# System Verilog 3.1 Donation
# Part IV: C-Modeling Interface

Version 1.1, May 2002

SYNOPSYS®

# 1

# The DirectC Interface

DirectC is an extended interface between the Verilog HDL and the C/C++ programming languages. It is an alternative to the PLI that, unlike the PLI, enables you to do the following:

- More efficiently pass values between Verilog module instances and C/C++ functions by calling the functions directly, along with actual parameters, in your Verilog code.
- Pass more kinds of data between Verilog and C/C++. With the PLI you can only pass Verilog information to and from a C/C++ application. With DirectC you do not have this limitation.

With DirectC, for example, you can model a simulation environment for your design in C/C++ in which you can pass pointers from the environment to your design and store them in Verilog signals, then at a later simulation time pass these pointers to the simulation environment.

Similarly you can use DirectC to develop applications to run with the simulator to which you can pass pointers to the location of simulation values for your design.

DirectC is an alternative to, but not a replacement for, the PLI. You can do things with the PLI that you cannot do with DirectC. For example there are PLI tf and acc routines to implement a callback to start a C/C++ function when a Verilog signal changes value. DirectC has not been implemented to also do this.

You call C/C++ functions like you call (or enable) a Verilog function or Verilog task.

## Making a Direct Call to a C/C++ Function

To make a call to a C/C++ function do the following:

1. Declare the function in your Verilog code.
2. Call the function in your Verilog code.

The declaration of these functions involves specifying a direction for the parameters of the C/C++ function. This is because they become in the Verilog environment analogous to Verilog tasks as well as functions. Verilog tasks are like void C functions in that they don't return a value. Verilog tasks do however have input, output, and inout arguments, whereas C/C++ function parameters do not have explicitly declared directions. See "Declaring The C/C++ Function" on page 1-5.

There are two access modes for C/C++ function calls. They pertain to the development only of the C/C++ function. They are as follows:

- The slightly more efficient direct access mode
  This mode has rules for how values of different types and sizes are passed to and from Verilog and C/C++. This mode is explained in detail in “Using Direct Access” on page 1-10.

- The slightly less efficient but with better error handling abstract access mode
  In this implementation there is a descriptor for each actual parameter of the C/C++ function. You access these descriptors using a specially defined pointer called a handle. All formal arguments are handles. DirectC comes with a library of accessory functions for using these handles. This mode is explained in detail in “Using Abstract Access” on page 1-14.

The abstract access library of accessory functions contains operations for reading and writing values and for querying about argument types, sizes, etc. An alternative library, with perhaps different levels of security or efficiency, can be developed and used in abstract access without changing your Verilog or C/C++ code.

Using abstract access is “safer” in that the library of accessory functions for abstract access have error messages to help you to debug the interface between the C/C++ and Verilog. With direct access errors simply result in segmentation faults memory corruption, etc.

Abstract access is more generalizable for your C/C++ function. For example with open arrays you can call the function with eight bit arguments at one point in your Verilog design and call it again some place else with 32 bit arguments. The accessory functions can manage the differences in size. With abstract access you can have the size of a parameter returned to you. With direct access you must know the size.

## How C/C++ Functions Work in a Verilog Environment

Like Verilog functions, and unlike Verilog tasks, no simulation time elapses during the execution of a C/C++ function.

The parameters of C/C++ functions, are analogous to the arguments of Verilog tasks. They can be input, output, or inout just like the arguments of Verilog tasks. You don’t specify them as such in your C code, but you do when you declare them in your Verilog code. Accordingly your Verilog code can pass values to parameters declared to be input or inout, but not output, in the function declaration in your Verilog code, and your C function can only pass values from parameters declared to be inout or output, but not input, in the function declaration in your Verilog code.

If a C/C++ function returns a value to a Verilog register (the C/C++ function is in an expression that is assigned to the register) the return value of the C/C++ function is restricted to the following:

- The value of a scalar `reg` or `bit`
  In two state simulation a `reg` has a new name, `bit`.
- The value of the C type `int`
- A pointer
- A short, 32 bits or less, vector `bit`

So C/C++ functions cannot return the value of a four state vector `reg`, long (longer than 32 bits) vector `bit`, or Verilog `integer`, `real`, `realtime`, or `time` data type. You can pass these type of values out of the C/C++ function using a parameter that you declare to be inout or output in the declaration of the function in your Verilog code.

## Declaring The C/C++ Function

You declare the C/C++ function outside the `module` - `endmodule` keywords that start and end a module definition. These C/C++ functions are globally accessible to your entire design and are never declared inside a module definition.

A partial EBNF specification for external function declaration is as follows:

```
source_text ::= description +

description ::= module | user_defined_primitive | extern_declaration

extern_declaration ::= extern access_mode ?  attribute ? return_type function_id
 (extern_func_args ? ) ;

access_mode ::= ( "A" | "C" )

attribute ::= pure

return_type ::= void | reg | bit | DirectC_primitive_type
| small_bit_vector

small_bit_vector::= bit [ (constant_expression : constant_expression ) ]

extern_func_args ::= extern_func_arg ( , extern_func_arg ) *

extern_func_arg ::= arg_direction ? arg_type arg_id ?
arg_direction ::= input | output | inout

arg_type ::= bit_or_reg_type | array_type | DirectC_primitive_type

bit_or_reg_type ::= ( bit | reg ) optional_vector_range ?

optional_vector_range ::= [ ( constant_expression : constant_expression ) ? ]

array_type ::= bit_or_reg_type array [ (constant_expression :
```

```
 constant_expression ) ? ]

DirectC_primitive_type ::= int | real | pointer | string
```

Where:

| | |
|---|---|
| `extern` | Is the keyword that begins the declaration of the C/C++ function declaration. |
| *access_mode* | Specifies the mode of access in the declaration. Enter `C` for direct access, `A` for abstract access. Using this entry enables some functions to use direct access while others use abstract access.<br>You typically use these entries when some functions use direct access and others use abstract access. |
| *attribute* | An optional attribute for the function.<br>The `pure` attribute enables some optimizations. Enter this attribute if the function has no side effects and is dependent only on the values of its input parameters. |
| *return_type* | The valid return types are `int`, `bit`, `reg`, `string`, `pointer`, and `void`. See Table 1-1 for a description of what these types specify. |
| *small_bit_vector* | Specifies a bit-width of a returned vector `bit`. A C/C++ function cannot return a four state vector `reg` but it can return a vector `bit` if its bit-width is 32 bits or less. |
| `function_id` | The name of the C/C++ function. |
| *direction* | One of the following keywords: `input`, `output`, `inout`. These keywords specify in a C/C++ function the same thing that they specify in a Verilog task, see Table 1-2. |

| | |
|---|---|
| *arg_type* | The valid argument types are `real`, `reg`, `bit`, `int`, `pointer`, `string`. |
| [*bit_width*] | Specifies the bit-width of a vector `reg` or `bit` that is an argument to the C/C++ function.<br>You can leave the bit-width open by entering `[]`. |
| `array` | Specifies that the argument is a Verilog memory. |
| [*index_range*] | Specifies a range of elements (words, addresses) in the memory.<br>You can leave the range open by entering `[]`. |
| `arg_id` | The Verilog register argument to the C/C++ function that becomes the actual parameter to the function. |

Note:

Argument direction, i.e. input, output, inout applies to all arguments that follow it until next direction occurs; the default direction is input.

*Table 1-1 C/C++ Function Return Types*

| **Return Type** | **What it specifies** |
|---|---|
| `int` | The C/C++ function returns a value for type int. |
| `bit` | The C/C++ function returns the value of a bit, which is a Verilog reg in two-state simulation, if it is 32 bits or less. |
| `reg` | The C/C++ function returns the value of a Verilog scalar reg. |
| `string` | The C/C++ function returns a pointer to a character string. |
| `pointer` | The C/C++ function returns a pointer. |
| `void` | The C/C++ function does not return a value. |

*Table 1-2 C/C++ Function Argument Directions*

| keyword | What it specifies |
|---|---|
| `input` | The C/C++ function can only read the value or address of the argument. If you specify an input argument first, you can omit the keyword `input`. |
| `output` | The C/C++ function can only write the value or address of the argument. |
| `inout` | The C/C++ function can both read and write the value or address of the argument. |

*Table 1-3 C/C++ Function Argument Types*

| keyword | What it specifies |
|---|---|
| `real` | The C/C++ function reads or writes the address of a Verilog real data type. |
| `reg` | The C/C++ function reads or writes the value or address of a Verilog reg. |
| `bit` | The C/C++ function reads or writes the value or address of a Verilog reg in two state simulation. |
| `int` | The C/C++ function reads or writes the address of a C/C++ int data type. |
| `pointer` | The C/C++ function reads or writes the address that a pointer is pointing to. |
| `string` | The C/C++ function reads from or writes to the address of a string. |

### Examples

```
extern "A" reg return_reg (input reg r1);
```

This example declares a C/C++ function named return_reg. This function returns the value of a scalar reg. When we call this function the value of a scalar reg named r1 is passed to the function. This function uses abstract access.

```
extern "C" bit [7:0] return_vector_bit (bit [7:0] r3);
```

This example declares a C/C++ function named return_vector_bit. This function returns an 8-bit vector `bit` (a reg in two state simulation). When we call this function the value of an 8-bit `bit` named r3 is passed to the function. This function uses direct access.

The keyword `input` is omitted. This keyword can be omitted if the first argument specified is an input argument.

```
extern string return_string();
```

This example declares a C/C++ function named return_string. This function returns a character string and takes no arguments.

## Calling The C/C++ Function

After declaring the C/C++ function you can call it in your Verilog code.

You call a void C/C++ function like a Verilog task enabling statement by entering the function name and its arguments on a separate line in an `always` or `initial` block or in the procedural statements in a Verilog task or function declaration. Unlike Verilog tasks, you can call a C/C++ function in a Verilog function.

