VERILOG-HDL PLI

Reference Manual

Version 1.0

November 1, 1991

Open Verilog International

Additional copies of this manual may be purchased by contacting OpenVerilog International at the address shown below.

Notices

The information contained in this manual represents the definition of the Programming Language Interface (PLI) as it existed at the time Cadence Design Systems, Inc. transferred PLI and its documentation to Open Verilog International (OVI). This manual does not contain any PLI changes or additions developed or approved by OVI. This information constitutes the basis from which OVI may make refinements and/or additions to PLI.

Open Verilog International makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this manual to a user’s requirements.

Open Verilog International reserves the right to make changes to PLI and this manual at any time without notice.

Open Verilog International does not endorse any particular simulator or other CAE tool based on the Verilog hardware description language and/or PLI.

Suggestions for improvements to the Verilog hardware description language, PLI, and/or this manual will be welcome. They should be sent to the address below.

Information about Open Verilog International and membership enrollment can be obtained by inquiring at the address below.

Published as: PLI Reference Manual, Version 1.0, November 1, 1991.

Printed in the United States of America.

Published by: Open Verilog International
Suite 408
1016 East El Camino Real
Sunnyvale, CA 94087

Phone: (408) 776-1684
FAX: (408) 776-0124

Table of Contents

1

Introduction

This reference manual outlines a C programming interface for a Verilog-HDL simulation environment. It is based on the Cadence PLI Reference Manual and describes the Verilog PLI as implemented by Cadence. This document in no way represents a standard established by the Open Verilog International (OVI), nor should it be interpreted to be the proposed or suggested method of implementation. Instead, it is intended as a baseline and beginning for defining an open Programming Language Interface (PLI). The OVI PLI Task Force is actively working on version 2.0 of the PLI Reference Manual, which will define major enhancements to PLI.

1.1
Procedural Interface Components

The components of an HDL procedural interface can be broken down into the following groups:

- Language interface

  Allows a procedure to be associated with a language construct. For PLI, this is implemented as a link between a C routine and a user-defined system task.

- Language access

  This includes read access to the information contained in the language (e.g. connectivity, along with read/write access to selected items (e.g. delays).

- Dynamic simulation

  This provides read/write access to certain dynamic values, and access to information such as simulation time and state.

- Simulation synchronization

  This can be achieved with a simulation callback mechanism which invokes user routines when predefined events occur such as value changes, simulation time advance, etc.

1.2
Contents of this Manual

This document presents the PLI in the following chapters:

- Access routines

  A set of routines which provide language access, dynamic simulation access, and some synchronization.

- Interface mechanism

  The PLI language interface, and some synchronization methods.

- Utility routines

  A set of routines aimed mainly at dynamic access and synchronization. Much of this code is made obsolete by analogous access routines.

2

Access Routines

2.1 Overview

This chapter describes the PLI access routines, beginning with a general discussion of how and why to use them, followed by descriptions of the individual routines.

2.1.1 Prerequisites

Before exploring access routines, we recommend that you acquire these skills:

1. a working knowledge of the C programming language
2. familiarity with the PLI mechanism
3. an understanding of how to use the Verilog Hardware Design Language (HDL) to connect structural objects in a design hierarchy

Please use the references listed below if you need to review any of these topics before reading further in this chapter:

To learn more about:	Refer to:
programming in C	any C programmer’s manual
how to use the PLI mechanism	chapter 3 in this manual
how to use the HDL	the Verilog-HDL Reference Manual

2.2
What Are Access Routines?

2.2.1
Definition

Access routines are C programming language routines that provide procedural access to information within Verilog-HDL.

2.2.2
What Access Routines Can Do

Access routines perform one of two operations:

1. ***read*** data about particular objects in your circuit design directly from internal data structures
2. ***write*** new information about objects in your circuit design into the internal data structures

Access routines can *read* information about the following objects:

- module instances
- module ports
- module paths
- inter-module paths
- top-level modules
- primitive instances
- primitive terminals
- nets
- registers
- parameters
- specparams
- timing checks
- named events
- integer, real and time variables

Access routines can *write* timing information—delays values or timing check limits—for the following objects:

- inter-module paths
- module paths
- primitive instances
- timing checks

2.2.3 Names

All access routine names indicate the type of information the access routine reads or writes. These names begin with the prefix `acc_` so you can recognize them easily. For example, `acc_fetch_fullname` is the name of an access routine that reads and returns the full hierarchical name of any named accessible object in a design.

2.3 Handles

2.3.1 Definition

A ***handle*** is a predefined data type that is a pointer to a specific object in the design hierarchy. Each handle conveys information to access routines about a unique instance of an accessible object—information about the object's type, plus how and where to find data about the object.

2.3.2 How Handles Work With Access Routines

Most access routines require a handle argument to indicate the objects about which they need to read or write information; many access routines also return handles.

The PLI provides two categories of access routines that return handles for objects: HANDLE routines, which begin with the prefix `acc_handle_`, and NEXT routines, which begin with the prefix `acc_next_`. Refer to Section 2.10 for a discussion of HANDLE routines and Section 2.12 for more information about NEXT routines.

2.3.3 Handle Variables

Handles are passed to and from access routines through ***handle variables***. To declare a handle variable, use the keyword `handle` (all lower case) followed by the variable name, as in this example:

```
handle  net_handle;
```

After you declare a handle variable, you can pass it to any access routine that requires a handle argument or use it to pick up any handle returned by an access routine. The following C-language code fragment uses the variable `net_handle` to store the handle returned by the access routine `acc_handle_object`:

```
                 .
                 .
                 .
handle  net_handle;
net_handle = acc_handle_object("top.mod1.w3");
                 .
                 .
```

2.4 Accessible Objects

Access routines may retrieve and examine information about the following objects:

- module instances
- module ports
- individual bits of a port
- module or data paths
- inter-module paths
- top-level modules
- primitive instances
- primitive terminals
- nets
- registers
- parameters
- specparams
- timing checks
- named events
- integer, real and time variables

Each object allows its own set of access operations. Table 2 - 1 through Table 2-15 in the following sections describe the operations that can be performed for each object type.

2.4.1
Operations on Module Instances

To:	*Use:*
obtain handles for all module instances tagged as cells within a hierarchical scope	***acc_next_cell***
obtain handles for all module instances within a particular module instance	***acc_next_child***
find instance name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find the name of the module definition	***acc_fetch_defname***
find parent (the module instance that contains the instance)	***acc_handle_parent***
obtain the *fulltype* of a module instance (cell instance, module instance, or top-level module)	***acc_fetch_fulltype***

Table 2 - 1: Operations on ***module instances***

2.4.2 Operations on Module Ports

To:	*Use:*
obtain handles for all ports of a module instance	***acc_next_port***
obtain handle for a particular port	***acc_handle_port***
find parent (the module instance that contains the port)	***acc_handle_parent***
find hierarchically higher connected nets	***acc_next_hiconn***
find hierarchically lower connected nets	***acc_next_loconn***
find direction (input, output, inout)	***acc_fetch_direction***
find index	***acc_fetch_index***
obtain the *fulltype* of a module port	***acc_fetch_fulltype***
find hierarchically higher connected net to a scalar module port or bit of a vector port	***acc_handle_hiconn***
find hierarchically lower connected net to a scalar module port or bit of a vector port	***acc_handle_loconn***

Table 2-2: Operations on ***module ports***

2.4.3
Operations on Bits of a Port

To:	*Use:*
obtain handles for all bits of a module port	***acc_next_bit***
read MIPD delays	***acc_fetch_delays***
modify MIPD delays	***acc_append_delays*** ***acc_replace_delays***
find lowest hierarchical name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
obtain the *fulltype* of a port's bit	***acc_fetch_fulltype***

Table 2-3: Operations on ***bits of a port***

2.4.4 Operations on Module or Data Paths

To:	*Use:*
obtain handles for all module paths within a scope	***acc_next_modpath***
find first connected nets	***acc_handle_pathin*** ***acc_handle_pathout***
read delays	***acc_fetch_delays***
modify delays	***acc_append_delays*** ***acc_replace_delays***
find a particular module path	***acc_handle_modpath***
find lowest-level name (name is derived; see Module path names in Section 2.15.7)	***acc_fetch_name***
find full hierarchical name (name is derived; see Module path names in Section 2.15.7)	***acc_fetch_fullname***
find the polarity of a path	***acc_fetch_polarity***
find a particular conditional expression for a path	***acc_handle_condition***
find an edge specifier for a path terminal	***acc_fetch_edge***
obtain handles for all input path terminals of a module or data path	***acc_next_input***
obtain handles for all output path terminals of a module or data path	***acc_next_output***
free the memory associated with an input or output terminal path	***acc_release_object***
find a particular datapath	***acc_handle_datapath***

*Table 2-4: Operations on **module paths***

2.4.5 Operations on Inter-Module Paths

To:	*Use:*
obtain handle for an inter-module path	***acc_handle_path***
read inter-module path delays	***acc_fetch_delays***
modify inter-module path delays	***acc_replace_delays***
find lowest hierarchical name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
obtain the *fulltype* of an inter-module path	***acc_fetch_fulltype***

Table 2-5: Operations on ***inter-module paths***

2.4.6 Operations on Top-Level Modules

To:	*Use:*
obtain handles for all top-level modules in a design	***acc_next_topmod***
find name	***acc_fetch_name***

Table 2-6: Operations on ***top-level modules***

2.4.7 Operations on Primitive Instances

To:	*Use:*
obtain handles for all primitive instances within a module instance	***acc_next_primitive***
find instance name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find definition name	***acc_fetch_defname***
find parent (the hierarchical module instance that contains the primitive)	***acc_handle_parent***
read delays	***acc_fetch_delays***
modify delays	***acc_append_delays*** ***acc_replace_delays***
obtain *fulltype*	***acc_fetch_fulltype***

Table 2-7: Operations on ***primitive instances***

2.4.8 Operations on Primitive Terminals

To:	*Use:*
obtain handles for all terminals of a primitive instance	***acc_next_terminal***
find parent (the primitive instance that contains the terminal)	***acc_handle_parent***
find connected net	***acc_handle_conn***
find direction (input, output, inout)	***acc_fetch_direction***
find index	***acc_fetch_index***
obtain *fulltype*	***acc_fetch_fulltype***

Table 2-8: Operations on ***primitive terminals***

2.4.9 Operations on Nets

To:	*Use:*
obtain handles for all nets within a module instance	***acc_next_net***
find name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find parent (the hierarchical module instance that contains the net)	***acc_handle_parent***
find all driver terminals	***acc_next_driver***
find all load terminals	***acc_next_load***
find connected load terminals, but only one per driven cell port	***acc_next_cell_load***
find logic or strength value	***acc_fetch_value***
find msb and lsb	***acc_fetch_range***
find scalar, vector, collapsed, or expanded nets	***acc_object_of_type***
find net type	***acc_fetch_fulltype***
find size	***acc_fetch_size***

Table 2-9: Operations on ***nets***

2.4.10 Registers

To:	*Use:*
retrieve handles to all objects within a given scope	***acc_next***
find instance name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find parent	***acc_handle_parent***
find bit size	***acc_fetch_size***
retrieve value	***acc_fetch_value***
set the value	***acc_set_value***
find msb and lsb	***acc_fetch_range***
find all load terminals	***acc_next_load***

Table 2-10: Operations on ***registers***

2.4.11 Operations on Parameters

To:	*Use:*
obtain handles for all parameters within a module instance	***acc_next_parameter***
find name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find parent (the module instance that contains the parameter)	***acc_handle_parent***
find integer, floating point or string value	***acc_fetch_paramval***
find type (integer, floating point, string)	***acc_fetch_paramtype*** ***acc_fetch_fulltype***

Table 2-11: Operations on ***parameters***

2.4.12 Operations on Specparams

To:	*Use:*
obtain handles for all specparams within a module instance	***acc_next_specparam***
find name	***acc_fetch_name***
find integer, floating point or string value	***acc_fetch_paramval***
find type (integer, floating point, string)	***acc_fetch_paramtype***

Table 2-12: Operations on ***specparams***

2.4.13 Operations on Timing Checks

To:	*Use:*
obtain handles for all timing checks within a module instance	***acc_next_tchk***
find connected nets	***acc_handle_tchkarg1*** ***acc_handle_tchkarg2***
read limit	***acc_fetch_delays***
modify limit	***acc_append_delays*** ***acc_replace_delays***
find a particular timing check	***acc_handle_tchk***
obtain *fulltype*	***acc_fetch_fulltype***

Table 2-13: Operations on ***timing checks***

2.4.14
Named events

To:	*Use:*
retrieve handles to all objects within a given scope	***acc_next***
find instance name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find parent	***acc_handle_parent***

Table 2-14: Operations on ***named events***

2.4.15
Integer, real, and time variables

To:	*Use:*
retrieve handles to all objects within a given scope	***acc_next***
find instance name	***acc_fetch_name***
find full hierarchical name	***acc_fetch_fullname***
find parent	***acc_handle_parent***
retrieve value	***acc_fetch_value***

Table 2-15: Operations on ***integer, real and time variables***

2.5
Using Access Routines

2.5.1
Header File

You must include the header file, `acc_user.h`, at the top of any C-language source file containing an application program that calls access routines. The *include* statement looks like this:

```
#include  "acc_user.h"
```

Refer to Appendix A to view the contents of `acc_user.h`.

2.5.2
Initializing Access Routines

The access routine `acc_initialize` initializes the environment for access routines and *must* be called from your C-language application program before the program invokes any other access routines.

See Section 2.15.47 for more information about `acc_initialize`.

2.5.3
Setting the Development Version

After initializing access routines, you must also set the configuration parameter `accDevelopmentVersion` to indicate which version of access routines you used to develop the application.

To set this parameter, call `acc_configure` in your C-language application program immediately after you call `acc_initialize`. Following is a sample call that sets `accDevelopmentVersion` to the PLI 1.6a version of access routines:

```
acc_configure(accDevelopmentVersion, "1.6a");
```

Configuring the parameter in this way guarantees that the C-language application code will run successfully with all future releases of access routines. See Section 2.15.5 for more information about `acc_configure`.

2.5.4
Exiting Access Routines

Before exiting a C-language application program that calls access routines, it is necessary to also exit the access routine environment by calling `acc_close` at the end of the program. See Section 2.15.2 for more information about `acc_close`.

2.5.5
Compiling and Linking

To create a new Verilog executable that includes your application program, you must perform the following steps:

1. In the file `veriuser.c`, define the new Verilog system tasks and functions to be associated with your C- language routines using the PLI mechanism (discussed in Chapter 3).
2. Compile your application source files and `veriuser.c`.
3. Link the resulting application and ***veriuser*** object files with the object files that are included with your release.

2.6 Error Handling

When an access routine detects an error, it performs the following operations:

1. sets the global error flag `acc_error_flag` to `true`
2. displays an informative error message at run time to standard output in a format similar to Verilog error messages (unless you specifically suppress error reporting as described in the next section)
3. returns an exception value

When an access routine is called, it automatically resets `acc_error_flag` to `false`.

2.6.1 Suppressing Error Messages

By default, access routines display error messages. Error messages can be suppressed with the use of the access routine `acc_configure` to set the configuration parameter `accDisplayErrors` to `"false"`.

2.6.2 Warnings

When access routines detect warning conditions, they set `acc_error_flag` to `true`, but, by default, do not display warning messages. To instruct the access routines to display warning messages, use the access routine `acc_configure` to set the configuration parameter `accDisplayWarnings` to `"true".`

2.6.3 Testing for Errors

If you decide to suppress automatic error reporting, you can perform your own error handling by checking `acc_error_flag` explicitly after calling a routine. This procedure is described in the flowchart in Figure 2-1:

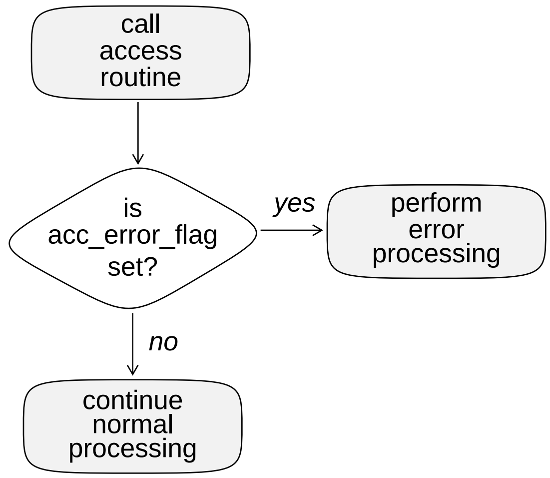


Figure 2-1: How to detect errors

2.6.4 Example: Error Checking for Access Routines

Figure 2-2 shows an example of C-language code that performs error checking for access routines.

```
#include "acc_user.h"

check_new_timing()
{
  handle   gate_handle;

  /* initialize and configure access routines */
  acc_initialize();

  /* suppress error reporting by access routines */
  acc_configure( accDisplayErrors, "false" );

  /* set development version */
  acc_configure( accDevelopmentVersion, "1.6a" );

  /* check type of first argument—the object */
  gate_handle = acc_handle_tfarg( 1 );

  /* check for valid argument */
  if (acc_error_flag)
   tf_error("Cannot derive handle from argument\n");
  else
  /* argument is valid */
  /* make sure it’s a primitive */
   if ( acc_fetch_type(gate_handle) != accPrimitive )
    tf_error("Invalid argument type:not a primitive\n");
  acc_close();
}
```

*Figure 2-2: A **checktf** routine*

This example uses `acc_configure` to suppress automatic error reporting. Instead, it checks `acc_error_flag` explicitly and displays its own specialized error message.

2.6.5 Exception Values

Access routines return one of three exception values when an error occurs:

When routines return:	*the value is:*
int *(integer) values* **or** ***double*** *(double precision floating point) values*	***0***
pointers **or** *handles*	***null***
bool *(boolean) values*	***false***

Table 2-16: Exception values returned by access routines on errors

Because access routines can return valid values that are the same as exception values, the only definitive way to detect errors explicitly is to check `acc_error_flag`.

Note that `null` and `false` are predefined constants, declared in `acc_user.h`.

2.7 String Handling

2.7.1 Access Routines Share an Internal String Buffer

Access routines share an internal buffer to store string values. This buffer is used by all routines that return pointers to strings, as in the following list:

- acc_fetch_defname
- acc_fetch_fullname
- acc_fetch_name
- acc_fetch_value
- acc_fetch_attribute
- acc_fetch_paramval

Each of these routines returns a pointer to the location in the buffer that contains the first character of the string, as illustrated in Figure 2-3. In this example, `mod_name` points to the location in the buffer where `top.m1`—the name of the module associated with `module_handle`—is stored.

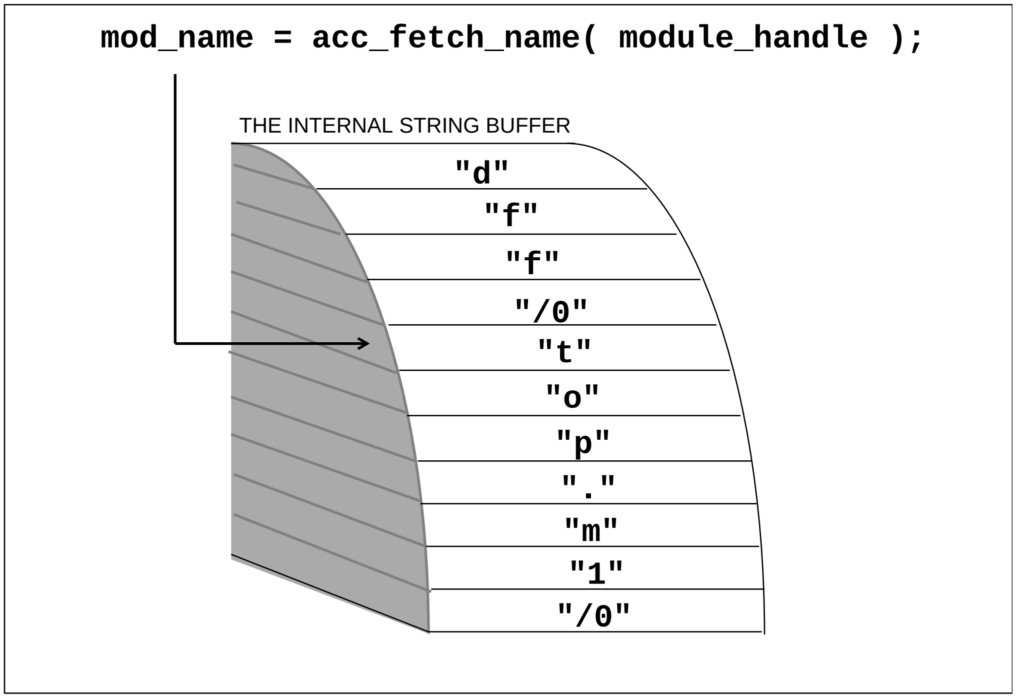
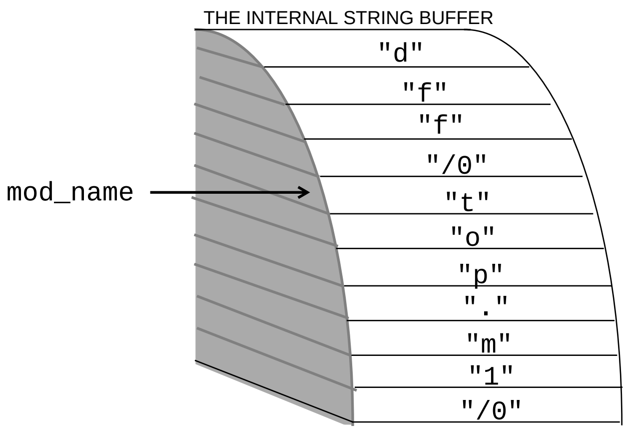
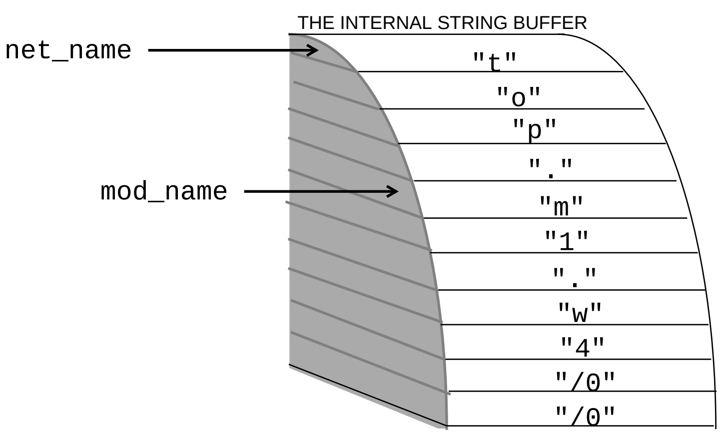


Figure 2-3: How access routines store strings in the internal buffer

2.7.2 Buffer Reset

Access routines always try to place strings at the next available sequential location in the string buffer which, by default, stores up to 4096 characters. However, if there is not enough room to store an entire string starting at that location, a condition known as *buffer reset* occurs.

When buffer reset occurs, the access routine places its string starting at the beginning of the buffer, overwriting data already stored there. The result is a loss of data, as shown in Figure 2-4.

<table>
<tr><th>Action:</th><th>Results:</th></tr>
<tr><td><code>mod_name = acc_fetch_fullname(module_handle);</code>
</td><td><code>mod_name</code> points to <code>"top.m1"</code></td></tr>
<tr><td><code>net_name = acc_fetch_fullname(net_handle);</code>
</td><td>buffer reset occurs

<code>net_name</code> points to <code>"top.m1.w4"</code>

<code>"mod_name</code> points to <code>"m1.w4"</code></td></tr>
</table>

*Figure 2-4: Buffer reset causes **acc_fetch_fullname** to overwrite the name of **module_handle** in the string buffer*

Figure 2-4 shows that the second call to `acc_fetch_fullname` corrupts the module name by overwriting it in the string buffer.

