are no restrictions on the use of coverage point names in a conditional associated with any coverage point of the coverage group.

User-Defined Cross Coverage Bins

You can define and name your own cross coverage bins and use them to:

- Define illegal cross products
- Define cross products that should be ignored
- Combine cross products into a single named bin or equivalence class

You can use the NTB-OV expressions to concisely specify cross coverage bins. The syntax is:

```
cross cross_name (coverage_point, coverage_point...) {
      [cross_bin_definitions]
      [attribute_definitions]
}
```

`cross_name`

is the name of the cross. You can use this name to query or set information about the cross in the testbench.

`cross_bin_definitions`

use NTB-OV expressions to define named cross coverage bins. To define a cross coverage bin that combines several cross products into single named bin or equivalence class, use the following syntax:

```
state state_name ((select_expression)[logical_operator
(select_expression)]) [ if (expression) ];
```

`select_expression`

is a subset of `bins_expression`. The syntax is:

```
([!]binsof(bins_expression) [intersect {value_range_list}])
```

`bins_expression`

is the expression that identifies the bins to be specified for a cover point identifier:

```
cover_point_identifier [.bins_identifier]
```

`cover_point_identifier`

is the name of a cover point.

`bins_identifier`

is the name of bin.

`value_range_list`

is a list of value ranges.

`logical_operator`

is `&&` or `||`

If the expression for the state evaluates to true, NTB-OV increments the hit count of the cross coverage bin state name. You can optionally specify a conditional for the cross coverage state. In that case the bin is only hit only if the expression that defines the bin and the conditional evaluate to true.

To define cross products that should not be considered for coverage calculations, use the `ignored` keyword. The syntax is:

```
ignored state_name ((select_expression)[logical_operator
(select_expression)]) [ if (expression) ];
```

If the expression for the ignored cross coverage bin evaluates to true, NTB-OV skips the enclosing cross and does not update its bins. To define cross products that are illegal and should not occur, use the `bad_state` keyword. The syntax is:

```
bad_state state_name ((select_expression)[logical_operator
(select_expression)]) [ if (expression) ];
```

If the expression for the `bad_state` evaluates to true, NTB-OV issues a verification error.

`attribute_definitions`

You can use attributes to control various aspects of a *cross* (see “Attribute Definitions” on page 552). You specify an attribute’s value as follows:

```
attribute_name = value_expression;
```

where *`attribute_name`* is the name of the attribute, and *`value_expression`* is an NTB-OV expression. For more information, see “Expressions in Coverage Group Definitions” on page 496.

Wildcard Support in binsof Expressions

You can use wildcards (`x` and `z`) to specify ranges in the `binsof` expression.

Extensions to OpenVera Syntax

The following extensions have been made to the OpenVera® syntax:

```
cross cross_name (coverage_point, coverage _point,
[,coverage_point_name])
{
     [cross_bin_definitions]
     [attribute_definitions]
}
```

An optional keyword `wildcard` is included in the cross bin definition.

The *cross_bin_definition* defines a cross coverage bin that combines several cross products into a single named bin or equivalence class, using the following syntax:

```
[wildcard] state state_name
((select_expression)[logical_operator
(select_expression)]) [ if (expression) ];
```

The following definitions apply to the preceding syntax:

`wildcard`

Treats the `x` or `z` values as wildcards in the state declarations.

`state`

Can be `state`, `ignored`, or `bad_state`.

`state_name`

Represents a user-specified name for the cross bin.

`select_expression`

Represents a subset of `bins_expression`. See the *OpenVera Language Reference Manual: Native Testbench* for details on `bins_expression`.

`logical_operator`

Represents `&&` or `||`.

Understanding Wildcard Usage

By default, a bit with a value of `x` must match a sampled value of `x` for a cross bin counter to be incremented. The wildcard keyword treats the `x` value as a wildcard in the state declarations:

Note:
Here `wildcard` essentially means either `1` or `0` at the specified location.

The following code provides an example wildcard usage.

```
coverage_group cg {
  sample_event = @(posedge CLOCK);
  sample v_a {
     state a1 (0:3);
     state a2 (4:7);
     state a3 (8:11);
     state a4 (12:15);
  }
  sample v_b {
     state b1 (0);
     state b2 (1:8);
     state b3 (9:13);
     state b4 (14:15);
  }
```

```
  cross c (v_a, v_b) {
     wildcard state c1  (binsof(v_a) intersect {4'b11zx});
     wildcard state c2  (!binsof(v_b) intersect {4'b1x0x});
  }
}
```

The example above defines a coverage group named `cg` that samples its coverage points on the positive edge of signal `clk` (not shown). The coverage group includes two samples, one for each of the two 4-bit variables `v_a` and `v_b`.

Sample `v_a` defines four equal-sized bins for each possible value of variable `v_a`.

Sample `v_b` defines four bins for each possible value of variable `v_b`.

Cross definition `c` specifies the cross coverage of the two samples `v` and `v_b`. If the cross coverage of samples `v_a` and `v_b` were defined without any additional cross bins (select expressions), then cross coverage of `v_a` and `v_b` would include 16 cross products corresponding to all combinations of bins `a1` through `a4` with bins `b1` through `b4`, that is, cross products

- `a1,b1`
- `a1,b2`
- `a1,b3`
- `a1,b4`
- `...`
- `a4, b1`
- `a4,b2`

- `a4,b3`
- `a4,b4`

The first user-defined cross bin, `c1`, specifies that `c1` should include only cross products of sample `v_a` that intersect the value range of `1100` to `1111`. This select expression excludes bins `a1`, `a2`, and `a3`. Thus, `c1` will cover only four cross products of

- `a4,b1`
- `a4,b2`
- `a4,b3`
- `a4,b4`

This is similar to the behavior of the following code:

```
state c1 (binsof(a) intersect {[12:15]});
```

The second user-defined cross bin, `c2`, specifies that bin `c2` should include only cross products of samples `v_b` that do not intersect the value range of `1000` to `1001` and `1100` to `1101`. This select expression excludes bins `b2` and `b3`. Thus, `c2` covers only four cross products of

- `a1,b1`
- `a1,b4`
- `a2,b1`
- `a2,b4`
- `a3,b1`
- `a3,b4`

- a4,b1
- a4,b4

Precedence Semantics for User-Defined Cross Bins

You can have several user-defined cross coverage bins of each kind (that is, `state`, `bad_state`, and `ignored`). When a cross construct includes one or more `state`, `bad_state`, and `ignored` bins, NTB-OV uses the following precedence semantics:

- If any of the illegal bins (`bad_state`) is matched (expression evaluates to true), NTB-OV issues a verification error.
- Otherwise, if any of the ignored bins (`ignore`) is matched, NTB-OV does not update or add any bins and proceeds to the next cross (if any).
- Otherwise, NTB-OV updates the cross coverage bins (increments their counters) if their expressions evaluate to true. This applies to user-defined, non-illegal, and non-ignored cross coverage bins.
- Otherwise, NTB-OV updates the automatic cross product bin and the hit count of that bin if necessary. The bin is associated with the state and/or transition bins that are hit in each coverage point involved in the cross.

Example 11-42 illustrates the NTB-OV precedence semantics.

Example 11-42 Cross Coverage bin Precedence Semantics

```
extern i, j, k;
coverage_group cov1 {
      sample_event = @(posedge CLOCK);
      sample i, j, k;
      cross cc1 (i,j) {
            state MyCrossBin(binsof(i) intersect {0:4});
            ignoredIgnoredCrossProds(binsof(i)intersect{4:7}&&
             binsof(j) intersect {2:6});
```

```
            bad_state BadCrossProds((binsof(i) intersect {4}) &&
              (binsof(j) intersect {2}));
        }
    }
```

In Example 11-42, if i is 4 and j is 2, then the expressions for all three bins evaluate to true, so NTB-OV issues a verification error, and does not update the hit count for MyCrossBin. NTB-OV does not ignore the cross, even though the expression for IgnoredCrossProds also evaluates to true, because bad_states have the highest precedence.

If i is 4 and j is 5, the expressions for both IgnoredCrossProd and MyCrossBin (but not BadCrossProds) evaluate to true. In this case, NTB-OV gives higher precedence to the ignored cross coverage state, and skips the MyCross cross (the hit count for MyCrossBin is not updated).

If i is 4 and j is 8, then MyCrossBin is the only user-defined bin whose expression evaluates to true. so NTB-OV increments that bin's hit count.

Finally, if i is 5 and j is 7, then none of the user-defined bin expressions evaluate to true. In that case NTB-OV automatically updates the cross product bin (auto[5:5], auto[7:7]) for the MyCross cross and increments the hit count.

Examples of Embedded Coverage Groups with User-Defined Cross Coverage Bins

Example 11-43 shows an embedded coverage group that defines a cc1 cross for sampled variables m_y, m_z, and g_var1.

Example 11-43 Embedded Coverage Group with Cross Coverage

```
    class MyClass {
```

```
...
coverage_group cov1 {
      sample_event = @(posedge CLOCK);
      sample m_x, m_y, g_var1;

      cross cc1(m_y, m_z, g_var1) {
      bad_state bad1((binsof(m_y) intersect {0:12}) &&
      (binsof(m_z) intersect {0}) || (binsof(m_y) intersect
      {7:15}) && (binsof(g_var1) intersect {0}));

      ignored ig1 ((binsof(m_y) intersect {0:15}) &&
      (binsof(m_z) intersect {0})); }
}
}
```

Example 11-43 uses an NTB-OV expression to define an illegal (`bad_state`) cross product named `bad1`. If the expression evaluates to true when the cross is sampled, NTB-OV issues a verification error. An NTB-OV expression is also used to define the set of cross products that should be ignored by coverage. This is accomplished using the `ignore` statement in conjunction with an NTB-OV expression that evaluates to true when `m_y` is between the value 0 and 15, and `m_z` is 0. NTB-OV does not update any cross product bins when the `ignore` state is satisfied. If the illegal and ignored cross states are not satisfied, NTB-OV updates the hit count for the corresponding automatic cross bins.

In Example 11-44, an embedded coverage_group defines a cross named `cc1` for the `m_y, m_z`, and `g_var1` sampled variables.

Example 11-44 Embedded Coverage Group Cross for Sampled Variables

```
extern bit [0:4] g_var1;
class MyClass {
      bit [0:4] m_x, m_y;
      coverage_group cov1 {
            sample_event = @(posedge CLOCK);
            sample m_x, m_y, g_var1;
            cross cc1(m_y, m_z, g_var1) {
                  state s1 ((binsof(m_y) intersect {0}) &&
                  (binsof(m_z) intersect {2}));
            }
```

```
        }
    }
```

Using Example 11-44, at each positive edge of the system clock, if the sampled values of `m_y` and `m_z` are 0 and 2, respectively, NTB-OV increments the hit count for bin `s1` of the `cc1` cross (regardless of the sampled value for `g_var1`). For example, for following set of sampled values, NTB-OV increments state `s1`:

```
m_y = 0, m_z = 2, g_var1 = 7
```

When NTB-OV samples coverage points, if the expression associated with a user-defined cross state does not evaluate to true, it updates the corresponding automatically generated cross product bin for the state and/or transitions that it hit in each coverage point. In Example 11-44, when the sampled values of `m_y`, `m_z`, and `g_var1` are 1, 2, and 7,respectively, NTB-OV updates the automatic bin (auto[1:1], auto[2;2], auto[7:7]) for the cross product of `cc1`.