You call a non-void (returns a value) C/C++ function like a Verilog function call by entering its name and arguments in an expression on the RHS of a procedural assignment statement in an `always` or `initial` block or in a Verilog task or function declaration.

### Examples

```
r2=return_reg(r1);
```

The value of scalar reg r1 is passed to C/C++ function return_reg. It returns a value to reg r2.

```
r4=return_vector_bit(r3);
```

The value of vector bit r3 is passed to C/C++ function return_vector_bit. It returns a value to vector bit r4.

## Using Direct Access

Direct access was implemented for C/C++ routines whose formal parameters are of the following types:

| | | | | |
|---|---|---|---|---|
| int | int* | double* | void* | void** |
| char* | char** | scalar | scalar* | U* |
| vec32 | UB* | | | |

Some of these type identifiers are standard C/C++ types, the ones that aren't were defined with the following `typedef` statements:

```
typedef unsigned int U;
typedef unsigned char UB;
typedef unsigned char scalar;
typedef struct {U c; U d;} vec32;
```

The type identifier you use depends on the corresponding argument direction, type, and bit-width that you specified in the declaration of the function in your Verilog code. The following rules apply:

- Direct access passes all output and inout arguments by reference, so their corresponding formal parameters in the C/C++ function must be pointers.
- Direct access passes by value a Verilog `bit` only if it is 32 bits or less. Direct access passes by reference a `bit` if it is larger than 32 bits, so their corresponding formal parameters in the C/C++ function must be pointers if they are larger than 32 bits.
- Direct access passes by value a scalar `reg`. A vector `reg` direct access passes by reference, so the corresponding formal parameter in the C/C++ function for a vector `reg` must be a pointer.
- An open bit-width for a `reg` makes it possible for you to pass a vector `reg` so the corresponding formal parameter for a `reg` argument specified with an open bit-width must be a pointer. Similarly an open bit-width for a `bit` makes it possible for you to pass a `bit` larger than 32 bits so the corresponding formal parameter for a `bit` argument specified with an open bit-width must be a pointer.
- Direct access passes by value the following types of input arguments: `int`, `string`, and `pointer`.
- Direct access passes input arguments of type `real` by reference.

The following tables show the mapping from the data types you use in the C/C++ function for arguments you specify in the function declaration in your Verilog code.

*Table 1-4 For Input Arguments*

| **argument type** | **C/C++ formal parameter data type** | Passed by |
| --- | --- | --- |
| `int` | `int` | value |
| `real` | `double*` | reference |
| `pointer` | `void*` | value |
| `string` | `char*` | value |
| `bit` | `scalar` | value |
| `reg` | `scalar` | value |
| `bit []` - 1-32 bit wide vector | `U` | value |
| `bit []` - open vector, any vector wider than 32 bits | `U*` | reference |
| `reg []` - 1-32 bit wide vector | `vec32*` | reference |
| `array []` - open vector, any vector wider than 32 bits | `UB*` | reference |

*Table 1-5 For Output and Inout Arguments*

| **argument type** | **C/C++ formal parameter data type** | Passed by |
| --- | --- | --- |
| `int` | `int*` | reference |
| `real` | `double*` | reference |
| `pointer` | `void**` | reference |
| `string` | `char**` | reference |
| `bit` | `scalar*` | reference |

*Table 1-5 For Output and Inout Arguments*

| **argument type** | **C/C++ formal parameter data type** | Passed by |
|---|---|---|
| `reg` | `scalar*` | reference |
| `bit []` - any vector, including open vector | `U*` | reference |
| `reg[]` - any vector, including open vector | `vec32*` | reference |
| `array[]` - any array, 2 state or 4 state, including open array | `UB*` | reference |

In direct access the return value of the function is always passed by value. The data type of the returned value is the same as the input argument.

## Passing Class Objects in Verilog/VeraLite and DirectC

Class objects declared in Verilog can be freely used in VeraLite and vice-versa. Class objects can also be passed as arguments to DirectC functions. To do so, you must use the struct declaration in the header file produced by VCS. For all class types passed to DirectC functions, VCS will produce an equivalent typedef in a header file. You must use this header file in your C code. It is your responsibility to use this struct in the appropriate fashion. For example, a class Packet declaration in the header file looks as shown below:

```
struct vec32 {
  unsigned int c;
  unsigned int d;
}
struct Packet {
  vec32 command;
  vec32 address[2];
```

```
  vec32 master_id;
  vec32 status;
}
```

Note that class defined in VeraLite/Verilog gets used in C as a simple struct. No member functions are available in C. Use of struct vec32 is needed since values in Verilog/VeraLite are 4-state and are represented using control and data bits. For class variables that are declared 2-valued in Verilog/VeraLite, unsigned int could be used instead of vec32.

## Using Abstract Access

In abstract access there is a descriptor for each argument in a function call. The corresponding formal parameters in the function uses a specially defined pointer to these descriptors called `vc_handle`. In abstract access you use these “handles” to pass data and values by reference to and from these descriptors.

The idea behind abstract access is that you don’t have to worry about the type you use for parameters, because you always use a special pointer type called vc_handle.

In abstract access there is a descriptor for every argument that you enter in the function call in your Verilog code. The vc_handle is a pointer to the descriptor for the argument.

## Using vc_handle

In the function header the vc_handle for a Verilog reg, `bit`, or memory is based on the order that you declare the vc_handle and the order that you entered its corresponding reg, bit, or memory in the function call in your verilog code, for example, in your verilog code you declared the function and called it like so:

```
extern "A" void my_function( input bit [31:0] r1,
                             input bit [32:0] r2);

module dev1;
reg [31:0] bit1;
reg [32:0] bit2;
initial
begin
:
my_function(bit1,bit2);
:
end
endmodule
```

Declare the function

Enter first bit1 then bit2 as arguments in the function call

There are descriptors for bit1 and bit2. These descriptors contain information about their value, but also other information such as whether they are scalar or vector, and whether they are simulation in two or four state simulation.

in the header for the C/C++ function:

```
   :
 my_function(vc_handle h1, vc_handle h2)
{
   :
  up1=vc_2stVectorRef(h1);
  up2=vc_2stVectorRef(h2);
   :
}
```

h1 is the vc_handle for bit1
h2 is the vc_handle for bit2

A routine that accesses the data structures for bit1 and bit2 using their vc_handles

After declaring the vc_handles you can use them to pass data to and from these descriptors.

---

## Using Access Routines

Abstract access comes with a set of access routines that enable your C/C++ function to pass values to and from the descriptors for the Verilog reg, bit, and memory arguments in the function call.

These access routines use the vc_handle to pass values by reference but the vc_handle is not the only type of argument for many of these routines. These routines also have the following types of arguments:

- scalar — which is defined as an unsigned char
- integers — uninterpreted 32 bits with no implied semantics
- other types of pointers — primitive types “string” and “pointer”
- real numbers

These routines were named to help you to remember their function. Routine names beginning with vc_get are for retrieving data from the descriptor for the Verilog argument. Routine names beginning with vc_put are for passing new values to these descriptors.

These routines can convert from Verilog representation of simulation values and strings to string representation in C/C++. Strings can also be created in a C/C++ function and passed to Verilog but you should bear in mind that they can be overwritten in Verilog. So you copy them to local buffers if you want them to persist.

The following are the access routines, their arguments, and return values.

## The Access Routines

`int vc_isScalar(vc_handle)`

Returns a 1 value if the vc_handle is for a one-bit reg or bit, returns a 0 value for a vector reg or bit or any memory including memories with scalar elements.

`int vc_isVector(vc_handle)`

This routine returns a 1 value if the vc_handle is to a vector reg or bit. It returns a 0 value for a vector bit or reg or any memory.

`int vc_isMemory(vc_handle)`

This routine returns a 1 value if the vc_handle is to a memory. It returns a 0 value for a bit or reg that is not a memory.

`int vc_is4state(vc_handle)`

This routine returns a 1 value if the vc_handle is to a reg or memory that simulates with four states. It returns a 0 value for a bit or a memory that simulates with two states.

`int vc_is2state(vc_handle)`

This routine does the opposite of the vc_is4state routine.

`int vc_is4stVector(vc_handle)`

This routine returns a 1 value if the vc_handle is to a vector reg. It returns a 0 value if the vc_handle is to a scalar reg, scalar or vector bit, or to a memory.

`int vc_is2stVector(vc_handle)`

This routine returns a 1 value if the vc_handle is to a vector bit. It returns a 0 value if the vc_handle is to a scalar bit, scalar or vector reg, or to a memory.

`int vc_width(vc_handle)`

Returns the width of a vc_handle.

`int vc_arraySize(vc_handle)`

Returns the number of elements in a memory.

`scalar vc_getScalar(vc_handle)`

Returns the value of a scalar reg or bit.

`void   vc_putScalar(vc_handle, scalar)`

Passes by reference to a vc_handle the value of a scalar reg or bit.

`char   vc_toChar(vc_handle)`

Returns the 0, 1, x, or z character.

`int    vc_toInteger(vc_handle)`

Returns and int value for a vc_handle to a scalar bit or a vector bit of 32 bits or less.

`char  *vc_toString(vc_handle)`

Returns a string that contains the 1, 0, x, and z characters.