The buffer reset warning

Access routines issue a warning whenever the internal string buffer resets. Remember that to view the warning message, you must use `acc_configure` to set the configuration parameter `accDisplayWarnings` to `"true"`.

2.7.3 Preserving String Values

Applications that use strings for short periods of time—for example, to print names of objects—generally do not need to be concerned about overwrites after a string buffer reset. The risk of losing data is greater, however, for applications that must preserve string values for a long time while calling many access routines that write to the string buffer.

To preserve a string value, use the C routine `strcpy` to copy the string to a local character array that you allocate in your C-language application. These are the steps to follow:

1. Allocate a character array that is large enough to store the string.
2. Call `strcpy` with the following arguments:
    - the pointer to your character array as the first argument
    - the character pointer returned by the access routine as the second argument

Example: Using strcpy to preserve a string value

Consider the example in Figure 2-5. If the module in this example contains many cells, one of the calls to `acc_fetch_name` could eventually overwrite the module name in the string buffer with a cell name. To preserve the module name, `strcpy` is used to store it locally in the array `mod_name`.

```
#include "acc_user.h"

#define  NAME_SIZE   256

void  display_cells_in_module(mod)
handle   mod;
{
  handle   cell;
  char     mod_name[NAME_SIZE];

  /*save the module name in local buffer mod_name*/

  strcpy( mod_name, acc_fetch_fullname( mod ) );


  cell = null;
  while (cell = acc_next_cell( mod, cell ) )
     io_printf( "%s.%s\n", mod_name, acc_fetch_name( cell ) );
}
```

strcpy saves the full module name in array ***mod_name***

the access routine call is passed as the second argument to ***strcpy***

*Figure 2-5: How to use **strcpy***

2.8 Types of Access Routines

The six categories of access routines are:

1. FETCH
2. HANDLE
3. MODIFY
4. NEXT
5. UTILITY
6. VCL

Sections 2.9 through 2.14 describe each *type* of access routine in detail, while Section 2.15 presents an alphabetical list of individual access routines, explaining their functions, syntax, and usage.

2.9 FETCH Routines

2.9.1 Function

FETCH routines return a variety of information about different objects in the design hierarchy.

2.9.2 Names

The name of each routine begins with the prefix `acc_fetch_` and indicates the type of information desired. For example, `acc_fetch_fullname` retrieves the full hierarchical path name for any named object, while `acc_fetch_paramval` retrieves the value of a ***parameter*** or ***specify parameter***.

2.9.3 How to Use FETCH Routines

Follow these steps to use FETCH routines to retrieve data from the design hierarchy:

1. Declare a variable of the same data type as the value returned by the routine.
2. Call the routine with the appropriate number and type of arguments, assigning the return value to the variable.

2.9.4 List of FETCH Routines

Routine name	*Reference*
acc_fetch_attribute	page 2-70
acc_fetch_defname	page 2-76
acc_fetch_delays	page 2-77
acc_fetch_direction	page 2-88
acc_fetch_fullname	page 2-93
acc_fetch_fulltype	page 2-95
acc_fetch_index	page 2-105
acc_fetch_name	page 2-109
acc_fetch_paramtype	page 2-112
acc_fetch_paramval	page 2-114
acc_fetch_size	page 2-120
acc_fetch_tfarg	page 2-122
acc_fetch_type	page 2-124
acc_fetch_type_str	page 2-129
acc_fetch_value	page 2-131
acc_fetch_edge	page 2-90
acc_fetch_location	page 2-107
acc_fetch_polarity	page 2-117
acc_fetch_range	page 2-119

2.10 HANDLE Routines

2.10.1 Function

HANDLE routines return handles to a variety of objects in the design hierarchy.

2.10.2 Names

The name of each routine begins with the prefix `acc_handle_` and indicates the type of handle desired. For example, `acc_handle_object` retrieves a handle for any named object, while `acc_handle_conn` retrieves a handle for a net connected to a particular terminal.

2.10.3 Return Values

Each HANDLE routine returns a handle to an object—a handle that can, in turn, be passed as an argument to another access routine.

2.10.4 How to Use HANDLE Routines

Follow these steps to use HANDLE routines for retrieving handles to objects in the design hierarchy:

1. Declare a handle variable, as described in Section 2.3.
2. Call the routine with the appropriate number and type of arguments, assigning the return value to the handle variable.

2.10.5 List of HANDLE Routines

2.11 MODIFY Routines

2.11.1 Function

MODIFY routines alter the values of a variety of objects in the design hierarchy. The following table shows the types of values that can be modified for particular objects:

MODIFY routines alter:	*for these objects:*
delay values	primitives module paths inter-module paths module input ports timing checks
value	register UDPs

Table 2-17: Values that can be modified

2.11.2 List of MODIFY Routines

Routine name	*Reference*
acc_append_delays	page 2-37
acc_replace_delays	page 2-229
acc_set_value	page 2-244

2.12 NEXT Routines

2.12.1 Function

When used inside a loop construct, NEXT routines find each object of a given type that is related to a particular reference object in the design hierarchy.

2.12.2 Names

The name of each routine begins with the prefix `acc_next_` and indicates the type of object desired—the *target* object. For example, `acc_next_net` retrieves each net in a module, while `acc_next_driver` retrieves each terminal driving a net.

2.12.3 Reference Objects and Target Objects

A *target* object is the type of object to be returned by a NEXT routine. A *reference* object indicates to the NEXT routine where it must look for its first target object. Often, a reference object relates to a target object in some way—for example, by *containing* or *connecting to* the target object.

2.12.4 Arguments

Typically, NEXT routines require *two* arguments:

1. The first argument is a handle to the reference object.
2. The second argument is an input handle that indicates whether to retrieve the *first* or *next* target object.

The one exception is `acc_next_topmod` which finds each top-level module in the design; its reference object—top-level module— is implied in the routine name, so it does not require the first argument.

2.12.5 Return Values

Each call to a NEXT routine returns a handle to the object it finds. You can pass this handle as an argument to other access routines.

2.12.6 How NEXT Routines Work

The following table summarizes how NEXT routines work (assuming their first argument is a valid reference object):

When:	***NEXT routine returns:***
the second argument is ***null***	handle to the first target object related to the reference object
the second argument is a handle to a target object returned by a previous call to the routine	handle to the next target object related to the reference object
no target objects remain for a particular reference object	***null*** handle
no target objects are found initially for a particular reference object	***null*** handle
an error occurs	***null*** handle

*Table 2-18: How **NEXT** routines work*

Looping

Each call to a NEXT routine returns only one handle; therefore, to retrieve all objects of a desired type for a particular reference object, place the NEXT routine in a ***while*** loop that terminates when the routine returns ***null***.

2.12.7 How to Use NEXT Routines

Follow these steps to use NEXT routines for retrieving all objects of a given type at a certain level in the design hierarchy:

1. Declare handle variables for the *reference object* argument and the *first/next* argument.
2. Use an access routine to retrieve the handle of the desired *reference* object.
3. Set the *first/next* handle variable to ***null***.
4. Call the NEXT routine using the same handle variable for the return value and for the *first/next* argument; this technique automatically updates the *first/next* argument to point to the last object found for a particular reference object.
5. Place the NEXT routine call inside a ***while*** loop that terminates when the return value is ***null***.

2.12.8 Example: Display Names of Nets Within a Module

The following sample C-language routine `display_net_names` uses a NEXT routine to display the names of all nets in a module.

```
#include "acc_user.h"

display_net_names()
{
   handle   module_handle;
   handle   net_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set the development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg(1);

   /*display names of all nets in the module*/
   net_handle = null;
   while( net_handle = acc_next_net( module_handle, net_handle ) )
      io_printf( "Net name is: %s\n", acc_fetch_fullname( net_handle ) );

   acc_close();
}
```

Figure 2-6: Displaying names of nets in a module

2.12.9 List of NEXT Routines

2.13 UTILITY Routines

2.13.1 Function

UTILITY routines perform a variety of operations, such as initializing and configuring the access routine environment.

2.13.2 List of UTILITY Routines

Routine name	*Reference*
acc_close	page 2-49
acc_collect	page 2-51
acc_compare_handles	page 2-53
acc_configure	page 2-55
acc_count	page 2-68
acc_free	page 2-134
acc_initialize	page 2-178
acc_object_in_typelist	page 2-221
acc_object_of_type	page 2-223
acc_set_scope	page 2-241
acc_version	page 2-253
acc_release_object	page 2-226
acc_product_version	page 2-225

2.14 VCL Routines

The Value Change Link (VCL) allows a PLI application to monitor the value changes of selected objects. It consists of two PLI access routines that tell the Verilog simulator to start or stop informing an application when an object changes value.

2.14.1 VCL Objects

The VCL can monitor value changes for the following objects:

- events
- scalar and vector registers
- scalar nets
- bit-selects of expanded vector nets
- unexpanded vector nets

The VCL cannot extract information about the following objects:

- bit-selects of unexpanded vector nets or registers
- part-selects
- memories
- expressions (such as a+b)

2.14.2 List of VCL Routines

Routine name	*Reference*
acc_vcl_add	page 2-247
acc_vcl_delete	page 2-250

2.15
Alphabetical List of Access Routines

This section describes the various PLI access routines, explaining their function, syntax, and usage. The routines are described in alphabetical order.

2.15.1 acc_append_delays

acc_append_delays for single delay per transition			
function	when accMinTypMaxDelays is "false", adds delays passed as arguments to existing delay values for primitives, module paths, timing checks or module input ports		
syntax			
primitives:	**acc_append_delays(** *primitive_handle, rise_delay, fall_delay, z_delay* **)**;		
module paths:	**acc_append_delays(** *path_handle, delay1, delay2, delay3*, *delay4*, *delay5, delay6* **)**;		
timing checks:	**acc_append_delays(** *timing_check_handle, limit* **)**;		
ports:	**acc_append_delays(** *port_handle, rise_delay, fall_delay, z_delay* **)**;		
port's bits:	**acc_append_delays(** *bit_handle, rise_delay, fall_delay, z_delay* **)**;		
arguments	name	type	description
inputs:	**object_handle**	*handle*	handle of a primitive, module path, timing check, module input port or bit of a module input port
primitives, ports, port's bits:	**rise_delay**	*double*	object's rise delay
	fall_delay	*double*	object's fall delay
	z_delay (depends on accToHiZDelay)	*double*	object's turn-off delay
module paths	**delay1**	*double*	module path's delay for transitions determined by accPathDelayCount
	delay2 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay3 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay4 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay5 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay6 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
timing checks	**limit**	*double*	timing check's limit
related routines	Use ***acc_configure(accMinTypMaxDelays, "false")*** for single delay per transition Use ***acc_configure(accPathDelayCount...)*** to set number of module path delays to append Use ***acc_configure(accToHiZDelay...)*** to calculate turn-off delays from rise and fall delay values		

<table>
<tr><th colspan="4">acc_append_delays (min:typ:max delays)</th></tr>
<tr><td>function</td><td colspan="3">when accMinTypMaxDelays is "true", adds min:typ:max delay values contained in an array to existing delay values for primitives, module paths, timing checks or module input ports</td></tr>
<tr><td>syntax</td><td colspan="3"></td></tr>
<tr><td>primitives:</td><td colspan="3">acc_append_delays(primitive_handle, array_ptr);</td></tr>
<tr><td>module paths:</td><td colspan="3">acc_append_delays(path_handle, array_ptr);</td></tr>
<tr><td>timing checks:</td><td colspan="3">acc_append_delays(timing_check_handle, array_ptr);</td></tr>
<tr><td>ports:</td><td colspan="3">acc_append_delays(port_handle, array_ptr);</td></tr>
<tr><td>port’s bits:</td><td colspan="3">acc_append_delays(bit_handle, array_ptr);</td></tr>
<tr><td>arguments</td><td>name</td><td>type</td><td>description</td></tr>
<tr><td>inputs:</td><td>object_handle</td><td>handle</td><td>handle of a primitive, module path, timing check, module input port or bit of a module input port</td></tr>
<tr><td></td><td>array_ptr</td><td>double address</td><td>pointer to array of min:typ:max values (see Table 2-20)</td></tr>
<tr><td>related routines</td><td colspan="3">Use acc_configure(accMinTypMaxDelays, "true") for min:typ:max delays for each transition</td></tr>
</table>

The access routine `acc_append_delays` works differently depending on how the configuration parameter `accMinTypMaxDelays` is set. When this parameter is set to `"false"`, `acc_append_delays` assumes a single delay per transition and expects each delay to be passed as an individual argument. For this ***single delay mode***, the first syntax table in this section applies.

When `accMinTypMaxDelays` is set to `"true"`, `acc_append_delays` expects one or more sets of *min:typ:max* delays to be passed in an array, rather than single delays passed as individual arguments. For this ***min:typ:max*** delay mode, the second syntax table in this section applies.

Different delays for different objects

The routine `acc_append_delays` writes different delay values for different objects, as summarized in Table 2-19 and Table 2-20. Table 2-19 applies when `acc_append_delays` is set for *single delay mode* (`accMinTypMaxDelays` is `"false"`) and Table 2-20 applies when this routine is set for *min:typ:max delay mode* (`accMinTypMaxDelays` is `"true"`).

For:	***acc_append_delays*** *writes:*
primitives with a Z state: ***bufif*** gates ***notif*** gates ***MOS*** switches	***two*** or ***three*** delays: depends on how you configure ***accToHiZDelay***
primitives with ***no*** Z state	***two*** delays: rise delay, ***rise_delay*** fall delay, ***fall_delay***
module paths	***one***, ***two***, ***three*** or ***six*** delays: depends on how you configure ***accPathDelayCount***
timing checks	***one*** delay: timing check limit, ***limit***
module input or inout ports (MIPDs)	***two*** or ***three*** delays: depends on how you configure ***accToHiZDelay***

Table 2-19: How ***acc_append_delays*** *writes delays in single delay mode*

In *single delay mode*, it is up to the user to supply the correct number of delay arguments to `acc_append_delays`, as follows:

- MIPDs and Z-state primitives require ***three*** delay arguments—`rise_delay`, `fall_delay` and `Z_delay`—if `accToHiZDelay` is set to "`from_user`" (the default).
- MIPDs and Z-state primitives require ***two*** delay arguments—`rise_delay` and `fall_delay`—if `accToHiZDelay` is set to "`average`", "`max`" or "`min`".
- Primitives with no Z state require ***two*** delay arguments—`rise_delay` and `fall_delay`.
- Module paths require ***one***, ***two***, ***three*** or ***six*** delay arguments, depending on how you configure `accPathDelayCount`.
- Timing checks require ***one*** limit argument—`limit`.

For more information on the configuration parameters `accToHiZDelay` and `accPathDelayCount`, refer to Table 2-21, Table 2-22 and Table 2-23.

Table 2-20 shows how `acc_append_delays` writes delays in *min:typ:max delay mode*.

For:	***acc_append_delays*:**	*The delay array must pass:*
module input ports primitives	writes *three* sets of *min:typ:max* delays: one set for rise delays one set for fall delays one set for turn-off delays	*nine* values: array[0] = minimum rise delay array[1] = typical rise delay array[2] = maximum rise delay array[3] = minimum fall delay array[4] = typical fall delay array[5] = maximum fall delay array[6] = minimum turn-off delay array[7] = typical turn-off delay array[8] = maximum turn-off delay ***(you must always declare an array of size 9, even if turn-off delays are not used)***
module paths	writes *one*, *two*, *three* or *six* sets of *min:typ:max* delays: depends on how you configure `accPathDelayCount` ***(see Table 2-22)***	*three*, *six*, *nine* or*18* values: ***(see Table 2-22)***
timing checks	writes *one* set of *min:typ:max* delays: timing check limit	*three* values: array[0] = minimum timing check limit array[1] = typical timing check limit array[2] = maximum timing check limit

Table 2-20: How ***acc_append_delays*** *writes delays in* ***min:typ:max*** *delay mode*

There are several points to note in Table 2-20:

- For module input ports and primitives, always declare an array of size 9, even if the primitives do not have a Z state.
- The routine `acc_append_delays` does not support inter-module paths.
- The configuration parameter `accPathDelayCount` affects the *min:typ:max* delays processed for module paths. Table 2-22 describes these effects in greater detail.

Setting delays for module paths (single delay mode)

In *single delay mode*, you can control how many delays `acc_append_delays` writes for module paths by using the access routine `acc_configure` to set the delay count parameter `accPathDelayCount` as shown in Table 2-21.

When ***accPathDelayCount*** *is:*	***acc_append_delays*** *writes:*
1	***one*** delay value, the same for all transitions: ***delay1*** *(last five delay arguments may be dropped)*
2	***two*** delay values: one for rising transitions: ***delay1*** one for falling transitions***: delay2*** *(last four delay arguments may be dropped)*
3	***three*** delay values: one for rising transitions: ***delay1*** one for falling transitions: ***delay2*** one for transitions to z: ***delay3*** *(last three delay arguments may be dropped)*
6 (the default)	all ***six*** delay values, a different delay for each possible transition among 0, 1 and z: one for 01 transitions: ***delay1*** one for 10 transitions: ***delay2*** one for 0z transitions: ***delay3*** one for z1 transitions: ***delay4*** one for 1z transitions: ***delay5*** one for z0 transitions: ***delay6***

Table 2-21: How ***accPathDelayCount*** *affects module path delays in single delay mode*

The minimum number of delay arguments to pass to `acc_append_delays` must equal the value of `accPathDelayCount`. You may drop any remaining arguments.

The following example shows how to set `accPathDelayCount` so that `acc_append_delays` writes rise and fall delays for module paths:

```
acc_configure( accPathDelayCount, "2" );
```

If you do not set `accPathDelayCount` explicitly, it defaults to ***6***; in this case, you must pass all six delay arguments when you call `acc_append_delays` in *single delay mode*.

Setting delays for module paths (min:typ:max mode)

The following rules apply when you modify *min:typ:max* delays for module paths:

- the number of *sets* of *min:typ:max* delays must equal the value of `accPathDelayCount`
- the size of the delay array must be three times the value of `accPathDelayCount`

Table 2-22 summarizes how `accPathDelayCount` affects *min:typ:max* delays for module paths.

When ***accPathDelayCount*** *is:*	*The number of sets of min:typ:max path delays is:*	*The delay array must pass or retrieve:*
"1"	**one**: the same minimum, typical and maximum delay for all transitions	**three values:** `array[0]` = minimum delay `array[1]` = typical delay `array[2]` = maximum delay
"2"	**two**: one set for rising transitions one set for falling transitions	**six values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay
"3"	**three**: one set for rising transitions one set for falling transitions one set for transitions to z	**nine values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay `array[6]` = minimum turn-off delay `array[7]` = typical turn-off delay `array[8]` = maximum turn-off delay
"6" *(the default)*	**six:** one set for 01 transitions one set for 10 transitions one set for 0z transitions one set for z1 transitions one set for 1z transitions one set for z0 transitions	**18 values:** `array[0]` = minimum 01 delay `array[1]` = typical 01 delay `array[2]` = maximum 01 delay `array[3]` = minimum 10 delay `array[4]` = typical 10 delay `array[5]` = maximum 10 delay `array[6]` = minimum 0z delay `array[7]` = typical 0z delay `array[8]` = maximum 0z delay `array[9]` = minimum z1 delay `array[10]` = typical z1 delay `array[11]` = maximum z1 delay `array[12]` = minimum 1z delay `array[13]` = typical 1z delay `array[14]` = maximum 1z delay `array[15]` = minimum z0 delay `array[16]` = typical z0 delay `array[17]` = maximum z0 delay

Table 2-22: The relationship between ***accPathDelayCount*** *and* ***min:typ:max*** *delays for module paths*

Setting module input port delays (MIPDs)

Use `acc_append_delays` to specify Module Input Port Delays (MIPDs). An MIPD is a delay associated with a module input port or inout port. The MIPD describes the delay between the module port and each of the loads in its fanout. In an MIPD you can specify rise, fall, and high impedance propagation delays.

You can write an MIPD for each individual bit of a vector port using `acc_append_delays` in conjunction with `acc_next_bit`. For more information, see Section 2.15.49.

Declaring the array that holds min:typ:max values

Use Table 2-20 and Table 2-22 to decide how large to make the array that passes or holds *min:typ:max* values. The array must be able to store the correct number of delays that will be processed. Declaring an array that is too small will cause errors or unpredictable results.

Calculating turn-off delays from rise and fall delays

In *single delay mode*, you can instruct `acc_append_delays` to automatically calculate turn-off delays from rise and fall delays by using the access routine `acc_configure` to set the parameter `accToHiZDelay` as follows:

*When **accToHiZDelay** is:*	***acc_append_delays:***
"average"	calculates turn-off delay as the average of the rise and fall delays
"min"	calculates turn-off delay as the smaller of the rise and fall delays
"max"	calculates turn-off delay as the larger of the rise and fall delays
"from_user" (default)	sets turn-off delay to the value passed as a user-supplied argument: ***z_delay*** for primitives, ports and port's bits.

*Table 2-23: How the value of **accToHiZDelay** affects **acc_append_delays***

The following example shows how to set `accToHiZDelay` so that `acc_append_delays` calculates turn-off delays as the average of rise and fall delays for Z-state primitives:

```
acc_configure( accToHiZDelay, "average" );
```

Note that the value you assign to `accToHiZDelay` also influences how `acc_replace_delays` derives turn-off delays.

Effect of timescales

The routine `acc_append_delays` writes delays in the timescale of the module that contains `object_handle`.

Usage example: single delay mode

The following C-language routine, `write_gate_delays`, is an example of back annotation. It reads new delay values from a file called `primdelay.dat` and uses `acc_append_delays` to add them to the current delays on a gate. The format of the file is shown in Figure 2-7. The source code appears in Figure 2-8.

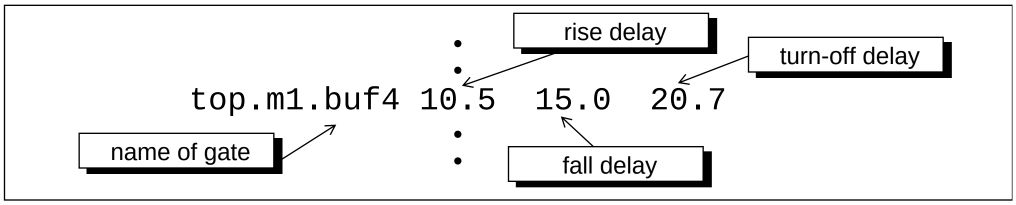


*Figure 2-7: Format of file **primdelay.dat***

```
#include <stdio.h>
#include "acc_user.h"

write_gate_delays()
{
   FILE     *infile;
   char     full_gate_name[NAME_SIZE];
   double   rise,fall,toz;
   handle   gate_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*read delays from file - "r" means read only*/
   infile = fopen("primdelay.dat","r");
   while( fscanf(infile, "%s %lf %lf %lf",
                 full_gate_name,rise,fall,toz) != EOF )
   {

      /*get handle for the gate*/
      gate_handle = acc_handle_object(full_gate_name);

      /*add new delays to current values for the gate*/
      acc_append_delays( gate_handle, rise, fall, toz );
   }
   acc_close();
}
```

*Figure 2-8: Using **write_gate_delays** to back annotate delay values*

Note that the identifier `EOF` is a predefined constant that stands for ***end of file***; `NAME_SIZE` is a user-defined constant that represents the maximum number of characters allowed for any object name in an input file.