Valid Expressions for Defining Cross Coverage Bins

Although NTB-OV allows for the use of expressions in a cross coverage bin specification, there are certain kinds of expressions that are flagged as semantic errors by the compiler. The validity of expressions used in cross coverage bin specification is governed by the following three rules:

- Since tasks and functions can be blocking, you cannot use task or function calls.
- A sample name associated with a sampled expression can be used in the expression, provided it is in the list of samples for the cross.

Measuring Coverage

NTB-OV computes a coverage number or percentage for the testbench run as whole. Here, the coverage number is referred to as coverage. The coverage for the testbench is the weighted average of the coverages of every coverage group in the testbench. When per-instance data is available, NTB-OV also computes instance coverage for the testbench. That number is the weighted average of the coverages of every coverage group instance.

The `cov_weight` attribute of a coverage group determines the contribution of that group to the testbench coverage. For more information, see "Attribute Definitions" on page 552.

The coverage for each coverage group is the weighted sum of that group's sample and coverage numbers. The `cov_weight` attribute of a sample determines the contribution of that sample to the coverage of the enclosing coverage group. Similarly, the `cov_weight` attribute of a `cross` determines the contribution of that cross to the coverage of the enclosing coverage group. Both attributes have a default value of 1.

NTB-OV computes the coverage number for a sample as the number of bins with the `at_least` number of hits divided by the total number of possible bins for the sample (multiplied by 100). When the sample is auto-binned (that is, there are no user-defined state or transition bins), the total number of possible bins for the sample is the minimum of the `auto_bin_max` attribute for that sample and the number of possible values for the coverage point.

By default, NTB-OV does not create automatic bins for X or Z values of a coverage point. For example, if a coverage point is a 4-bit bit-vector and the `auto_bin_max` attribute is set to the default of 64, the total number of possible bins for the coverage point is 16 (2^4). On the other hand, if NTB-OV coverage is sampling the coverage point when it has X or Z values (`auto_bin_include_xz` attribute of the sample is set to `ON` or `1`), then the total number of possible bins for the 4-bit, bit-vector is 64 (MIN(auto_bin_max attribute, 4^4)). Finally, if the `auto_bin_max` attribute is set to 5, the total number of possible bins for the 4 bit, bit-vector is 5.

NTB-OV computes the coverage number of a `cross` as the number of bins of that cross with the `at_least` number of hits divided by the total number of bins for that cross (multiplied by 100). By default, the number of possible bins for a cross is the sum of the user-defined bins and the number of possible automatically generated bins for that cross. The number of possible automatically generated bins is the product of the number of possible bins for each of the samples being crossed.

Reporting and Querying Coverage Numbers

Testbench coverage is reported in the VCS coverage HTML and text reports. These reports also include detailed information for each coverage group as well as the samples and crosses of each group.

You can query the testbench coverage during the simulation run. This allows you to react to the coverage statistics dynamically (for example, stop the run when the testbench achieves a particular coverage). The following system function returns the cumulative coverage (an integer between 0 and 100) for the testbench:

```
function integer get_coverage();
```

The following system function returns the instance-based coverage (an integer between -1 and 100) for the testbench:

```
function integer get_inst_coverage();
```

Note that the `get_inst_coverage()` system function returns -1 when there is no instance-based coverage information, meaning the cumulative attribute of the coverage group was not set to 0.

Attribute Definitions

This section explains the attributes you can use as part of a coverage group specification. You can specify attributes at the group, sample, and cross levels. For a summary of coverage attributes, their types and default values, see Table 11-14. This table also indicates whether an attribute can be set at the coverage group, sample, or cross level. The syntax for an attribute definition is:

```
attribute_name = value_expression;
```

where *attribute_name* is the name of the attribute, and *value_expression* is an NTB-OV expression. For information on expressions allowed in coverage group definitions, see “Expressions in Coverage Group Definitions” on page 496.

When you set an attribute in a coverage group’s definition, NTB-OV evaluates the *value_expression* when the coverage group is instantiated. You can use coverage group attributes for two purposes:

- To control the behavior of a coverage group definition and its instances. For example, the `coverage_goal` attribute defines the coverage goal percentage for the coverage group.
- To provide a shorthand for setting the corresponding attribute for all samples and crosses of the coverage group that do not explicitly set that attribute.

In Example 11-45 on page 557 , the at_least attribute of the PacketCov coverage group is set to 2. Sample m_packetSize of that coverage group sets at_least to 5. Therefore, NTB-OV sets the m_packetSize sample's at_least attribute to 5, whereas for sample m_packetId and the MyCross cross, it sets at_least to 2.

NTB-OV does not force a particular order in the definition of coverage-group-level attributes and the sample and cross definitions. Similarly, NTB-OV does not force any particular order in the definition of sample or cross-level attributes and the definition of the sample or cross bins.

Coverage Group Attributes

The following sections explain the coverage group attributes.

at_least

specifies the minimum number of times a bin should be hit for it to be considered covered. The default is 1. You can set this attribute at the coverage group, sample, or cross levels. When used at the sample or cross level, at_least applies to all bins of the sample or cross. When used at the coverage group level it applies to all bins of all samples and crosses that do not explicitly set the attribute.

auto_bin_max

specifies the maximum number of automatically created bins for samples. The default is 64. You can set this attribute at the coverage group or sample level. When used at the coverage group level, it applies to all samples in the coverage group that do not explicitly set the attribute.

bin_activation

defines the default active or inactive state for user-defined bins of a coverage construct. NTB-OV evaluates the value of the expression when the coverage group containing the expression is instantiated. If the expression evaluates to `0` (`OFF`), NTB-OV assumes that all bins for the construct (coverage group, sample, or cross) containing the attribute are inactive for the current and all subsequent instantiations of the coverage group. If the expression evaluates to a non-zero value (`ON`), NTB-OV assumes that all bins for the construct (coverage group, sample, or cross) containing the attribute are active for the current and all subsequent instantiations of the coverage group. The default is `ON`.

collect

turns data collection on or off.The default is `ON` (or `1`). You can set the collect attribute at the coverage group, sample, or cross levels. When set at the coverage group level, it applies to all samples in the coverage group that do not explicitly set the attribute.

cov_comment

specifies user-defined comments (character string literals) in the coverage group, sample, and cross constructs. You can use the comment as a mnemonic device to help interpret coverage results organize coverage results in custom ways. The syntax is:

```
cov_comment = string_literal;
```

`string_literal`

is character string enclosed in double quotes. Note that you cannot use an NTB-OV string variable to initialize this attribute.

cov_weight

When specified for a coverage group, this attribute affects how the coverage group contributes to the overall testbench coverage number. When specified for a sample or cross, it affects how the sample or cross contributes to the enclosing coverage group's coverage number. The default is 1. Setting this attribute at the coverage_group level does not affect the value of the `cov_weight` attribute for any sample or crosses of the coverage group.

coverage_goal

When specified for a coverage group, this attribute designates the desired coverage percentage for the group. The default is 100 (percent). When specified at the sample or cross level, this attribute specifies the desired coverage percentage for the sample or cross. Setting `coverage_goal` at the coverage group level does not affect the value of the coverage goal attribute for the sample and crosses of the coverage group. The `coverage_goal` attribute retains its default value for the sample and crosses where coverage goal is not explicitly set.

cumulative

You can accumulate coverage data on a per-instance basis for a coverage group or cumulatively across all coverage group instances. Use the `cumulative` attribute to make this choice. The default is `ON.` In this case, NTB-OV accumulates data on a cumulative basis only. If the value is `OFF`, NTB-OV accumulates data on a per-instance and cumulative basis.

overlap_state

specifies whether NTB-OV checks for states with overlapping values. When set to ON (or 1), NTB-OV prints a warning if any two states of a sample have overlapping values. The default value is OFF (or 0). You can set the overlap_state attribute at the coverage group or sample level. When set at the coverage group level, it applies to all samples in the coverage group that do not explicitly set the attribute.

overlap_trans

specifies whether NTB-OV checks for trans constructs that define overlapping transitions. When set to ON (or 1), NTB-OV prints a warning if any two trans constructs of a sample define overlapping transitions. The default is OFF (or 0). You can set the overlap_trans attribute at the coverage group or sample level. When set at the coverage group level, it applies to all samples in the coverage group that do not explicitly set the attribute.

auto_bin_include_xz

specifies whether NTB-OV should consider X and Z values for an automatically binned coverage point. The default is OFF (or 0). In this case, NTB-OV does not create a bin for any X or Z values of the coverage point. The NTB-OV functional coverage group treats the coverage point as having 2^N values (where N is the number of bits for a bit-vector or 32 for an integer). When the auto_bin_include_xz attribute is set to ON (or 1), NTB-OV creates a bin for the X or Z values of the coverage point. In this case, the NTB-OV functional coverage group treats bit-vector coverage points as having 4^N values (where N is the number of bits), and integer coverage points as having $2^{32} + 1$ values (-2^{31} to 2^{31}, and X). You can set the auto_bin_include_xz attribute at the coverage group or sample level. When set at the coverage group level, it applies to all samples in the coverage group that do not explicitly set the attribute. Note that this attribute only applies to coverage points that are automatically binned. Example 11-45 shows an example.

Example 11-45 Coverage Group Attributes

```
class Packet {
bit [7:0] m_packetSize;
bit [7:0] m_packetId;
coverage_group PacketCov {
      at_least = 2;
      sample m_packetSize {
            at_least = 5;
      }
      sample m_packetId;
      cross MyCross (m_packetSize, m_packetId);
      sample_event = @(posedge CLOCK);
}
```

Table 11-14 summarizes the coverage attributes, their types, and default values. It also indicates whether an attribute can be set at the coverage_group, sample, or cross level.

Table 11-14 Coverage Attributes and Related Constructs

Attribute Name	Type	Default Value	Defined in coverage group?	Defined in sample?	Defined in cross?
`at_least`	integer (>=1)	1	Yes	Yes	Yes
`auto_bin_max`	integer	64	Yes	Yes	No
`collect`	boolean	ON	Yes	Yes	Yes
`cov_weight`	integer (>=0)	1	Yes	Yes	Yes
`coverage_goal`	integer (0-100)	90	Yes	Yes	Yes
`cumulative`	boolean	ON	Yes	No	No
`overlap_state`	boolean	OFF	Yes	Yes	No

Table 11-14 Coverage Attributes and Related Constructs

Attribute Name	Type	Default Value	Defined in coverage group?	Defined in sample?	Defined in cross?
`overlap_trans`	boolean	OFF	Yes	Yes	No
`auto_bin_include_xz`	boolean	OFF	Yes	Yes	No
`bin_activation`	integer	1	Yes	Yes	Yes

Coverage Group Instances with Different Attribute Values

Each instance of a coverage group can have a different value for a particular attribute. For example, the first instance of a coverage group can set the `at_least` attribute to 5, while the second instance of the same coverage group sets it to 3.