`char  *vc_toStringF(vc_handle, char)`

Returns a string that contains the 1, 0, x, and z characters and allows you to specify the format or radix for the display. The char argument can be `'b'`, `'o'`, `'d'`, or `'x'`.

```
void   vc_putReal(vc_handle, double)
```
Passes by reference a real (double) value to a vc_handle.

```
double vc_getReal(vc_handle)
```
Returns a real (double) value from a vc_handle.

```
void   vc_putValue(vc_handle, char *)
```
This function passes, by reference through the vc_handle, a value represented as a string containing the 0, 1, x, and z characters.

```
void   vc_putValueF(vc_handle, char, char *)
```
This function passes by reference through the vc_handle a value for which you specify a radix with the third parameter. The valid radixes are `'b'`, `'o'`, `'d'`, and `'x'`.

```
void   vc_putPointer(vc_handle, void*)
```

```
void  *vc_getPointer(vc_handle)
```
These functions pass by reference to a vc_handle a generic type of pointer or string. Do not use these functions for passing Verilog data (the values of Verilog signals). Use it for passing C/C++ data. vc_putPointer passes this data by reference to Verilog and vc_getPointer receives this data in a pass by reference from Verilog. You can also use these functions for passing Verilog strings.

```
void vc_StringToVector(char *, vc_handle)
```
Converts a C string (a pointer to a sequence of ASCII characters terminated with a null character) into a Verilog string (a vector with
8-bit groups representing characters).

```
void vc_VectorToString(vc_handle, char *)
```
Converts a vector value to a string value.

```
int    vc_getInteger(vc_handle)
```
Same as vc_toInteger.

```
void   vc_putInteger(vc_handle, int)
```

Passes an int value by reference through a vc_handle to a scalar reg or bit or a vector bit that is 32 bits or less.

```
vec32 *vc_4stVectorRef(vc_handle)
```

Returns a vec32 pointer to a four state vector. Returns NULL if the specified vc_handle is not to a four state vector reg.

```
U  *vc_2stVectorRef(vc_handle)
```

This routine returns a U pointer to a bit vector that is larger than 32 bits. If you specify a short bit vector (32 bits or fewer) this routine returns a NULL value.

```
void   vc_get4stVector(vc_handle, vec32 *)
```

```
void   vc_put4stVector(vc_handle, vec32 *)
```

Passes a four state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get4stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog.

```
void   vc_get2stVector(vc_handle, U *)
```

```
void   vc_put2stVector(vc_handle, U *)
```

Passes a two state vector by reference to a vc_handle to and from an array in C/C++ function. vc_get2stVector receives the vector from Verilog and passes it to the array. vc_put4stVector passes the array to Verilog.

```
UB *vc_MemoryRef(vc_handle)
```

Returns a pointer of type UB that points to a memory in Verilog.

```
UB *vc_MemoryElemRef(vc_handle, U indx)
```

Returns a pointer to an element (word, address or index) of a Verilog memory. You specify the vc_handle of the memory and the element.

`scalar vc_getMemoryScalar(vc_handle, U indx)`

Returns the value of a one bit memory element.

`void   vc_putMemoryScalar(vc_handle, U indx, scalar)`

Passes a value, of type scalar, to a Verilog memory element. You specify the memory by vc_handle and the element by the indx argument.

`int  vc_getMemoryInteger(vc_handle, U indx)`

Returns the integer equivalent of the data bits in a memory element whose bit-width is 32 bits or less.

`void   vc_putMemoryInteger(vc_handle, U indx, int)`

Passes an integer value to a memory element that is 32 bits or fewer. You specify the memory by vc_handle and the element by the indx argument.

`void   vc_get4stMemoryVector(vc_handle, U indx, vec32 *)`

Copies the value in an Verilog memory element to an element in an array. This routine copies both the data and control bytes. It copies them into and array of type vec32.

`void   vc_put4stMemoryVector(vc_handle, U indx, vec32 *)`

Copies Verilog data from a vec32 array to a Verilog memory element.

`void   vc_get2stMemoryVector(vc_handle, U indx, U *)`

Copies the data bytes, but not the control bytes, from a Verilog memory element to an array in your C/C++ function.

`void   vc_put2stMemoryVector(vc_handle, U indx, U *)`

Copies Verilog data from a U array to a Verilog memory element. This routine is used in the previous example.

`void  vc_putMemoryValue(vc_handle, U indx, char *)`

This routine works like the vc_putValue routine except that is for passing values to a memory element instead of to a reg or bit. You enter an argument to specify the element (index) you want the routine to pass the value to.

`void vc_putMemoryValueF(vc_handle, U indx, char, char *)`

This routine works like the vc_putValueF routine except that it is for passing values to a memory element instead of to a reg or bit. You enter an argument to specify the element (index) you want the routine to pass the value to.

`char  *vc_MemoryString(vc_handle, U indx)`

This routine works like the vc_toString routine except that it is for passing values to from memory element instead of to a reg or bit. You enter an argument to specify the element (index) you want the routine to pass the value of.

`char  *vc_MemoryStringF(vc_handle, U indx, char)`

This routine works like the vc_MemoryString function except that you specify a radix with the third parameter. The valid radixes are `'b'`, `'o'`, `'d'`, and `'x'`.

`void vc_FillWithScalar(vc_handle, scalar)`

This routine fills all the bits or a reg, bit, or memory with all 1, 0, x, or z values (you can choose only one of these four values).

`char *vc_argInfo(vc_handle)`

Returns a string containing the information about the argument in the function call in your Verilog source code.

# Storing Vector Values in Machine Memory

Verilog four state simulation values (1, 0, x, and z) are represented in machine memory with data and control bits. The control bit differentiates between the 1 and x and the 0 and z values, as shown in the following table:

| Simulation Value | Data Bit | Control Bit |
|---|---|---|
| 1 | 1 | 0 |
| x | 1 | 1 |
| 0 | 0 | 0 |
| z | 0 | 1 |

When a routine returns Verilog data to a C/C++ function, how that data is stored depends on whether it is from a two or four state value and whether it is from a scalar, a vector, or from an element in a Verilog memory.

For a four state vector (denoted by the keyword reg) the Verilog data is stored in type vec32, which for abstract access is defined as follows:

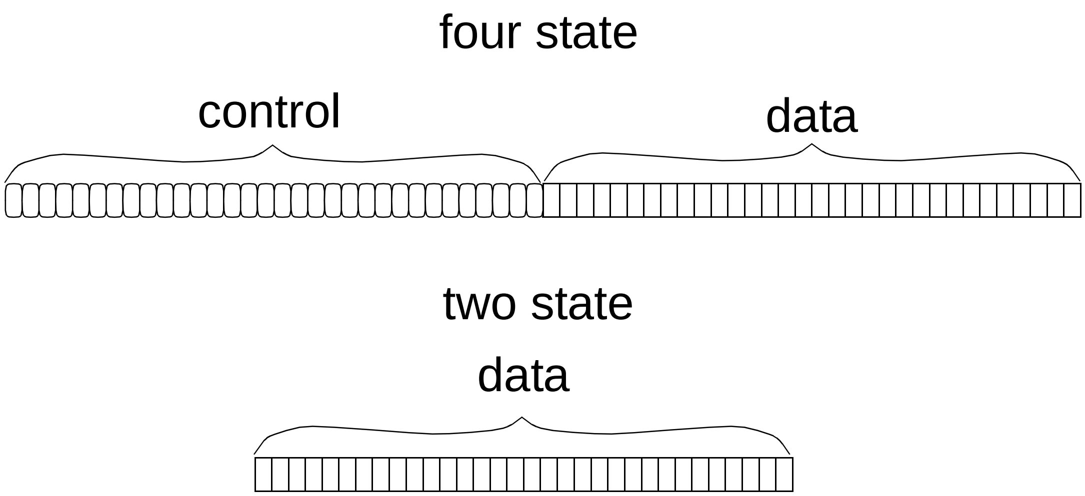
```
typedef unsigned int U;
typedef struct { U c; U d;} vec32;
```

So type vec32* has two members of type U, member c is for control bits and member d is for data bits.

For a two state vector bit the Verilog data is stored in type U*.

Vector values are stored in arrays of chunks of 32 bits. For four state vectors there are chunks of 32 bits for data values and 32 bits for control values. For two state vectors there are chunks of 32 bits for data values.

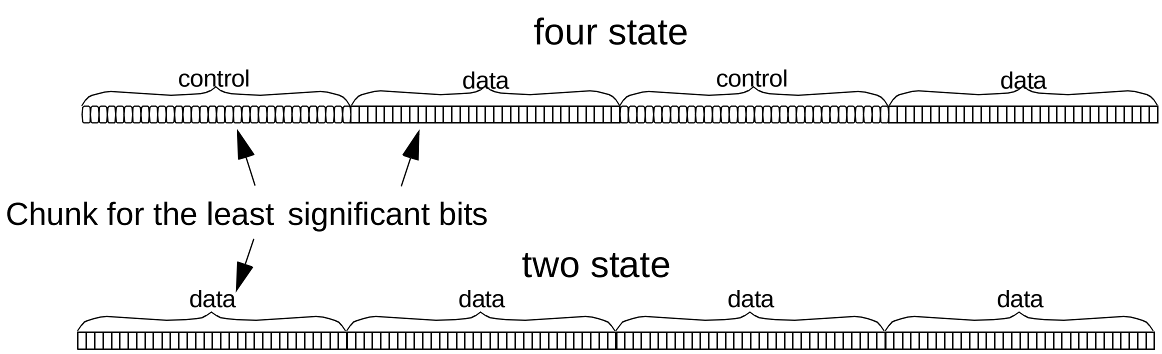
*Figure 1-1 Storing Vector Values*




Long vectors, more than 32 bits, have their value stored in more than one group of 32 bits and can accessed by chunk. Short vectors, 32 bits or less, are stored in a single chunk.

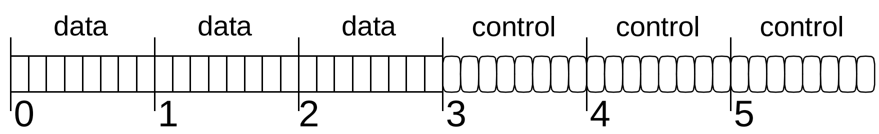
For long vectors the chunk for the least significant bits come first, followed by the chunks for the more significant bits.

