

# 9

Figure 9-0  
Example 9-0  
Syntax 9-0  
Table 9-0

# Tasks and Functions

Tasks and functions provide the ability to execute common procedures from several different places in a description. They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions. Input, output, and inout argument values can be passed into and out of both tasks and functions. The next section discusses the differences between tasks and functions. Subsequent sections describe how to define and invoke tasks and functions and present examples of each.

## 9.1 Distinctions Between Tasks and Functions

The following rules distinguish tasks from functions:

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task; a task can enable other tasks and functions.
- A function must have at least one input argument; a task can have zero or more arguments of any type.
- A function returns a single value; a task does not return a value.

The purpose of a *function* is to respond to an input value by returning a single value. A *task* can support multiple goals and can calculate multiple result values. However, only the output or inout arguments

pass result values back from the invocation of a task. A Verilog model uses a function as an operand in an expression; the value of that operand is the value returned by the function.

For example, you could define either a task or a function to switch bytes in a 16-bit word. The *task* would return the switched word in an output argument, so the source code to enable a task called `switch_bytes` could look like the following example:

```
switch_bytes (old_word, new_word);
```

The task `switch_bytes` would take the bytes in `old_word`, reverse their order, and place the reversed bytes in `new_word`. A word-switching *function* would return the switched word directly. Thus, the function call for the function `switch_bytes` might look like the following example:

```
new_word = switch_bytes (old_word);
```

## 9.2 Tasks and Task Enabling

A task is enabled from a statement that defines the argument values to be passed to the task and the variables that will receive the results. Control is passed back to the enabling process after the task has completed. Thus, if a task has timing controls inside it, then the time of enabling can be different from the time at which control is returned. A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled. Regardless of how many tasks have been enabled, control does not return until all enabled tasks have completed.

## 9.2.1 Defining a Task

The following is the syntax for defining tasks:

```

<task>
  ::= task <name_of_task> ;
     <tf_declaration>*
     <statement_or_null>
     endtask
<name_of_task>
  ::= <IDENTIFIER>
<tf_declaration>
  ::= <parameter_declaration>
     || = <input_declaration>
     || = <output_declaration>
     || = <inout_declaration>
     || = <reg_declaration>
     || = <time_declaration>
     || = <integer_declaration>
     || = <real_declaration>
     || = <event_declaration>
  
```

*Syntax 9-1: Syntax for <task>*

Task and function declarations specify the following:

- local variables
- I/O ports
- registers
- times
- integers
- real
- events

These declarations all have the same syntax as for the corresponding declarations in a module definition.

If there is more than one output, input, and inout port declared in a task these must be enclosed within a block.

## 9.2.2

### Task Enabling and Argument Passing

The statement that enables a task passes the I/O arguments as a comma-separated list of expressions enclosed in parentheses. The following is the formal syntax of the task enabling statement:

```
<task_enable>  
 ::= <name_of_task> ;  
    || = <name_of_task> ( <expression> <,<expression>>* ) ;
```

*Syntax 9-2: Syntax of the task enabling statement*

The first form of a task enabling statement applies when there are no I/O arguments declared in the task body. In the second form, the list of <expression> items is an ordered list that must match the order of the list of I/O arguments in the task definition.

If an I/O argument is an input, then the corresponding <expression> can be any expression. If the I/O argument is an output or an inout, then Verilog restricts it to an expression that is valid on the left-hand side of a procedural assignment. The following items satisfy this requirement:

- reg, integer, real, and time variables
- memory references
- concatenations of reg, integer, real, and time variables
- concatenations of memory references
- bit-selects and part-selects of reg, integer, real, and time variables

The execution of the task enabling statement passes input values from the variables listed in the enabling statement to the variables specified within the task. Execution of the return from the task passes values from the task output and inout variables to the corresponding variables in the task enabling statement. Verilog passes all arguments by value (that is, Verilog passes the *value* rather than a *pointer* to the value).

Example 9-1 illustrates the basic structure of a task definition with five arguments.

---

```
module this_task;
  task my_task;
    input a, b;
    inout c;
    output d, e;
    reg foo1, foo2, foo3;
    begin
      <statements>          // the set of statements that
                          // performs the work of the task

      c = foo1;           // the assignments that initialize
      d = foo2;           // the results variables
      e = foo3;
    end
  endtask
endmodule
```

---

*Example 9-1: Task definition with arguments*

The following statement enables the task in Example 9-1:

```
my_task (v, w, x, y, z);
```

The calling arguments (v, w, x, y, z) correspond to the I/O arguments (a, b, c, d, e) defined by the task. At task enabling time, the input and inout arguments (a, b, and c) receive the values passed in v, w, and x. Thus, execution of the task enabling call effectively causes the following assignments:

```
a = v; b = w; c = x;
```

As part of the processing of the task, the task definition for my\_task must place the computed results values into c, d, and e. When the task completes, the processing software performs the following assignments to return the computed values to the calling process:

```
x = c; y = d; z = e;
```

### 9.2.3 Task Example

Example 9-2 illustrates the use of tasks by redescribing the traffic light sequencer that was introduced in Chapter 8, *Behavioral Modeling*.