However, NTB-OV enforces that all instances of a coverage group have the same setting for the cumulative attribute. The first instance of the coverage group determines the value of the attribute even if other instances have a different setting.

When different instances of a coverage group have different values for the `at_least`, `auto_bin_max`, `coverage_goal`, or `cross_bin_max` attributes, NTB-OV uses the largest value of each attribute for computing cumulative coverage statistics (that is, coverage numbers for the coverage group). When different instances of a coverage group have different values for the `cov_weight` attribute, NTB-OV uses the smallest value of that attribute for computing cumulative coverage statistics for the coverage group.

Predefined Tasks and Functions

You can invoke predefined coverage group methods on an instance of a coverage group. The methods follow the same syntax as when you are invoking class functions and tasks on an object. For more information, see “Predefined Coverage Group Tasks and Functions” on page 289.

Loading Coverage Data

You can load cumulative coverage and instance-specific coverage data. Loading coverage data from a previous simulation run adds bin hits from the previous run to the current run.

Runtime Access to Coverage Results

You can use the `query()` function to monitor functional coverage group statistics dynamically during simulation, and dynamically react to those coverage results. You can invoke `query()` on a coverage group instance, or a sample or cross of a coverage group instance. For more information, details see “Reporting and Querying Coverage Numbers” on page 300 .

Instance Names

Coverage objects are named so that they can be identified in the coverage reports. By default, the automatically generated name is based on the variable name of the NTB-OV coverage object. In Example 11-46, the name of the coverage instance is `cov1`. The full name is `covType:cov1`.

Example 11-46   Coverage Instance Auto-name

```
coverage_group covType {
      sample_event = @(posedge CLOCK);
      sample gVar;
}
...
covType cov1 = new();
```

For an array of coverage objects, the names generated for the coverage instances include the array index. In Example 11-47, the coverage instances are named `covInst[0]` and `covInst[1]`.

Example 11-47   Coverage Instance Name with Array

```
coverage_group covType(sample integer thisVar) {
      sample_event = @(posedge CLOCK);
      sample thisVar;
}
...
covType covInst[2];
covInst[0] = new(localVar1);
covInst[1] = new(localVar2);
```

User-Specified Names

NTB-OV creates coverage instances dynamically during the simulation run. In some testbenches, a large number of coverage instances may be created. For greater control, you can explicitly name the coverage objects. The syntax for stand-alone coverage groups is:

```
coverage_instance.set_name("user_specified_name");
```

The syntax for an embedded coverage group is:

```
class_instance.coverage_group_name.set_name("user_
      specified_name");
```

Coverage Shapes and Meaningful Shape Names

The concept of "shapes" is important to detect the instances of a coverage_group that have different parameters passed in to them. For example, consider two instances of the embedded coverage_group cg called w1.cg and w2.cg. Although both are instances of the same coverage_group cg, w1.cg and w2.cg could have been constructed with, different parameters being passed in, making the two instances different. When these two instances are merged for coverage reporting, NTB-OV keeps them separate, as they may have differences, such as different bins or bin value ranges, as you can see in the example below.

Example 11-48 Coverage Shapes

```
class W {
      rand bit [3:0] addr;
      rand bit [3:0] resp;
      coverage_group cg(bit [3:0] lower, bit [3:0] upper) {
            sample_event = @(posedge CLOCK);
            sample resp {
                  m_state(lower:upper);
            }
            sample addr {
                  state state_of_interest(lower:upper);
            }
      }
      task new(bit [3:0] lower, bit [3:0] upper) {
            cg = new(lower,upper);
      }
      task display(integer id = -1) {
            printf("%d -> \t%h \t%s \n", id, addr, resp);
      }
}
program prog {
      W w1;
      W w2;
      reg [3:0] u,l;
      l = 4'h0;
      u = 4'h3;
      printf("Creating inst w1 with lower=%0h, upper=%0h\n", l, u);
      w1 = new(l, u);
```

```
        l = 4'h4;
        u = 4'h7;
        printf("Creating inst w2 with lower=%0h, upper=%0h\n", l, u);
        w2 = new(l, u);
        @(posedge CLOCK);
        repeat(1) {
              void = w1.randomize() with {addr == 4'h6;};
              void = w2.randomize() with {addr == 4'h6;};
              w1.display(1);
              w2.display(2);
              @(posedge CLOCK);
        }
  }
```

In Example 11-48, the parameters `upper` and `lower` are passed into coverage_group cg in class W. These two parameters are used to:

- Define bins to automatically create in m_state
- Create state bin state_of_interest.

The coverage group cg tracks the values of both addr and resp. When class W is constructed, two parameters are passed in. These parameters are used to construct an instance of cg, where the parameters dictate what values to count as hit for state state_of_interest. w1.cg is constructed with 4'h0:4'h3 as the range for state_of_interest, whereas w2.cg is constructed with 4'h0:4'h3 as the range for state_of_interest. Although both w1 and w2 are instances of the same class, the shapes feature keeps track of the fact that they are different from each other.

Coverage Report With Shapes and Meaningful Shape Names

```
Group : W::cg::SHAPE{lower=0,upper=3}
==================================================================
Score   Weight  Goal
0.00  1       100
------------------------------------------------------------------
```

```
Summary for variable resp
                 Expected Covered Percent
User Defined Bins 4        0       0.00
User Defined Bins for resp
Uncovered bins
name            count at least
s_lower_upper_0 0     1
s_lower_upper_1 0     1
s_lower_upper_2 0     1
s_lower_upper_3 0     1
-----------------------------------------------------------------
Summary for variable addr
                  Expected Covered Percent
User Defined Bins 1        0       0.00
User Defined Bins for addr
Uncovered bins
name              count at least
state_of_interest 0     1

Group : W::cg::SHAPE{lower=4,upper=7}
=================================================================
Score   Weight  Goal
 50.00  1       100

Summary for variable resp
                 Expected Covered Percent
User Defined Bins 4        0       0.00
User Defined Bins for resp
Uncovered bins
name            count at least
s_lower_upper_4 0     1
s_lower_upper_5 0     1
s_lower_upper_6 0     1
s_lower_upper_7 0     1

-----------------------------------------------------------------
Summary for variable addr
                  Expected Covered Percent
User Defined Bins 1        1       100.00
User Defined Bins for addr
Bins
name              count at least
state_of_interest 1     1
```

Coverage Shape Creation During Simulation

Instantiating a covergroup or a class that contains a covergroup does not trigger the creation of a coverage shape and once the shape is created, it is not allowed to change. The following rules in NTB-OV determine whether a coverage shape is created:

The following events would typically lead to the creation of a coverage shape if it does not exist:

- Sampling event for a coverage group occurs
- Simulation ends
- Loading an existing coverage database.
- Call to a query function

Coverage Shapes in Instance Merging

The concept of "shapes" was introduced to match identical instances of coverage groups so that merging yields useful results. When instances of coverage groups are to be merged, a mechanism is needed not only to ensure that the targeted instances are of the same coverage group, but also that the instances have the same parameter(s), since instances of the same coverage group can be instantiated with different parameters. There are two scenarios:

- Instances are in the same testbench run.
- Instances are in different testbench runs.

Example 11-49 Instances

```
#define UPPER 4'h7
#define LOWER 4'h0
class W {
```

```
        rand bit [3:0] addr;
        rand bit [3:0] resp;
        coverage_group cov0(bit [3:0] lower, bit [3:0] upper){
              sample_event = @(posedge CLOCK);
              sample resp;
              sample addr;
              cross cc1 (resp, addr) {
              state cross_low_range(addr >= lower && addr <= upper);
              }
              cumulative = 1;
        }

        task new(bit [3:0] lower, bit [3:0] upper) {
              cov0 = new(lower,upper);
        }
        task display(integer id = -1) {
              printf("%d -> \t%h \t%s \n", id, addr, resp);
        }
  }
  program prog {
        W w1, w2;
        w1 = new(LOWER, UPPER);
        w2 = new(LOWER+8, UPPER+8);
        @(posedge CLOCK);
        void = w1.randomize() with {addr == 4'h6;};
        w1.display(1);
        void = w2.randomize() with {addr == 4'h6;};
        w2.display(2);
        @(posedge CLOCK);
  }
```

Example 11-49 has two instances of W (w1 and w2). Coverage group .cov0. was instantiated with different parameters in w1 and w2. The coverage results for w1 and w2 are found in W::cov0_SHAPE_(lower=8, upper=15) and W::cov0_SHAPE_(lower=0, upper=7), respectively.

```
==============================================================
Group : W::cov0::SHAPE{lower=8,upper=15}
==============================================================
Summary for cross cc1
Samples crossed: resp addr
                                   Expected Covered Percent Missing
Total                              129      1       0.78    127
Automatically Generated Cross Bins 128      1       0.78    127
```

```
User Defined Cross Bins             1        0       0.00
Automatically Generated Cross Bins for cc1
Uncovered bins
resp                 addr                 count at least
[auto[0] - auto[2]]  [auto[0] - auto[7]] --    1        (24 bins)
[auto[3]]            [auto[0] - auto[5]] --    1        (6 bins)
[auto[3]]            [auto[7]]           0     1
[auto[4] - auto[15]] [auto[0] - auto[7]] --    1        (96 bins)

Covered bins
resp    addr    count at least
auto[3] auto[6] 1     1
User Defined Cross Bins for cc1
Uncovered bins
name            count at least
cross_low_range 0     1


==============================================================
Group : W::cov0::SHAPE{lower=0,upper=7}
==============================================================
Summary for cross cc1
Samples crossed: resp addr
                                   Expected Covered Percent Missing
Total                              129      1       0.78    128
Automatically Generated Cross Bins 128      0       0.00    128
User Defined Cross Bins            1        1       100.00

Automatically Generated Cross Bins for cc1
Uncovered bins
resp                 addr                  count at least
[auto[0] - auto[15]] [auto[8] - auto[15]] --    1        (128 bins)
User Defined Cross Bins for cc1
Bins
name            count at least
cross_low_range 1     1
```

Cumulative and Instance-based Coverage

Coverage statistics can be gathered both cumulatively and on a per-instance basis. Cumulative implies that coverage statistics (that is, bin hit counts and coverage numbers) are computed for the coverage_group definition. In this case all instances of the coverage_group contribute to a single set of statistics maintained for the coverage_group definition. By default, NTB-OV computes cumulative coverage information.

An example where cumulative coverage is very useful is when covering a packet class and the cumulative coverage information for all packets (instances) is of interest.