Usage example: min:typ:max delay mode

To append *min:typ:max* delays for a primitive that does not have a Z state, follow these steps in your C routine:

1. **Declare an array of 9 double-precision floating-point values** to hold three sets of *min:typ:max* values, one each for rising transitions, falling transitions and transitions to Z.
2. **Set the configuration parameter `accMinTypMaxDelays` to "`true`"** to instruct `acc_append_delays` to write delays in *min:typ:max* format.
3. **Call `acc_append_delays`** with a valid primitive handle and the array pointer.

The following example C-language application shows how to append *min:typ:max* delays for a primitive. Assume for this example that `append_mintypmax_delays` is associated through the interface mechanism with a user-defined system task called `$appendprimdelays`. A primitive with no Z state and new delay values are passed as arguments to `$appendprimdelays` as follows:

primitive (no Z state)

```
$appendprimdelays( g1, 3.0, 5.0, 6.7, 2.4, 8.1, 9.1 );
```

minimum rise delay **typical rise delay** **maximum rise delay** **minimum fall delay** **typical fall delay** **maximum fall delay**

```
#include "acc_user.h"
#include "veriuser.h"

append_mintypmax_delays( )
{
    handle   prim;
    double   delay_array[9];
    int      i;

    acc_configure( accMinTypMaxDelays, "true" );

    /* get handle for primitive */
    prim = acc_handle_tfarg( 1 );

    /* store new delay values in array */
    for ( i = 0; i < 9; i++ )
       delay_array[i] = acc_fetch_tfarg(i+2);

    /* append min:typ:max delays */
    acc_append_delays( prim, delay_array );
}
```

delay_array must be large enough to hold *nine* values to handle both Z-state primitives and primitives with no Z states

*Figure 2-9: Appending **min:typ:max** delays on a primitive*

In this example, `acc_append_delays` modifies delays as follows:

- sets the minimum rise delay to 3.0 time units
- sets the typical rise delay to 5.0 time units
- sets the maximum rise delay to 6.7 time units
- sets the minimum fall delay to 2.4 time units
- sets the typical fall delay to 8.1 time units
- sets the maximum fall delay to 9.1 time units
- does not write minimum turn-off, typical turn-off and maximum turn-off delays because `prim` has no Z state

2.15.2 acc_close

acc_close	
function	frees internal memory used by access routines; resets all configuration parameters to default values
syntax	**acc_close*( )*;**
arguments	*none*
related routines	***acc_initialize()***

The difference between acc_close and acc_free

The routine `acc_close` frees memory that all access routines use for internal storage; it also resets all configuration parameters to their default values. In contrast, the routine `acc_free` frees memory that you allocate to store the array of handles returned by `acc_collect`.

When to call acc_close

Call `acc_close` at the end of any C-language routine that calls access routines. It is important that you do not call any other access routines *after* calling `acc_close`.

Avoiding application interference

Potentially, multiple PLI applications running in the same simulation session can interfere with each other because they share the same set of configuration parameters. To guard against application interference, both `acc_initialize` and `acc_close` reset any configuration parameters that have changed from their default values.

Usage example

The following example presents a C-language routine, `show_versions`, that calls `acc_close` before exiting.

```
#include "acc_user.h"

 show_versions()
{
 handle  module_handle;

 /*initialize environment for access routines*/
 acc_initialize();

 /*set development version*/
 acc_configure( accDevelopmentVersion, "1.6a" );

 /*show version of access routines*/
 /* and version of simulator that is linked to access routines*/
 io_printf("Running %s with %s\n",acc_version(),acc_product_version() );

   acc_close();
}
```

*Figure 2-10: Using **show_versions** to display the version of access routines linked to a Verilog simulator*

2.15.3 acc_collect

acc_collect			
function	returns a pointer to an array that contains handles for all objects related to a particular reference object		
syntax	*handle_array_pointer* **= acc_collect(***NEXT_routine, object_handle, number_of_objects***)***;*		
arguments	name	type	description
inputs:	**NEXT_routine**	*pointer to a NEXT routine:*	actual name (unquoted) of the NEXT routine that finds the object of interest
	object_handle	*handle*	handle of the reference object
output:	**number_of_objects**	*integer address*	number of objects collected
related routines	all ***NEXT*** routines Use ***acc_free*** to free memory allocated by ***acc_collect*** for the handle array		

What is a reference object?

A *reference object* provides a frame of reference for a NEXT routine to indicate where it must look for its first target object. Often, a reference object relates to a target object in some way—for example, by *containing* or *connecting to* the target object.

The object associated with `object_handle` must be the same as the reference object required by `NEXT_routine`.

When to use acc_collect instead of a NEXT routine

Use `acc_collect` in either of these situations:

- to retrieve data that you plan to use more than once
- instead of using nested or concurrent calls to `acc_next_loconn`, `acc_next_hiconn`, `acc_next_load`, and `acc_next_cell_load` routines

Otherwise, it is more efficient to use NEXT routines.

You cannot pass acc_next_topmod to acc_collect

The access routine `acc_next_topmod` does not work with `acc_collect`. However, you can collect top-level modules by passing `acc_next_child` with a null reference object argument. Here is a sample call that collects top-level modules:

```
acc_collect( acc_next_child, null, &count );
```

Managing memory

The routine `acc_collect` allocates memory for the array of handles it returns. When you no longer need the contents of the handle array, it is important to free this memory by calling the access routine `acc_free`, described in Section 2.15.26.

Usage example

The following example presents a C-language routine, `display_nets`, that uses `acc_collect` to collect and display all nets in a module.

```
#include "acc_user.h"

display_nets()
{
   handle   *list_of_nets,module_handle;
   int      net_count, i;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for the module*/
   module_handle = acc_handle_tfarg(1);

   /*collect all nets in the module*/
   list_of_nets = acc_collect( acc_next_net, module_handle, &net_count);

   /*display names of net instances*/
   for( i=0; i < net_count; i++ )
      io_printf( "Net name is: %s\n", acc_fetch_name( list_of_nets[i] ));

   /*free memory used by array list_of_nets*/
   acc_free( list_of_nets );

   acc_close();
}
```

*Figure 2-11: Using **display_nets** to display all nets in a module*

2.15.4 acc_compare_handles

acc_compare_handles			
function	returns true if the two input handles refer to the same object		
syntax	*bool_value* **= acc_compare_handles(***handle1, handle2***)***;*		
arguments	name	type	description
inputs:	**handle1** **handle2**	*handle*	handles to any objects

In some cases, two different handles can reference the same object if each handle is retrieved in a different way—for example, if a NEXT routine returns one handle and `acc_handle_object` returns the other. The access routine `acc_compare_handles` allows you to determine if two handles refer to the same object.

Use acc_compare_handles instead of ==

Do ***not*** use the following `if` construct to compare two handles:

```
if ( handle1 == handle2 )   /* this is not correct */
```

You cannot use the `==` operator to determine if two handles reference the same object.

Usage example

The following C-language function uses `acc_compare_handles` to determine if a primitive drives the specified output of a scalar port of a module.

```
#include "acc_user.h"
#include "veriuser.h"

bool prim_drives_scalar_port( prim, mod, port_num )
handle prim, mod;
int    port_num;
{
  /* retrieve net connected to scalar port */
  handle  port = acc_handle_port( mod, port_num );
  handle  port_conn = acc_next_loconn( port, null );

  /* retrieve net connected to primitive output */
  handle  out_term = acc_handle_terminal( prim, 0 );
  handle  prim_conn = acc_handle_conn( out_term );

  /* compare handles */
  if ( acc_compare_handles( port_conn, prim_conn ) )
    return( true );
  else
    return( false );
}
```

If port_conn and prim_conn refer to the same connection, then the prim drives port

*Figure 2-12: Using **acc_compare_handles** to compare two handles*

2.15.5 acc_configure

acc_configure			
function	sets parameters that control the operation of various access routines		
syntax	*load_handle* **= acc_configure(***configuration_parameter, configuration_value***);**		
arguments	name	type	description
inputs:	**configuration_parameter**	*character_string_constant*	a parameter that controls the operation of one or more access routines, chosen from one of the following: ***accDefaultAttr0*** ***accDevelopmentVersion*** ***accDisplayErrors*** ***accDisplayWarnings*** ***accEnableArgs*** ***accMapToMipd*** ***accMinTypMaxDelays*** ***accPathDelayCount*** ***accPathDelimStr*** ***accToHiZDelay***
	configuration_value	*character string pointer:* quoted string literal *or* character pointer variable	one of a fixed set of values for ***configuration_parameter***
related routines	For ***accDefaultAttr0***: ***acc_fetch_attribute*** For ***accDevelopmentVersion*** all access routines For ***accDisplayErrors***: all access routines For ***accDisplayWarnings*** all access routines For ***accEnableArgs*** ***acc_handle_modpath*** ***acc_handle_tchk*** ***acc_set_scope***	For ***accMapToMipd***: ***acc_append_delays*** ***acc_replace_delays*** For ***accMinTypMaxDelays***: ***acc_append_delays*** ***acc_fetch_delays*** ***acc_replace_delays*** For ***accPathDelayCount***: ***acc_append_delays*** ***acc_fetch_delays*** ***acc_replace_delays***	For ***accPathDelimStr***: ***acc_fetch_attribute*** ***acc_fetch_fullname*** ***acc_fetch_name*** For ***accToHiZDelay***: ***acc_append_delays*** ***acc_replace_delays***

List of parameters and their values

The following tables describe each parameter and its set of values. A call to either `acc_initialize` or `acc_close` sets each configuration parameter to its default value (see Sections 2.15.2 and 2.15.47 for further information).

	Set of values	Effect	Default
accDefaultAttr0	"true"	***acc_fetch_attribute*** returns zero when it does not find the attribute requested, and ignores the argument ***default_value***	"false"
	"false"	***acc_fetch_attribute*** returns the value passed as the argument ***default_value*** when it does not find the attribute requested	

*Table 2-24: Using **accDefaultAttr0** configuration parameter*

	Set of values	Effect	Default
accDevelopmentVersion	a sequence of letters, numbers, and the period character that form a valid PLI version, such as "1.6a"	ensures backward compatibility by indicating to the PLI which version of access routines was used to develop a particular C-language application	current version of access routines

*Table 2-25: Using **accDevelopmentVersion** configuration parameter*

	Set of values	Effect	Default
accDisplayErrors	"true"	access routines display error messages	"true"
	"false"	access routines do *not* display error messages	

*Table 2-26: Using **accDisplayErrors** configuration parameter*

	Set of values	Effect	Default
accDisplayWarnings	"true"	access routines display warning messages	"false"
	"false"	access routines do *not* display warning messages	

*Table 2-27: Using **accDisplayWarnings** configuration parameter*

	Set of values	Effect	Default
accEnableArgs	"acc_handle_modpath"	***acc_handle_modpath*** recognizes its optional arguments	"no_acc_handle_modpath" "no_acc_handle_tchk" "no_acc_set_scope"
	"no_acc_handle_modpath"	***acc_handle_modpath*** ignores its optional arguments	
	"acc_handle_tchk"	***acc_handle_tchk*** recognizes its optional arguments	
	"no_acc_handle_tchk"	***acc_handle_tchk*** ignores its optional arguments	
	"acc_set_scope"	***acc_set_scope*** recognizes its optional arguments	
	"no_acc_set_scope"	***acc_set_scope*** ignores its optional arguments	

*Table 2-28: Using **accEnableArgs** configuration parameter*

	Set of values	Effect	Default
accMapToMipd	"max"	Map the longest inter-module path delay to the MIPD	"max"
	"min"	Map the shortest inter-module path delay to the MIPD	
	"latest"	Map the inter-module path delay last modified by ***acc_replace_delays*** to the MIPD	

*Table 2-29: Using **accMapToMipd** configuration parameter*

	Set of values	Effect	Default
accMinTypMaxDelays	"true"	***acc_append_delays***, ***acc_fetch_delays***, ***acc_replace_delays*** handle one, two, three, or six sets of min:typ:max delays passed in one array the number of sets depends on whether min:typ:max delays apply to primitives, module paths, timing checks, module input ports, or inter-module paths	"false"
	"false"	***acc_append_delays***, ***acc_fetch_delays***, ***acc_replace_delays*** handle one, two, three, or six delays passed as individual arguments each argument represents a single delay the number of arguments depends on whether delays apply to primitives, module paths, timing checks, module input ports, or inter-module paths	

*Table 2-30: Using **accMinTypMaxDelays** configuration parameter*

	Set of values	Effect	Default
accPathDelayCount	"1"	***acc_append_delays*****,** ***acc_fetch_delays***, ***acc_replace_delays*** take one delay argument when ***accMinTypMaxDelays*** is ***"false"*** and a pointer to an array of size 3 when ***accMinTypMaxDelays*** is ***"true"***	"6"
	"2"	***acc_append_delays*****,** ***acc_fetch_delays***, ***acc_replace_delays*** take two delay arguments when ***accMinTypMaxDelays*** is ***"false"*** and a pointer to an array of size 6 when ***accMinTypMaxDelays*** is ***"true"***	
	"3"	***acc_append_delays*****,** ***acc_fetch_delays***, ***acc_replace_delays*** take three delay arguments when ***accMinTypMaxDelays*** is ***"false"*** and a pointer to an array of size 9 when ***accMinTypMaxDelays*** is ***"true"*** take three delay argument pairs	
	"6"	***acc_append_delays*****,** ***acc_fetch_delays***, ***acc_replace_delays*** take six delay arguments when ***accMinTypMaxDelays*** is ***"false"*** and a pointer to an array of size 18 when ***accMinTypMaxDelays*** is ***"true"***	

Table 2-31: Using ***accPathDelayCount*** *configuration parameter*

	Set of values	Effect	Default
accPathDelimStr	any sequence of letters, numbers, ***$*** or _	use the string literal as the delimiter that separates the source from the destination in module path names	"$"

*Table 2-32: Using **accPathDelimStr** configuration parameter*

	Set of values	Effect	Default
accToHiZDelay		***acc_append_delays*** and ***acc_replace_delays*** derive turn-off delays as follows:	"from_user"
	"average"	from the average of the rise and fall delays	
	"max"	from the larger of the rise and fall delays	
	"min"	from the smaller of the rise and fall delays	
	"from_user"	from user-supplied argument(s)	

*Table 2-33: Using **accToHiZDelay** configuration parameter*

Usage example: accDefaultAttr0

The following example presents a C-language routine, `display_load_capacitance`, that obtains the load capacitance of all scalar nets connected to the ports in a module. This routine uses `acc_configure` to direct `acc_fetch_attribute` to return the value zero if a load capacitance is not found for a net; as a result, the third argument, `default_value`, can be dropped from the call to `acc_fetch_attribute`.

```
#include "acc_user.h"

display_load_capacitance()
{
   handle   module_handle, port_handle, net_handle;
   double   cap_val;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*configure acc_fetch_attribute to return 0 when it does not find*/
   /* the attribute*/
   acc_configure( accDefaultAttr0, "true" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );

   /*scan all ports in module; display load capacitance*/
   port_handle = null;
   while( port_handle = acc_next_port( module_handle, port_handle ) )
   {
      /*ports are scalar, so pass "null" to get single net connection*/
      net_handle = acc_next_loconn( port_handle, null );

      /*since accDefaultAttr0 is "true", drop default_value argument*/
      cap_val = acc_fetch_attribute( net_handle,"LoadCap_" );

      if (!acc_error_flag)
         io_printf("Load capacitance of net #%d = %1f\n",
                   acc_fetch_index( port_handle ), cap_val );
   }
   acc_close();
}
```

default_value
argument is dropped

*Figure 2-13: Setting **accDefaultAttr0***

Usage example: accDevelopmentVersion

The following example presents a C-language routine, `display_net_names`, that displays the names of all nets in a module. It uses `acc_configure` to set `accDevelopmentVersion` to *"1.6a"* to indicate that it was developed under the PLI version 1.6a access routines.

```
#include "acc_user.h"

display_net_names()
{
  handle   module_handle;
  handle   net_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set version of access routines used to develop this application*/
  acc_configure(accDevelopmentVersion, "1.6a");

  /*get handle for module*/
  module_handle = acc_handle_tfarg(1);

  /*display names of all nets in the module*/
  net_handle = null;
  while( net_handle = acc_next_net( module_handle, net_handle ) )
     io_printf( "Net name is: %s\n", acc_fetch_fullname( net_handle ) );

  acc_close();
}
```

configure
accDevelopmentVersion
after initializing access routines

Figure 2-14: Setting ***accDevelopmentVersion***

Usage example: accDisplayErrors

The routine `display_top_modules` uses `acc_configure` to suppress automatic error reporting. Instead, the routine checks `acc_error_flag` explicitly and displays its own specialized error message which pinpoints the system task or function that invokes the C-language routine—in this case, `$displaytopmods`.

```
#include "acc_user.h"

display_top_modules()
{
   handle   top_mod_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*disable error message display*/
   acc_configure( accDisplayErrors, "false" );

   top_mod_handle = null;
   while( top_mod_handle = acc_next_topmod(top_mod_handle) )
      io_printf("Top module: %s\n", acc_fetch_name(top_mod_handle));

   /* if exit while loop due to error condition, report */
   if (acc_error_flag)
       io_printf("Error in $displaytopmods : exiting task\n");

   acc_close();
}
```

*Figure 2-15: Setting **accDisplayErrors***

Usage example: accDisplayWarnings

The following example presents a checktf routine, `check_new_timing`, that validates arguments passed to its associated system task. It uses `acc_configure` to make sure warning messages issued by `acc_handle_tfarg` are displayed.

```
#include "acc_user.h"

check_new_timing()
{
  handle   gate_handle;

  /* initialize and configure access routines */
  acc_initialize();

  /* set development version */
  acc_configure( accDevelopmentVersion, "1.6a" );

  /* enable warning reporting by access routines*/
  acc_configure( accDisplayWarnings, "true" );

  /* check type of first argument—the object */
  gate_handle = acc_handle_tfarg( 1 );

  /* check for valid argument */
  if (!acc_error_flag)
  /* make sure it's a primitive */
     if ( acc_fetch_type(gate_handle) != accPrimitive )
       tf_error("Invalid argument not a primitive\n");

  acc_close();
}
```

Figure 2-16: Setting ***accDisplayWarnings***

Example: accEnableArgs

The following example presents a C-language routine, `get_path`, that displays the name of a module path. It uses `acc_configure` to set `accEnableArgs` and, therefore, force `acc_handle_modpath` to ignore its null *name* arguments and recognize its optional *handle* arguments, `src_handle` and `dst_handle`.

```
#include "acc_user.h"

get_path()
{
   handle   path_handle,mod_handle,src_handle,dst_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*set accEnableArgs for acc_handle_modpath*/
   acc_configure( accEnableArgs, "acc_handle_modpath" );

   /*get handles to the three system task arguments:*/
   /*    arg 1 is module name              */
   /*    arg 2 is module path source       */
   /*    arg 3 is module path destination */
   mod_handle = acc_handle_tfarg( 1 );
   src_handle = acc_handle_tfarg( 2 );
   dst_handle = acc_handle_tfarg( 3 );

   /*display name of module path*/
   path_handle = acc_handle_modpath( mod_handle,
                                     null, null,
                                     src_handle, dst_handle );
   io_printf( "Path is %s \n", acc_fetch_fullname( path_handle ) );

   acc_close();
}
```

acc_handle_modpath uses optional handle arguments ***src_handle*** and ***dst_handle*** because:

accEnableArgs is set
and
the name arguments are null

Figure 2-17: Setting ***accEnableArgs***

Usage example: accPathDelayCount

In the next example, the C-language routine `set_path_delays` fetches the rise and fall delays of each path in a module, and back annotates the maximum delay value as the delay for all transitions. The value of `accPathDelayCount` specifies the minimum number of arguments you must pass to routines that read or write delay values—in this case, `acc_fetch_delays` and `acc_replace_delays`.

```
#include "acc_user.h"

set_path_delays()
{
 handle  mod_handle;
 handle  path_handle;
 double  rise_delay,fall_delay,max_delay;

/*initialize environment for access routines*/
acc_initialize();

/*set development version*/
acc_configure( accDevelopmentVersion, "1.6a" );

/*get handle to module*/
mod_handle = acc_handle_tfarg( 1 );

/*fetch rise delays for all paths in module "top.m1"*/
 path_handle = null;
 while( path_handle = acc_next_modpath(mod_handle, path_handle) )
 {
  /*configure accPathDelayCount for rise and fall delays only*/
  acc_configure( accPathDelayCount, "2" );
  acc_fetch_delays(path_handle, &rise_delay, &fall_delay);

  /*find the maximum of the rise and fall delays*/
  max_delay = (rise_delay > fall_delay) ? rise_delay : fall_delay;

  /*configure accPathDelayCount to apply one delay for all transitions*/
  acc_configure( accPathDelayCount, "1" );
  acc_replace_delays(path_handle, max_delay);
 }
 acc_close();
}
```

only ***2*** delay arguments are needed

only ***1*** delay argument is needed

*Figure 2-18: Setting **accPathDelayCount***

By setting `accPathDelayCount` to the minimum number of arguments needed for `acc_fetch_delays` and again for `acc_replace_delays`, all unused arguments can be eliminated from each call.

Usage example: accToHiZDelay

The following example shows how the C-language routine `set_buf_delays` sets `accToHiZDelay` to direct `acc_replace_delays` to automatically derive the turn-off delay for a Z-state primitive as the larger of its rise and fall delays.

```
#include "acc_user.h"

set_buf_delays()
{
 handle  primitive_handle;
 handle  path_handle;
 double  added_rise, added_fall;

 /*initialize environment for access routines*/
 acc_initialize();

 /*set development version*/
 acc_configure( accDevelopmentVersion, "1.6a" );

 /*configure accToHiZDelay so acc_append_delays derives turn-off */
 /* delay from the smaller of the rise and fall delays*/
  acc_configure( accToHiZDelay, "min" );

 /*get handle to Z-state primitive*/
 primitive_handle = acc_handle_tfarg( 1 );

 /*get delay values*/
 added_rise = tf_getrealp(2);
 added_fall = tf_getrealp(3);

 acc_append_delays(primitive_handle, added_rise, added_fall);

 acc_close();
}
```

*Figure 2-19: Setting **accToHiZDelay***

2.15.6 acc_count

acc_count			
function	returns an integer count of the number of objects related to a particular reference object		
syntax	*integer_variable* **= acc_count(***NEXT_routine_name, object_handle***)***;*		
arguments	name	type	description
inputs:	**NEXT_routine_name**	*pointer to a NEXT routine*	actual name (unquoted) of the NEXT routine that finds the object of interest
	object_handle	*handle*	handle of the reference object
related routine	all ***NEXT*** routines except ***acc_next_topmod***		

What is a reference object?

A *reference object* provides a frame of reference for a NEXT routine to indicate where it must look for its first target object. Often, a reference object relates to a target object in some way—for example, by *containing* or *connecting to* the target object.

The object associated with `object_handle` must be the same as the reference object required by `NEXT_routine_name`.