---

```
module traffic_lights;
    reg clock, red, amber, green;
    parameter on = 1, off = 0, red_tics = 350,
              amber_tics = 30, green_tics = 200;

    // initialize colors
    initial
        red = off;
    initial
        amber = off;
    initial
        green = off;

    // sequence to control the lights
    always begin
        red = on; // turn red light on
        light(red, red_tics); // and wait.
        green = on; // turn green light on
        light(green, green_tics); // and wait.
        amber = on; // turn amber light on
        light(amber, amber_tics); // and wait.
    end

    // task to wait for 'tics' positive edge clocks
    // before turning 'color' light off
    task light;
        output color;
        input [31:0] tics;
        begin
            repeat (tics)
                @(posedge clock);
            color = off; // turn light off
        end
    endtask

    // waveform for the clock
    always begin
        #100 clock = 0;
        #100 clock = 1;
    end
endmodule // traffic_lights
```

---

Example 9-2: Using tasks

## 9.2.4

### **Effect of Enabling an Already Active Task**

Because Verilog supports concurrent procedures, and tasks can have non-zero time duration, you can write a model that invokes a task when that task is already executing (a special case of invoking a task that is already active is where a task recursively calls itself). Verilog-XL allows multiple copies of a task to execute concurrently, but it does not copy or otherwise preserve the task arguments or local variables. Verilog-XL uses the same storage for each invocation of the task. This means that when the simulator interrupts a task to process another instance of the same task, it overwrites the argument values from the first call with the values from the second call. The user must manage what happens to the variables of a task that is invoked while it is already active.

## 9.3 Functions and Function Calling

The purpose of a function is to return a value that is to be used in an expression. The rest of this chapter explains how to define and use functions.

### 9.3.1 Defining a Function

To define functions, use the following syntax:

```
<function>  
 ::= function <range_or_type>? <name_of_function> ;  
   <tf_declaration>+  
   <statement_or_null>  
   endfunction  
<range_or_type>  
 ::= <range>  
   || = integer  
   || = real  
<name_of_function>  
 ::= <IDENTIFIER>  
<tf_declaration>  
 ::= <parameter_declaration>  
   || = <input_declaration>  
   || = <reg_declaration>  
   || = <time_declaration>  
   || = <integer_declaration>  
   || = <real_declaration>  
   || = <event_declaration>
```

*Syntax 9-3: Syntax for function*

A function returns a value by assigning the value to the function's name. The <range\_or\_type> item which specifies the data type of the function's return is optional.

Example 9-3 defines a function called `getbyte`, using a `<range>` specification.

---

```

module fact;
  function [7:0] getbyte;
    input [15:0] address;
    reg [3:0] result_expression;
    begin
      //<statements> code to extract low-order
      // byte from addressed word
      getbyte = result_expression;
    end
  endfunction
endmodule

```

---

*Example 9-3: A function definition using range*

### 9.3.2 Returning a Value from a Function

The function definition implicitly declares a register, internal to the function, with the same name as the function. This register either defaults to one bit or is the type that `<range_or_type>` specifies. The `<range_or_type>` can specify that the function's return value is a real, an integer, or a value with a range of `[n:m]` bits. The function assigns its return value to the internal variable bearing the function's name. The following line from Example 9-3 illustrates this concept:

```

  getbyte = result_expression;

```

### 9.3.3 Calling a Function

A function call is an operand within an expression. The operand has the following syntax:

<pre> &lt;function_call&gt;   ::= &lt;name_of_function&gt; ( &lt;expression&gt; &lt;,&lt;expression&gt;&gt;* ) &lt;name_of_function&gt;   ::= &lt;identifier&gt; </pre>
---

*Syntax 9-4: Syntax for function\_call*

The following example creates a word by concatenating the results of two calls to the function `getbyte` (defined in Example 9-3).

```
word = control ? {getbyte(msbyte), getbyte(lsbyte)} : 0;
```

### 9.3.4 Function Rules

Functions are more limited than tasks. The following five rules govern their usage:

- A function definition cannot contain any time controlled statements—that is, any statements introduced with `#`, `@`, or `wait`.
- Functions cannot enable tasks.
- A function definition must contain at least one input argument.
- A function definition must include an assignment of the function result value to the internal variable that has the same name as the function.
- A function definition can not contain an `inout` declaration or an `output` declaration.

### 9.3.5 Function Example

Example 9-4 defines a function called `factorial` that returns a 32-bit register. The `factorial` function then calls itself recursively and prints some results.

---

```

module tryfact;
    // define function
    function [31:0] factorial;
        input [3:0] operand;
        reg [3:0] index;
        begin
            factorial = operand ? 1 : 0;

for(index = 2; index <= operand; index = index + 1)
            factorial = index * factorial;
        end
    endfunction

    // Test the function
    reg [31:0] result;
    reg [3:0] n;
    initial
        begin
            result = 1;
            for(n = 2; n <= 9; n = n+1)
                begin

$display("Partial result  n=%d result=%d",
                    n, result);

result = n * factorial(n) / ((n * 2) + 1);
            end
            $display("Final result=%d", result);
        end
endmodule // tryfact

```

---

*Example 9-4: Defining and calling a function*