*Figure 1-2 Storing Vector Values of More Than 32 Bits*




In an element in a Verilog memory, for each eight bits in the element there is a data byte and a control byte with an additional set of bytes for remainder bit, so if a memory had 9 bits it would need two data bytes and two control bytes. If it had 17 bits it would need three data bytes and three control bytes. All the data bytes precede the control bytes. For two state memories there are still data and control bytes but the bits in the control bytes always have a zero value.

*Figure 1-3 Storing Verilog Memory Elements in Machine Memory*




## Converting Strings

There are no true strings in Verilog and a string literal, like "some_text," is just a notation for vectors of bits, based on the same principle as binary, octal, decimal, hexadecimal numbers. So there is a need for a conversion between the two representations of strings: the C/C++ representation (which actually is a pointer to the sequence of bytes terminated with null byte) and the Verilog vector encoding a string.

DirectC comes with the following routines for string conversion:

`void vc_ConvertToString(vec32 *, int, char *)`
Converts a Verilog string to a C string.

`void vc_VectorToString(vc_handle, char *)`
Converts a vector to a C string.

`void vc_StringToVector(char *, vc_handle)`
Converts a string to a vector.

## Avoiding a Naming Problem

In a module definition do not call an external C/C++ function with the same name as the module definition. The following is an example of source code you should avoid:

```
extern void receive_string (input string r5);
:
module receive_string;
:
always @ r5
begin
:
receive_string(r5);
:
end
endmodule
```

# 2

# Cmodules

- Conditional Compilation
- Pre-Defined Functions
- Cmodule Ports
- Accessing Port Values

---

## Introducing Cmodules

A cmodule is a way to model hardware using the significant advantages of C/C++ to model at a high level, using sophisticated constructs like multi-dimensional arrays and data structures, while still having what would be the advantages of Verilog in hardware modeling, specifically parallel processes and timing.

Cmodules are another way to use C/C++ in your design or the modeled environment for your design. They also borrow a few concepts from Verilog. These borrowed concepts are:

- They have a Verilog-like outer appearance — they have a header and ports like Verilog modules. You instantiate them using a port connection list instead of calling C/C++ functions with arguments for their formal parameters.
- They can contain `initial` and `always` blocks that work like Verilog `initial` and `always` blocks. These blocks are an additional level of hierarchy within the cmodule and like their Verilog counterparts are separate activity flows that execute in parallel.
- They can have named events and you can use them to halt activity in an `initial` or `always` block, or a function defined in a cmodule, until you trigger the named event.

- They can contain event control statements that halt activity in an `initial` or `always` block, or a function defined in a cmodule, until the event expression is true.
- They can contain delay statements that halt activity in an `initial` or `always` block, or a function defined in a cmodule, for a certain amount of simulation time.

These are all the concepts that are borrowed from Verilog, all other actions specified in a cmodule are specified in C/C++. Most of the statements in a cmodule should be C/C++. Except for port declarations, named events, and `always` and `initial` block specifications, all declarations in the cmodule are C/C++ declarations. A cmodule is not intended to be a way to use all of Verilog and C/C++ interchangeably.

When you use cmodules, just like when you call C/C++ functions using direct or abstract access, you set up a simulation in which there are two domains, one for Verilog values and one for C/C++ values. Each domain is largely invisible to the other except through the portals where values can pass back and forth between these domains: the parameters of the C/C++ functions and the ports of the cmodules. A cmodule, like a C/C++ function in direct or abstract access, is like a “black box” to the Verilog domain. The Verilog design has no access to the values of the variables inside the cmodule unless these values propagate out through the cmodule ports.

A cmodule is intended to model hardware that has the following requirements:

- The model needs to be sensitive to Verilog four-state values (0, 1, x, and z) or Verilog two-state values (0 and 1). The model is instantiated in a Verilog design, Verilog simulation values propagate into it, changing its behavior, and Verilog values propagate out of it.
- The model needs to use the sophisticated construct in C/C++ like multi-dimensional arrays and data structures.
- The model needs a certain amount of simulation time to perform its function in your design and some of its operations need to be performed in parallel.
- The model needs to do an extensive amount of processing of input values before it can output resulting values.

If the hardware you are modeling has these requirements, then modeling it with a cmodule is something that you should consider.

Note:

VCS can monitor or dump the values of ports of cmodules from the Verilog side. C/C++ variables defined in a cmodule cannot be monitored or dumped.

Cmodules require a C++ compiler, since VCS will translate cmodules into C++ code. All C-compiler compile and link options can be specified under the existing `-CFLAGS` and `-LDFLAGS` options of VCS.

All standard C/C++ debuggers (gdb, dbx, etc.) will be supported for debugging code in DirectC and in cmodules. The C/C++ debugger of choice will be invoked along with VirSim for concurrent debugging of Verilog, Cmodule, and C/C++ source during a debug simulation (Currently not supported).

Existing PLI 1.0 routines may be called from within direct C functions or cmodules. TF routines that handle parameters to PLI task/function are not relevant here. TF routines that handle delays, events and callbacks are disallowed (since there is better mechanism to achieve this through delay/event constructs in cmodule). Use of these functions will produce unpredictable results. No explicit error checking will be performed. ACC routines that walk the design hierarchy will treat a cmodule as an empty Verilog wrapper module with visibility only to its instantiation and ports. Internal details of the cmodule are within behavioral scope and so will not be visible to PLI 1.0 routines.

You enable the use of cmodules with the `+cmod` compile-time option.

## Cmodule Example

This section describes the parts of a cmodule definition. Let's begin with an example:

```
'stacksize 32k
'timescale 1ns/1ns
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include "rom.h"
#include "display.h"

#define AttrLength (RS_ATTR_HI - RS_ATTR_LO ) + 1

int i, j;
Rom chRom;
UCHAR display_page[DisplayPageSize];
UCHAR attribute [AttrLength];

cmodule osd_test(clk, reset_l, vr_data, pixel_clk, vsync)
   input reg clk;
```

```
      output reg reset_l;
      output reg [7:0] vr_data;
      input reg pixel_clk;
      input reg vsync;
   {
      vc_event wait_for_vsync;

      initial {
         int int_true = 1;
         printf("Starting osd_test\n");
         chRom.load("/cmod-testing/ext-ex/character.rom");
         @(posedge clk);
         reset_l = 0;
         for(int idx = RESET_LENGTH;idx>0;idx--){
            @(posedge clk);
         }
         vc_delay(5);
         reset_l = int_true;
         @(posedge clk);

         i = RS_RAM_LO;

         rs_write (0x53, i++);
         rs_write (0x45, i++);
         rs_write (0x41, i++);
         rs_write (CC_RTN, i);

         vc_trigger(wait_for_vsync);
         compare_display();
      }

      initial
      {
         @(wait_for_vsync);
         @(posedge vsync);
         @(negedge vsync);
         for(int idx=40; idx>0; idx--){
            @(negedge pixel_clk);
         }
         printf("Test completed after 1 video frame\n");
      }
```

```
   /**************************************************/
   void rs_write (UCHAR data, UINT addr)
   {
      if ((addr >= RS_RAM_LO) && (addr <= RS_RAM_HI)) {
         display_page[addr] = data;
      }
      else if ((addr >= RS_ATTR_LO)&&(addr <= RS_ATTR_HI))
      {
         attribute[addr - RS_ATTR_LO] = data;
      }
      vr_data = data;
      @(posedge clk);
   }
   /**************************************************/

   void compare_display()
   {
      Screen screen;
      Display display;
      Pixel p;
      screen.build();
      display.fill(screen,
         attribute[RS_HZ_DELAY - RS_ATTR_LO],
         attribute[RS_TOP_MARGIN - RS_ATTR_LO], 0);
      int x = 0;
      int y = 0;
      for (int idx=40; idx>0; idx--) {
         @(posedge pixel_clk);
         if (!(vsync.toInteger())) {
            p = display.get(x, y);
            x++;
            if (x >= DisplayPixelWidth) {
               x = 0;
               y++;
               if (y >= DisplayPixelLength) {
                  y = 0;
               }
            }
         }
      }
   }
} //End of cmodule
```

The example begins with preprocessor directives, declaring global variables and special compiler directives for cmodules: `‘stacksize` (which enables you to change the stacksize of a thread) and `‘timescale` (which is similar to the Verilog compiler directive):

```
‘stacksize 32k
‘timescale 1ns/1ns
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include "rom.h"
#include "display.h"

#define AttrLength (RS_ATTR_HI - RS_ATTR_LO ) + 1

int i, j;
Rom chRom;
UCHAR display_page[DisplayPageSize];
UCHAR attribute [AttrLength];
```

See “Timescale Specification” on page 2-26 and “Stack Size Specification” on page 2-27.

Next is the cmodule header:

```
cmodule osd_test(clk, reset_l, vr_data, pixel_clk, vsync)
```

The cmodule header begins with the keyword `cmodule`, followed by the cmodule name or identifier and then a port connection list. Notice that the cmodule header does not end with a semicolon, just like a C/C++ function. The port connection list is optional, however a cmodule is always at the leaf level of the design hierarchy (has no hierarchy under it) and so a cmodule without ports would only be able to communicate with other cmodules and only by passing values to and from global external variables.

In this example the port connection list contains ports named clk, reset_l, vr_data, pixel_clk, and vsync.

After the cmodule header are the port declarations:

```
input reg clk;
output reg reset_l;
output reg [7:0] vr_data;
input reg pixel_clk;
input reg vsync;
```

Like Verilog module ports, cmodule ports have either the input, output, or output direction and the `input`, `output`, and `inout` keywords begin a declaration of one or more ports.

Unlike Verilog module ports, cmodule ports are always registers, never nets. The keyword `reg` specifies a port for four-state simulation values. The keyword `bit` specifies a port for two-state simulation values.