NTB-OV also supports computation of per-instance coverage statistics. In this case NTB-OV computes coverage statistics for every instance of a coverage_group as well as the coverage_group definition as a whole. NTB-OV computes per-instance coverage statistics when the cumulative attribute of the coverage_group is set to OFF.

It should be noted that in cumulative mode only cumulative can be queried for. Furthermore, the coverage reports only report on cumulative data for the coverage group definitions, and not instances.

Activation/Deactivation: User-defined Bins

The NTB-OV functional coverage engine assumes that all user-defined bins (state and transition) in a coverage_group definition are of interest to the user, therefore the coverage data is computed and reported for all user-defined bins.

By default, all user defined bins are considered active. NTB-OV functional coverage has provided an attribute based mechanism for setting all user-defined bins in a coverage construct as inactive. Further, they also provide two built-in functions that select a subset of user defined bins for a coverage construct to be active or inactive.

The attribute is called “bin_activation,” and the built-in functions are `set_bin_activation()` and `inst_set_bin_activation()`.

The bin_activation attribute defines the default active/inactive state for user defined bins of a coverage construct. As with other coverage attributes, this attribute can be set for a coverage_group, sample or cross construct.

The syntax for setting this attribute is:

```
bin_activation = value_expression;
```

`value_expression`

  is an OpenVera expression.

The value of the expression will be evaluated when the coverage group containing the expression is first instantiated.

If the expression evaluates to 0 (OFF), the NTB-OV coverage engine assumes that all bins for the construct (coverage_group, sample, cross) containing the attribute will be inactive for the current instance and all subsequent instantiations of the coverage group.

If the expression evaluates to a non-zero value (ON), the NTB-OV coverage engine assumes that all bins for the construct (for example, coverage_group, sample, cross) containing the attribute will be active for the current instance and all subsequent instantiations of the coverage group.

The default active/inactive state of user defined bins can be overridden using the functions, `set_bin_activation()` and `inst_set_bin_activation()`.

`set_bin_activation()`

This function call activates/deactivates a user-defined state/transition bin for a coverage shape as well as the instance. The bin hit count is not updated for shape as well as instance if the bin is deactivated but is updated if the bin is reactivated. In the new implementation even if the bin is deactivated it contributes to the coverage calculations.

Syntax

```
integer function inst_set bin_activation(integer command,
          [integer bin_type[, string bin_name_pattern]]);
```

`command`

Is either OFF (for deactivating a bin) or ON (for reactivating a bin).

`bin_type`

Is an optional argument, which can be any of the valid bin types: STATE, BAD_STATE, TRANS, BAD_TRANS. The bin_type argument controls the set of bins on which the command is to be applied. Multiple bin types can be specified using the ‘or’ operator (|). If the *bin_type* is not specified, the command is applied to all bins. If specified, the *bin_type* argument must follow the command argument.

`bin_pattern`

Is an optional argument that is used to further control the set of bins upon which the command is applied. It can be any Perl regular expression. Only those user defined bins of type *bin_type*, whose names match the *bin_pattern* are affected by the command. If no *bin_pattern* is specified, then the command is applied to all bins of the type *bin_type*. If the bin_pattern is specified, then it must follow the command and bin_type arguments.

When a user-defined state/transition bin for a coverage_group definition is deactivated, their instance and cumulative hit counts are not updated, but the bins are considered when computing instance and cumulative coverage numbers for the affected instance. Note that if bins are deactivated using this command, subsequent instantiations of the coverage group will also have the bins deactivated.

If a particular instance of the coverage_group needs to be activated or deactivated, a separate function `inst_set_bin_activation()` can be used.

`inst_set_bin_activation()`

This function call activates/deactivates a user-defined state/transition bin for an instance The bin hit count is not updated for the instance if the bin is deactivated but is updated if the bin is reactivated. In the new implementation even if the bin is deactivated it contributes to the coverage calculations for the instance.

Syntax

```
integer function inst_set_bin_activation(integer command,
          [integer bin_type[, string bin_name_pattern]]);
```

`command`

is either OFF (for deactivating a bin) or ON (for reactivating a bin).

`bin_type`

Is an optional argument, which can be any of the valid bin types: STATE, BAD_STATE, TRANS, BAD_TRANS. The *bin_type* argument controls the set of bins upon which the command is to be applied. Multiple bin types can be specified using the ‘or’ operator (|). If the bin_type is not specified, the command will be applied to all bins. If specified, the *bin_type* argument must follow the command argument.

`bin_pattern`

Is an optional argument that is used to further control the set of bins upon which the command is applied. It can be any Perl regular expression. Only those user defined bins of type *bin_type*, whose names match the *bin_pattern* are affected by the command. If no *bin_pattern* is specified, then the command is applied to all bins of the type *bin_type*. If the bin_pattern is specified, then it must follow the command and *bin_type* arguments.

When a user-defined state/transition bin for a coverage_group instance is deactivated, their instance and the cumulative hit counts for the inactive bin is not updated. But the inactive bin is considered when computing instance-based coverage numbers, and the inactive bin reflects in the instance-based coverage report for the coverage_group instance.

Including/Excluding: User-Defined Bins

In previous versions of VCS, you could use the function calls set_bin_activation () and inst_set_bin_activation() to activate/ deactivate user-defined state/transition bins for a cumulative at the coverage definition level or an instance of the cover group definition.

If you deactivate a user-defined state/transition bin from a coverage_group NTB-OV does not consider the bin for either cumulative coverage calculation or instance coverage calculation for the instance for which the function was invoked. Instance and cumulative hit counts are not updated. Even if the bin of an instance is deactivated at the end of a simulation, it is not included in the coverage report.

The functional coverage flow with coverage shapes does not allow this behavior. A coverage group can have multiple shapes and there can be several instances having the same shape. There are several factors that decide the shape of an instance. One such factor is the user-defined states. Shape calculation for an instance takes place once when the first sample event is encountered after the instantiation. If you were to turn a bin OFF (after the first sampling event i.e. after the shape had been finalized) using the functions mentioned earlier, the bin would not contribute to the coverage numbers, thereby changing the shape of the instance making it inconsistent with the previously computed shape for that instance. This results in inconsistent behavior. To avoid this inconsistent behavior, use the `inst_set_cov_shape()` function.

`inst_set_cov_shape()`

This function includes/excludes a user-defined state/transition bin for a coverage shape as well as the coverage instance on which the function is invoked. This function call for an instance would override its previous calls if they were called on the same bin. The function calls for a particular instance have meaning until the shape for that instance is finalized. Once the shape is created/finalized for that instance, any calls on the instance lead to a runtime warning and the shapes are not affected by the calls.

Note:

> This function has to be called before the creation of the shape. Events that declare the creation of shape are discussed later.

Syntax

```
integer function inst_set_cov_shape(integer command,
      [integer bin_type[, string bin_name_pattern]]);
```

`command`

The command argument can be either OFF (for deactivating a bin) or ON (for reactivating a bin).

`bin_type`

The bin_type argument controls the set of bins on which the command is to be applied. Multiple bin types can be specified using the 'or' operator (|).This is an optional argument, which can be any of the valid bin types: STATE, BAD_STATE, TRANS, and BAD_TRANS. If the bin_type is not specified, the command is applied to all bins. If specified, the bin_type argument must follow the command argument.

`bin_name_pattern`

The bin_name_pattern argument is an optional argument that is used to further control the set of bins upon which the command is applied. It can be any Perl regular expression. Only those user defined bins of type bin_type, whose names match the bin_pattern are affected by the command. If no bin_pattern is specified, then the command is applied to all bins of the type bin_type. If the bin_pattern is specified, then it must follow the command and bin_type arguments.

When a user-defined state/transition bin for a coverage_group definition is deactivated, they are completely excluded and not considered for coverage reporting and coverage calculation.

For rules deciding the creation of shapes see “Coverage Shape Creation During Simulation” on page 564.

Important Behavior

- Function call inst_set_cov_shape( ) can be called on an instance handle before the coverage shape is created. If called after the coverage shape is created for that instance, a runtime warning is issued and the function call is ignored.
- You can call the function inst_set_bin_activation( ) on an instance handle before or after NTB-OV creates the coverage shape.
- You can call the the function set_bin_activation( ) on an instance handle after NTB-OV creates the coverage shape. This function call has no meaning until the shape of that particular instance is created.

Example 11-50 illustrates the use of the `set_bin_activation()` and `inst_set_bin_activation( )` functions. These functions do not affect the coverage shape of the instance but only decide whether bin hit counts have to be updated for that instance and the corresponding shape.

Example 11-50 Set Bin Functions

```
 Coverage_group Cov{
    Sample  {
       State s1;
       State s2;
       State s3;
       State s4;
       State s5;
    }
    Sample_event = …;
  }

Program MyProg {
  Cov cov1=new, cov2=new, cov3=new;

  Cov1.inst_set_cov_shape(OFF, STATE, s1);
  Cov2.inst_set_cov_shape(OFF, STATE, s3);

  Cov1.inst_set_bin_activation(OFF, STATE, s2);
  Cov2.inst_set_bin_activation(OFF, STATE, s1);
```

```
     @(posedge CLOCK); //shape is created for the instances

     Cov1.set_bin_activation(OFF, STATE, s3);
     Cov1.inst_set_bin_activation(ON, STATE, s2);
     ...
  }
```

In this example, Cov1 has s1 excluded from the shape and Cov2 has s3 excluded from the shape. Cov1 has s2 deactivated which means bin hits are not collected for this instance. Cov2 has s1 deactivated in the similar manner. On seeing the sample event NTB-OV creates the coverage shape for the instances.

Once the coverage shape is created, the function set_bin_activation is called on the instance Cov1 for the bin s3. This will deactivate s3 for all the instances of the same shape as that of Cov1.

Example 11-51

```
 Coverage_group Cov{
     Sample  {
          State s1;
          State s2;
          State s3;
          State s4;
          State s5;
     }
     Sample_event = @(posedge CLOCK);
}
Program MyProg {
     Cov cov1=new, cov2=new, cov3=new;

     Cov1.set_bin_activation(OFF, STATE, s2);
     Cov2.set_bin_activation(OFF, STATE, s1);

     @(posedge CLOCK); //shape is created for the instances
     …….
}
```

In the example, the calls to `set_bin_activation` would result in a warning because no shape is created for that instance. Only instance-specific calls have meaning before shape creation.

Example 11-52

```
 Coverage_group Cov{
      Sample  {
            State s1;
            State s2;
            State s3;
            State s4;
            State s5;
      }
      Sample_event = @(posedge CLOCK);
}
Program MyProg {
      Cov cov1=new, cov2=new, cov3=new;

      Cov1.inst_set_cov_shape(OFF, STATE, s1);
      Cov2.inst_set_bin_activation(OFF, STATE, s1);

      @(posedge CLOCK); //shape is created for the instances
      …….
}
```

This example shows that Cov2 and Cov3 have the same shape and Cov1 has a different shape.