You cannot pass acc_next_topmod to acc_count

The access routine `acc_next_topmod` does not work with `acc_count`. However, you can count top-level modules by passing `acc_next_child` with a null reference object argument. Here is a sample call that counts top-level modules:

```
acc_count( acc_next_child, null );
```

Usage example

The following example shows a C-language routine, `count_nets`, that uses `acc_count` to count the number of nets in a module.

```
#include "acc_user.h"

count_nets()
{
  handle  module_handle;
  int     number_of_nets;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg(1);

   /*count and display number of nets in the module*/
   number_of_nets = acc_count( acc_next_net, module_handle );
   io_printf( "number of nets = %d\n", number_of_nets );

   acc_close();
}
```

*Figure 2-20: Using **count_nets** to count the number of nets in a module*

2.15.7 acc_fetch_attribute

acc_fetch_attribute			
function	returns the value of a parameter or specparam named as an attribute in your source description		
syntax	*double_value* **= acc_fetch_attribute(***object_handle, attribute_string, default_value***)***;*		
arguments	name	type	description
inputs:	**object_handle**	*handle*	handle of a named object
	attribute_string	*character string pointer:* quoted string literal *or* character pointer variable	the ***attribute*** portion of the parameter or specparam declaration
	default_value *(depends on accDefaultAttr0)*	*double*	double precision value to be returned if the attribute is not found
related routines	Use ***acc_configure(accDefaultAttr0...)*** to set default value returned when attribute is not found Use ***acc_fetch_paramval*** for parameters or specparams ***not*** declared in ***attribute/object*** format Use ***acc_handle_object*** to obtain the handle for the first argument, ***object_handle***		

Naming attributes

The routine `acc_fetch_attribute` obtains the value of a parameter or specparam that you declare as an attribute in your source description. Any parameter or specparam can be an attribute, as long you name it in one of the following ways:

- to associate the attribute with a particular object in the module where the attribute is declared
- to specify a more general attribute—that is, one that may be associated with more than one object in the module where the attribute is declared

Each of these methods uses its own naming convention, as described in Table 2-34.

When:	*use a name:*	*in this format:*
you want to associate an attribute with a particular object	that concatenates a mnemonic name that describes the attribute with the name of the object	**attribute_string** | **object_name**
you want the attribute to be general	that describes only the attribute	**attribute_string**

Table 2-34: Naming conventions for attributes

For either convention, attribute_string names the attribute and must be passed as the second argument to acc_fetch_attribute. The object_name is the actual name of a design object in your source description.

The following example shows how to name a specparam as an attribute associated with a particular object—the drive strength of a gate called g1:

```
specparam  DriveStrength$g1 = 2.8;
```

Here, attribute_string is DriveStrength$ and object_name is g1.

To make the drive strength a more general attribute, specify it this way:

```
specparam  DriveStrength$ = 2.8;
```

Now, the name contains only the attribute_string, which is DriveStrength$.

Module path names

The access routine acc_fetch_attribute identifies module paths in terms of their sources and destinations in the following format:

source	**path_delimiter**	**destination**

In particular, acc_fetch_fullname and acc_fetch_name return names of module paths in this format, and acc_fetch_attribute looks for module path names in this format. Therefore, you must use the same naming convention when associating an attribute with a module path.

Names of module paths with multiple sources or destinations are derived from the first source or destination only.

By default, the path_delimiter is the character "$". However, you can override this default by using the access routine acc_configure to set the delimiter parameter accPathDelimStr to another character string (see Section 2.15.5).

The following examples show how to name module paths using different delimiter strings:

For module path:	*if accPathDelimStr is:*	*then module path name is:*
`(a => q) = 10;`	"$"	***a$q***
`(b *> q1,q2) = 8;`	"**_$_**"	***b_$_q1***
`(d,e,f *> r,s)= 8;`	"_"	***d_r***

Table 2-35: Naming module paths using different delimiter strings

The next example shows how to use this naming convention to describe the rise strength of module path ***(a => q) = 10*** as an attribute in a source description:

```
specparam  RiseStrength$a$q = 20;
```

Here, the `attribute_string` is `RiseStrength$`, the `object_name` is `a$q`, and the `path_delimiter` is `$`, the default.

How the routine works

The following flow chart illustrates how `acc_fetch_attribute` works:

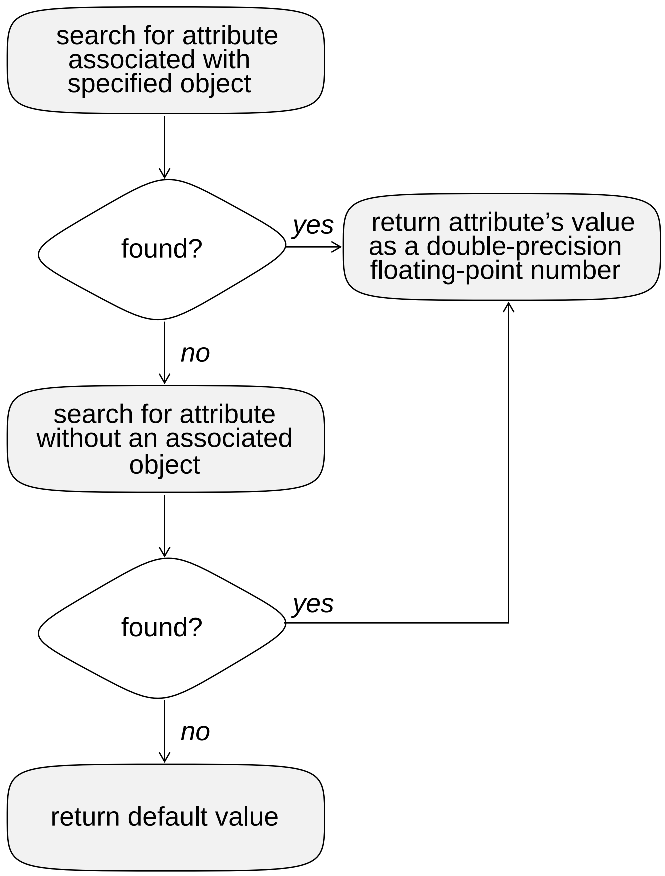


*Figure 2-21: How **acc_fetch_attribute** works*

This illustration shows that when `acc_fetch_attribute` finds the attribute you request, it returns the attribute's value as a double-precision floating point number.

The routine first looks for the attribute name that concatenates `attribute_string` with the name associated with `object_handle`. For example, to find an attribute `InputLoad$` for a net `n1`, `acc_fetch_attribute` searches for `InputLoad$n1`.

If `acc_fetch_attribute` does not find the attribute associated with the object you specify with `object_handle`, the routine then searches for a name that matches `attribute_string`. Assume that, in the previous example, `acc_fetch_attribute` does not find `InputLoad$n1`. It then looks for `InputLoad$.` Other variants of that name, such as `InputLoad$n3` or `InputLoad$n`, are not considered matches.

Failing both search attempts, the routine returns a default value. You can control the default value by using the access routine `acc_configure` to set or reset the configuration parameter `accDefaultAttr0` as follows:

*When **accDefaultAttr0** is:*	***acc_fetch_attribute** returns:*
true	***zero*** when the attribute is not found; the ***default_value*** argument may be dropped
false	the value passed as the ***default_value*** argument when the attribute is not found

*Table 2-36: Controlling the default value returned by **acc_fetch_attribute***

Usage example

The following example presents a C-language routine, `display_load_capacitance`, that uses `acc_fetch_attribute` to obtain the load capacitance of all scalar nets connected to the ports in a module.

```
#include "acc_user.h"

display_load_capacitance()
{
   handle   module_handle, port_handle, net_handle;
   double   cap_val;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*configure acc_fetch_attribute to return 0 when it does not find*/
   /* the attribute*/
   acc_configure( accDefaultAttr0, "true" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );

   /*scan all ports in module; display load capacitance*/
   port_handle = null;
   while( port_handle = acc_next_port( module_handle, port_handle ) )
   {
      /*ports are scalar, so pass "null" to get single net connection*/
      net_handle = acc_next_loconn( port_handle, null );

      /*since accDefaultAttr0 is "true", drop default_value argument*/
      cap_val = acc_fetch_attribute( net_handle,"LoadCap_" );

      if (!acc_error_flag)
         io_printf("Load capacitance of net #%d = %1f\n",
                        acc_fetch_index( port_handle ), cap_val );
   }
   acc_close();
}
```

Figure 2-22: Obtaining the load capacitance of all scalar nets connected to module ports

Note that `acc_fetch_attribute` does not require its third argument, `default_value`, because `acc_configure` sets `accDefaultAttr0` to `"true"`.

2.15.8 acc_fetch_defname

acc_fetch_defname			
function	returns a pointer to the defining name of a module instance or primitive instance		
syntax	*character_pointer* **= acc_fetch_defname(** *object_handle* **)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of the module instance or primitive instance

Usage example

The following example presents a C-language routine, `get_primitive_definitions`, that uses `acc_fetch_defname` to display the defining names of all primitives in a module.

```
#include "acc_user.h"

get_primitive_definitions(module_handle)
handle  module_handle;
{
   handle   prim_handle;

   /*get and display defining names of all primitives in the module*/
   prim_handle = null;
   while( prim_handle = acc_next_primitive( module_handle,prim_handle))
      io_printf( "primitive definition is %s\n",
                 acc_fetch_defname( prim_handle ) );
}
```

Figure 2-23: Displaying the defining names of all primitives in a module

2.15.9 acc_fetch_delays

acc_fetch_delays (single delay per transition)

function	when accMinTypMaxDelays is "false", passes back typical delay values for primitives, module paths, timing checks or module input ports as output arguments
syntax	
primitives:	**acc_fetch_delays(** *primitive_handle, rise_delay_addr, fall_delay_addr, z_delay_addr* **)**;
module paths:	**acc_fetch_delays(** *path_handle, delay_addr1, delay_addr2, delay_addr3, delay_addr4, delay_addr5, delay_addr6* **)**;
timing checks:	**acc_fetch_delays(** *timing_check_handle, limit_addr* **)**;
ports:	**acc_fetch_delays(** *port_handle, rise_delay_addr, fall_delay_addr, z_delay_addr* **)**;
port's bits	**acc_fetch_delays(** *bit_handle, rise_delay_addr, fall_delay_addr, z_delay_addr* **)**;

arguments	name	type	description
input:	**object_handle**	*handle*	handle of a primitive, module path, timing check, module input port or bit of a module input port
outputs:			
primitives, ports, port's bits:	**rise_delay_addr**	*double address*	pointer to object's rise delay
	fall_delay_addr	*double address*	pointer to object's fall delay
	z_delay_addr	*double address*	pointer to object's turn-off delay
module paths	**delay_addr1**	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
	delay_addr2 (depends on accPathDelayCount)	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
	delay_addr3 (depends on accPathDelayCount)	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
	delay_addr4 (depends on accPathDelayCount)	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
	delay_addr5 (depends on accPathDelayCount)	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
	delay_addr6 (depends on accPathDelayCount)	*double address*	pointer to module path's delay for transitions determined by accPathDelayCount
timing checks	**limit_addr**	*double address*	pointer to timing check's limit
related routines	Use ***acc_configure(accPathDelayCount...)*** to set number of delays for module path Use ***acc_configure(accMinTypeMaxDelays, "false")*** for single delays per transition		

acc_fetch_delays (min:typ:max delays)			
function	when accMinTypMaxDelays is "true", passes back pointer to array of min:typ:max delay values for primitives, module paths, timing checks, module input ports or inter-module paths		
syntax			
primitives:	**acc_fetch_delays(** *primitive_handle, array_ptr* **)**;		
module paths:	**acc_fetch_delays(** *path_handle, array_ptr* **)**;		
timing checks:	**acc_fetch_delays(** *timing_check_handle, array_ptr* **)**;		
ports:	**acc_fetch_delays(** *port_handle, array_ptr* **)**;		
port's bits:	**acc_fetch_delays(** *bit_handle, array_ptr* **)**;		
inter-module paths	**acc_fetch_delays(***intermod_path_handle, array_ptr* **)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of a primitive, module path, inter-module path, timing check, module input port or bit of a module input port
output:	**array_ptr**	*double address*	pointer to array of min:typ:max values (see Table 2-38)
related routines	Use ***acc_configure(accMinTypeMaxDelays, "true")*** for min:typ:max delays for each transition		

The access routine `acc_fetch_delays` works differently depending on how the configuration parameter `accMinTypMaxDelays` is set. When this parameter is set to `"false"`, `acc_fetch_delays` assumes a single delay per transition and passes each delay as an individual argument. For this ***single delay mode***, the first syntax table in this section applies.

When `accMinTypMaxDelays` is set to `"true"`, `acc_fetch_delays` passes one or more sets of *min:typ:max* delays in an array, rather than passing single delays as individual arguments. For this ***min:typ:max*** delay mode, the second syntax table in this section applies.

Different delays for different objects

The routine `acc_fetch_delays` fetches different delay values for different objects, as summarized in Table 2-37 and Table 2-38. Table 2-37 applies when `acc_fetch_delays` is set for *single delay mode* (`accMinTypMaxDelays` is `"false"`) and Table 2-38 applies when this routine is set for *min:typ:max delay mode* (`accMinTypMaxDelays` is `"true"`).

For:	***acc_fetch_delays*** *returns:*
primitives with a Z state: ***bufif*** gates ***notif*** gates ***MOS*** switches	***three*** delays: rise delay, ***rise_delay_addr*** fall delay, ***fall_delay_addr*** turn-off delay, ***z_delay_addr***
primitives with ***no*** Z state:	***two*** delays: rise delay, ***rise_delay_addr*** fall delay, ***fall_delay_addr*** *(if a third argument is supplied, it is ignored)*
module paths	***one***, ***two***, ***three*** or ***six*** delays: depends on how you configure ***accPathDelayCount***
timing checks	***one*** delay: timing check limit, ***limit***
module input or inout ports (MIPDs)	***three*** delays: rise delay, ***rise_delay_addr*** fall delay, ***fall_delay_addr*** turn-off delay, ***z_delay_addr***

*Table 2-37: How **acc_fetch_delays** reads delays in single delay mode*

In *single delay mode*, you must supply the correct number of delay arguments to `acc_fetch_delays`, as follows:

- MIPDs and Z-state primitives require ***three*** delay arguments—`rise_delay_addr`, `fall_delay_addr` and `Z_delay_addr`.
- Primitives with no Z state require ***two*** delay arguments—`rise_delay_addr` and `fall_delay_addr`.
- Module paths require ***one***, ***two***, ***three*** or ***six*** delay arguments, depending on how you configure `accPathDelayCount`.
- Timing checks require ***one*** limit argument—`limit`.

For more information on the configuration parameter `accPathDelayCount`, refer to Table 2-39 and Table 2-40.

Table 2-38 shows how `acc_fetch_delays` reads delays in *min:typ:max delay mode*.

For:	***acc_fetch_delays*:**	*The delay array must retrieve:*
module input ports primitives inter-module paths	reads *three* sets of *min:typ:max* delays: one set for rise delays one set for fall delays one set for turn-off delays	*nine* values: array[0] = minimum rise delay array[1] = typical rise delay array[2] = maximum rise delay array[3] = minimum fall delay array[4] = typical fall delay array[5] = maximum fall delay array[6] = minimum turn-off delay array[7] = typical turn-off delay array[8] = maximum turn-off delay ***(you must always declare an array of size 9, even if turn-off delays are not used)***
module paths	reads *one*, *two*, *three* or *six* sets of *min:typ:max* delays: depends on how you configure `accPathDelayCount` ***(see Table 2-40)***	*three*, *six*, *nine* or*18* values: ***(see Table 2-40)***
timing checks	reads *one* set of *min:typ:max* delays: timing check limit	*three* values: array[0] = minimum timing check limit array[1] = typical timing check limit array[2] = maximum timing check limit

*Table 2-38: How **acc_fetch_delays** reads delays in **min:typ:max** delay mode*

There are a couple of points to note in Table 2-38:

- For module input ports, primitives and inter-module paths, always declare an array of size 9, even if the objects do not have a Z state.
- The configuration parameter `accPathDelayCount` affects the *min:typ:max* delays processed for module paths. Table 2-40 describes these effects in greater detail.

Fetching delays for module paths (single delay mode)

In most cases, you will want to fetch the number of delays that appears in your Verilog-HDL source description for a particular module path. In *single delay mode*, you can control the number of delays returned for module paths by using the access routine `acc_configure` to set the delay count parameter `accPathDelayCount` shown in Table 2-39.

When ***accPathDelayCount*** *is:*	***acc_fetch_delays*** *returns:*
1	***one*** delay value, ***delay_addr1*** *(last five delay arguments may be dropped)*
2	***two*** delay values: ***delay_addr1*** ***delay_addr2*** *(last four delay arguments may be dropped)*
3	***three*** delay values: ***delay_addr1*** ***delay_addr2*** ***delay_addr3*** *(last three delay arguments may be dropped)*
6 (the default)	all ***six*** delay values, a different delay for each possible transition among 0, 1 and z: one for 01 transitions: ***delay_addr1*** one for 10 transitions: ***delay_addr2*** one for 0z transitions: ***delay_addr3*** one for z1 transitions: ***delay_addr4*** one for 1z transitions: ***delay_addr5*** one for z0 transitions: ***delay_addr6***

Table 2-39: How ***accPathDelayCount*** *affects module path delays in single delay mode*

The minimum number of delay arguments to pass to `acc_fetch_delays` must equal the value of `accPathDelayCount`. You may drop any remaining arguments.

The following example shows how to set `accPathDelayCount` so that `acc_fetch_delays` retrieves rise and fall delays for module paths:

```
acc_configure( accPathDelayCount, "2" );
```

If you do not set `accPathDelayCount` explicitly, it defaults to ***6***; in this case, you must pass all six delay arguments when you call `acc_fetch_delays` in *single delay mode*.

When `accPathDelayCount` is less than 6, `acc_fetch_delays` returns delays in the following order:

1. delay for 01 transitions
2. delay for 10 transitions
3. delay for 0z transitions

Fetching delays for module paths (min:typ:max mode)

The following rules apply when you retrieve *min:typ:max* delays for module paths:

- the number of *sets* of *min:typ:max* delays must equal the value of `accPathDelayCount`
- the size of the delay array must be three times the value of `accPathDelayCount`

Table 2-40 summarizes how `accPathDelayCount` affects *min:typ:max* delays for module paths.

When ***accPathDelayCount*** *is:*	*The number of sets of min:typ:max path delays is:*	*The delay array must pass or retrieve:*
"1"	**one**: the same minimum, typical and maximum delay for all transitions	**three values:** `array[0]` = minimum delay `array[1]` = typical delay `array[2]` = maximum delay
"2"	**two**: one set for rising transitions one set for falling transitions	**six values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay
"3"	**three**: one set for rising transitions one set for falling transitions one set for transitions to z	**nine values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay `array[6]` = minimum turn-off delay `array[7]` = typical turn-off delay `array[8]` = maximum turn-off delay
"6" *(the default)*	**six:** one set for 01 transitions one set for 10 transitions one set for 0z transitions one set for z1 transitions one set for 1z transitions one set for z0 transitions	**18 values:** `array[0]` = minimum 01 delay `array[1]` = typical 01 delay `array[2]` = maximum 01 delay `array[3]` = minimum 10 delay `array[4]` = typical 10 delay `array[5]` = maximum 10 delay `array[6]` = minimum 0z delay `array[7]` = typical 0z delay `array[8]` = maximum 0z delay `array[9]` = minimum z1 delay `array[10]` = typical z1 delay `array[11]` = maximum z1 delay `array[12]` = minimum 1z delay `array[13]` = typical 1z delay `array[14]` = maximum 1z delay `array[15]` = minimum z0 delay `array[16]` = typical z0 delay `array[17]` = maximum z0 delay

Table 2-40: The relationship between ***accPathDelayCount*** *and* ***min:typ:max*** *delays for module paths*

Fetching delays for inter-module paths

An inter-module path is a wire path that connects an output or inout port of one module to an input or inout port of another module. You can use the access routines `acc_fetch_delays` and `acc_replace_delays` to fetch and replace delays for inter-module paths.

For inter-module paths, you *must* fetch or replace delays in *min:typ:max delay mode*. Therefore, set `accMinTypMax` to `"true"` before calling `acc_fetch_delays` or `acc_replace_delays` for inter-module paths.

Fetching module input port delays (MIPDs)

Use `acc_fetch_delays` to fetch Module Input Port Delays (MIPDs). An MIPD is a delay associated with a module input port or inout port. The MIPD describes the delay between the module port and each of the loads in its fanout. In an MIPD you can specify rise, fall, and high impedance propagation delays.

You can fetch an MIPD for each individual bit of a vector port using `acc_fetch_delays` in conjunction with `acc_next_bit`. For more information, see Section 2.15.49.

Declaring the array that holds min:typ:max values

Use Table 2-38 and Table 2-40 to decide how large to make the array that passes or holds *min:typ:max* values. The array must be able to store the correct number of delays that will be processed. Declaring an array that is too small will cause errors or unpredictable results.

Effect of timescales

The routine `acc_fetch_delays` retrieves delay values in the timescale of the module that contains `object_handle`.

Usage example: single delay mode

The following example presents a C-language routine, `display_path_delays`, that uses `acc_fetch_delays` to retrieve the rise, fall and turn-off delays of all paths through a module.

```
#include "acc_user.h"

display_path_delays()
{
  handle  mod_handle;
  handle  path_handle;
  double  rise_delay,fall_delay,toz_delay;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*set accPathDelayCount to return rise, fall and turn-off delays */
  acc_configure( accPathDelayCount, "3" );

  /*get handle to module*/
  mod_handle = acc_handle_tfarg(1);

  /*fetch rise delays for all paths in module "top.m1"*/
  path_handle = null;
  while( path_handle = acc_next_modpath(mod_handle, path_handle) )
  {
   acc_fetch_delays(path_handle,
                    &rise_delay,&fall_delay,&toz_delay);

   /*display rise, fall and turn-off delays for each path*/
   io_printf("For module path %s,delays are:\n",
              acc_fetch_fullname(path_handle) );
   io_printf("rise = %lf, fall = %lf, turn-off = %lf\n",
              rise_delay,fall_delay,toz_delay);
  }
  acc_close();
}
```

Figure 2-24: Displaying the rise, fall and turn-off delays of all paths through a module

Usage example: min:typ:max delay mode

To fetch *min:typ:max* delays for an inter-module path, follow these steps in your C routine:

1. **Declare an array of nine double-precision floating-point values** as a buffer for storing three sets of *min:typ:max* values: one set each for rise, fall and turn-off delays
2. **Set the configuration parameter `accMinTypMaxDelays` to "`true`"** to instruct `acc_fetch_delays` to retrieve delays in *min:typ:max* format.
3. **Call `acc_fetch_delays`** with a valid inter-module path handle and the array pointer.

The following C-language code fragment of an application shows how to fetch *min:typ:max* delays for the inter-module path referenced by `intermod_path`. Assume the Verilog-HDL source description contains only one delay per transition.

```
#include "acc_user.h"
fetch_mintypmax_delays( port_output, port_input )
handle port_output, port_input;
{
                              •
                              •
                              •
   handle  intermod_path;
   double  delay_array[9];
                              •
                              •
                              •
   acc_configure( accMinTypMaxDelays, "true" );
                              •
                              •
                              •
   intermod_path = acc_handle_path( port_output, port_input);
   acc_fetch_delays( intermod_path, delay_array );
                              •
                              •
                              •
}
```

acc_handle_path returns a handle to a wire path that represents the connection from an output (or inout) port to an input (or inout) port

acc_fetch_delays places the following values in delay_array:

delay_array[0] =
delay_array[1] =
delay_array[2] =
rise delay

delay_array[3] =
delay_array[4] =
delay_array[5] =
fall delay

delay_array[6] =
delay_array[7] =
delay_array[8] =
turn-off delay

Figure 2-25: Fetching **min:type:max** *delays for an inter-module path*

If for this example the Verilog-HDL description specified *min:typ:max* delays on `intermod_path` for rise, fall and to-Z transitions, then `acc_fetch_delays` would place the following values in `delay_array`:

`delay_array[0]` = minimum rise delay
`delay_array[1]` = typical rise delay
`delay_array[2]` = maximum rise delay
`delay_array[3]` = minimum fall delay
`delay_array[4]` = typical fall delay
`delay_array[5]` = maximum fall delay
`delay_array[6]` = minimum turn-off delay
`delay_array[7]` = typical turn-off delay
`delay_array[8]` = maximum turn-off delay

For another example of using acc_fetch_delays to retrieve *min:typ:max* delays, see *Usage example: scaling min:typ:max delays* on page 2-239.