You can declare more than one port in a declaration if all the ports in that declaration have the same direction and size and all of them in the declaration simulate all together in two-state or all together in four-state simulation.

Also, unlike Verilog ports, you must declare them in the same order that you list them in the port connection list.

Notice that, like Verilog port declarations, cmodule port declarations end with a semicolon (;). As stated earlier, having ports at all is optional, but if a cmodule header contains a port connection list then, as you might expect, the ports must be declared.

After the port declarations is the beginning of the main set of braces:

```
{
                         The cmodule body goes in here.


} //End of cmodule
```

Like the body of a C/C++ function, the body of a cmodule is always enclosed in braces.

Just inside the main braces is the declaration for a named event in a cmodule.

```
vc_event wait_for_vsync;
```

A named event is similar to a named event in a Verilog module. You declare it with the `vc_event` keyword. You trigger the event with a pre-defined function shown later in this example. Named events are, like in Verilog module definitions, optional. For more information on named events see “Named Events” on page 2-28.

Next is the first of two `initial` blocks in this example:

```
initial {
   int int_true = 1;
   printf("Starting osd_test\n");
   chRom.load("/cmod-testing/ext-ex/character.rom");
   @(posedge clk);
   reset_l = 0;
   for (int idx = RESET_LENGTH; idx>0; idx--) {
```

```
      @(posedge clk);
    }
    vc_delay(5);
    reset_l = int_true;
    @(posedge clk);

    i = RS_RAM_LO;

    rs_write (0x53, i++);
    rs_write (0x45, i++);
    rs_write (0x41, i++);
    rs_write (CC_RTN, i);

    vc_trigger(wait_for_vsync);
    compare_display();
}
```

Notice that unlike a Verilog `initial` block, in a cmodule `initial` block all statements are enclosed in braces { }. The different types of statements in this `initial` block include:

- A call to standard C/C++ functions:

  ```
  printf("Starting osd_test\n");
  ```

- A call to a function of an instance of a class:

  ```
  chRom.load("/cmod-testing/ext-ex/character.rom");
  ```

- An event control statement:

  ```
  @(posedge clk);
  ```

  This is a unique statement in a cmodule. This statement halts execution of the initial block until there is a rising edge on input port clk. This statement is the only statement that is not a C/C++ statement that you can use in a cmodule. See "Event Control Statements" on page 2-29.

- An assignment to an output port:

```
reset_l = 0;
```

- A C/C++ for loop for repeated executions of an event control statement:

```
for (int idx = RESET_LENGTH; idx>0; idx--) {
    @(posedge clk);
}
```

- A call to a predefined function for a simulation delay:

```
vc_delay(5);
```

  This is another unique statement in a cmodule. This statement halts execution of the initial block for five time units (as specified by the `'timescale` compiler directive). See“Delay Statements” on page 2-30.

- An assignment to an output port of the value of a local variable:

```
reset_l = int_true;
```

- An assignment of a definition from a .h file that is used with this .vc file to a global external variable.

```
i = RS_RAM_LO;
```

- A call to a function that is defined within the cmodule:

```
rs_write(0x53, i++);
```

- A call to a predefined function for triggering the named event:

```
vc_trigger(wait_for_vsync);
```

  See “Pre-Defined Functions” on page 2-31.

Next is the second initial block:

```
initial
{
   @(wait_for_vsync);
   @(posedge vsync);
   @(negedge vsync);
   for (int idx=40; idx>0; idx--) {
      @(negedge pixel_clk);
   }
   printf("Test completed after 1 video frame\n");
}
```

Notice that the event expression in the event control statement is the named event that was declared earlier with the `vc_event` keyword. For more information on named events see “Named Events” on page 2-28.

Next is a function defined within the cmodule:

```
void rs_write (UCHAR data, UINT addr)
{
   if ((addr >= RS_RAM_LO) && (addr <= RS_RAM_HI)) {
      display_page[addr] = data;
   }
   else if ((addr >= RS_ATTR_LO) && (addr <= RS_ATTR_HI)) {
      attribute[addr - RS_ATTR_LO] = data;
   }
   vr_data = data;
   @(posedge clk);
}
```

Notice that an event control statement is in the function definition. Functions defined within a cmodule can only be called within that cmodule but you can include event control and delay statements in them.

Next, and in this example last, is another function defined within the cmodule:

```
void compare_display()
{
   Screen screen;
   Display display;
   Pixel p;
   screen.build();
   display.fill(screen,
      attribute[RS_HZ_DELAY - RS_ATTR_LO],
      attribute[RS_TOP_MARGIN - RS_ATTR_LO], 0);
   int x = 0;
   int y = 0;
   for (int idx=40; idx>0; idx--) {
      @(posedge pixel_clk);
      if (!(vsync.toInteger())) {
         p = display.get(x, y);
         x++;
         if (x >= DisplayPixelWidth) {
            x = 0;
            y++;
            if (y >= DisplayPixelLength) {
               y = 0;
            }
         }
      }
   }
}
```

This function also contains an event control statement.

A partial EBNF specification for cmodule definition is as follows:

```
cmod_defn  ::= cmodule cmod_id (port_list?)
   port_defn_list? {
      cmod_body
   }
port_list::= port_id, port_list | port_id
port_defn_list::= port_defn; port_defn_list | port_defn
port_defn::= direction_id port_type port_id
direction_id::= input | output | inout
port_type::= (reg | bit) ([number:number])?
```

```
cmod_body::= (cmod_local_decl | cmod_always_blk
   | cmod_initial_blk | cmod_func_definition )*
cmod_always_blk ::= always (@(event_list))? {
   cmod_statements }
cmod_initial_blk::= initial { cmod_statements }
cmod_statements::= (c_statement | @(event_list); *

event_list::= simple_event | simple_event or event_list
simple_event::= event_id | (posedge | negedge)?
input_port_id
```

Bold-face identifiers and punctuation are terminals in the above productions. (In BNF terminals are entries for which there can be no further substitutions and productions are the formal term for rules.) Plain-face punctuation are the standard meta-symbols for alternation, grouping, etc.

## Cmodule Instantiation

You instantiate a cmodule in a Verilog module so that it can pass values back and forth from Verilog through its ports. Cmodule instantiation is just like Verilog module instantiation. The instantiation statement begins with the cmodule identifier, followed by an instance name, followed by an order based connection list or a name based connection list. The following is an example cmodule header followed by two instantiation statements, one with a name based, the other with an order based connection list:

```
cmodule osd_test(clk, reset_l, vr_data, pixel_clk, vsync)

osd_test vshell(.clk(top_clk),.reset_l(top_reset_l),
.vr_data(top_vr_data),.pixel_clk(top_pixel_clk),
.vsync(top_vsync));

osd_test vshell(top_clk,top_reset_l,top_vr_data,
top_pixel_clk,top_vsync);
```

Also like instantiating Verilog modules you can leave a null port in the instantiation. In the above examples of instantiation statements, signal top_vr_data is connected to the cmodule through its port named vr_data. We could leave out this connection as follows:

```
osd_test vshell(.clk(top_clk),.reset_l(top_reset_l),
.pixel_clk(top_pixel_clk),.vsync(top_vsync));

osd_test vshell(top_clk,top_reset_l, ,top_pixel_clk,
top_vsync);
```

Just like Verilog module instantiation statements, you can include bit-selects, part-selects, and concatenations in the port connection list in the cmodule instantiation statement, for example:

```
osd_test osd1(tclk,rst_top[7],{data1,data2[3:0]},pclk,
syncher);
```

Also just like Verilog module instantiation statements, you can connect a register only to input ports but you can connect a net to an input, inout, or output port.

Just like Verilog modules there is no limit to the number of instantiations you can make of a cmodule.

Cmodule instances must be leaf-level instances, that is, they are at the bottom of their hierarchical tree and cannot contain cmodule or Verilog module instantiation statements.

You can choose not to instantiate a cmodule. If you do there is no way for the cmodule to pass values to the Verilog part of your design because communication to Verilog is only through a cmodule's ports. You cannot make a cross-module reference (sometimes called an upwards name reference) to a Verilog signal from inside a cmodule or a cross module reference to a variable inside a cmodule from a Verilog module.

If you choose not to instantiate the cmodule, it can still communicate with other cmodules by passing values to global variables declared outside of a cmodule.

## Cmodule Always and Initial Blocks

You model concurrency in a cmodule with `always` and `initial` blocks, their syntax is as follows:

```
 cmod_always_blk  ::= always (@(event_expression))?
{ cmod_statements }
 cmod_initial_blk ::= initial { cmod_statements }
```

As you can see in this syntax there is a formal “sensitivity list” for an `always` block, see “The always Block Sensitivity List” on page 2-18. This sensitivity list is almost the same as an event control statement (see “Event Control Statements” on page 2-29) but does not end with a semicolon.

Cmodule `always` and `initial` blocks are an additional level of hierarchy in a cmodule.

The `always` and `initial` blocks in cmodules are similar to Verilog `always` and `initial` blocks in the following ways:

- They execute in parallel. The order of their execution is arbitrary and cannot be specified. They also execute in parallel with the `always` and `initial` blocks in Verilog modules.

- The `initial` blocks execute at the start of simulation and the `always` blocks execute continuously through out the simulation. (There are delay statement and event control mechanisms for interrupting the execution of `always` blocks.)
- An event at time zero that makes the event expression true in the sensitivity list for an `always` block, triggers the execution of the `always` block at time zero.

This is where the similarity ends between Verilog and cmodule `always` and `initial` blocks. Their differences include:

- Cmodule `always` and `initial` blocks begin and end with curly braces `{}`. There are no `begin-end` or `fork-join` blocks within them.
- The statements inside cmodule `always` and `initial` blocks are C/C++ statements, they are never Verilog statements. There are also special statements that you can use in these blocks, see "Event Control Statements" on page 2-29 and "Delay Statements" on page 2-30.