12

Direct Programming Interface (DPI)

DPI is an interface between OpenVera Native Testbench (NTB-OV) and another programming language. In the current DPI implementation, C is the only supported language (an extern C statement enables C++ code). This chapter explains how to use the DPI in the following sections:

Overview

DPI allows you to call C functions in your NTB-OV code and call NTB-OV tasks and functions in your C code. Execution of such tasks and functions take zero simulation time. There is no synchronization mechanism between the layers. You can pass only valid NTB-OV data types through the interface. C language functions called in NTB-OV are referred to as import functions. NTB-OV tasks and functions called in C are referred to as export tasks and functions.

For a complete description of the interface, see the *SystemVerilog 3.1a Language Reference Manual*, Section 27, Direct Programming Interface (DPI), and Annex E, DPI C-Layer.

Mapping Between NTB-OV and C DPI Data Types

Table 12-1 shows the NTB-OV types supported with the DPI and their corresponding C types.

Table 12-1 NTB-OV and C DPI Data Types

NTB-OV Data Types	C DPI Data Types
`[var] integer`	`int*, int`
`[var] reg`	`svScalar*, svScalar`
`[var] reg vector`	`svLogicVec32*, svLogicVec32*`
`[var] string`	`char**, char*`
`[var] dynamic array`	`svOpenArrayHandle`
`[var] fixed size SDA`	`type[]`
`[var] fixed size MDA`	`type[]`

In Table 12-1:

- SDA stands for single-dimensional array.
- MDA stands for multidimensional array. Multidimensional arrays are passed to C as single-dimensional arrays.
- `svLogicVec32*` is defined as:

```
typedef struct {
        unsigned int c; // control
        unsigned int d; // data
    } svLogicVec32;
```

The reg data type is considered to be 2-state in the return value of a import DPI function. Everywhere else, reg is considered to be 4-state. The reg encoding as a 4-state value is as follows:

Table 12-2 4-State

	c	d
0	0	0
1	0	1
z	1	0
x	1	1

- `svScalar` is defined as an unsigned `char`. The value for `Z` in C is `sv_z` and the value for `X` is `sv_x`.
- the `svOpenArrayHandle` function is defined as `void*`.
- The `svScalar`, `svLogicVec32`, and `svOpenArrayHandle` data types are defined in the `svdpi.h` file. You must include this header file in your C code to use these data types:.

```
#include <svdpi.h>
```

The DPI passes all arrays by reference even when you specify them as non-var arguments (except integer and string arrays). Therefore, if a non-var array argument changes on the C side, DPI propagates the change to the NTB-OV side. You must ensure that an input array argument does not change on the C side. For integer and string arrays passed as non-var arguments, DPI does not propagate changes made on the C side to the NTB-OV side.

Note:

Associative arrays and Smart Queues are not supported in either export or import declarations. Dynamic arrays are not supported in export declarations, but are supported in import declarations.

When you pass an array as an argument, you must also specify the size of the array (see Example 12-1).

Example 12-1 Specifying Size of an Array

```
// NTB-OV code example:
import "DPI" function void foo(integer i[4], integer size);

// Corresponding C code:
void foo(int *pInt, int size);
```

As with arrays, when you pass a reg vector as an argument to a DPI import function, you must also specify the width of the reg vector.

You pass arrays of reg vectors to C as arrays of elements of type `svLogicVec32`. The DPI represents each element as a number of groups of type `svLogicVec32`. For example, if you have an array with four elements of `type reg[74:0]`, the C representation of this is an array of `svLogicVec32` with 3*4 elements.

You pass dynamic arrays as open arrays on the C side. You can use array querying functions to get the size and elements of an open array on the C side.

Calling C Functions from NTB-OV

To call C functions from NTB-OV using the DPI, follow these steps:

1. Declare the C DPI functions
2. Call the C DPI functions
3. Compile the NTB-OV and C code using the `-ntb` option:

```
% vcs -ntb file.c file.vr
```

4. Run the simulation (`simv`).

DPI Function Declarations

To call C functions from NTB-OV, declare them in the NTB-OV code as follows:

```
import "DPI" function  type  func_name (argument_list);
```

`import “DPI”`

are keywords that declare the function is a DPI function.

`type`

valid return types are integer, string, reg, and reg vector with no more than 32 bits (also, void bit vectors less than 32 bits wide).

`func_name`

is the name of the C function called.

`argument_list`

pass arguments by value or by reference (var). Fixed size and dynamic arrays are both supported.

Example 12-2 shows an import DPI declaration and corresponding C declaration.

Example 12-2 NTB-OV calls C Task Passing in integer, reg and string by Value

```
// NTB-OV code
import "DPI" function void vtask0(integer i, reg b, reg[3:0] bv,string s);

// C code
void vtask0(int i, svScalar b, svLogicVec32 *bv, char *s);
```

Example 12-3 shows an import DPI declaration and corresponding C declaration. This one shows returning a reg value.

Example 12-3 NTB-OV calls C Function Passing in integer, reg and string by Value, Returning a reg Value

```
// NTB-OV code
import "DPI" function reg cfunc(integer i, reg b, reg[3:0] bv, string s);

// C code
svScalar cfunc(int i, svScalar b, svLogicVec32 *bv, char *s);
```

Example 12-4 shows an import DPI declaration and corresponding C declaration. This one shows returning a reg vector.

Example 12-4 NTB-OV calls C Function Passing in integer, reg and string by Reference, Returning a Small reg Vector

```
// NTB-OV code
import "DPI" function reg[31:0] cfunc(var integer i, var reg b,var
reg[3:0] bv, var string s);

// C code
svScalar cfunc(int *i, svScalar *b, svLogicVec32 *bv, char **s)
```

In Example 12-4, the `var` modifier for the `reg[3:0]` bit vector is optional. If you omit it, the DPI still treats it as a call-by-reference variable.

Example 12-5 shows a complete example with an array of reg vectors.

Example 12-5 Complete Example with Array of reg Vectors

bv_array.c File

```
#include <svdpi.h>
void print_bv_array(svLogicVec32 *bv_array, int array_size, int
elem_bit_width) {
      /* per each array's element */
      int chunks = SV_CANONICAL_SIZE(elem_bit_width);

      /* #bits in the last chunk */
      int n = (elem_bit_width&31 ? elem_bit_width&31 : 32);
      int i, j;
      int c, d;
      printf("C values:\n");

      for (i = 0; i < array_size; i++) {
            /* get most significant bits (the last chunk), apply
             masking */
            c=SV_GET_UNSIGNED_BITS(bv_array[i*chunks+chunks-1].c, n);
            d=SV_GET_UNSIGNED_BITS(bv_array[i*chunks+chunks-1].d, n);

            /* ignore control bits, print only data bits */
            printf(" bv_array[%d].d = %0x", i, d);

            /* go through remaining chunks, from more- to less-
             significant bits */
            for (j = chunks-2; j >= 0; j--) {
                  /* full 32-bits, no masking needed */
                  d = bv_array[i*chunks+j].d;
                  printf("%08x", d);
               }
            printf("\n");
      }
}
```

bv_array.vr File

```
import "DPI" function void print_bv_array(reg[74:0] im[4], integer
array_size, integer bv_elem_size);
program test {
      integer i;
      reg[74:0] im[4];

      im[0] = 74'h1030000000700000009;
      im[1] = 74'h0033000007700000b0a;
      im[2] = 74'h0230000300700000109;
      im[3] = 74'h023000030070000010a;

      print_bv_array(im, 4, 75);
      printf("\nNTB-OV values: \n");

      for (i = 0; i < 4; i++) {
            printf(" im[%d] = %h\n", i, im[i]);
      }
}
```

Here is the command line to compile Example 12-5:

```
% vcs -R -ntb bv_array.c bv_array.vr
```

Example 12-5 uses the `SV_CANONICAL_SIZE` and `SV_GET_UNSIGNED_BITS` macros, which are defined in the svdpi.h file as:

```
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)
#define SV_MASK(N) (~(-1<<(N)))
#define SV_GET_UNSIGNED_BITS(VALUE,N)
      ((N)==32?(VALUE):((VALUE)&SV_MASK(N)))
```

Example 12-6 shows a complete example using a dynamic array.

Example 12-6   Complete Example using Dynamic Array

dynamicArray.c File

```
#include <svdpi.h>
void foo(const svOpenArrayHandle h) {
      int dim, size, i;
```

```
      svScalar sc;
      void *hh = svGetArrayPtr(h);
      if (hh) {
            printf("pointer to array is OK\n");
      }
      else {
            printf("NULL array pointer!\n");
      }

      dim = svDimensions(h);
      if (dim <= 0) {
            printf("Dimension <= 0\n");
      }

      printf("dimension = %d\n", dim);

      size = svSizeOfArray(h);
      printf("---- Total size is %d byte(s) ------------\n", size);
      for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {
            sc = svGetLogicArrElem1(h, i);
                  printf("in C h[%d] = %", i);
            if (sc == sv_x) {printf("x\n");
            }
            else if (sc == sv_z) {printf("z\n");
            } else {
                  printf("%d\n", sc);
            }
        }
}
```

dynamicArray.vr File

```
import "DPI" function void foo( reg b [*]);
program test {
      reg im[*];
      integer i;

      im = new[4];
      for (i= 0; i < 4; i++) {
            im[i] = i;
      }

      foo(im);
      for (i= 0; i < 4; i++) {
            printf("In VCS im[%d] = %d\n", i, im[i]);
      }
```

```
}
```

Here is the command line to compile Example 12-6:

```
% vcs -ntb dynamicArray.c dynamicArray.vr -R
```

In Example 12-6:

- `void *svGetArrayPtr(const svOpenArrayHandle)` gives a pointer to the actual representation of the whole array of any type, or `NULL` if not in the C layout.
- `int svSizeOfArray(const svOpenArrayHandle)` gives total size in bytes, or 0 if not in the C layout.
- `int svDimensions(const svOpenArrayHandle h)` gives the number of dimensions of an array.
- `int svLow(const svOpenArrayHandle h, int d)` and `int svHigh(const svOpenArrayHandle h, int d)` give the low and high boundaries, respectively, of an array dimension, where `d` is the dimension.
- `svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1)` gives the array element at position `indx1` for a single-dimensional array.

Example 12-7 shows a complete example using a mutidimensional array.