.

2.15.10 acc_fetch_direction

acc_fetch_direction			
function	returns the direction of a port or terminal as one of three predefined integer constants: ***accInput*** ***accOutput*** ***accInout***		
syntax	*integer_variable* **= acc_fetch_direction(** *object_handle* **);**		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of a port or terminal

How the routine works

When direction is:	***acc_fetch_direction*** *returns:*
input only	***accInput***
output only	***accOutput***
input and output	***accInout***

Table 2-41: How ***acc_fetch_direction*** *works*

Usage example

The following example presents a C-language routine, `is_port_input`, that uses `acc_fetch_direction` to determine whether or not a port is an input.

```
#include "acc_user.h"

bool  is_port_input(port_handle)
handle port_handle;
{
  int   direction;

  direction = acc_fetch_direction(port_handle);

  /*return "true" if an input port*/
  if (direction == accInput || direction == accInout)
     return(true);
  else
     return(false);
}
```

Figure 2-26: Determining if a port is an input

2.15.11 acc_fetch_edge

acc_fetch_edge			
function	returns the edge specifier (type) of a path input or output terminal as one of these predefined integer constants: ***accNoedge*** ***accEdge01*** ***accEdgex1*** ***accPosedge*** ***accEdge10*** ***accEdge1x*** ***accNegedge*** ***accEdge0x*** ***accEdgex0***		
syntax	*integer_variable* **= acc_fetch_*edge* (** *pathio_handle);*		
arguments	name	type	description
input:	**pathio_handle**	*handle*	handle to a module path input or output

Description

The returned value is a masked integer representing the edge specifier for `pathio` upon success, or 0 (`accNoedge`) upon error or no edge specifier being given.

Table 2-42 lists the predefined edge specifiers in `acc_user.h`.

Edge type	Defined constant	Binary value
None	accNoedge	0
Positive edge (0→1,0→x,x→1)	accPosedge	00001011
Negative edge (1→0,1→x,x→0)	accNegedge	01100010
0→1 edge	accEdge01	00000001
1→0 edge	accEdge10	00000010
0→x edge	accEdge0x	00000100
x→1 edge	accEdgex1	00001000
1→x edge	accEdge1x	00010000
x→0 edge	accEdgex0	00100000

Table 2-42: Edge specifiers defined in ***acc_user.h***

The integer mask returned by this routine usually equals either `accPosedge` or `accNegedge`. Occasionally, however, the mask is a hybrid mix of specifiers that is equal to neither. The example below illustrates how to check for these hybrid edge specifiers. The value `accNoEdge` is returned if no edge is found.

Example

The following example takes a path input or output and returns the string corresponding to its edge specifier. It provides analogous functionality to that of `acc_fetch_type_str` in that it returns a string corresponding to an integer value that represents a type.

This example first checks to see whether the returned mask is equal to `accPosedge` or `accNegedge`, which are the most likely cases. If it does not, the routine does a bitwise AND with the returned mask and each of the other edge specifiers to find out which types of edges it contains. If an edge type is encoded in the returned mask, the corresponding edge type string suffix is appended to the string `“accEdge”.`

```
char *acc_fetch_edge_str(pathio)
handle pathio;
{
   int edge = acc_fetch_edge(pathio);
   static char edge_str[32];

   if (! acc_error_flag)
   {
      if (edge == accNoEdge)
         strcpy(edge_str, “accNoEdge”);

      /* accPosedge == (accEdge01 & accEdge0x & accEdgex1) */
      else if (edge == accPosEdge)
         strcpy(edge_str, “accPosEdge”);

      /* accNegedge == (accEdge10 & accEdge 1x & accEdgex0) */
      else if (edge == accNegEdge)
         strcpy(edge_str, “accNegEdge”);

      /* edge is neither posedge nor negedge, but some combination
         of other edges */
      else {
         strcpy(edge_str, “accEdge”);
         if (edge & accEdge01) strcat(edge_str, “_01”);
         if (edge & accEdge10) strcat(edge_str, “_10”);
         if (edge & accEdge0x) strcat(edge_str, “_0x”);
         if (edge & accEdgex1) strcat(edge_str, “_x1”);
         if (edge & accEdge1x) strcat(edge_str, “_1x”);
         if (edge & accEdgex0) strcat(edge_str, “_x0”);
      }

      return(edge_str);
   }
   else
      return(NULL);
}
```

*Figure 2-27: Using **acc_fetch_edge** to get the string that corresponds to an edge specifier*

2.15.12 acc_fetch_fullname

acc_fetch_fullname			
function	returns a pointer to the full hierarchical name of any named object or module path		
syntax	*character_pointer* **= acc_fetch_fullname(** *object_handle***)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of the object
related routines	Use ***acc_fetch_name*** to find the lowest-level name of the object Use ***acc_configure(accPathDelimStr...)*** to set the delimiter string for module path names		

What is the full hierarchical name?

The ***full hierarchical name*** is the name that uniquely identifies an object. Consider this example: the top-level module `top1` contains a module instance `mod3` which, in turn, contains a net `w4`, as shown in Figure 2-28.

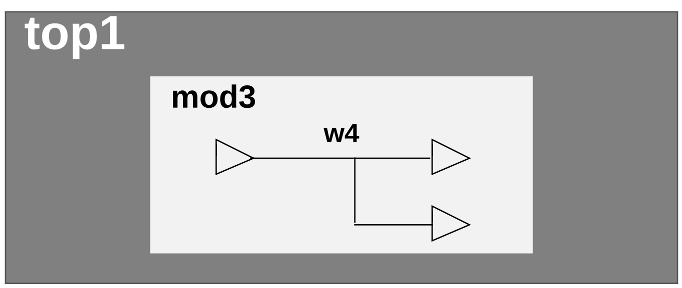


Figure 2-28: A design hierarchy

The full hierarchical name of the net is `top1.mod3.w4`.

Module path names

Module path names are derived from their sources and destinations in the following format:

source	**path_delimiter**	**destination**

Names of module paths with multiple sources or destinations are derived from the first source and destination only.

By default, the ***path_delimiter*** is the character `$`. However, you can override this default by using the access routine `acc_configure` to set the delimiter parameter `accPathDelimStr` to another character string.

The following examples show names of paths within a top-level module `m3`, as returned by `acc_fetch_fullname` when the `path_delimiter` is `$`:

*For module **m3** path:*	***acc_fetch_fullname** returns pointer to:*
`(a => q) = 10;`	***m3.a$q***
`(b *> q1,q2) = 8;`	***m3.b$q1***
`(d,e,f *> r,s)= 8;`	***m3.d$r***

*Table 2-43: Module path names returned by **acc_fetch_fullname***

Default names

If a Verilog simulator creates default names for unnamed instances, `acc_fetch_fullname` returns the full hierarchical default name. Otherwise, the routine returns *null* for unnamed instances.

Usage example

In the example in Figure 2-29, the routine `display_if_net` uses `acc_fetch_fullname` to display the full hierarchical name of an object if the object is a net.

```
#include "acc_user.h"

display_if_net(object_handle)
handle   object_handle;
{
      /*get and display full name if object is a net*/
      if (acc_fetch_type(object_handle) == accNet)
         io_printf( "Object is a net: %s\n",
                     acc_fetch_fullname(object_handle) );
      else
         io_printf( "Object is not a net: %s\n",
                     acc_fetch_fullname(object_handle) );
}
```

Figure 2-29: Displaying the full hierarchical name of a net

2.15.13 acc_fetch_fulltype

acc_fetch_fulltype

function

returns the **fulltype** of an object as one of these predefined integer constants:

accAlwaysStat	***accIfStat***	***accPmosGate***	***accTask***
accAndGate	***accInitialStat***	***accPort***	***accTaskCall***
accAssignDelayStat	***accInoutTerminal***	***accPortBit***	***accStringParam***
accAssignEventStat	***accInputTerminal***	***accPrimPath***	***accTimeVar***
accAssignmentStat	***accIntegerParam***	***accPulldownGate***	***accTerminal***
accAssignStat	***accIntegerVar***	***accPullupGate***	***accTopModule***
accAtEventStat	***accInterModPath***	***accRcmosGate***	***accTranGate***
accBitSelectPort	***accModPath***	***accRealParam***	***accTranif0Gate***
accBufGate	***accModule***	***accRealVar***	***accTranif1Gate***
accBufif0Gate	***accModuleInstance***	***accRecovery***	***accTri***
accBufif1Gate	***accNamedBeginStat***	***accRegister***	***accTri0***
accCaseStat	***accNamedForkStat***	***accReleaseStat***	***accTri1***
accCaseXStat	***accNamedEvent***	***accRepeatStat***	***accTriand***
accCaseZStat	***accNandGate***	***accRnmosGate***	***accTrior***
accCellModule	***accNet***	***accRpmosGate***	***accTrireg***
accCmosGate	***accNmosGate***	***accRtlDelayStat***	***accUserFunction***
accCombPrim	***accNochange***	***accRtlEventStat***	***accUserRealFunction***
accConcatPort	***accNorGate***	***accRtranGate***	***accUserTask***
accContAssignStat	***accNotGate***	***accRtranif0Gate***	***accUnnamedBeginStat***
accDataPath	***accNotif0Gate***	***accRtranif1Gate***	***accUnnamedForkStat***
accDeassignStat	***accNotif1Gate***	***accScalarPort***	***accVectorPort***
accDelayStat	***accNullStat***	***accSeqPrim***	***accWand***
accDisableStat	***accOrGate***	***accSetup***	***accWaitStat***
accForceStat	***accOutputTerminal***	***accSkew***	***accWhileStat***
accForStat	***accPartSelectPort***	***accSupply0***	***accWidth***
accFunction	***accPath***	***accSupply1***	***accWire***
accFunctionCall	***accPathInput***	***accSystemFunction***	***accWor***
accGenEventStat	***accPathOutput***	***accSystemRealFunction***	***accXnorGate***
accHold	***accPeriod***	***accSystemTask***	***accXorGate***

syntax

integer_variable **= acc_fetch_fulltype(** *object_handle* **);**

arguments

	name	type	description
input:	**object_handle**	*handle*	handle of the object

related routines

acc_fetch_type
acc_fetch_type_str

The difference between type and fulltype

The *type* of an object is its general Verilog-HDL data type classification. Table 2-60 in Section 2.15.23 lists these data types, along with the predefined integer constants that represent them.

The *fulltype* of an object is a finer classification of a particular Verilog-HDL data type. Currently, *fulltypes* are defined for the following objects:

- primitives
- terminals
- ports
- bits of vector or concatenated ports
- parameters and specparams
- nets
- modules
- module paths and module path terminals
- timing checks and timing check terminals
- registers
- named events
- integer, real, and time variables
- data paths
- inter-module paths
- statements

Table 2-44 through Table 2-53 list the *fulltypes* for each of these objects.

The access routine `acc_fetch_type` returns the *type* of an object, while `acc_fetch_fulltype` returns the *fulltype* of an object. Table 2-54 illustrates the difference between `acc_fetch_fulltype` and `acc_fetch_type` for selected objects.

Primitive fulltype	*Predefined integer constant*
gates with one or more inputs, one output	accAndGate accNandGate accNorGate accOrGate accXnorGate accXorGate
gates with one input, one or more outputs	accBufGate accNotGate
gates that model tri-state drivers	accBufif0 accBufif1 accNotif0 accNotif1
MOS gates	accNmosGate accPmosGate accRnmosGate accRpmosGate
CMOS gates	accCmosGate accRcmosGate
bidirectional pass gates	accRtranGate accRtranif0Gate accRtranif1Gate accTranGate accTranif0Gate accTranif1Gate
pulldown, pullup gates	accPulldownGate accPullUpGate
combinational user-defined primitive	accCombPrim
sequential user-defined primitive	accSeqPrim

Table 2-44: The ***fulltypes*** *of primitives*

Terminal Fulltype	*Predefined Integer Constant*
input terminal	accInputTerminal
output terminal	accOutputTerminal
inout terminal	accInoutTerminal

Table 2-45: The ***fulltypes*** *of terminals*

Port or Port Bit Fulltype	*Predefined Integer Constant*
scalar port	accScalarPort
vector port bit select	accBitSelectPort
vector port part select	accPartSelectPort
vector port	accVectorPort
concatenated port	accConcatPort
a bit on a port	accPortBit

Table 2-46: The ***fulltypes*** *of ports and bits of a vector port*

Parameter Fulltype	*Predefined Integer Constant*
character string parameter	accStringParam
integer parameter	accIntegerParam
real number parameter	accRealParam

Table 2-47: The ***fulltypes*** *of parameters and specparams*

Net fulltype	*Predefined integer constant*
net of type wire	accWire
net of type tri	accTri
wired-AND nets	accWand accTriand
wired-OR nets	accWor accTrior
pulldown, pullup nets	accTri0 accTri1
nets that model power supplies	accSupply0 accSupply1
net that stores a value	accTrireg

*Table 2-48: The **fulltyppes** of nets*

Module Fulltype	*Predefined Integer Constant*
top-level module	accTopModule
module instance that is a cell	accCellModule
module instance that is not a cell	accModuleInstance

*Table 2-49: The **fulltypes** of modules*

Fulltype	*Predefined Integer Constant*
data path	accDataPath
input terminal path	accPathInput
inter-module path	accInterModPath
module path	accModPath
output module path	accPathOutput
path between primitives	accPrimPath
Verilog-HDL function definition	accFunction
Verilog-HDL task definition	accTask

*Table 2-50: The **fulltypes** of paths, and task and function definitions*

Timing check fulltype	*Predefined integer constant*
HOLD check	accHold
NOCHANGE check	accNochange
PERIOD check	accPeriod
RECOVERY check	accRecovery
SETUP check	accSetup
SKEW check	accSkew
WIDTH check	accWidth

*Table 2-51: The **fulltypes** of timing checks*

New fulltype	*Predefined integer constant*
register	accRegister
named event	accNamedEvent
integer variable	accIntegerVar
real variable	accRealVar
time variable	accTimeVar

*Table 2-52: The **fulltypes** of registers, named events and variables*

Verilog-HDL statement fulltype	*Predefined integer constant*
assignment statements	accAssignDelayStat accAssignEventStat accAssignmentStat accAssignStat accConAssignStat accDeassignStat accDelayStat accForceStat accReleaseStat accRt1DelayStat
call statements	accFunctionCall accSystemTask accSystemRealFunction accSystemFunction accTaskCall accUserFunction accUserRealFunction accUserTask
control statements	accCaseStat accCaseXStat accCaseZStat accIfStat
disable statement	accDisableStat
event statements	accAssignEventStat accAtEventStat accGenEventStat accRt1EvenStat accWaitStat
loop statements	accForStat accRepeatStat accWhileStat
null statement	accNullStat
parallel block statements	accNamedForkStat accUnnamedForkStat
procedure block statements	accAlwaysStat accInitialStat
sequential block statements	accNamedBeginStat accUnnamedBeginStat

Table 2-53: The fulltypes of Verilog-HDL statements

For:	***acc_fetch_fulltype** returns:*	***acc_fetch_type** returns:*
a setup timing check	accSetup	accTchk
an AND gate	accAndGate	accPrimitive
a sequential primitive	accSeqPrim	accPrimitive

*Table 2-54: The difference between **acc_fetch_fulltype** and **acc_fetch_type***

Usage example

The following two C-language routines in Figure 2-30 and Figure 2-31, `display_timing_check_type` and `display_primitive_type`, use `acc_fetch_fulltype` to find and display the *fulltypes* of timing checks and primitive objects passed as input arguments. These routines are called by a higher-level routine, `display_object_type`, presented as the usage example for *acc_fetch_type*.

```
#include "acc_user.h"

display_timing_check_type(tchk_handle)
handle  tchk_handle;
{
  /*display timing check type*/
  io_printf("Timing check is");
  switch( acc_fetch_fulltype( tchk_handle ) )
     {
        case  accHold:
           io_printf(" hold\n");
           break;
        case  accNochange:
           io_printf(" nochange\n");
           break;
        case  accPeriod:
           io_printf(" period\n");
           break;
        case  accRecovery:
           io_printf(" recovery\n");
           break;
        case  accSetup:
           io_printf(" setup\n");
           break;
        case  accSkew:
           io_printf(" skew\n");
           break;
        case  accWidth:
           io_printf(" width\n");
     }
}
```

Figure 2-30: Displaying the fulltypes of timing checks

```
#include "acc_user.h"

display_primitive_type(primitive_handle)
handle  primitive_handle;
{
  /*display primitive type*/
  io_printf("Primitive is");
  switch( acc_fetch_fulltype( primitive_handle ) )
  {
    case accAndGate:
           io_printf(" and gate\n"); break;
    case accBufGate:
           io_printf(" buf gate\n"); break;
    case accBufif0Gate:case accBufif1Gate:
           io_printf(" bufif gate\n"); break;
    case accCmosGate:case accNmosGate:case accPmosGate:
    case accRcmosGate:case accRnmosGate:case accRpmosGate:
           io_printf(" MOS or Cmos gate\n"); break;
    case accCombPrim:
           io_printf(" combinational UDP\n"); break;
    case accSeqPrim:
           io_printf(" sequential UDP\n"); break;
    case accNotif0Gate:case accNotif1Gate:
           io_printf(" notif gate\n"); break;
    case accRtranGate:
           io_printf(" rtran gate\n"); break;
    case accRtranif0Gate:case accRtranif1Gate:
           io_printf(" rtranif gate\n"); break;
    case accNandGate:
           io_printf(" nand gate\n"); break;
    case accNorGate:
           io_printf(" nor gate\n"); break;
    case accNotGate:
           io_printf(" not gate\n"); break;
    case accOrGate:
           io_printf(" or gate\n"); break;
    case accPulldownGate:
           io_printf(" pulldown gate\n"); break;
    case accPullupGate:
           io_printf(" pullup gate\n"); break;
    case accXnorGate:
           io_printf(" xnor gate\n"); break;
    case accXorGate:
           io_printf(" xor gate\n");
  }
}
```

Figure 2-31: Displaying the fulltypes of primitives

2.15.14 acc_fetch_index

acc_fetch_index			
function	returns a zero-based integer index for a port or terminal		
syntax	*integer_variable* **= acc_fetch_index(** *object_handle***)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of the port or terminal

What is an index?

The index of a *port* is its position in a module definition in your source description; the index of a *terminal* is its position in a gate, switch, or UDP instance. Indexes are integers that start at zero and increase from left to right. The following table shows how indexes are derived:

For:	*Indexes are:*
implicit ports: `module A( q,a,b);`	***0*** for port ***q*** ***1*** for port ***a*** ***2*** for port ***b***
terminals: `nand g1(out,in1,in2);`	***0*** for terminal ***out*** ***1*** for terminal ***in1*** ***2*** for terminal ***in2***

explicit ports:
`module top;`
`reg  ra,rb;`
`wire wq;`
`explicit_port_mod epm1( .b(rb),.a(ra),.q(wq) );` ← port instance
`endmodule`
 	
`module m1;`
`explicit_port_mod( q,a,b );` ← port definition
`input  a,b;`
`output q;`
`nand( q,a,b );`
`endmodule`
	0 for explicit port ***epm1.q***
	1 for explicit port ***epm1.a***
	2 for explicit port ***epm1.b***

Table 2-55: Deriving indexes

Usage example

The following example presents a C-language routine, `display_inputs`, that uses `acc_fetch_index` to find and display the input ports of a module.

```
#include "acc_user.h"

display_inputs(module_handle)
handle module_handle;
{
  handle   port_handle;
  int      direction;

  /*get handle for the module and each of its ports*/
  port_handle = null;
  while ( port_handle = acc_next_port( module_handle, port_handle) )
  {
     /*determine if port is an input*/
     direction = acc_fetch_direction( port_handle );
     /*give the index of each input port*/
     if ( direction == accInput )
         io_printf( "Port #%d of %s is an input\n",
                     acc_fetch_index( port_handle ),
                     acc_fetch_fullname( module_handle) );
  }
}
```

Figure 2-32: Displaying the input ports of a module

2.15.15 acc_fetch_location

acc_fetch_location			
function	returns the location of an object in a Verilog-HDL source file		
syntax	**acc_fetch_location (** *loc_p, object_handle* **)**;		
arguments	name	type	description
input:	**loc_p**	*p_location*	pointer to location data structure
	object_handle	*handle*	handle to object

Description

This function returns the *file name* and *line number* in the file for the specified object. The *file name* and *line number* are returned in an `s_location` data structure. This data structure is defined in Figure 2-33.

```
typedef struct t_location
{
   int line_no; /* line number in the file */
   char *filename; /* file name */
} s_location, *p_location;
```

*Figure 2-33: **t_location** data structure*

The `filename` field is a character pointer. The `line_no` field is a nonzero positive number.

Example

The following example prints the file name and line number for an object.

```
void find_object_location (object)
   handle object;
{
   s_location s_loc;
   p_location loc_p = &s_loc;
   acc_fetch_location(loc_p, object); /* To get the filename
                                          and line_no */
   if (! acc_error_flag)  /* On success */
      io_printf (“Object located in file %s on line %d \n,
                  loc_p->filename, loc_p->line_no);
}
```

*Figure 2-34: Using **acc_fetch_location** to find the file name and line number for an object*

2.15.16 acc_fetch_name

acc_fetch_name			
function	returns a pointer to the instance name of any named object or module path		
syntax	*character_pointer* **= acc_fetch_name(** *object_handle***)***;*		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of the named object
related routines	Use ***acc_configure(accPathDelimStr...)*** to set the naming convention for module paths		

What is the name of an object?

The ***name*** of an object is its lowest-level name. Consider this example: the top-level module `top1` contains a module instance `mod3` which, in turn, contains a net `w4`, as shown in Figure 2-35.

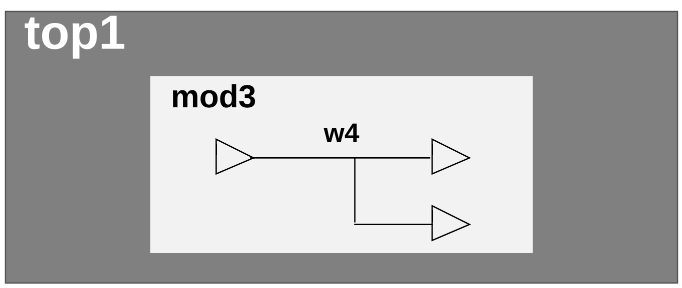


Figure 2-35: A design hierarchy

The name of the net is `w4`.

Module path names

Module paths are identified in terms of their sources and destinations in the following format:

source	**path_delimiter**	**destination**

Names of module paths with multiple sources or destinations are derived from the first source or destination only.

By default, the ***path_delimiter*** is the character `$`. However, you can override this default by using the access routine `acc_configure` to set the delimiter parameter `accPathDelimStr` to some other character string.