## The always Block Sensitivity List

A cmodule `always` block sensitivity list is similarly to an event control "sensitivity list" for an `always` block in Verilog. (The IEEE Standard 1364-1995 does not mention the concept of a sensitivity list for an `always` block but in practical usage and event control immediately following the `always` keyword is a sensitivity list for the `always` block and its use has important performance considerations in VCS.)

A cmodule `always` block sensitivity list is between the `always` keyword and the opening curly brace `{`, and begins with `@` (the "at" character), for example:

```
always @(posedge inport1 or negedge inport2) {
    .
    .
    .
}
```

The sensitivity list contains an event expression that is enclosed in parentheses and can include one or more input or inout port (if more than one, separated by the `or` keyword) and the `negedge` or `posedge` keyword (to specify a falling or rising edge on the port, just like in Verilog).

VCS does not begin to execute the `always` block until the event expression is true. In this example VCS does not begin to execute the `always` block until there is a rising edge on port inport1 or a falling edge on port inport2.

Cmodule sensitivity lists differ from cmodule event control statements in that they do not end with a semicolon (;). See "Event Control Statements" on page 2-29.

## User-Defined Functions

You can define a C/C++ function in a .vc file that contains a cmodule definition. You can define these functions inside the cmodule definition (where they are local to the cmodule) or above a cmodule definition (where they are global to all cmodules that your design uses).

These functions, like `initial` and `always` blocks can include delay and event control statements, see "Event Control Statements" on page 2-29 and "Delay Statements" on page 2-30. Synopsys recommends that you use these types of statements with caution in user-defined functions and only in `void` function because the order of execution of functions with these types of statements cannot be defined.

For each function there is a separate stack for each call of the function, provided the different calls come from different concurrent blocks like different `initial` or `always` block or another user-defined function. Their local data is private to each call.

Variables declared `static` in such a function will be shared across calls of the same function, for all cmodule instances, as per C/C++ semantics. Two concurrently executing copies of the function can overwrite the values of such shared static variables.

## Scope

For cmodules there are three levels of scope:

| | |
|---|---|
| Global Scope | Variables and named events declared, and functions defined, outside and above the cmodule definition but inside the .vc file that contains the cmodule definition.<br>"This scope is equivalent to the scope of C++ variables declared globally outside any class. As in C/C++, static variables declared in this scope have visibility restricted to the source file. |
| Cmodule Scope | Variables and named events declared inside a cmodule definition but outside an `always` or `initial` block or a function defined inside a cmodule definition.<br>Functions defined inside a cmodule definition.<br>`always` and `initial` blocks are always at the cmodule scope, they cannot be specified in another `always` or `initial` block or a function |
| Block or Function Scope | Variables and declared inside an `always` or `initial` block or a function.<br>Functions cannot be defined inside an `always` or `initial` block. |

Global functions can do the following:

- Call global functions defined above them in the.vc file
- Access global variables declared above them in the .vc file as well as variables declared inside the global function
- Trigger global named events declared above them in the .vc file. (You can declare a named event in a global function, but if you do you cannot trigger it from outside the global function or use it in an event control statement in the cmodule.)

Cmodule scope functions can do the following:

- Call global functions defined above them in the.vc file and cmodule scope functions defined above them in the cmodule
- Access global variables declared above them in the .vc file, cmodule scope variables declared above them in the cmodule, as well as variables declare inside the cmodule scope function
- Trigger global named events declared above them in the .vc file, and cmodule scope named events declared above or below them in the cmodule. (You can declare a named event in a cmodule function but if you do you can't trigger it from outside the cmodule function or use is in an event control statement outside the cmodule function.)

`always` and `initial` blocks can:

- Call global functions defined above them in the.vc file and cmodule scope functions defined above or below them in the cmodule

- Access global variables declared above them in the .vc file, cmodule scope variables declared above or below them in the cmodule, as well as variables declare inside the `always` and `initial` block
- Trigger global named events declared above them in the .vc file and cmodule scope named events declared above them in the cmodule. (You can declare a named event in an `always` or `initial` block but if you do you cannot trigger it from outside the `always` or `initial` block or use is in an event control statement outside the `always` or `initial` block.)

The following .vc files, used for an example design, show the different levels of scope:

```
// ex1.vc

'stacksize 32k
'timescale 1ns/1ns
#include <stdio.h>

int i, j;

void glob_func1 (char data)
    {
.
.
.
    }
```

global variables

global function

global event

```
cmodule mycmod(clk, flag,indata,outdata)
    input reg clk;
    output reg flag;
    input reg [7:0] indata;
    output reg [7:0] outdata;
{
    vc_event do_it;

    char karak;


initial {
int int_true = 1;
karak = 'a';
glob_func1(karak);
.
.
.
vc_trigger(do_it);
.
.
.
vc_trigger(glob_event);
.
```

cmodule event

cmodule variable

initial block variable

accessing a cmodule variable

calling a global function

triggering a cmodule event

triggering a global event

```
    always
    {
@(do_it);
@(glob_event);
.
.
.
printer(i);
    }

void printer (int arg)
    {
.
.
.
```

cmodule event in event control statement

global event in event control statement

call of a cmodule function

cmodule function

```
// ex2.vc

'stacksize 32k
'timescale 1ns/1ns
#include <stdio.h>
vc_event glob_event1,glob_event2;
int j2;
char k;
void glob_func2(char char1)
{
.
.
.
```

global events in different .vc files can't have the same name

global functions in different .vc files can't have the same name

```
void glob_func3()
{
char c = 'c';
glob_func2(c);
.
.
```

A global function can call a global function that precedes it in the .vc file

```
cmodule mycmod2(clk, flag)
    input reg clk;
    output reg flag;
{
.
.



void cmod_funk1 ()
{
.
.
.
      cmod_funk3();
.
.
.
}
.
.
.



   initial{
       cmod_funk4();
       botint=1;
       }

int botint;

void cmod_funk4()
 {
 .
.
.
  }
```

In a cmodule function, call of a cmodule function defined later in the cmodule

In an initial block, call of a function defined later in the cmodule

In an initial block, access of a cmodule variable declared later in the cmodule

### Static Variables

You can define static variables in the following scopes:

- Globally, outside the cmodule definition but inside the .vc file. These static variables are accessible by all functions and `always` and `initial` blocks.

  Global static variables can be access by functions and `always` and `initial` blocks in other .vc files if there is an `extern` declaration for it.

- At the cmodule level. These static variables are accessible by all functions and `always` and `initial` blocks inside the cmodule definition.
- At the block or function level. These static variables cannot be accessed from another function or `always` or `initial` block.

## Timescale Specification

The possibility of delay statements in a cmodule require the ability to specify a time scale. You can use a `‘timescale` compiler directive in a .vc file, just like to `‘timescale` compiler directive in your Verilog source files, to specify the time scale and time precision for these delay statements.

There is nothing different about a `‘timescale` compiler directive in a .vc file. It takes two arguments *time_unit* and *time_precision* just like it is specified in IEEE Standard 1364-1995 pages 225-227. For example,

```
‘timescale 10ns/1ns
```

This compiler directive specifies that the delay value in a delay statement is multiplied by 10 ns, rounded to the nearest 1 ns.

When you enter a `‘timescale` compiler directive in a .vc file, the specified time scale and precision apply to the next cmodule definition in the .vc file. It also applies to all cmodule definitions that follow in the .vc file, and all cmodule definitions in subsequent .vc files on the command line, until VCS encounters another a `‘timescale` compiler directive.

A `‘timescale` compiler directive in a Verilog source file has no effect on a cmodule definition, however a `‘timescale` compiler directive in a .vc file effects all Verilog module definitions not under a `‘timescale` compiler directive in the Verilog source files.

## Stack Size Specification

Each concurrent block runs a separate thread. The default stacksize of a thread is 8K. Users can change the stacksize using the `‘stacksize` specification.

The syntax for stacksize specification is as follows:

```
‘stacksize n[k]
```

Where *n* is a decimal number and the optional suffix `k` specifies multiplying n by 1000. So for example `‘stacksize 4k` and `‘stacksize 4000` are equivalent.

In case of a highly recursive function, you might need to set the stacksize to a very high value. DirectC attempts to detect stack overflow but is not always successful.

## Named Events

Cmodules can contain and use named events just like Verilog modules can. In cmodules they work the same way, they don't have value, they are just triggered to make some other event happen.

The details of their declaration, triggering statement, and how you use them to make other events happen are different:

- In Verilog you declare a named event with a declaration using the `event` keyword. You can declare a named event inside a module definition or inside a named begin-end or fork-join block.

  You trigger a Verilog named event with the `->` event triggering operator followed by the event name. You can specify a hierarchical name for the event.

  You can use a Verilog named even in an event control on a statement or block of statements to control when the statement or statements are executed.

- In cmodules you declare a named event using the `vc_event` keyword, for example:

  ```
  vc_event event1, event2;
  ```

  You can declare a named event at the global scope, in the .vc file but outside the cmodule definition, or at the cmodule scope inside the cmodule definition. (You can also declare them inside a global function, cmodule function, or an `initial` or `always` block, but if you do you cannot trigger or use them outside of the function or `initial` or `always` block.)

You trigger a cmodule named event with the predefined function `vc_trigger(named_event)`, for example:

```
vc_trigger(event1);
```

You can use a named event is an event control statement. In cmodules event controls are separate statements, for example:

```
@(event1);
```

This statement halts execution of a function or an `initial` or `always` block until the event is triggered by the `vc_trigger` function.

## Event Control Statements

Event control statements halt the execution of an `always` or `initial` block, or a function defined inside a cmodule until the event expression in these statements become true. For example:

```
always {
@(port1);
    .
    .
    .
}
```

In this example the execution of the `always` block stops until there is a change of value on port port1.