Example 12-7   Complete Example Using Multidimensional Array of Strings

string_mda.c File

```
static char *my_str_array[] = { "RED", "BLUE", "GREEN" };
void string_mda_var(char *str_array[], int s1, int s2){
      int i, j;
      for (i = 0; i < s1; i++) {
```

```
            for (j = 0; j < s2; j++) {
            printf("c value before change is im[%d][%d] = %s\n",
                  i, j, str_array[i*s2 + j]);
            str_array[i*s2 + j] = my_str_array[i];
            printf("c value after change is im[%d][%d] = %s\n", i, j,
                  str_array[i*s2 + j]);
                 }
      } }
```

string_mda.vr File

```
import "DPI" function void string_mda_var(var string i[3][4],
integer s1, integer s2);
program test {
integer i, j;
string im[3][4] = {{ "red", "red1", "red2", "red3"}, {"blue",
"blue1","blue2", "blue3"}, {"green", "green1", "green2", "green3"}
};
      string_mda_var(im, 3, 4);

      for (i = 0; i < 3; i++){
            for (j = 0; j < 4; j++){
            printf("Vera value is im[%0d][%0d] = %s\n", i, j,
                  im[i][j]);
                 }
        }
}
```

Here is the command line to compile Example 12-7:

```
% vcs -ntb string_mda.c string_mda.vr -R
```

Calling NTB-OV Functions and Tasks from C

To call NTB-OV functions and non-blocking tasks from C using the DPI, follow these steps:

1. Declare the NTB-OV DPI functions and tasks in C.
2. Call the NTB-OV DPI functions and tasks in C.

3. Compile the NTB-OV and C code using the `-ntb` option.
4. Run the simulation (`simv`).

DPI Function Declarations

In order for C to call NTB-OV functions and tasks they must be declared in the NTB-OV code as follows:

```
export "DPI" function function_name;
export "DPI" task task_name;
```

`export “DPI”`

are the keywords that declare the functions or tasks as DPI functions or tasks.

Note:

You cannot export class methods.

The export declaration and the definition of the corresponding function/task can occur in any order. Only one export declaration is permitted per NTB-OV function/task.

Example 12-8 shows an export function.

Example 12-8 Export DPI Function

data.vr File

```
import "DPI" function  integer ntb_call_c(integer in0, reg reg0, reg
      [33:0] regvec0, integer size, string s0);
export "DPI" function  c_call_ntb;

function  integer c_call_ntb(integer in0, reg reg0, reg [33:0] regvec0,
      integer size, string s0) {
      integer i;
      printf("Hello from NTB-OV - c_call_ntb() \n");
```

```
    printf(" int0 = %0d \n", int0);
    printf(" reg0 = %0d \n", reg0);
    printf(" regvec0 = %x\n", regvec0);
    printf(" s0 = %s\n", s0);
    c_call_ntb = 2*in0;
}

program test {
    integer int0 = 007;
    reg reg0 = 1'b0;
    reg [33:0] regvec0 = 34'h3_1234_5678;
    string s0 = "Good bye, NTB-OV";
    integer ret_val=2;
    ret_val = ntb_call_c(int0, reg0, regvec0, 2, s0);
    printf("NTB - ret_val = %0d \n", ret_val);
}
```

data.c File

```
#include <svdpi.h>
extern int c_call_ntb(int int0, svLogic reg0, svLogicVec32 *regvec0,
    int size, char *s);
int ntb_call_c(int int0, svLogic reg0, svLogicVec32 *regvec0,
int size,char *s0) {
int i;
    printf("Hello from C - ntb_call_c() \n");
    printf(" int0 = %d \n", int0);
    if (reg0 == sv_x)
        printf(" reg0 = x \n");
    else if (reg0 == sv_z)
        printf(" reg0 = z\n");
    else
        printf(" reg0 = %d \n", reg0);

    for (i=0; i<size; i++) {
        printf(" regvec0[%d](d) = %x \n", i, regvec0[i].d);
        printf(" regvec0[%d](c) = %x \n", i, regvec0[i].c);
    }
    printf(" s0 = %s\n", s0);
    i = c_call_ntb(int0, reg0, regvec0, size, s0);
    printf(" After export, i = %d\n", i);
    return i;
}
```

Here are the commands to compile and run Example 12-8:

```
% vcs -ntb datas.vr data.c
% simv
```

Example 12-8 generates output similar to the following:

```
Chronologic VCS simulator copyright 1991-2004
Contains Synopsys proprietary information.
Compiler version X-2005.06-B (ENG); Runtime version X-2005.06-B (ENG);
Feb 23 11:12 2005

Hello from C - ntb_call_c()
int0 = 7
reg0 = 0
regvec0[0](d) = 12345678
regvec0[0](c) = 0
regvec0[1](d) = 3
regvec0[1](c) = 0
s0 = Good bye, NTB-OV
Hello from NTB - c_call_ntb()
int0 = 7
reg0 = 0
regvec0 = 312345678
s0 = Good bye, NTB-OV
After export, i = 14
NTB - ret_val = 14
$finish at simulation time                    0
V C S   S i m u l a t i o n   R e p o r t
Time: 0
CPU Time:      0.020 seconds;       Data structure size:   0.0Mb
Wed Feb 23 11:12:05 2005
```

Example 12-9 shows an export task.

Example 12-9 Export DPI Task

test.vr File

```
import "DPI" function void add_int();
import "DPI" function void dummy();

interface intf {
     input clk CLOCK ;
}

program test {
```

```
      integer  a ;
      export "DPI" task export_fun ;
      task export_fun(integer foo, string str, integer
            int_arr3d[3][4][5], bit [63:0]bit_vec, bit [7:0]
            bit_arr2d[3][4]){
            integer i, j, k ;
            a = foo ;
            printf(" the integer from export is %d\n", foo) ;
      printf(" the string from export is %s\n", str) ;
            for (i = 0 ; i < 3 ; i++) {
                  for ( j = 0 ; j < 4 ; j++) {
                        for ( k = 0 ; k < 5 ; k++)
                              printf(" the int array 3d from export is %d
                              %d %d %d\n", i, j , k , int_arr3d[i][j][k]) ;
                  }
            }

            printf (" the bit vec from export is %b\n ", bit_vec) ;

            for (i = 0 ; i < 3q ; i++) {
                  for ( j = 0 ; j < 5 ; j++)
                        printf ("the bit array 2d from export is %d\n",
                              bit_arr2d[i][j]);
            }
      }
            dummy();
            @(posedge intf.clk) ;
            add_int() ;
}
```

test.c File

```
#include <stdio.h>
#include "acc_user.h"
#include "vcs_acc_user.h"
#include "svdpi.h"

extern void export_fun(int, char*, svOpenArrayHandle, svBitVec32*,
svOpenArrayHandle);
svScope scope;

void dummy() {
      scope = svGetScope();
}

int add_int(){
      int i ;
```

```
    int j , k , l ;
    char *str = "hello" ;
    int  int_arr3d[3][4][5] ;
    svBitVec32* bit_vec ;
    svBitVec32* bit_arr2d ;

    bit_vec = (svBitVec32*)
    malloc(SV_CANONICAL_SIZE(64)*sizeof(svBitVec32));
    bit_arr2d =(svBitVec32*)
          malloc(SV_CANONICAL_SIZE(8)*20*sizeof(svBitVec32));
    i = 3 ;
    for (j = 0 ; j < 3 ; j++){
          for (k = 0; k < 4 ; k++) {
                for (l = 0 ; l < 5 ; l++){
                      int_arr3d[j][k][l] = j+k+l ;
                }
          }
    }
    bit_vec[0].d=14;
    bit_vec[0].c =0;
    bit_vec[1].d =14;
    bit_vec[1].c =0;

    for (j = 0 ; j < 3 ; j++) {
          for (k = 0; k < 4 ; k++) {
                bit_arr2d[j*5+k].d = j+k;
                bit_arr2d[j*5+k].c = 0;
          }
    }
          svSetScope(scope);
          export_fun(i, str, int_arr3d, bit_vec, bit_arr2d) ;
}
```

Here are the commands to compile and run Example 12-9:

```
% vcs -ntb test.vr test.c
% simv
```

Example 12-9 generates output similar to the following:

```
Chronologic VCS simulator copyright 1991-2004
Contains Synopsys proprietary information.
Compiler version 7.2R13; Runtime version 7.2R13;
Mar 23 11:44 2005

the integer from export is 3
```

```
the string from export is "hello"
the int array 3d from export is 0 0 0 0
the int array 3d from export is 0 0 1 1
      ....//and so on
the int array 3d from export is 2 3 3 8
the int array 3d from export is 2 3 4 9
the bit vec from export is 0000000e0000000e
the bit array 2d from export is 0
the bit array 2d from export is 1
      ....//and so on
the bit array 2d from export is 4
the bit array 2d from export is 5
$finish at simulation time                  500
      V C S   S i m u l a t i o n   R e p o r t
Time: 5000 ps
CPU Time:      0.130 seconds;       Data structure size:   0.2Mb
Wed Mar 23 12:02:15 2005
```

Memory Management Rules

Observe the following rules when using the DPI:

- You cannot use C to deallocate memory allocated by NTB-OV. For example:

```
void ctask(svLogicVec32 *pVec){
/* Error--this memory is owned by NTB-OV */
free(pVec);
}
```

- You cannot use C to keep a reference to memory allocated by NTB-OV after a function call returns. For example:

```
svLogicVec32 *pGlobal;
void ctask(svLogicVec32 *pVec) {
/* Error--cannot keep refrence to NTB-OV memory */
     pGlobal = pVec;
}
```

- You must use C to deallocate all memory allocated on the C side. For example:

```
char *pMystr = "C_str";
void ctask(char **ppStr) {
      /* NTB-OV will not free this memory.*/
      *ppStr =  (char *) malloc(20*sizeof(char));
      strcpy(*ppStr, pMystr);
      return;
}
```

Indirect Interface with C++

Because NTB-OV expects C functions, you need to write wrapper code if you want to use C++ code. You must compile with C linkage. Example 12-10 shows how to make your DPI wrapper code compiler-independent:

Example 12-10   C++ Wrapper

```
#ifdef __cplusplus
extern "C" {
#endif
extern void abc (int i){
      ....
      ....
#ifdef __cplusplus
}
#endif
```

The `extern “C”` construct is required only when compiling with a C++ compiler.

The Defined C Layer

The C-layer of the DPI is provided in the $VCS_HOME/include/svdpi.h main include file, which is defined in the SystemVerilog 3.1 standard. This include file defines the canonical representation, basic types, and interface functions.The svdpi.h file also provides function headers and defines a number of helper macros and constants. For more information, see section D.9.1 of the *SystemVerilog 3.1 LRM*. Some of the functions specified as standard in the svdpi.h file are not yet implemented in NTB-OV. For the latest information on what functions are implemented, see the release notes.

13

Testbench to HDL Task Calls

You can call HDL tasks from an OpenVera Native Testbench (NTB-OV) testbench, and vice-versa. This way you can reuse tasks and functions. This chapter explains how to do both, in the following major sections:

- Calling HDL Tasks from the Testbench
- Calling Testbench Tasks from HDL

Calling HDL Tasks from the Testbench

To declare an HDL task in the testbench, use the following syntax in your main program module:

```
hdl_task task_name (argument_list) "inst_path";
```

`task_name`

is the name of the HDL task you want to call.

`argument_list`

is passed from the testbench to the HDL when the task is called. Arguments can be of type integer or bit vector. You can pass strings as regs using this construct:

reg[(8**string_length*)-1:0] *str*

where `string_length` is the number of characters in *str*.

`inst_path`

is the instantiation path of the task in the HDL. It identifies the task within the HDL hierarchy starting at the top-level module (that is, *inst_path* specification should begin with a top-level module name and not the design module name).