Here are some examples of module path names returned by `acc_fetch_name` when the ***path_delimiter*** is `$`:

For module path:	***acc_fetch_name*** *returns pointer to:*
`(a => q) = 10;`	***a$q***
`(a *> q1,q2) = 8;`	***a$q1***
`(d,e,f *> q,r)= 8;`	***d$q***

Table 2-56: Module path names returned by ***acc_fetch_name***

Default names

If the Verilog simulator default names for unnamed instances, `acc_fetch_name` returns the default name. Otherwise, the routine returns *null* for unnamed instances.

Usage example

The following example presents a C-language routine, `show_top_modules`, that uses `acc_fetch_name` to display the names of all top-level modules.

```
#include "acc_user.h"

show_top_modules()
{
   handle  module_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*scan all top-level modules*/
   io_printf("The top-level modules are:\n");
   module_handle = null;
   while ( module_handle = acc_next_topmod( module_handle ) )
         io_printf(" %s\n",acc_fetch_name(module_handle));

   acc_close();
}
```

Figure 2-36: Displaying the names of all top-level modules

2.15.17 acc_fetch_paramtype

acc_fetch_paramtype			
function	returns the data type of a parameter as one of three predefined integer constants: ***accIntegerParam*** ***accRealParam*** ***accStringParam***		
syntax	*integer_variable* **= acc_fetch_paramtype(** *parameter_handle***)**;		
arguments	name	type	description
input:	**parameter_handle**	*handle*	handle of a parameter
related routines	Use ***acc_fetch_paramval*** to retrieve the value of a parameter Use ***acc_next_parameter*** to scan all parameters within a module Use ***acc_next_specparam*** to scan all specparams within a module		

How the routine works

When type is:	***acc_fetch_paramtype*** *returns:*
integer	***accIntegerParam***
floating point	***accRealParam***
string	***accStringParam***

*Table 2-57: How **acc_fetch_paramtype** works*

Usage example

The following example presents a C-language routine, `print_parameter_values`, that uses `acc_fetch_paramtype` to display the values of all parameters within a module.

```
#include "acc_user.h"

print_parameter_values()
{
   handle   module_handle;
   handle   param_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );

   /*scan all parameters in the module and display their values*/
   /*  according to type*/
   param_handle = null;
   while( param_handle = acc_next_parameter( module_handle,param_handle))
   {
      io_printf( "Parameter %s has value: ",
                  acc_fetch_fullname( param_handle ));
      switch( acc_fetch_paramtype( param_handle ) )
      {
         case accRealParam:
           io_printf( "%lf\n", acc_fetch_paramval( param_handle) );
           break;
         case accIntegerParam:
           io_printf( "%d\n", (int)acc_fetch_paramval( param_handle) );
           break;
         case accStringParam:
           io_printf("%s\n",(char*)(int)acc_fetch_paramval(param_handle) );
           break;
      }
   }
   acc_close();
}
```

two-step cast

Figure 2-37: Displaying the values of all parameters within a module

Please note: Most C-language compilers do not allow you to cast a double-precision value directly to a **character pointer**. For string parameters in this example, it is therefore necessary to use a two-step cast, **(char*)(int)**, to first convert the **double** value to an integer and then convert the integer to a character pointer.

2.15.18 acc_fetch_paramval

acc_fetch_paramval			
function	returns the value of a parameter or specparam		
syntax	*double_variable* **= acc_fetch_paramval(** *parameter_handle* **)**;		
arguments	name	type	description
input:	**parameter_handle**	*handle*	handle of a parameter
related routines	Use ***acc_fetch_paramtype*** to retrieve the data type of a parameter Use ***acc_next_parameter*** to scan all parameters within a module Use ***acc_next_specparam*** to scan all specparams within a module		

Three types of parameter values

A parameter value can be stored as one of three data types:

1. ***double***, a double-precision floating point number
2. ***int***, an integer value
3. ***char***, a string

It is therefore often necessary to check the *type* of the parameter—using `acc_fetch_paramtype`—to determine the format for displaying its value, as shown in the example in Figure 2-38.

Casting return values

The routine `acc_fetch_paramval` returns values as double precision floating point numbers. However, you can convert these values to ***integers*** or ***character pointers*** using the C-language ***cast*** mechanism, as shown in the following table:

To convert to:	*Follow these steps:*
integer	1. declare an integer variable ***int_val*** 2. cast the return value to the integer data type, using the C-language cast operator ***(int)*** 3. ***assign the return*** integer to the variable: **`int_val=(int)acc_fetch_paramval(..);`**
string	1. declare a character pointer variable ***ptr*** 2. cast the return value to a character pointer, using the C-language cast operators ***(char*)(int)*** 3. assign the return pointer to the variable: **`ptr=(char*)(int)acc_fetch_paramval(..);`**

*Table 2-58: Casting **acc_fetch_paramval** return values*

Usage example

The following example presents a C-language routine, `print_parameter_values`, that uses `acc_fetch_paramval` to display all parameter values for a module.

```
#include "acc_user.h"

print_parameter_values()
{
  handle   module_handle;
  handle   param_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg( 1 );

  /*scan all parameters in the module and display their values*/
  /*  according to type*/
  param_handle = null;
  while( param_handle = acc_next_parameter( module_handle,param_handle))
  {
      io_printf( "Parameter %s has value: ",
                  acc_fetch_fullname( param_handle ));
      switch( acc_fetch_paramtype( param_handle ) )
      {
         case accRealParam:
           io_printf( "%lf\n", acc_fetch_paramval( param_handle) );
           break;
         case accIntegerParam:
           io_printf( "%d\n", (int)acc_fetch_paramval( param_handle) );
           break;
         case accStringParam:
           io_printf("%s\n",(char*)(int)acc_fetch_paramval(param_handle) );
           break;
      }
  }
  acc_close();
}
```

two-step cast

Figure 2-38: Displaying all parameter values for a module

Please note: Most C-language compilers do not allow you to cast a double-precision value directly to a **character pointer**. For string parameters in this example, it is therefore necessary to use a two-step cast, **(char*)(int)**, to first convert the **double** value to an integer and then convert the integer to a character pointer.

2.15.19 acc_fetch_polarity

acc_fetch_polarity			
function	***returns the polarity of the specified path***		
syntax	*integer_variable* **= acc_fetch_polarity(** *path_handle***);**		
arguments	name	type	description
input:	**path_handle**	*handle*	handle to a module path or data path

Description

This routine returns the polarity of the specified path. The polarity of a path determines how a signal transition at its source propagates to its destination in the absence of logic simulation events. The return value is one of the predefined integer constant polarity types listed in Table 2-59.

Predefined Integer Constant	**Description**
`accPositive`	A rise at the source causes a rise at the destination. A fall at the source causes a fall at the destination.
`accNegative`	A rise at the source causes a fall at the destination. A fall at the source causes a rise at the destination.
`accUnknown`	Unpredictable. A rise or fall at the source causes either a rise or fall at the destination.

Table 2-59: Polarity types

Example

The following example takes a path argument and returns the string corresponding to its polarity.

```
char *fetch_polarity_str(path)
{
   switch (acc_fetch_polarity(path)) {
      case accPositive:  return(“accPositive”);
      case accNegative: return(“accNegative”);
      case accUnknown: return(“accUnkinown”);
      default: return(NULL);
   }
}
```

*Figure 2-39: Using **acc_fetch_polarity** to get the polarity of a path*

2.15.20 acc_fetch_range

acc_fetch_range			
function	retrieves the most significant bit and least significant bit range values for a vector; returns zero if successful and nozero upon error		
syntax	*integer_variable* **= acc_fetch_range(** *vector_handle, msb, lsb***)**;		
arguments	name	type	description
input:	**vector_handle**	*handle*	handle to a vector net or register
	msb	*integer pointer*	pointer to an integer variable to hold the most significant bit of *vector_handle*
	lsb	*integer pointer*	pointer to an integer variable to hold the least significant bit of *vector_handle*

Description

The ‘lsb’ will be the right range element, while the ‘msb’ will be the left range element.

Example

This system task takes as its only input a handle to a module instance. It displays the name and range of each vector net found in the module as: <name>[<msb>:<lsb>].

```
display_vector_nets()
{
   handle mod = acc_handle_tfarg(1);
   handle net;
   integer msb, lsb;

   io_printf (“Vector nets in module %s:\n:,
      acc_fetch_fullname (mod));

   net = null;
   while (net = acc_next_net(mod, net))
      if (acc_object_of_type(net, accVector))
      {
         acc_fetch_range(net, &msb, &lsb);
         io_printf(“  %s[%d:%d]\n”,
            acc_fetch_name(net), msb, lsb);
      }
```

Figure 2-40: Displaying the name and range for each vector net found in a scope

2.15.21 acc_fetch_size

acc_fetch_size			
function	returns the bit size of a net, register, or port		
syntax	*size_in_bits* **= acc_fetch_size(***object_handle***);**		
	name	type	description
return value	**size_in_bits**	*integer*	number of bits in the net, register, or port
input	**object_handle**	*handle*	handle of a net, register, or port

The access routine `acc_fetch_size` returns the number of bits of a net, register, or port.

Usage example

In the following example, the routine `display_vector_size` uses `acc_fetch_size` to determine the size of a vector net. The routine then displays the name of the vector net and its size in bits.

```
#include "acc_user.h"

void  display_vector_size()
{
   handle   net_handle;
   int      size_in_bits;

   /*reset environment for access routines*/
   acc_initialize();
   acc_configure( accDevelopmentVersion, "1.6" );

   /*get first argument passed to user-defined system task*/
   /* associated with this routine*/
   net_handle = acc_handle_tfarg(1);

   /*if net is a vector, find and display its size in bits*/
   if( acc_object_of_type( net_handle, accVector ) )
   {
      size_in_bits = acc_fetch_size( net_handle );
      io_printf("Net %s is a vector of size %d\n",
                   acc_fetch_fullname( net_handle ),size_in_bits );
   }
   else
      io_printf( "Net %s is not a vector net\n",
                  acc_fetch_fullname( net_handle ) );
}
```

Figure 2-41: Determining the size of a vector net

2.15.22 acc_fetch_tfarg

acc_fetch_tfarg			
function	returns value of the specified argument of the system task or function associated (through the PLI mechanism) with your C-language routine		
syntax	*double_variable* **= acc_fetch_tfarg(***argument_number***)***;*		
arguments	name	type	description
input:	**argument_number**	*integer*	value that references an object—passed as a variable in the system task or function call—by its position in the argument list

The access routine `acc_fetch_tfarg` returns the value of arguments passed to user-defined system tasks and functions.

How arguments are numbered

Argument numbers start at **1** and increase from left to right in the order that they appear in the system task or function call.

Casting values

Values are returned as double-precision floating-point numbers, but they can be cast to integers and character pointers as well. The following example shows how to use `acc_fetch_tfarg` and cast its return values appropriately.

```
#include "acc_user.h"
#include "veriuser.h"

display_arg_value()
{
 int      arg_type;

 /*initialize environment for access routines*/
 acc_initialize();

 /*set development version*/
 acc_configure( accDevelopmentVersion, "1.5b.3" );

 /*check type of argument*/
 io_printf( "Argument value is " );

 switch( tf_typep( 1 ) )
 {
    case tf_readonlyreal:case tf_readwritereal:
       io_printf("%1f\n", acc_fetch_tfarg(1) );
       break;
    case tf_readonly:case tf_readwrite:
       io_printf("%d\n", (int)acc_fetch_tfarg(1) );
       break;
    case tf_string:
       io_printf("%s\n", (char*)(int)acc_fetch_tfarg(1) );
       break;
    default:
       io_printf("Error in argument specification\n");
       break;
 }
 acc_close();
}
```

returns value as **double-precision floating-point** number

casts value to **integer**

casts value to a **character pointer**

*Figure 2-42: Using **acc_fetch_tfarg** to return the value of arguments*

2.15.23 acc_fetch_type

<table>
<tr><th colspan="4">acc_fetch_type</th></tr>
<tr><td>function</td><td colspan="3">returns the type of an object as one of these predefined integer constants:

accDataPath
accFunction
accIntegerVar
accModule
accNamedEvent
accNet
accParameter
accPath
accPathTerminal
accPort
accPrimitive
accPrimPath
accRealVar
accRegister
accSpecparam
accStatement
accTask
accTchk
accTerminal
accTimeVar
accWirePath</td></tr>
<tr><td>syntax</td><td colspan="3">integer_variable = acc_fetch_type(object_handle);</td></tr>
<tr><td>arguments</td><td>name</td><td>type</td><td>description</td></tr>
<tr><td>input:</td><td>object_handle</td><td>handle</td><td>handle of the object</td></tr>
<tr><td>related routines</td><td colspan="3">acc_fetch_type_str
acc_fetch_fulltype</td></tr>
</table>

The difference between type and fulltype

The *type* of an object is its general Verilog-HDL data type classification. Table 2-60 lists these data types, along with the predefined integer constants that represent them.

The *fulltype* of an object is a finer classification of a particular Verilog-HDL data type. Currently, specific *fulltypes* are defined for the following objects:

- primitives
- terminals
- ports
- bits of vector or concatenated ports
- parameters and specparams
- nets
- modules
- module paths and module path terminals
- timing checks and timing check terminals
- registers
- named events
- integer, real, and time variables
- data paths
- inter-module paths
- statements

The access routine `acc_fetch_type` returns the *type* of an object, while `acc_fetch_fulltype` returns the *fulltype* of an object. Table 2-61 illustrates the difference between `acc_fetch_fulltype` and `acc_fetch_type` for selected objects.

Verilog-HDL data type	*Predefined integer constant*
data path	accDataPath
integer variable	accIntegerVar
module	accModule
module path	accPath
module path terminal	accPathTerminal
named event	accNamedEvent
net	accNet
parameter	accParameter
path between primitives	accPrimPath
port	accPort
primitive	accPrimitive
real variable	accRealVar
register	accRegister
specparam	accSpecparam
terminal	accTerminal
time variable	accTimeVar
timing check	accTchk
Verilog-HDL function definition	accFunction
Verilog-HDL statement	accStatement
Verilog-HDL task definition	accTask
wire path	accWirePath

*Table 2-60: The **types** of Verilog-HDL objects*

For:	***acc_fetch_fulltype** returns:*	***acc_fetch_type** returns:*
a setup timing check	accSetup	accTchk
an AND gate	accAndGate	accPrimitive
a sequential primitive	accSeqPrim	accPrimitive

*Table 2-61: The difference between **acc_fetch_fulltype** and **acc_fetch_type***

Usage example

The following example presents a C-language routine, `display_object_type`, that uses `acc_fetch_type` to identify the type of an object.

```
#include "acc_user.h"

display_object_type()
{
   handle object_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   object_handle = acc_handle_tfarg(1);

   /*display object type*/
   switch( acc_fetch_type( object_handle ) )
      {
         case accModule:
            io_printf( "Object is a module\n" );
            break;
         case accNet:
            io_printf( "Object is a net\n" );
            break;
         case accPath:
            io_printf("Object is a module path\n" );
            break;
         case accPort:
            io_printf("Object is a module port\n" );
            break;
         case accPrimitive:
            display_primitive_type( object_handle );
            break;
         case accTchk:
            display_timing_check_type( object_handle );
            break;
         case accTerminal:
            io_printf("Object is a primitive terminal\n" );
            break;
      }
   acc_close();
}
```

Figure 2-43: Identifying the type of an object

Other routines may be called to identify an object's *fulltype*. Two example routines—`display_primitive_type` and `display_timing_check_type`—use the access routine `acc_fetch_fulltype` to provide more specific identification of a primitive or timing check. These are presented in the section on *acc_fetch_fulltype*.

2.15.24 acc_fetch_type_str

acc_fetch_type_str			
function	returns a pointer to a string that indicates the type of its argument		
syntax	*character_string_pointer* **= acc_fetch_type_str(type);**		
arguments	name	type	description
input:	**type**	*integer*	a predefined integer constant that stands for an object type
related routines	***acc_fetch_type*** ***acc_fetch_fulltype***		

Purpose of the routine

This routine returns the character string that specifies the type of its argument. Use this routine when you are debugging or need type information.

Usage example

In the following example, a handle to an argument is passed to a C routine. The routine displays the name of the object and the object's type.

```
#include "acc_user.h"
void        display_object_type(object)
handle      object;
{
  int   type = acc_fetch_type(object);

  io_printf("Object %s is of type %s \n",
            acc_fetch_fullname(object),
            acc_fetch_type_str(type));
}
```

Figure 2-44: Displaying an object type

In Figure 2-44, if you pass a handle to an object named `top.param1`, the routine `display_object_type` produces the following output:

```
Object top.param1 is of type accParameter
```

The access routine `acc_fetch_type_str` returns the character string that is equivalent to the defined integer constant that represents the object in the `acc_user.h` file. In this output, `accParameter` is the name of the integer constant that represents the type `parameter`.

2.15.25 acc_fetch_value

<table>
<tr><th colspan="4">acc_fetch_value</th></tr>
<tr><td>function</td><td colspan="3">returns a pointer to a character string indicating the logic or strength value of a net, register or variable</td></tr>
<tr><td>syntax</td><td colspan="3">character_pointer = acc_fetch_value(object_handle, format_string);</td></tr>
<tr><td>arguments</td><td>name</td><td>type</td><td>description</td></tr>
<tr><td>inputs:</td><td>object_handle</td><td>handle</td><td>handle of the net, register or variable</td></tr>
<tr><td></td><td>format_string</td><td>character string pointer:
quoted string literal or
character pointer variable</td><td>one of the following specifiers for formatting the return value:
"%b"
"%d"
"%h"
"%o"
"%v"
"%x"</td></tr>
</table>

Size of objects

The access routine `acc_fetch_value` returns ***logic*** simulation values for ***scalar*** or ***vector*** nets, registers and variables; it returns ***strength*** values for ***scalar*** nets and registers only.

How the routine formats return value strings

The access routine `acc_fetch_value` returns logic or strength values as character strings. You can format these value strings by passing a format specifier as the second argument.

Consider this example: At a particular time during simulation, a vector contains the value ***92*** (decimal). Table 2-62 shows how `acc_fetch_value` returns this value in different formats.

When specifier is:	*type is:*	***acc_fetch_value** returns:*
"%b"	binary	***"01011100"***
"%d"	decimal	***"92"***
"%h"	hexadecimal	***"5c"***
"%o"	octal	***"134"***
"%v"	strength	***null*** and sets ***acc_error_flag*** (strength only valid for scalar objects)
"%x"	hexadecimal	***"5c"***

*Table 2-62: How **acc_fetch_value** uses format specifiers*

Note that when specified for scalar objects, %v produces the standard Verilog-HDL strength format. Refer to the *Verilog-HDL Reference Manual* for information about %v.

Usage example

In the example in Figure 2-45, the routine `display_net_values` uses `acc_fetch_value` to retrieve the logic values of all nets in a module.

```
#include "acc_user.h"

display_net_values()
{
  handle  mod_handle, net_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  mod_handle = acc_handle_tfarg(1);

  /*get all nets in the module and display their values*/
  /* in binary format*/
  net_handle = null;
  while( net_handle = acc_next_net(mod_handle,net_handle) )
      io_printf("Net value: %s\n",acc_fetch_value(net_handle,"%b"));

  acc_close();
}
```

Figure 2-45: Retrieving the logic values of all nets in a module

2.15.26 acc_free

acc_free			
function	frees memory allocated by ***acc_collect***		
syntax	**acc_free(** *handle_array_pointer* **);**		
arguments	name	type	description
input:	**handle_array_pointer**	*handle pointer*	pointer to the array of handles allocated by ***acc_collect***—the area in memory to free
related routines	***acc_collect***		

The difference between acc_free and acc_close

The routine `acc_free` frees memory allocated by `acc_collect` to store an array of handles; the routine `acc_close` frees memory that all access routines use for internal storage.

Usage example

The following example presents a C-language routine, `display_nets`, that uses `acc_free` to deallocate memory used by the array returned by `acc_collect`.

```
#include "acc_user.h"

display_nets()
{
  handle   *list_of_nets, module_handle;
  int      net_count, i;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg( 1 );

  /*collect all nets in the module*/
  list_of_nets = acc_collect( acc_next_net, module_handle, &net_count);

  /*display names of net instances*/
  for( i=0; i < net_count; i++ )
     io_printf( "Net name is: %s\n", acc_fetch_name( list_of_nets[i] ));

  /*free memory used by array list_of_nets*/
  acc_free( list_of_nets );

  acc_close();
}
```

*Figure 2-46: Using **acc_collect** to gather all nets in a module for display*

2.15.27 acc_handle_by_name

acc_handle_by_name			
function	returns the handle to an object based on its name and scope		
syntax	*object_handle* **= acc_handle_by_name (** *object_name, scope_handle***)**;		
arguments	name	type	description
input:	**object_name**	*character string pointer; quoted string literal or character pointer variable*	character pointer to object name
	scope_handle	*handle*	handle to scope

Description

This routine returns the handle to a Verilog-HDL object based on the specified name and scope.

This routine can be used in place of the combination of routines `acc_set_scope` and `acc_handle_object`. While the functionality is the same as calling `acc_set_scope` followed by a call to `acc_handle_object`, this routine makes the code cleaner and easier to understand and maintain.

Example

The following example shows how a C-language routine, is_net_in_module, uses acc_handle_by_name to set the scope and get the handle to an object if the object is in the module.

```
#include "acc_user.h"

is_net_in_module(module_handle, net_name)
handle module_handle;
char *net_name;
{
  handle net_handle;
  handle load_handle, load_net_handle;

  /*set scope to module and get handle for net */
  net_handle = acc_handle_by_name(net_name, module_handle);

  if (net_handle)
     io_printf("Net %s found in module %s\n",
                 net_name,
                 acc_fetch_fullname(module_handle) );
  else
     io_printf("Net %s not found in module %s\n",
                 net_name,
                 acc_fetch_fullname(module_handle) );
}
```

Figure 2-47: Setting a scope and getting a handle

In this example:

```
net_handle = acc_handle_by_name(net_name, module_handle);
```

could also have been written as follows:

```
acc_set_scope(module_handle);
net_handle = acc_handle_object(net_name);
```

Related routines

`acc_set_scope()`

`acc_handle_object()`

Notes

The routines `acc_handle_object` and `acc_set_scope` will continue to be supported.

2.15.28 acc_handle_condition

acc_handle_condition			
function	returns a handle to the conditional expression for the specified path		
syntax	*condition_handle* **= acc_handle_condition (** *path_handle* **)**;		
arguments	name	type	description
input:	**path_handle**	*handle*	handle to a module path

Description

The return value is the conditional expression for the specified module or data path. If there is no condition, null is returned.