Cmodule event control statements are similar to Verilog event controls in that they both contain event expressions and they halt execution until the event expression is true.

Cmodule event control statements are different from a Verilog event control in that a Verilog event control applies to the statement or block of statements that follow it, or, when used as an intra-assignment timing control, controls when the value of the RHS is assigned to the LHS, and is not in and of itself a statement. In cmodules there are event control statements and you cannot apply an event control to another statement.

The event expression is enclosed in parentheses, can include one or more ports or named events (but not other types of local of external variables) and can include, like Verilog, the `posedge`, `negedge`, and `or` keywords.

You can use an event control statement in an `initial` or `always` block or in a C/C++ function that you define inside a cmodule.

## Delay Statements

Delay statements halt the execution of an `always` or `initial` block, or a function defined inside a cmodule for a specific amount of simulation time. For example:

```
always {
vc_delay(5);
    .
    .
    .
}
```

The delay statement is designed to look like a predefined function call. It is not the same as a Verilog delay specification. Cmodule delay statements do not apply to other statements or constructs, instead they are individual statements of themselves.

You can use a delay statement in an `initial` or `always` block or in a C/C++ function that you define inside a cmodule.

### Zero Delay Statements

Zero delay statements are possible and they serve the same purpose as zero delays in Verilog. A zero delay statement in an `initial` or `always` block or in a C/C++ function that you define inside a cmodule suspends the execution of that block or function until all other events scheduled to occur in the current time step have executed.

## Conditional Compilation

You can use the `#ifdef`, `#ifndef`, and `#endif` preprocessor directives in you cmodule file to specify identifiers for conditional compilation.

You define these conditional compilation identifiers with the `+cmoddefine+`*`identifier`* compile-time option.

## Pre-Defined Functions

There are predefined functions that enable you to control the simulation or obtain the simulation time. You can call these functions from any `initial` or `always` block or from any global or cmodule scope user-defined function.This section describes these functions:

### void vc_trigger(*named_event*)

This function triggers a named event. The named event must be visible to the `initial` or `always` block or user-defined function that calls this function. Name events declared in other `initial` or `always` blocks or user-defined functions are not visible to a different `initial` or `always` block of user-defined function.

You declare a named event with the `vc_event` keyword, for example:

```
vc_event my_named_event;
```

### void vc_delay(*number_of_time_units*)

Halts the execution of the statements in an `initial` or `always` block for the specified number of time units.

### void vc_finish()

Stops simulation just like the `$finish` system task.

### unsigned int vc_lowtime()

Simulation time is stores in two 32-bit words. This function returns an unsigned integer value for the first word used for recording the simulation time.

### unsigned int vc_hightime()

This function returns an unsigned integer value for the second word used for recording the simulation time.

### double vc_time()

This function returns a double value for the entire simulation time.

## Cmodule Ports

Like Verilog module ports, cmodule ports are listed in the connection list in the cmodule header and then declared separately listing their direction (input, output, or inout) and size. Both Verilog and cmodule ports can be scalar or vector ports.

Unlike Verilog module ports, cmodule port declarations also specify their type and they can have two types:

- `reg` — for four-state simulation
- `bit` — for two-state simulation

These types are included in the port declaration. The following is an example of a cmodule header and its port declarations:

In cmodules all ports are four-state or two-state registers; they are never considered nets. Assignments to them are in cmodule scope functions or in `always` or `initial` blocks.

Also unlike Verilog module ports, cmodule port declarations must be in the same order as the ports are listed in the port connection list in the cmodule header.

The following code shows example port declarations:

```
#define width 7
cmodule vector (in4,in2,out4,out2)
    input reg [width:0] in4;
    input bit in2;
```

```
        output reg [width:0] out4;
        output bit out2;
        {
         .
         .
         .
        }
```

In this example:

- The ports are listed in the following order in the cmodule header port connection list:

  in4 in2 out4 out2

  So the port declarations for these ports are in the same order:

  in4

  in2

  out4

  out2

- Ports in4 and out 4 can propagate all four simulation states: 1, 0, x, and z. Ports in2 and out 2 can propagate only two simulation states: 1 and 0.
- Ports in2 and out2 are scalar ports and ports in4 and out4 are eight-bit vector ports.

Cmodule ports of the same direction, type, and size can be declared together, for example:

```
cmodule des1 (in1,in2,out1,out2)
    input reg [7:0] in1,in2;
    output reg [7:0] out1,out2;
    {
     .
```

```
        .
        .
      }
```

There is no such thing as port coercion in cmodules. If you declare a port to be an input port, then the cmodule cannot contain an assignment statement to that port. In Verilog modules you can make an assignment to an input port, VCS will coerce it to an inout port, so that values can propagate both into and out of this port. For example:

```
module top;
reg r1,r2;
wire w1,w2;

initial
begin
#10 r1=1;
    r2=r1;
#100 $finish;
end
inst inst1(r1,w1);
cinst cinst1(r2,w2);
endmodule
```

port coerced to inout

```
module inst (in1,in2);
input in1,in2;
reg r2;

assign #3 in2 = r2;
```

Valid assignment to an input port

```
always @ in1
#5 r2 = ~in1;

endmodule
```

In a cmodule this is not acceptable, therefore in a corresponding example cmodule:

```
cmodule cinst (in1,in2)
input reg in1,in2;
{
always{
    @(in1);
    vc_delay(8);
    in2 = !in1.toChar();
    }
}
```

Assignment to in2 results in a compiler error

In this example you must declare in2 to be an inout or output port.

### Connecting To Cmodule Ports

When you instantiate a cmodule, connect only nets to the inout or output ports. You can connect both registers and nets to input ports.

## Accessing Port Values

In cmodules you use the assignment operator = to assign values to an inout or an output port. You must assign values to the entire port, there is no way to assign to a bit-select or a part-select of an inout or output port.

In cmodules you use access functions to obtain information about ports including their value. You use these access functions to assign the value of an input or inout port to a variable or an inout or output port. You invoke these access functions using the C++ invoke by method technique.

You can assign the value of a bit-select or part-select of a port to a variable including an inout or output port.

The access functions return different data types (like `int` and `char*`) depending on what sort of information you are trying to obtain about the port (such as its size or value) and the nature of the port (such as whether it contains two- or four-state simulation values, or whether it's a scalar or vector port). When obtaining the values of ports logic values are converted to int, char, and string values.

In assignments of values to inout and output ports that can contain four-state simulation values, string and char values such as "x/X", "z/Z", "0", and "1" will be converted to logic values X, Z, 0, and 1. In assignment to inout and output ports that contain two-state simulation values, a "1" is converted to 1 and all string and char values that are not "1" are converted to 0.

The following line-numbered .vc file shows an example cmodule to show you how values are obtained and assigned:

```
#include <iostream.h>
#include <stdio.h>

cmodule bitselect (a,b,c,d,e,f)
input reg a;
input reg [7:0] b;
inout  reg [7:0] c;
output reg d;
output reg [7:0] e;
output reg [1:0] f;
{
    int i;
    always @(b) {
        vc_delay(10);
        c = b.toString();
        i = b.toInteger() ;
        vc_delay(10);
```

```
18         d = a.toChar();
19         vc_delay(10);
20         d = b[7];
21         f = b.range(7,6);
22         vc_delay(10);
23         assigner();
24         vc_delay(10);
25         e = c.toString();
26     }
27
28     void assigner(){
29         d = a.toChar();
30     }
31 }
```

In the always block, on line 15, inout port c is assigned the value of input port b. The access function toString() that was written for the type of port that b is, returns a string value to inout port c.

On line 16 int variable i is assigned the value of input port b, converted to an integer by access function toInteger().

On line 18 output port d is assigned the value of input port a using the access function toChar(). The access function returns a char type, possibly “x”, “z”, “1”, or “0” and the assignment operator converts these char values to their corresponding logic values.

On line 20 output port d is assigned a bit-select of input port b. Using the square brackets [ ] to specify a bit-select also invokes a special access function to return the value of the bit.

On line 21 output port f is assigned a part-select of input port b. The range() access function for this type of port returns a string with the values of the specified bits.

On line 25 output port e is assigned the value of inout port c using the access function toString() that was written for this type of port.

On line 29, in a cmodule function, output port d is assigned the value of input port a using the toChar access function.

## Port Classes

Ports, depending on their direction, size, and type (four-state or two-state), are organized into classes. This implementation detail needs to be pointed out because the access functions that have been implemented for each class of port and error messages from these access functions sometimes refer to the classes of ports. The classes of ports are as follows:

| | |
|---|---|
| `vcInputBit` | Scalar input port for two-state simulation |
| `vcInputBitSv` | Vector input port, 32-bits or fewer, for two-state simulation |
| `vcInputBitv` | Vector input port, more than 32-bits, for two-state simulation |
| `vcInputReg` | Scalar input port for four-state simulation |
| `vcInputRegSv` | Vector input port, 32-bits or fewer, for four-state simulation |
| `vcInputRegv` | Vector input port, more than 32-bits, for four-state simulation |
| `vcInOutBit` | Scalar inout port for two-state simulation |
| `vcInOutBitSv` | Vector inout port, 32-bits or fewer, for two-state simulation |
| `vcInOutBitv` | Vector inout port, more than 32-bits, for two-state simulation |
| `vcInOutReg` | Scalar inout port for four-state simulation |
| `vcInOutRegSv` | Vector inout port, 32-bits or fewer, for four-state simulation |
| `vcInOutRegv` | Scalar inout port for two-state simulation |
| `vcOutputBit` | Scalar output port for two-state simulation |
| `vcOutputBitSv` | Vector output port, 32-bits or fewer, for two-state simulation |
| `vcOutputBitv` | Vector output port, more than 32-bits, for two-state simulation |

| | |
|---|---|
| `vcOutputReg` | Scalar output port for four-state simulation |
| `vcOutputRegSv` | Vector output port, 32-bits or fewer, for four-state simulation |
| `vcOutputRegv` | Scalar output port for two-state simulation |

For each of these classes there is a set of different access functions. Each access function is unique and is implemented for a particular operation for passing data to and from a particular class of port or obtaining other information about that class of port.