Here is an example that shows how to call an HDL task call from an NTB-OV testbench:

```
hdl_task chip_init (reg [7:0] init_reg)
      "top.chip.chip_init";
```

This example calls the `chip_init` Verilog task and passes the `reg` field variable `init_reg`. You can find the task in the `top.chip` hierarchy in the Verilog declaration. Because the testbench identifies this declaration as a task definition, it must occur in the top code block. However, if you want to call an HDL task from other files, you can define it as an external task using the extern construct:

```
extern hdl_task task_name (argument_list);
```

For example, to declare the `chip_init()` task so it can be called from other files, use:

```
extern hdl_task chip_init (reg [7:0] init_reg);
```

HDL Tasks in the Testbench: Outputs/Inouts

For HDL outputs and inouts, the testbench reflects the output value if the testbench formal argument is prefixed by the `var` keyword. It is important to define the `var` parameter in the HDL as inout.

Example 13-1 shows NTB-OV testbench code that illustrates how to use the `var` construct with HDL task calls.

Example 13-1   HDL Task Calls

```
hdl_task get_status (var reg [7:0] status_word)
      "top.cpu.status";

extern hdl_task chip_init (reg [7:0] init_reg);

program test{
      reg [7:0] word;
      integer i;
      chip_init (8'b0000_0000); // OK
      get_status(word);// OK
      get_status(i); // Error: formal is reg[7:0]; actual is integer

      get_status(8'b0000_0000);
/*Error: formal is var; actual is constant*/
}
```

In Example 13-1, the `chip_init()` task call succeeds because the Verilog task is defined to accept any `reg[7:0]` as a parameter. The `get_status(word)` task call succeeds because `word` matches the formal variable type specified in the declaration. However, the `get_status(i)` task call does not succeed because the variable type of the argument (`integer`) does not match the formal declaration (`reg [7:0]`). And the `get_status(8'b0000_0000)` task call does not succeed because the argument passed in is a constant and the formal declaration is a variable.

Caveats for Using HDL tasks Called From NTB

Verilog HDL tasks (IEEE 1364-1195) are not re-entrant. This implies that multiple invocation of tasks overwrite the respective local data space of tasks. Note that SystemVerilog 3.0 automatic tasks do provide re-entrancy.

It is bad form to call Verilog tasks in threads or tasks that terminate before the task completes (for example, in `fork-join none`).

Calling Testbench Tasks from HDL

You can call an NTB-OV task from your HDL code using the following syntax:

```
ntb_program_name.taskname
```

`ntb_program_name`

is the instantiation of the NTB-OV code in the Verilog testbench environment.

Arguments can be of type integer, bit, reg, or string.

14

Preprocessor Directives

The preprocessor directives are special lines that will be replaced with NTB-OV source code by the preprocessor before the actual compilation begins.

The preprocessor makes your programs easier to develop, read, and maintain. You must always precede these preprocessor directives with a pound sign (#). You use these preprocessor directives for:

- Including Files
- Substituting Macros
- Including Conditional Compilation Directives

The preprocessor directives extend only across a single line in your NTB-OV code. The preprocessor directive ends as soon as it finds a new line character. You must also not use a semi colon to end a preprocessor directive.

If you want to use a preprocessor directive that extends beyond one line, then you must precede a new line character with a back slash (\).

Including Files

You can include an NTB-OV source file in another NTB-OV source file by reference, using the `#include` directive. You can place `#include` anywhere in the program, but is typically one of the first lines for readability. The syntax is:

```
#include "filename"
#include <filename>
```

`filename`

is the name of the file to include. When you use quotation marks around the `filename`, the preprocessor first searches for the file in the directory where the primary source file is located, and then in any of the directories specified using the optional `-ntb_incdir` command-line option. Use the `-ntb_incdir` option to specify the relative or absolute path to include files. For example, if the command line is:

```
% vcs -ntb_incdir ./sonet  -ntb_incdir ../amba  file.vr
```

the preprocessor first searches for the include file in the current working directory, followed by the `sonet` and `amba` directories. The search stops as soon as the file is found.

If you want to search system header files, use angle brackets (<>). When you use this method, the preprocessor searches for the specified `filename` first in a list of directories that you specify, followed by the $VCS_HOME/include directory.

Note:

- NTB-OV automatically includes the <vera_defines.vrh> file present in the $VCS_HOME/include directory. You can use macros inside this file (such as ON, OFF, and so on) without explicitly specifying them in your source code.
- When the preprocessor encounters a `#include` directive, it replaces the directive with the entire file specified.
- Include files can be complete code or fragments of code or text.
- You cannot split comments and string constants across multiple files. This results in a compilation error.
- The preprocessor treats the line following a `#include` as a new line, even if the include file does not end with a newline.

Substituting Macros

NTB-OV supports macros with arguments and without arguments. You can use these macros anywhere in your testbench program.

The next section describes the following two topics:

- Macros Without Arguments

  When you use a macro without arguments, it substitutes the macro with its value.

  The syntax of macros without arguments is illustrated as follows:

  Macro syntax:

  ```
  #define macro_name macro_value;
  ```

  `macro_name`

It is the name of the macro.

`macro_value`

It is a value assigned to *macro_name*. It must be a constant numeric.

- Macros with Arguments

  When you use this macro, NTB-OV expands it into inline code. NTB-OV replaces each occurrence of a formal parameter by its corresponding argument. You can also concatenate macro variables using a symbol. The syntax for declaring this macro is:

  ```
  #define macro_name(list_of_arguments) macro_value;
  ```

  `macro_name`

  It is the name of the macro.

  `list_of_arguments`

  They are the parameters of the macro.

  `macro_value`

  It is the value assigned to *macro_name*.

Example: Macros without Arguments

The following are a few examples of macros without arguments:

```
#define word_width 32
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
```

Examples: Macros with Arguments

This section describes macros with arguments. A few examples are as follows:

Example 14-1 Macro with arguments

```
#define CALCULATE(A,B) ((A)*100 + (B)*10)
```

For Example 14-1, the line:

```
x = CALCULATE(r+t, v+w)
```

is replaced by:

```
x = (((r+t)*100)+((v+w)*10))
```

Example 14-2 illustrates how a parameterized macro is used to define virtual ports:

Example 14-2 Macro with arguments

```
#define iport_bind(name, bitnum) \
     bind router_iport name { \
     frame_n router.frame_n[bitnum]; \
     valid_n router.valid_n[bitnum]; \
     din router.din[bitnum]; \
     cycles router.din; \
}

iport_bind(iport_0, 0)
iport_bind(iport_1, 1)
iport_bind(iport_2, 2)
```

For Example 14-2, the line:

```
iport_bind(iport_0, 0)
```

is replaced by:

```
    bind router_iport iport_0 { \
         frame_n router.frame_n[0]; \
         valid_n router.valid_n[0]; \
         din router.din[0]; \
         cycles router.din; \
}
```

In the above example, the name parameter is replaced by iport_0 while the bitnum parameter is replaced by ‘0’.

The next example, Example 14-3, shows you how to use the `/**/` symbol to concatenate macro variables.

Example 14-3   Concatenating Macro Variables

```
#define Concat(a, b) a/**/b
printf("%d\n", Concat(20,30));
```

Example 14-3 produces the following result:

```
2030
```

Including Conditional Compilation Directives

As the name suggests, you can include lines of NTB-OV code (part of your program) optionally during compilation using the conditional compilation directives. There are about six conditional directives NTB-OV supports that enable you to omit or include part of the code for compilation if a certain condition is met.

The NTB-OV preprocessor supports the following directives:

- `#ifdef`
- `#ifndef`
- `#if`

- `#else`
- `#elif`
- `#endif`

#ifdef

The `#ifdef` conditional compilation directive causes the preprocessor to compile the code defined within the `#ifdef..#endif` if you define the specified text macro. The syntax for this directive is:

```
#ifdef macro_name
     code
#endif
```

`macro_name`

It is the name of the text macro.

`code`

It is the code to be compiled if you define the macro.

Example 14-4 illustrates `#ifdef` clearly.

Example 14-4 #ifdef Example

```
#define DEBUG 1
#ifdef DEBUG
printf("In debug mode"); // NTB-OV compiles this
#endif
```

In this example, you have defined the text macro `DEBUG` in the first line. In order for your program to compile, you have to define the macro in your source code.

#ifndef

If you do not define your macro, the `#ifndef` conditional compilation directive causes the preprocessor to compile the code defined within the `#ifndef..#endif`. The syntax for this directive is:

```
#ifndef macro_name
     code
#endif
```

`macro_name`

It is the name of the text macro.

`code`

It is the code to be compiled if you do not define the macro in your program.

Example 14-5 illustrates clearly the `#ifndef` directive.

Example 14-5 #ifndef Example

```
#define DEBUG 1
#ifndef DEBUG
printf("In debug mode"); // NTB-OV does not compile this
#endif
```

This print statement does not compile because you have defined the macro. You must not define the macro if you want a part of the code in your program to be compiled.

#if

When you use the `#if` directive, NTB compiles the code defined between `#if (expression)` and `#endif` if the `expression` you specified evaluates true. The syntax for this directive is:

```
#if expression
    code
#endif
```

`expression`

It is an NTB-OV expression of type integer, which may be:

- an integer constant
- character constants, which are interpreted as they would be in the code
- arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations (`&&` and `||`). The latter two obey the standard C short-circuiting rules.
- macros; The tool expands all macros in the expression before evaluating the expression's value.
- a defined operator, which lets you check whether macros are defined in the middle of a `#if`.
- identifiers that are not macros are all considered to be the number zero. This allows you to write `#if MACRO` instead of `#ifdef MACRO` if you know that `MACRO` always has a non zero value when it is defined. NTB-OV also treats function-like macros used without their function call parentheses as zero. Example 14-6 clearly illustrates a `#if` example.

Example 14-6 #if Example

```
#define WIDTH 32
#define HEIGHT 2
#if (WIDTH*HEIGHT <= 64)
     printf("There is not enough memory \n");
#endif
```

In the above example, the code you defined between the #if directive and #endif directive is compiled because the expression evaluates true.

#else

You can use `#if`, `#ifdef`, and `#ifndef` directives with the `#else` directive, which conditionally includes the text after the directive if the previous `#if`, `#ifdef`, or `#ifndef` fail. The syntax is:

```
#if (expression)
     code1
#else
     code2
#endif
```

If `expression` evaluates false, NTB-OV compiles `code2`, which is defined between `#else` and `#endif`. Example 14-7 illustrates a `#else` example.

Example 14-7 #else Example

```
#define WIDTH 32
#define HEIGHT 2
#if (WIDTH*HEIGHT < 64)
printf("There is not enough memory \n");
#else
printf("There is enough memory \n");
#endif
```

The preprocessor does not compile the expression associated with `#if` directive because it evaluates false. It proceeds to the next statement and compiles because of the `#else` directive.