Examples

The following routines provide functionality to see if a path is conditional, and, if it is, whether it is level-sensitive or edge-sensitive. These routines assume that the input is a valid handle to a module path.

```
bool is_path_conditional(path)
{
   if (acc_handle_condition(path))
      return(TRUE);
   else
      return(FALSE);
}

bool is_level_sensitive(path)
{
   bool flag;
   handle path_in = acc_next_pathin(path, null);

   if (is_path_conditional(path) && acc_fetch_edge(path_in))
         flag = FALSE; /* path is edge-sensitive */
      else
         flag = TRUE; /* path is level_sensitive */

   acc_release_object(path_in);

   return (flag);
}
```

Figure 2-48: Finding conditional edge-sensitive and level-sensitive paths

2.15.29 acc_handle_conn

acc_handle_conn			
function	returns handle to the net connected to a primitive terminal		
syntax	*net_handle* **= acc_handle_conn(** *terminal_handle***)**;		
arguments	name	type	description
input:	**terminal_handle**	*handle*	handle of the primitive terminal
related routine	Call ***acc_next_terminal*** or ***acc_handle_terminal*** to obtain **terminal_handle** Call ***acc_next_load*** to obtain terminal loads Call ***acc_next_driver*** to obtain terminal drivers		

Usage example

The following example presents a C-language routine, `display_driven_net`, that displays the net connected to the output terminal of a gate.

```
#include "acc_user.h"

display_driven_net()
{
   handle  gate_handle, terminal_handle, net_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for the gate*/
   gate_handle = acc_handle_tfarg( 1 );

   /*get handle for the gate's output terminal*/
   terminal_handle = acc_handle_terminal( gate_handle, 0 );

   /*get handle for the net connected to the output terminal*/
   net_handle = acc_handle_conn( terminal_handle );

   /*display net name*/
   io_printf( "Gate %s drives net %s\n",
               acc_fetch_fullname( gate_handle ),
               acc_fetch_name( net_handle ) );

   acc_close();
}
```

Figure 2-49: Displaying the net connected to the output terminal of a gate

2.15.30 acc_handle_datapath

acc_handle_datapath			
function	returns a handle to a datapath for a module instance for the specified edge-sensitive module path		
syntax	*datapath_handle* **= acc_handle_datapath (** *modpath_handle* **)**;		
arguments	name	type	description
input:	**modpath_handle**	*handle*	handle to a module path

Description

The return value is a handle to the data path associated with the module path. If there is no data path, null is returned.

Example

The following routine finds the data path corresponding to the specified module path and displays the source and destination port names for the data path. It uses `acc_next_input()` and `acc_next_output()` to get the input and output, respectively, for a given path. Since a data path has only one input and one output, you must call `acc_release_object` to free the memory allocated for the input and output handles.

```
display_datapath_terms(modpath)
handle modpath;
{
   handle datapath = acc_handle_datapath(modpath);
   handle pathin  = acc_next_input(datapath, NULL);
   handle pathout = acc_next_output(datapath,  NULL);

   /* there is only one input and output to a datapath */
   io_printf(“DATAPATH INPUT:     %s\n”, acc_fetch_fullname(pathin));
   io_printf(“DATAPATH OUTPUT:  %s\n”, acc_fetch_fullname(pathout));
   acc_release_object(pathin);
   acc_release_object(pathout);
}
```

Figure 2-50: Finding the data path that corresponds to a module path

2.15.31 acc_handle_hiconn

acc_handle_hiconn			
function	returns the hierarchically higher net connection to a scalar module port or a bit of a vector port		
syntax	*hiconn_handle* **= acc_handle_hiconn(** *port_ref_handle* **)**;		
arguments	name	type	description
input:	**port_ref_handle**	*handle*	handle to a scalar port or a bit of a vector port

Description

The function returns the hierarchically higher net connection for a scalar port or a bit of one of the following:

- vector port
- part select of a port
- concatenation of any scalar ports, vector ports, part-selects of ports, or other concatenations

Example

For the indicated port, display the high and low connection(s).

```
display_port_info(mod, index)
handle mod;
int index;
{
   handle port = acc_handle_port (mod, index);
   handle hiconn, loconn, port_bit;

   if (acc_fetch_size(port) = 1) {
      hiconn = acc_handle_hiconn (port);
      loconn = acc_handle_loconn (port);
      io_printf (“    hi: %s  lo: %s\n”,
         acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
   } else {
      port_bit = null;
      while (port_bit = acc_next_bit (port, port_bit))
      {
         hiconn = acc_handle_hiconn (port);
         loconn = acc_handle_loconn (port);
         io_printf (“    hi: %s  lo: %s\n”,
            acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
      }
}
```

Figure 2-51: Displaying the hiconn and loconn of a port

2.15.32 acc_handle_loconn

acc_handle_loconn			
function	returns the hierarchically lower net connection to a scalar module port or a bit of a vector port		
syntax	*loconn_handle* **= acc_handle_loconn(** *port_ref_handle* **)**;		
arguments	name	type	description
input:	**port_ref_handle**	*handle*	handle to a scalar port or a bit of a vector port

Description

The function returns the hierarchically lower net connection for a scalar port or a bit of one of the following:

- vector port
- part select of a port
- concatenation of any scalar ports, vector ports, part selects of ports, or other concatenations

Example

For the indicated port, display the high and low connection(s).

```
display_port_info(mod, index)
handle mod;
int index;
{
   handle port = acc_handle_port (mod, index);
   handle hiconn, loconn, port_bit;

   if (acc_fetch_size(port) = 1) {
      hiconn = acc_handle_hiconn (port);
      loconn = acc_handle_loconn (port);
      io_printf (“    hi: %s  lo: %s\n”,
         acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
   } else {
      port_bit = null;
      while (port_bit = acc_next_bit (port, port_bit))
      {
         hiconn = acc_handle_hiconn (port);
         loconn = acc_handle_loconn (port);
         io_printf (“    hi: %s  lo: %s\n”,
            acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
      }
}
```

Figure 2-52: Displaying the hiconn and loconn of a port

2.15.33 acc_handle_modpath

acc_handle_modpath			
function	returns handle to the path of a module		
syntax	*path_handle* **= acc_handle_modpath(***module_handle, source_name, destination_name***,** *source_handle, destination_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	source_name	*character string pointer:* quoted string literal *or* character pointer variable	name of a net connected to a module path source
	destination_name	*character string pointer:* quoted string literal *or* character pointer variable	name of a net connected to a module path destination
	source_handle (required if accEnableArgs is set and source_name is null)	*handle*	handle of a net connected to a module path source
	destination_handle (required if accEnableArgs is set and destination_name is null)	*handle*	handle of a net connected to a module path destination
related routine	Use ***acc_configure(accEnableArgs, "acc_handle_modpath")*** to use **source_handle** and **destination_handle** arguments		

How the routine works

If:	*acc_handle_modpath:*
you call: *acc_configure(accEnableArgs*, *"acc_handle_modpath")* and you pass: either **source_name** or **destination_name** as a null pointer	uses the associated handle argument, **source_handle** or **destination_handle**, instead of the name argument
you do *not* call: *acc_configure(accEnableArgs*, *"acc_handle_modpath")*	always ignores both handle arguments and uses the name arguments instead *(both handle arguments may be dropped)*

*Table 2-63: How **acc_handle_modpath** works*

Optional arguments

When an optional argument is required for a particular call to `acc_handle_modpath`, you must set the configuration parameter `accEnableArgs` by calling `acc_configure` as follows:

```
acc_configure(accEnableArgs, "acc_handle_modpath");
```

If `accEnableArgs` is not set for `acc_handle_modpath`, the routine always ignores its optional arguments.

When an optional argument is *not* required for a particular call to `acc_handle_modpath`, it can be dropped as long as it does not precede any required arguments.

However, when an optional argument does precede one or more required arguments, it must be supplied even if it is ignored by the access routine. In this case, you can specify the argument as a null value.

Usage example

The following example shows how the C-language routine `get_paths` uses `acc_handle_modpath` to obtain handles for the paths that connect each of the sources and destinations listed in the file `pathconn.dat`: The format of `pathconn.dat` appears in Figure 2-53; the source code for `get_paths` appears in Figure 2-54.

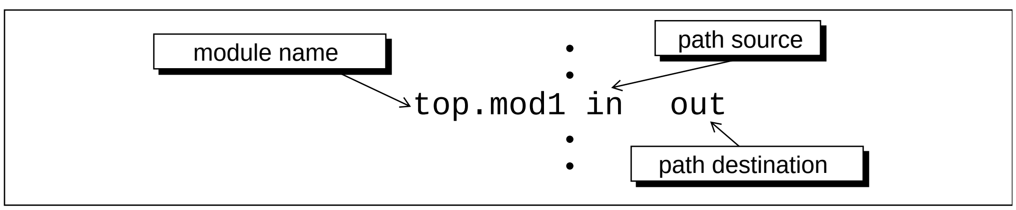


*Figure 2-53: Format of file **pathconn.dat***

```
#include <stdio.h>
#include "acc_user.h"

#define NAME_SIZE 256

get_paths()
{
   FILE   *infile;
   char   mod_name[NAME_SIZE], src_name[NAME_SIZE], dest_name[NAME_SIZE];
   handle   path_handle, mod_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*set accPathDelimStr to "_"*/
   acc_configure( accPathDelimStr, "_" );

   /*read delays from file - "r" means read only*/
   infile = fopen("pathconn.dat","r");
   while( fscanf(infile,"%s %s %s",mod_name,src_name,dest_name) != EOF )
   {
     /*get handle for module mod_name*/
     mod_handle = acc_handle_object( mod_name );
     path_handle = acc_handle_modpath( mod_handle, src_name, dest_name );
     if( !acc_error_flag )
       io_printf( "Path %s was found\n",
                  acc_fetch_fullname( path_handle ) );
     else
       io_printf( "Path %s_%s was not found\n",src_name,dest_name );
   }
   acc_close();
}
```

Figure 2-54: Displaying specific paths in a module

The identifier EOF is a predefined constant that stands for ***end of file***. NAME_SIZE is a user-defined constant that represents the maximum number of characters allowed for any object name in an input file.

2.15.34 acc_handle_object

acc_handle_object			
function	returns handle for any named object		
syntax	*net_handle* **= acc_handle_object(***object_instance_name***)***;*		
arguments	name	type	description
input:	**object_instance_name**	*character string pointer:* quoted string literal *or* character pointer variable	full or relative hierarchical path name of the object instance
related routine	To call ***acc_handle_object*** with a simple object instance name, call ***acc_set_scope*** first to set scope to the appropriate level in the design hierarchy		

Usage example

The following example presents a C-language routine, `write_prim_delays`, that uses `acc_handle_object` to retrieve handles for net names read from a file called `primdelay.dat`. The format of the file is shown in Figure 2-55; the example appears in Figure 2-56.

Note that `write_prim_delays` assumes that each net is driven by only one primitive.

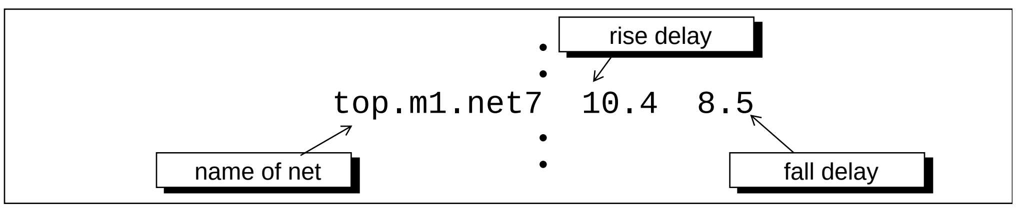


Figure 2-55: Format of file ***primdelay.dat***

```
#include <stdio.h>
#include "acc_user.h"

#define NAME_SIZE 256

write_prim_delays()
{
   FILE     *infile;
   char     full_net_name[NAME_SIZE];
   double   rise,fall;
   handle   net_handle, driver_handle, prim_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*set accPathDelayCount parameter for rise and fall delays only*/
   acc_configure( accPathDelayCount, "2" );

   /*read delays from file - "r" means read only*/
   infile = fopen("primdelay.dat","r");
   while( fscanf(infile,"%s %lf %lf",full_net_name,&rise,&fall) != EOF )
   {
      /*get handle for the net*/
      net_handle = acc_handle_object(full_net_name);

      /*get primitive connected to first net driver*/
      driver_handle = acc_next_driver( net_handle, null);
      prim_handle = acc_handle_parent( driver_handle );

      /*replace delays with new values*/
      acc_replace_delays( prim_handle, rise, fall );
   }
   acc_close();
}
```

Figure 2-56: Writing new rise and fall delays for primitives

The identifier `EOF` is a predefined constant that stands for ***end of file***. `NAME_SIZE` is a user-defined constant that represents the maximum number of characters allowed for any object name in an input file.

2.15.35 acc_handle_parent

acc_handle_parent			
function	returns handle for the parent primitive instance or module instance of an object		
syntax	*parent_handle* **= acc_handle_parent(***object_handle***);**		
arguments	name	type	description
input:	**object_handle**	*handle*	handle of an object

What is a parent?

A parent is an object that *contains* another object. The parent of a *terminal* is the *primitive object* that contains it; the parent of *any other object* (except a top-level module) is the *module instance* that contains argthe object.

Top-level modules do not have parents. Therefore, when you pass a top-level module handle to `acc_handle_parent`, it returns null.

Usage example

This example shows the C-language routine `get_primitives` that uses `acc_handle_parent` to determine which primitives' terminals drive a net.

```
#include "acc_user.h"

get_primitives(net_handle)
handle   net_handle;
{
   handle   primitive_handle;
   handle   driver_handle;

   /*get primitive that owns each terminal that drives the net*/
   driver_handle = null;
   while( driver_handle = acc_next_driver( net_handle, driver_handle ) )
   {
      primitive_handle = acc_handle_parent( driver_handle );
      io_printf( "Primitive %s drives net %s\n",
                     acc_fetch_fullname( primitive_handle ),
                     acc_fetch_fullname( net_handle ) );
   }
}
```

Figure 2-57: Determining which primitives' terminals drive a net

2.15.36 acc_handle_path

<table>
<tr><th colspan="4">acc_handle_path</th></tr>
<tr><td>function</td><td colspan="3">returns a handle to an inter-module path that represents the connection from an output port to an input port</td></tr>
<tr><td>syntax</td><td colspan="3">InterModPath_handle = acc_handle_path(port_output_handle, port_input_handle);</td></tr>
<tr><td>arguments</td><td>name</td><td>type</td><td>description</td></tr>
<tr><td>inputs:</td><td>port_output_handle</td><td>handle</td><td>handle to one of the following:
• a scalar output port
• a scalar bidirectional port
• one bit of a vector output port
• one bit of a vector bidirectional port</td></tr>
<tr><td></td><td>port_input_handle</td><td>handle</td><td>handle to one of the following:
• a scalar input port
• a scalar bidirectional port
• one bit of a vector input port
• one bit of a vector bidirectional port</td></tr>
<tr><td>related routines</td><td colspan="3">Use acc_next_port, acc_handle_port to retrieve a handle to a scalar port
Use acc_next_bit to retrieve a handle to a bit of a vector port or a bit of a concatenated port
Use acc_fetch_direction to determine whether a port is an input, an output, or bidirectional</td></tr>
</table>

Use `acc_handle_path` to retrieve handles to *inter-module paths*. An inter-module path is a wire path that connects an output or inout port of one module to an input or inout port of another module.

Arguments

The access routine `acc_handle_path` takes two arguments. Table 2-64 outlines the requirements for each argument.

The first argument must be:	*The second argument must be:*
m an output port *or* bidirectional port	m an input port *or* bidirectional port
m scalar *or* be a bit of a vector port or concatenated port	m scalar *or* be a bit of a vector port or of a concatenated port

Table 2-64: Arguments to ***acc_handle_path***

Paths supported

In this release, `acc_handle_path` returns handles *only* for wire paths that are inter-module paths.

Paths Not Supported

This routine `acc_handle_path` does not return handles for the following paths:

- module paths
- wire paths with connections to terminals

Usage example

The following C-language code fragment shows how to fetch *min:typ:max* delays for the inter-module path referenced by `intermod_path`. Assume the Verilog-HDL source description contains only one delay per transition.

```
#include "acc_user.h"
fetch_mintypmax_delays( port_output, port_input )
handle port_output, port_input;
{
                              •
                              •
                              •
   handle  intermod_path;
   double  delay_array[9];
                              •
                              •
                              •
   acc_configure( accMinTypMaxDelays, "true" );
                              •
                              •
                              •
   intermod_path = acc_handle_path( port_output, port_input );
   acc_fetch_delays( intermod_path, delay_array );
                              •
                              •
                              •
}
```

acc_handle_path returns a handle to a wire path that represents the connection from an output (or inout) port to an input (or inout) port

acc_fetch_delays places the following values in delay_array:

delay_array[0] =
delay_array[1] =
delay_array[2] =
min:typ:max rise delay

delay_array[3] =
delay_array[4] =
delay_array[5] =
min:typ:max fall delay

delay_array[6] =
delay_array[7] =
delay_array[8] =
min:typ:max turn-off delay

Figure 2-58: Fetching **min:typ:max** *delays for an inter-module path*

2.15.37 acc_handle_pathin

acc_handle_pathin			
function	returns handle for the first net connected to a module path source		
syntax	*pathin_handle* **= acc_handle_pathin(***path_handle***);**		
arguments	name	type	description
input:	**path_handle**	*handle*	handle of the module path
related routine	Use ***acc_next_modpath*** or ***acc_handle_modpath*** to get path_handle		

Usage example

The following example shows how the C-language routine `get_path_nets` uses `acc_handle_pathin` to find the net connected to the input of a path.

```
#include "acc_user.h"

get_path_nets(path_handle)
handle   path_handle;
{
  handle   pathin_handle, pathout_handle;

  pathin_handle = acc_handle_pathin( path_handle );
  pathout_handle = acc_handle_pathout( path_handle );
  io_printf( "Net connected to input is: %s\n",
             acc_fetch_name( pathin_handle ) );
  io_printf( "Net connected to output is: %s\n",
             acc_fetch_name( pathout_handle ) );
}
```

Figure 2-59: Finding the net connected to the input of a path

2.15.38 acc_handle_pathout

acc_handle_pathout			
function	returns handle for the first net connected to a module path destination		
syntax	*pathout_handle* **= acc_handle_pathout(***path_handle***)***;*		
arguments	name	type	description
input:	**path_handle**	*handle*	handle of the module path
related routine	Use ***acc_next_modpath*** or ***acc_handle_modpath*** to get path_handle		

Usage example

The following example shows how the C-language routine `get_path_nets` uses `acc_handle_pathout` to find the net connected to the output of a path.

```
#include "acc_user.h"

get_path_nets(path_handle)
handle   path_handle;
{
  handle   pathin_handle, pathout_handle;

  pathin_handle = acc_handle_pathin( path_handle );
  pathout_handle = acc_handle_pathout( path_handle );
  io_printf( "Net connected to input is: %s\n",
             acc_fetch_name( pathin_handle ) );
  io_printf( "Net connected to output is: %s\n",
             acc_fetch_name( pathout_handle ) );
}
```

Figure 2-60: Finding the net connected to the output of a path

2.15.39
acc_handle_port

acc_handle_port			
function	returns handle for a module port		
syntax	*port_handle* **= acc_handle_port(***module_handle, port_index***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of a module
	port_index	*integer*	index of the desired port

What is an index?

The index of a ***port*** is its position in a ***module*** definition in your source description. Indexes are integers that start at zero and increase from left to right. The following table shows how port indexes are derived:

For:	*Indexes are:*
implicit ports: `module A( q,a,b);`	***0*** for port ***q*** ***1*** for port ***a*** ***2*** for port ***b***

explicit ports:
`module top;`
`reg  ra,rb;`
`wire wq;`
`explicit_port_mod epm1( .b(rb),.a(ra),.q(wq) );`
`initial`
`$$systemtask;`
`endmodule`
 	
`module m1;`
`explicit_port_mod( q,a,b );`
`input  a,b;`
`output q;`
`nand( q,a,b );`
`endmodule`
	0 for explicit port ***epm1.q***
	1 for explicit port ***epm1.a***
	2 for explicit port ***epm1.b***

Table 2-65: Deriving port indexes

Usage example

The following example shows how the C-language routine `is_port_output` uses `acc_handle_port` to identify whether a particular module port is an output.

```
#include "acc_user.h"

bool  is_port_output(module_handle,port_index)
handle   module_handle;
int      port_index;
{
   handle   port_handle;
   int      direction;

   /*check port direction*/
   port_handle = acc_handle_port( module_handle, port_index);
   direction = acc_fetch_direction( port_handle );
   if( direction == accOutput || direction == accInout )
      return( true );
   else
      return( false );
}
```

Figure 2-61: Identifying if a module port is an output

2.15.40 acc_handle_scope

acc_handle_scope			
function	returns a handle to the scope which contains an object		
syntax	*scope_handle* **= acc_handle_scope (** *object_handle***)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle to an object

Description

This function returns the handle to the scope of an object. The scope can be either a module, task, function, named parallel block, or named sequential block.

Example

The following example displays the scope which contains an object.

```
get_scope(obj)
handle obj;
{
   handle scope = acc_handle_scope (obj);

   io_printf (“Scope %s contains object %s\n”,
      acc_fetch_fullname(scope), acc_fetch_name(obj);
}
```

Figure 2-62: Displaying the scope of an object

2.15.41 acc_handle_simulated_net

acc_handle_simulated_net			
function	returns the simulated net associated with the collapsed net passed as an argument		
syntax	*simulated_net_handle* **= acc_handle_simulated_net(***collapsed_net_handle***);**		
	name	type	description
return value	**simulated_net_handle**	*handle*	handle of the simulated net
input	**collapsed_net_handle**	*handle*	handle of a collapsed net

The access routine `acc_handle_simulated_net` returns a handle to the simulated net that is associated with a specified collapsed net. Note that if you pass a handle to a net that is not collapsed, `acc_handle_simulated_net` returns a handle to that same net.

Usage example

In the following example, the routine `display_simulated_nets` uses `acc_handle_simulated_net` to find all simulated nets within a particular scope. The routine then displays each collapsed net, along with the simulated net.

```
#include "acc_user.h"

void  display_simulated_nets()
{
   handle   mod_handle;
   handle   simulated_net_handle;
   handle   net_handle;

   /*reset environment for access routines*/
   acc_initialize();
   acc_configure( accDevelopmentVersion, "1.6" );

   /*get scope-first argument passed to user-defined system task*/
   /* associated with this routine*/
   mod_handle = acc_handle_tfarg(1);
   io_printf( "In module %s:\n",acc_fetch_fullname(mod_handle) );
   net_handle = null;

   /*display name of each collapsed net and its net of origin*/
   while( net_handle = acc_next_net(mod_handle,net_handle) )
   {
      if (acc_object_of_type( net_handle,accCollapsedNet) )
      {
         simulated_net_handle = acc_handle_simulated_net(net_handle);
         io_printf("   net %s was collapsed onto net %s\n",
                   acc_fetch_name( net_handle ),
                   acc_fetch_name( simulated_net_handle) );
      }
   }
}
```

Figure 2-63: Displaying collapsed nets in a particular scope

Note the use of the property `accCollapsedNet` to determine whether `net_handle` has been collapsed onto another net.

2.15.42 acc_handle_tchk

acc_handle_tchk			
function	returns handle for the specified timing check of a module (or cell)		
syntax	*timing_check_handle* **= acc_handle_tchk(***module_handle, timing_check_type, first_arg_conn_name***,** *first_arg_edge_type, second_arg_conn_name, second_arg_edge_type, first_arg_conn_handle, second_arg_conn_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module (or cell)
	timing_check_type	*one of these predefined constants*: accHold, accSetup, accNochange, accSkew, accPeriod, accWidth, accRecovery	type of timing check
	first_arg_conn_name	*character string pointer:* quoted string literal *or* character pointer variable	name of the net connected to first timing check argument
	first_arg_edge_type	*one of these predefined constants:* accNegedge, accNoedge, accPosedge ***or*** *list of these constants, separated by* **+***:* accEdge01 accEdge0x accEdgex1 ***or*** *list of these constants, separated by* **+***:* accEdge10 accEdge1x accEdgex0	edge of the net connected to first timing check argument
	second_arg_conn_name (depends on type of timing check)	*(same as for **first_arg_conn_name**)*	name of the net connected to second timing check argument
	second_arg_edge_type (depends on type of timing check)	*(same as for **first_arg_edge_type**)*	edge of the net connected to second timing check argument
	first_arg_conn_handle (required if accEnableArgs is set and first_arg_conn_name is null)	*handle*	handle of the net connected to first timing check argument
	second_arg_conn_handle (required if accEnableArgs is set and second_arg_conn_name is null)	*handle*	handle of the net connected to second timing check argument
related routine	Use ***acc_configure(accEnableArgs, "acc_handle_tchk")*** to use **first_arg_conn_handle** and **second_arg_conn_handle** arguments		

How the routine works

The following table shows how `acc_handle_tchk` works:

If:	***acc_handle_tchk:***
tchk_type is ***accWidth*** or ***accPeriod***	ignores ***second_arg_conn_name***, ***second_arg_edge_type*** and ***second_arg_conn_handle*** *(you can drop any one of these three optional arguments as long as it does not precede any required arguments; otherwise, the optional argument must be provided)*
you call: *acc_configure(accEnableArgs, "acc_handle_tchk")* and you pass: **first_arg_conn_name or** **second_arg_conn_name** as a null pointer	uses the associated handle argument, ***first_arg_conn_handle*** or ***second_arg_conn_handle***, instead of the name argument
you do *not* call: *acc_configure(accEnableArgs, "acc_handle_tchk")*	always ignores both handle arguments and uses the name arguments instead *(the handle arguments can be dropped)*

Table 2-66: How ***acc_handle_tchk*** *works*

Edge group constants represent groups of transitions

Access routines recognize predefined edge group constants that represent *groups* of transitions among ***0***, ***1*** and ***x*** edge values that trigger timing checks, as described in Table 2-67.