However access functions in different classes have the same name if they have the same purpose. For this reason, to avoid unnecessary repetitiveness, these access functions are presented in the remainder of this section, by function, instead of by class.

The following are the access functions from the various classes of ports:

## vcSignal getType()

Type vcSignal is a range of integers from 11 to 28. The getType() functions return a vcSignal that stands for the type of a port specified as the argument. There is, of course, an access function with this name that returns this integer in all classes. The following is an example of its use:

```
#include <stdio.h>
#define SHORT 31
#define LONG 32
cmodule getType (in0,in1,in2,in3,in4,in5,io0,io1,io2,
    io3,io4,io5,out0,out1,out2,out3,out4,out5)
    input reg [LONG:0] in0;
    input reg [SHORT:0] in1;
    input reg in2;
    input bit [LONG:0] in3;
    input bit [SHORT:0] in4;
```

```
    input bit in5;
    inout reg [LONG:0] io0;
    inout reg [SHORT:0] io1;
    inout reg io2;
    inout bit [LONG:0] io3;
    inout bit [SHORT:0] io4;
    inout bit io5;
    output reg [LONG:0] out0;
    output reg [SHORT:0] out1;
    output reg out2;
    output bit [LONG:0] out3;
    output bit [SHORT:0] out4;
    output bit out5;
{
    initial{
        printf("\n input reg in2 type is %d\n",
               in2.getType());
       printf("\n input reg [LONG:0] in0 type is %d\n",
               in0.getType());
      printf("\n input reg [SHORT:0] in1 type is %d\n",
               in1.getType());
        printf("\n input bit in5 type is %d\n",
               in5.getType());
       printf("\n input bit [LONG:0] in3 type is %d\n",
               in3.getType());
      printf("\n input bit [SHORT:0] in4 type is %d\n",
               in4.getType());
        printf("\n output reg out2 type is %d\n",
               out2.getType());
     printf("\n output reg [LONG:0] out0 type is %d\n",
               out0.getType());
    printf("\n output reg [SHORT:0] out1 type is %d\n",
               out1.getType());
        printf("\n output bit out5 type is %d\n",
               out5.getType());
    printf("\n output bit [SHORT:0] out4 type is %d\n",
               out4.getType());
        printf("\n output bit out5 type is %d\n",
               out5.getType());
        printf("\n inout reg io2 type is %d\n",
               io2.getType());
       printf("\n inout reg [LONG:0] io0 type is %d\n",
```

```
                io0.getType());
        printf("\n inout reg [SHORT:0] io1 type is %d\n",
                io1.getType());
         printf("\n inout bit io5 type is %d\n",
                io5.getType());
        printf("\n inout bit [LONG:0] io3 type is %d\n",
                io3.getType());
        printf("\n inout bit [SHORT:0] io4 type is %d\n",
                io4.getType());
    }

}
```

This example prints the following:

```
input reg in2 type is 11

input reg [LONG:0] in0 type is 12

input reg [SHORT:0] in1 type is 13

input bit in5 type is 14

input bit [LONG:0] in3 type is 15

input bit [SHORT:0] in4 type is 16

output reg out2 type is 17

output reg [LONG:0] out0 type is 18

output reg [SHORT:0] out1 type is 19

output bit out5 type is 20

output bit [SHORT:0] out4 type is 22

output bit out5 type is 20

inout reg io2 type is 23
```

```
inout reg [LONG:0] io0 type is 24

inout reg [SHORT:0] io1 type is 25

inout bit io5 type is 26

inout bit [LONG:0] io3 type is 27

inout bit [SHORT:0] io4 type is 28
```

## int toInteger()

The toInteger() functions return an integer value for a scalar port or a short (32-bits or fewer) vector port. These toInteger() functions are in all classes of ports except those for vector ports wider that 32 bits. The following is an example of a cmodule that uses these access functions.

```
#include <stdio.h>
cmodule toInteger
(in1,in2,in3,in4,in5,in6,out1,out2,out3,out4,out5,out6)
input reg in1;
input reg [32:0] in2;
input reg [31:0] in3;
input bit in4;
input bit [32:0] in5;
input bit [31:0] in6;
output reg out1;
output reg [32:0] out2;
output reg [31:0] out3;
output bit out4;
output bit [32:0] out5;
output bit [31:0] out6;
{
always{
   @(in1 or on2 or in3 or in4 or in5 or in6);
   out1 = in1.toInteger();
   out2 = in2.toString();
   out3 = in3.toInteger();
```

```
    out4 = in4.toInteger();
    out5 = in5.toString();
    out6 = in6.toInteger();
    }
}
```

This simple example cmodule passes values from input ports to output ports. The corresponding ports are the same size and are both declared for two-state or four-state simulation.

Notice that assignment statements are for assigning values to output or output ports but that access functions are needed to obtain port values.

Also notice that toInteger() functions were used for scalar and short vector ports, but different functions named toString() were needed for the ports wider that 32 bits.

These functions return a 0 for a Z value and a 1 for an X value.

## char toChar()

The toChar() functions return ‘0’, ‘1’, ‘X’, or ‘Z’ characters for the values of scalar ports. These functions are only in the classes for scalar ports. The following is an example of a cmodule that uses these access functions.

```
#include <stdio.h>
cmodule toChar
(in1,in2,in3,in4,in5,in6,out1,out2,out3,out4,out5,out6)
input reg in1;
input reg [32:0] in2;
input reg [31:0] in3;
input bit in4;
input bit [32:0] in5;
input bit [31:0] in6;
output reg out1;
```

```
output reg [32:0] out2;
output reg [31:0] out3;
output bit out4;
output bit [32:0] out5;
output bit [31:0] out6;
{
always{
   @(in1 or in2 or in3 or in4 or in5 or in6);
   out1 = in1.toChar();
   out2 = in2.toString();
   out3 = in3.toString();
   out4 = in4.toChar();
   out5 = in5.toString();
   out6 = in6.toString();
   }
}
```

Notice that the toChar() functions are only used with scalar ports.

## int getLeftRange()
## int getRightRange()

All vector ports are declared with a bit width that assigns an integer to its most significant bit and its least significant bit. This bit width specifies the size of the vector port because the bit width specifies a range of bits bound by the most and least significant bits.

The getLeftRange() functions return the integer for the most significant bit of a vector port, and the getRightRange() functions return the integer for the least significant bit of a vector port. These functions are in all classes for vector ports. The following cmodule uses these functions:

```
#include <stdio.h>
cmodule ranges (in, out)
input reg [7:0] in;
output reg [7:0] out;
{
```

```
    always{
        @(in);
        int l = in.getLeftRange();
        int r = in.getRightRange();
        int i;
        for (i = l; i >= r; i--){
                printf("\n working on bit [%d]\n\n",i);
        .
        .
        .
        }
    }
}
```

## unsigned int* toArray()

The toArray() functions are for assigning the values of vector ports larger that 32 bits. These toArray() functions are only in the classes for these long vector ports. The following cmodule uses one of these functions:

```
#include <iostream.h>

cmodule inputLvTest (c,d)
input  reg [33:1] c ;
output reg [33:1] d;
{
    always @(c) {
        unsigned int* cVal ;
      .
      .
      .
        cVal = c.toArray()  ;
      .
      .
      .
        d = cVal;
        }
}
```

## int getWord()

Data for vector ports is stored in 32-bit chunks. These functions returns an integer values that is the number of 32-bit chunks needed for a class of vector ports. The getWord() functions are in all classes for vector ports. The following cmodule uses one of these functions:

```
#include <iostream.h>

cmodule inputLvTest (a)
input  reg [131:0] a ;
{
  always @(a) {
    //Array of return value
    unsigned int* aVal = a.toArray();

    cout << " Size of a = " << a.getSize() << endl;
   cout << " No. of 32 bit chunks = " << a.getWord() << endl;
  }
}
```

This cmodule outputs the following:

```
Size of a = 132
No. of 32 bit chunks = 5
```

## char* toString()

The toString() functions return a string of characters for the values of the bits in a vector port. These functions are in all classes for vector ports. The toString() functions for four-state simulation vector ports can return x and z values. The cmodule example for the toChar() functions also uses a toString() function.

## int getSize()

The getSize() functions return the number of bits in a vector port. These functions are in all classes for vector ports. The following cmodule uses a getSize() function:

```
#include <stdio.h>

cmodule getSize (in0,out0)
    input reg [32:0] in0;
    output reg [32:0] out0;
{
   always{
      @(in0);
      if (in0.getSize() == out0.getSize())
         out0 = in0.toString();
   }
}
```

## char* range(int,int)

The range() functions return a string of characters for a part-select of a vector port. You specify the part-select with the int arguments to the functions. There is a range() function in all classes for vector ports. The range() functions for four-state simulation vector ports can return x and z values for bit with these values. The following cmodule uses one of these functions:

```
#include <iostream.h>

cmodule range (inport,outport)
input reg [7:0] inport;
output reg [7:0] outport;
{
   always{
      @(inport);
      cout <<" Middle 4 bits of inport are "
         << inport.range(5,2) << endl ;
```

```
    }
}
```

If the Verilog module that instantiate this cmodule passes a value to the input port:

```
module test;
reg [7:0] a;
wire [7:0] b;

range r1 (a,b);

initial
a = 8'b10xz10xz;

endmodule
```

The cmodule outputs the following:

```
Middle 4 bits of inport are xz10
```