#elif

The `#elif` preprocessor directive is equivalent to `else-if`. The syntax is:

```
#if expression1
      code1
#elif expression2
      code2
[#else ...]
#endif
```

code2

> is processed only if the original `#if` condition fails and the `#elif` condition succeeds. If the original `#if` condition evaluates true, then *code1* is processed.

You can put more than one `#elif` in the same `#if- #endif` group. NTB-OV processes the text after an `#elif` only if that `#elif` condition succeeds, the original `#if` fails, and any previous `#elif` directives within the `#if- #endif` group fails. The `#else` is allowed after any number of `#elif` directives. However, `#elif` cannot follow `#else`. Example 14-8 shows an `#elif` example.

Example 14-8 #elif Example

```
#define BUSWIDTH 32
#if (BUSWIDTH < 32)
      printf("Bus in word enabled \n");
#elif (BUSWIDTH = 32)
      printf("Bus in byte mode \n");
#else
      printf("Bus in invalid mode \n");
#endif
```

The preprocessor does not compile the expression associated with the `#if` directive because it evaluates false. The preprocessor proceeds to the next statement, evaluates it true and compiles the expression because it is associated with `#elif` directive.

#endif

You must terminate conditional preprocessor directives using a `#endif` directive. The syntax is:

```
conditional_directive
    ...
    #endif
```

`conditional_directive`

`#if`, `#ifdef`, `#ifndef`, `#elif`, or `#else`.

A

Linked Lists

OpenVera Native Testbench (NTB-OV) supports any type of list (for example, integer, string, or class object), with the exception of reg vectors. To use a particular type of linked list, you must create it before the main program and before any list declarations. The syntax is:

```
MakeVeraList (data_type)
```

Note the absence of a terminating semicolon, which is not used for linked lists. You should only enable a particular type of linked list once, regardless of the number of lists you use. You must enable a list type before you declare or use a list of that type.

To use linked lists in NTB-OV testbenches containing multiple source files, you must:

- call `MakeVeraList(type)`.

- call `ExternVeraList(type)` in the file where you want to use the list if you want to use lists across multiple files.

With linked lists, the values `TRUE` and `FALSE` are defined as `_VERA_TRUE` and `_VERA_FALSE`, respectively.

List Definitions

Here are the NTB-OV list definitions.

list — is a doubly-linked list, where every element has a predecessor and successor. It is a sequence that supports both forward and backward traversal, as well as amortized constant time insertion and removal of elements at the beginning, end, or middle.

container — is a collection of objects of the same type (for example, a container of network packets, a container of microprocessor instructions, etc.). Containers are objects that contain and manage other objects and provide iterators that allow the contained objects (elements) to be addressed. A container has methods for accessing its elements. Every container has an associated iterator type that can be used to iterate through the container's elements.

iterator — provide interfaces to containers. They also provide a means to traverse the container elements. Iterators are pointers to nodes within a list. If an iterator points to an object in a range of objects and the iterator is incremented, the iterator then points to the next object in the range.

List Declaration

Linked lists are supported via a package shipped with NTB-OV (ListMacros.vrh). Alternatively, you can write your own linked list package. To use the NTB-OV linked list package, you must:

- enable the list type
- declare the lists
- declare the iterators
- include the ListMacros.vrh header file in the file that uses the list:

```
#include <ListMacros.vrh>
```

Creating Lists

NTB-OV supports any type of list (for example, integer, string, and packet). For more information, see "Linked Lists" on page 613.

Declaring Lists

You must declare all lists before using them via the `VeraList` construct. The syntax is:

```
VeraList_data_type list1, list2, ..., listN;
```

The `VeraList` construct declares lists of the indicated `date_type`. You must declare the list before the main program and after the list enabling statements. Data stored in the list elements must be of the same type as the list declaration.

Declaring List Iterators

You must declare all list iterators before using them via the `VeraListIterator` construct. The syntax is:

`VeraListIterator_`*`data_type iterator1, iterator2, ..., iteratorN`***`;`**

The `VeraListIterator` construct declares list iterators of the indicated `data_type`. You must declare iterators as you would any other variable declaration.

Creating an Instance of a List

Before you can use a list, create an instance of the list. The syntax is:

`list` **`= new;`**

Example A-1 shows an example.

Example A-1 Creating List Instance

```
#include <ListMacros.vrh>
MakeVeraList(integer)

program atclock_test { // start of top block
      VeraListIterator_integer  it1;
      VeraList_integer list1;

      list1 = new;
      printf("first message\n");
      list1.push_front(66);
      printf(" The size of list %d",list1.size() );
}
```

You can declare and initialize a list on the same line as follows:

```
VeraList_integer list1 = new;
```

Size Methods

Here are the NTB-OV list methods you can use to analyze list sizes.

size ()

The `size()` method returns the size of the elements in the list container. The syntax is:

```
list1.size();
```

empty ()

The `empty()` method returns 1 if the container is empty; otherwise, it returns 0. The syntax is:

```
list1.empty();
```

Element Access Methods

Here are the NTB-OV list methods you can use to access list elements.

front ()

The `front()` method returns the first element in the list. The syntax is:

```
list1.front();
```

back ()

The `back()` method returns the last element in the list. The syntax is:

```
list1.back();
```

Iteration Methods

Here are the NTB-OV list methods you can use for iteration.

start ()

The `start()` method returns an iterator pointing to the first element in the list:

```
list1.start();
```

finish ()

The `finish()` method returns an iterator pointing to the very end of the list—past the end value (last element) of the list. The syntax is:

```
list1.finish();
```

To access the last element in the list, use `list.finish()` followed by `iterator.prev()`.

Modifying Methods

Here are the NTB-OV list methods you can use to modify list containers.

assign ()

The `assign()` method assigns elements of one list to another. The syntax is:

```
list1.assign(start_iterator, finish_iterator);
```

The `assign()` method assigns the elements between the two iterators to `list1`. If the `finish_iterator` points to an element before the `start_iterator`, the range wraps around the end of the list.

The range iterators must be valid list iterators. If they point to a non-existent elements or different lists, NTB-OV issues an error message.

swap ()

The `swap()` method swaps the contents of two lists. The syntax is:

```
list1.swap(list2);
```

The method assigns the elements of `list1` to `list2`, and vice-versa. Swapping a list with itself has no effect. Swapping lists of different sizes causes NTB-OV to issue an error message.

clear ()

The `clear()` method removes all the elements of the specified list and releases all the memory allocated for the list except the list header. The syntax is:

```
list1.clear();
```

purge ()

The `purge()` method removes all the elements of the specified list, and releases all memory allocated for the list, including the list header, to avoid memory leaks. The syntax is:

```
list1.purge();
```

To use a list that has been purged, you must `new()` the list. This creates a new list header.

The `purge()` and `clear()` methods both delete all elements in the list. However, the `purge()` method also deletes the list header. Because the `clear()` method does not delete the list header, subsequent list addition methods such as `push_back()` work without having to do a `new()` on the list. If you want to use the same list again, use `list1.clear()`. If you don't want to use the list again, use `list1.purge()`.

erase ()

The `erase()` method removes the indicated element. The syntax is:

```
new_iterator = list1.erase(position_iterator);
```

NTB-OV removes the element in the indicated position of `list1` from the list. After the element is removed, subsequent elements are moved up (there is no resultant empty element). When you call the `erase()` method, NTB-OV makes the position iterator invalid and returns a new iterator.

The `position_iterator` must be a valid list iterator. If it points to a non-existent element or an element from another list, NTB-OV issues an error message.

erase_range ()

The `erase_range()` method removes the elements in the specified range. The syntax is:

```
list1.erase_range(start_iterator, finish_iterator);
```

The `erase_range()` method removes the elements in the range from `list1`. NTB-OV removes elements from the `start_iterator` up to but not including the `finish_iterator`. After the elements are removed, NTB-OV moves subsequent up (there is no resultant empty element). If the `finish_iterator` points to an element before the `start_iterator`, the range wraps around the end of the list. Any iterators pointing to elements within the range are made invalid.

The range iterators must be valid list iterators. If they point to non-existent element or different lists, NTB-OV issues an error message.

push_back ()

The `push_back()` method inserts the specified `data` at the end of the list. The syntax is:

```
list1.push_back(data);
```

NTB-OV adds the `data` as another element at the end of `list1`. If the list already has the maximum allowed elements, NTB-OV does not add the element and issues an overflow error. The `data` must be of the same type as the elements in `list1`.

push_front()

The `push_front()` method inserts the specified `data` at the front of the list. The syntax is:

```
list1.push_front(data);
```

NTB-OV adds the `data` as another element at the end of `list1`. If the list already has the maximum allowed elements, NTB-OV does not add the element and issues an overflow error. The `data` must be the same type as the elements in `list1`.

pop_front ()

The `pop_front()` method removes the first element of the list. The syntax is:

```
list1.pop_front();
```

NTB-OV removes the first element of `list1`. If `list1` is empty, NTB-OV issues an error message.

pop_back ()

The `pop_back()` method removes the last element of the list. The syntax is:

```
list1.pop_back();
```

NTB-OV removes the last element of `list1`. If `list1` is empty, NTB-OV issues an error message.

insert ()

The `insert()` method inserts `data` before the position specified by `position_iterator`. The syntax is:

```
list1.insert(position_iterator, data);
```

NTB-OV moves subsequent elements backward. The position iterator must point to an element in the call list. The `data` must be the same type as the elements in `list1`.

insert_range ()

The `insert_range()` method inserts elements in a given range before the indicated position. The syntax is:

```
list1.insert_range(position_iterator, start_iterator,
finish_iterator);
```

The `insert_range()` method inserts the elements in the range between `start_iterator` and `finish_iterator` before the `position_iterator`. NTB-OV inserts the elements from start up to but not including finish. If the finish iterator points to an element before the start iterator, the range wraps around the end of the list. The range iterators can specify a range in another list or a range in list1.

The `position_iterator` must point to an element in the calling list. The start and finish range iterators must be valid list iterators. If either points to a non-existent element or to a different list, NTB-OV issues an error message.

Iterator Methods

Here are the NTB-OV methods you can use for iterators.

next ()

The `next()` method moves the iterator so that it points to the next item in the list. The syntax is:

```
I1.next();
```

prev ()

The `prev()` method moves the iterator so that it points to the previous item in the list. The syntax is:

```
I1.prev();
```

eq ()

The `eq()` method compares two iterators. The syntax is:

```
I1.eq(I2);
```

This method returns 1 if both iterators point to the same location in the same list. Otherwise, it returns 0.

neq ()

The `neq()` method compares two iterators. The syntax is:

```
I1.neq(I2);
```

This method returns 1 if the `I1` and `I2` iterators point to different locations (either different locations in the same list or any location in different lists). Otherwise, it returns 0.

data ()

The `data()` method returns the data stored at a particular location. The syntax is:

```
I1.data();
```

The method returns the data stored at the location pointed to by iterator `I1`. The data type is the same type used in `MakeVeraList(type)`.

Index

D

E

L

M

N

O

P

Q

R

S

T

U

V

W