Edge group constant	*Description of edge trigger*
`accNegedge`	any negative transition: **1** to **0** **1** to **x** **x** to **0**
`accNoedge`	any transition: **0** to **1** **0** to **x** **x** to **1** **1** to **0** **1** to **x** **x** to **0**
`accPosedge`	any positive transition**:** **0** to **1** **0** to **x** **x** to **1**

Table 2-67: Edge group constants

Edge specific constants represent individual transitions

Access routines recognize predefined edge specific constants that represent *individual* transitions among ***0***, ***1*** and ***x*** edge values that trigger timing checks, as described in Table 2-68.

Edge specific constant	*Description of edge trigger*
`accEdge01`	transition from **0** to **1**
`accEdge0x`	transition from **0** to **x**
`accEdgex1`	transition from **x** to **1**
`accEdge10`	transition from **1** to **0**
`accEdge1x`	transition from **1** to **x**
`accEdgex0`	transition from **x** to **0**

Table 2-68: Edge specific constants

Edge sums

Edge sums are lists of edge specific constants connected by plus (+) signs. They represent the Verilog-HDL edge control specifiers used by particular timing checks. Figure 2-64 shows the call to `acc_handle_tchk` that accesses a `$width` timing check containing edge control specifiers.

This access routine call:	*accesses this timing check:*
`acc_handle_tchk( cell_handle, accWidth, "clk", `**`accEdge10+accEdgex0`**` );`	`$width( `**`edge[10,x0]`**`clk, limit );`

edge sum
models
edge control specifier

Figure 2-64: Edge sums model edge control specifiers

Optional arguments

When an optional argument is required for a particular call to `acc_handle_tchk`, you must set the configuration parameter `accEnableArgs` by calling `acc_configure` as follows:

```
acc_configure(accEnableArgs, "acc_handle_tchk");
```

If `accEnableArgs` is not set for `acc_handle_tchk`, the routine always ignores its optional arguments.

When an optional argument is *not* required for a particular call to `acc_handle_tchk`, the argument can be dropped as long as it does not precede any required arguments. However, when an optional argument does precede one or more required arguments, it must be supplied even if it is ignored by the access routine. In this case, you can specify the argument as a null value.

Figure 2-65 shows an example in which optional arguments to `acc_handle_tchk` can be dropped; Figure 2-66 shows an example in which optional arguments must be specified.

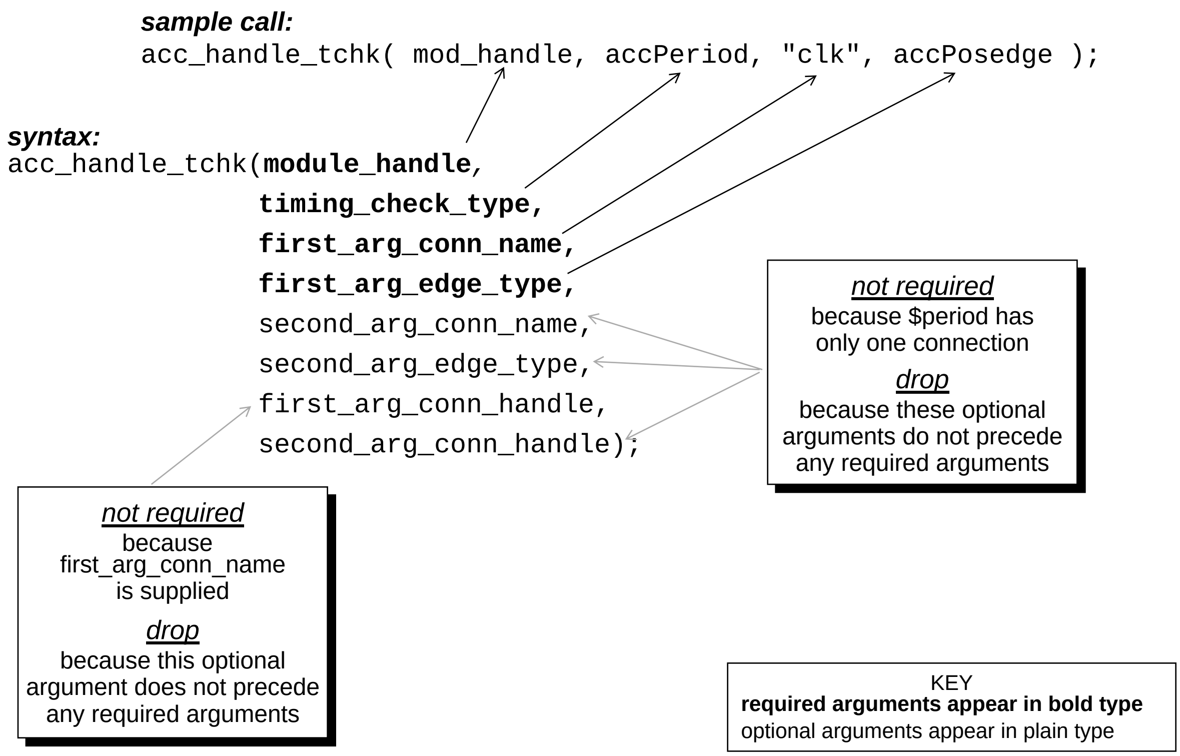


*Figure 2-65: Example showing optional arguments that may be dropped in **acc_handle_tchk***

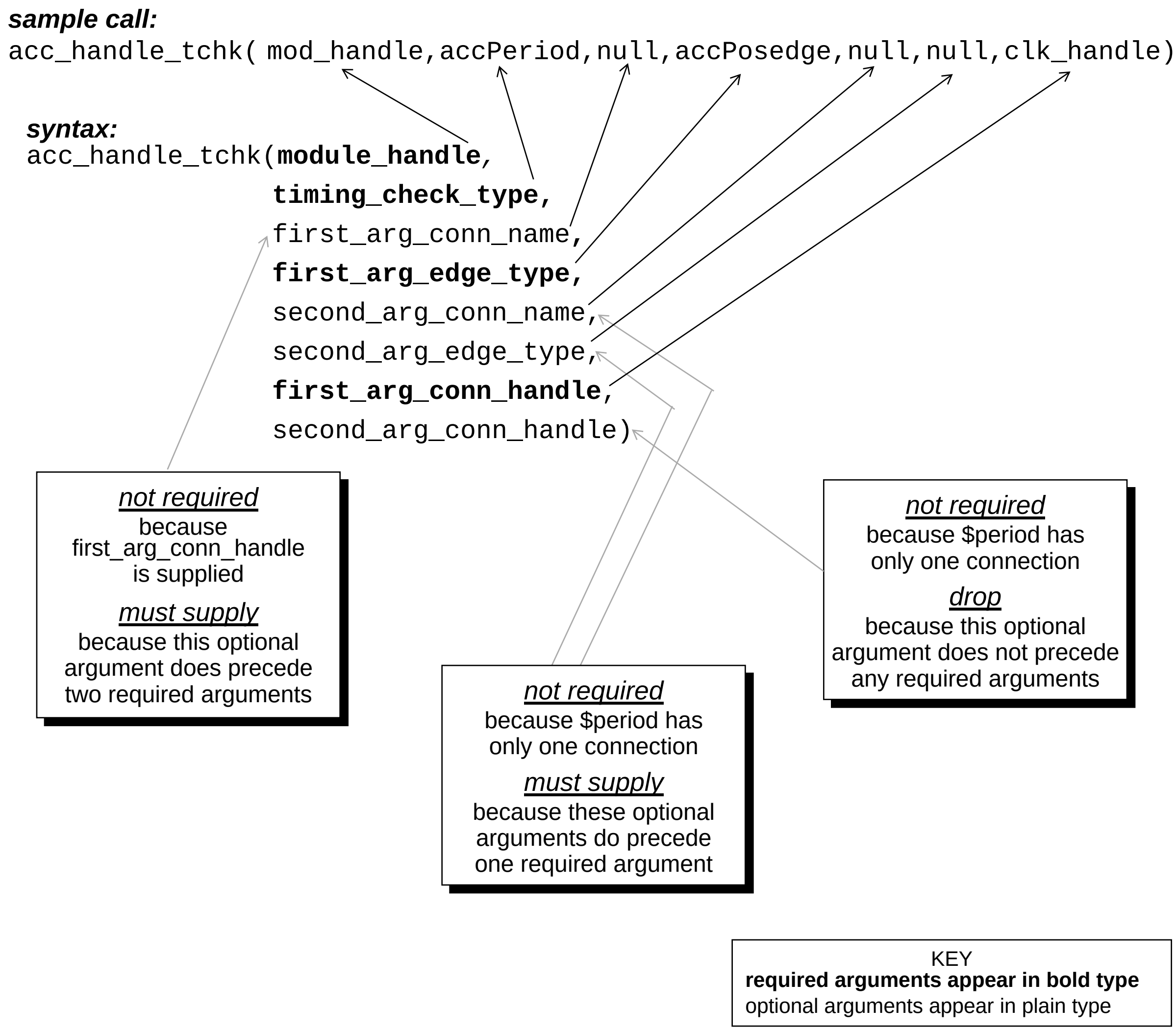


*Figure 2-66: Example showing optional arguments that may and may **not** be dropped in **acc_handle_tchk***

Usage example

The following example shows how the C-language routine `get_ps_tchks` uses `acc_handle_tchk` to identify all cells in a module that contain either or both of the following timing checks:

- a period timing check triggered by a positive edge on the clock signal `clk`
- a setup timing check triggered on signal `d` by any transition and on signal `clk` by either of these clock edge transitions: ***1 to 0*** or ***x to 0***

```
#include "acc_user.h"

get_ps_tchks()
{
  handle   module_handle, port_handle, net_handle, cell_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg( 1 );
  io_printf( "Module is %s\n", acc_fetch_name(module_handle) );

  /*scan all cells in module for:                              */
  /*   period timing checks triggered by a positive clock edge    */
  /*   setup timing checks triggered by 1->0 and x->0 clock edges*/
  cell_handle = null;
  while( cell_handle = acc_next_cell(module_handle, cell_handle) )
  {
     if( acc_handle_tchk( cell_handle,accPeriod,"clk",accPosedge ) )
      io_printf("positive clock edge triggers period check in cell %s\n",
                  acc_fetch_fullname( cell_handle ) );
     if( acc_handle_tchk( cell_handle,accSetup,"d",accNoedge,
                               "clk",accEdge10+accEdgex0 ) )
      io_printf("10 and x0 edges trigger setup check in cell %s\n",
                  acc_fetch_fullname( cell_handle ) );
  }
  acc_close();
}
```

Figure 2-67: Identifying all cells in a module that contain particular period and setup timing checks

There are several constructs to note in this example:

- Both calls to `acc_handle_tchk` supply *names* for all relevant connections; therefore, the arguments `first_arg_conn_handle` and `second_arg_conn_handle` are not supplied.
- `acc_handle_tchk` ignores `second_arg_conn_name`, `second_arg_edge_type` and `second_arg_conn_handle` for period timing checks; therefore, these arguments are not supplied.

2.15.43 acc_handle_tchkarg1

acc_handle_tchkarg1			
function	returns handle for the net connected to the first argument of a timing check		
syntax	*first_arg_conn_handle* **= acc_handle_tchkarg1*(****tchk_handle**)*;*		
arguments	name	type	description
input:	**tchk_handle**	*handle*	handle of a timing check
related routine	Use ***acc_next_tchk*** or ***acc_handle_tchk*** to obtain handles for timing checks		

Usage example

The following example presents a C-language routine, `show_check_nets`, that uses `acc_handle_tchkarg1` to obtain the net connected to the first argument of each setup timing check in each cell under a module.

```
#include "acc_user.h"

show_check_nets()
{
   handle  module_handle,cell_handle;
   handle  tchk_handle,tchkarg1_handle,tchkarg2_handle;
   int     tchk_type,counter;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );
   io_printf("module is %s\n", acc_fetch_fullname( module_handle ) );

   /*scan all cells in module for timing checks*/
   cell_handle = null;
   while ( cell_handle = acc_next_cell( module_handle, cell_handle ) )
   {
      io_printf( "cell is: %s\n", acc_fetch_fullname( cell_handle ) );
      counter = 0;
      while ( tchk_handle = acc_next_tchk( cell_handle, tchk_handle) )
      {
        /*get nets connected to timing check arguments*/
        tchk_type = acc_fetch_type( tchk_handle );
        if ( tchk_type == accSetup )
        {
          counter++;
          io_printf("    for setup check #%d:\n",counter);
          tchkarg1_handle = acc_handle_tchkarg1( tchk_handle );
          tchkarg2_handle = acc_handle_tchkarg2( tchk_handle );
          io_printf("       data net is %s\n        reference net is %s\n",
                       acc_fetch_name( tchkarg1_handle ),
                       acc_fetch_name( tchkarg2_handle ) );
        }
      }
   }
   acc_close();
}
```

Figure 2-68: Obtaining the nets connected to first arguments of setup timing checks in cells

2.15.44 acc_handle_tchkarg2

acc_handle_tchkarg2			
function	returns handle for the net connected to the second argument of a timing check		
syntax	*second_arg_conn_handle* **= acc_handle_tchkarg*2(*** *tchk_handle* ***)*** *;*		
arguments	name	type	description
input:	**tchk_handle**	*handle*	handle of a timing check
related routine	Use ***acc_next_tchk*** or ***acc_handle_tchk*** to obtain handles for timing checks		

Usage example

The following C-language routine, `show_check_nets`, uses `acc_handle_tchkarg2` to obtain the net connected to the second argument of each setup timing check in each cell under a module. Note that `acc_handle_tchkarg2` returns null if you pass it a handle to a timing check that requires only one net argument.

```
#include "acc_user.h"

show_check_nets()
{
   handle  module_handle,cell_handle;
   handle  tchk_handle,tchkarg1_handle,tchkarg2_handle;
   int     tchk_type,counter;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );
   io_printf("module is %s\n", acc_fetch_fullname( module_handle ) );

   /*scan all cells in module for timing checks*/
   cell_handle = null;
   while ( cell_handle = acc_next_cell( module_handle, cell_handle ) )
   {
      io_printf( "cell is: %s\n", acc_fetch_fullname( cell_handle ) );
      counter = 0;
      while ( tchk_handle = acc_next_tchk( cell_handle, tchk_handle) )
      {
        /*get nets connected to timing check arguments*/
        tchk_type = acc_fetch_type( tchk_handle );
        if ( tchk_type == accSetup )
        {
          counter++;
          io_printf("    for setup check #%d:\n",counter);
          tchkarg1_handle = acc_handle_tchkarg1( tchk_handle );
          tchkarg2_handle = acc_handle_tchkarg2( tchk_handle );
          io_printf("       data net is %s\n        reference net is %s\n",
                       acc_fetch_name( tchkarg1_handle ),
                       acc_fetch_name( tchkarg2_handle ) );
        }
      }
   }
   acc_close();
}
```

Figure 2-69: Obtaining the nets connected to second arguments of setup timing checks in cells

2.15.45 acc_handle_terminal

acc_handle_terminal			
function	returns handle for a primitive_terminal		
syntax	*term_handle* **= acc_handle_terminal(***primitive_handle, terminal_index***)***;*		
arguments	name	type	description
inputs:	**primitive_handle**	*handle*	handle of a primitive
	terminal_index	*integer*	index of the desired terminal

What is an index?

The index of a ***terminal*** is its position in a ***gate, switch***, or ***UDP*** declaration. Indexes are integers that start at zero and increase from left to right.

The following table shows how terminal indexes are derived:

For:	*Indexes are:*
`nand g1(out,in1,in2);`	***0*** for terminal ***out*** ***1*** for terminal ***in1*** ***2*** for terminal ***in2***

Table 2-69: Deriving terminal indexes

Usage example

The following example shows how the C-language routine `print_terminal_net` uses `acc_handle_terminal` to identify the name of a net connected to a primitive terminal.

```
#include "acc_user.h"

print_terminal_net(gate_handle, term_index)
handle   gate_handle;
int      term_index;
{
   handle   term_handle;
   term_handle = acc_handle_terminal( gate_handle, term_index );
   io_printf( "%s terminal net #%d is %s\n",
                  acc_fetch_name(gate_handle),
                  term_index,
                  acc_fetch_name( acc_handle_conn(term_handle) ) );
}
```

Figure 2-70: Identifying the name of a net connected to a primitive terminal

2.15.46 acc_handle_tfarg

acc_handle_tfarg			
function	returns handle for the specified argument of the system task or function associated (through the PLI mechanism) with your C-language routine		
syntax	*argument_handle* **= acc_handle_tfarg(***argument_number***)***;*		
arguments	name	type	description
input:	**argument_number**	*integer*	value that references an object—passed as a variable in the system task or function call—by its position in the argument list

Types of arguments

The routine retrieves handles to the following types of system task or function arguments:

- nets
- module instances
- primitives

How arguments are numbered

Argument numbers start at ***1*** and increase from left to right in the order that they appear in the system task or function call.

Passing arguments to system tasks and functions

For `acc_handle_tfarg` to retrieve handles to arguments, you must pass these arguments to system tasks or functions in the following ways:

- For ***primitives***, pass the lowest-level names or full hierarchical names as quoted strings.
- For ***module instances*** and ***nets***, pass the Verilog-HDL identifiers without quotation marks or pass the full hierarchical names as quoted strings.

Following is a sample call to a system task called `$mytask` that takes two arguments—a module instance and a gate instance:

```
$mytask(top.mod1,"top.mod1.nand6")'
```

Note that the module instance is passed as a Verilog-HDL identifier without quotation marks, while the gate instance name is passed as a quoted string.

How the routine works

When:	*acc_handle_tfarg:*
the system task or function argument is a Verilog-HDL identifier for a net or module	returns a handle to the net or module
the system task or function argument is a quoted string name of any object	calls ***acc_handle_object*** with the string as an argument and, if found, returns a handle to the object

Table 2-70: How ***acc_handle_tfarg*** *works*

Identifying collapsed nets

The access routine `acc_handle_tfarg` can be used to return a handle to the collapsed net passed as an argument to a user-defined system task or function. This enables Value Change Link (VCL) applications such as waveform displays to show the name of a monitored collapsed net along with its value changes.

Usage example

The following example shows a C-language routine called `new_timing` that has the following characteristics:

- changes the rise and fall delays of a gate
- takes three arguments—the first is a Verilog-HDL gate and the others are double-precision floating-point constants representing rise and fall delay values
- links through the PLI mechanism with a Verilog-HDL system task called `$timing_task`

To invoke the routine `new_timing`, you must first call the system task `$timing_task` from your Verilog-HDL source description or interactively from the command line. Here is a sample call:

```
$timing_task( "top.g12", 8.4, 9.2 );
```

When Verilog encounters this call, it executes `new_timing`, which contains the following code:

```
#include "acc_user.h"

new_timing()
{
  handle   gate_handle;
  double   new_rise, new_fall;

  /*initialize and configure access routines*/
  acc_initialize();
  acc_configure( accDevelopmentVersion, "1.6a" );
  acc_configure(accToHiZDelay, "max");

  /*get handle to gate*/
  gate_handle = acc_handle_tfarg( 1 );     top.g12

  /* get new delay values */                8.4
  new_rise = tf_getrealp( 2 );
  new_fall = tf_getrealp( 3 );              9.2

  /*place new delays on the gate*/
  acc_replace_delays( gate_handle,new_rise,new_fall);

  /* report action */
  io_printf("Primitive %s has new delays %d %d\n",
             acc_fetch_fullname( gate_handle ),
             new_rise, new_fall );

  acc_close();
}
```

Figure 2-71: Changing the rise and fall delays on a gate

A handle to the first argument—the gate `top.g12`—is retrieved using `acc_handle_tfarg`, while the other two arguments—the delay values—are retrieved using `tf_getrealp`.

2.15.47 acc_initialize

acc_initialize	
function	initializes the environment for access routines
syntax	**acc_initialize*()*;**
arguments	*none*
related routines	Call ***acc_close*** before exiting your C-language routine

The initialization functions

The routine `acc_initialize` performs the following functions:

- initializes all configuration parameters to their default values
- allocates memory for string handling and other internal uses

When to call acc_initialize

You must call `acc_initialize` in your C-language routine before invoking any other access routines.

Avoiding application interference

Potentially, multiple PLI applications running in the same simulation session can interfere with each other because they share the same set of configuration parameters. To guard against application interference, both `acc_initialize` and `acc_close` reset any configuration parameters that have changed from their default values.

Usage example

The following example presents a C-language routine, `display_inputs`, that uses `acc_initialize` to initialize the environment for access routines.

```
#include "acc_user.h"

display_inputs()
{
  handle   module_handle,port_handle;
  int      direction;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg( 1 );
  port_handle = null;
  while (port_handle = acc_next_port( module_handle, port_handle) )
  {
     /*determine if port is an input*/
     direction = acc_fetch_direction( port_handle );
     /*give the index of each input port*/
     if ( direction == accInput )
        io_printf( "Port #%d of module %s is an input",
                   acc_fetch_index( port_handle ),
                   acc_fetch_fullname( module_handle) );
  }
  acc_close();
}
```

Figure 2-72: Displaying the input ports of a module

2.15.48 acc_next

acc_next			
function	within a scope, returns the next object of each type specified in `object_type_array`		
syntax	*object_handle* **= acc_next(***object_type_array, module_handle, object_handle***);**		
	name	type	description
return value	**object_handle**	*handle*	handle of the object found
inputs	**object_type_array**	*integer array*	array containing one or more predefined integer constants that represent the types of objects desired; the last element must be 0
	module_handle	*handle*	handle of the desired scope
	object_handle	*handle*	handle of the object found

The access routine `acc_next` allows you to scan one or more types of objects within a scope. This routine performs a more general function than the object-specific NEXT routines—such as `acc_next_net` and `acc_next_primitive`—which scan only one type of object within a scope.

How to set up the array of object types

Declare the array of object types as a static, integer array inside the C routine that calls `acc_next`. The array can contain any number and combination of the predefined integer constants listed in Table 2-71 through Table 2-74, and must be terminated by a `0`. The predefined integer constants specify the types and fulltypes of objects that `acc_next` will return.

The following C-language statement shows how to declare an array of object types called `net_reg_list`:

```
static int net_reg_list[3] = {accNet,accRegister,0};
```

If you pass this array to `acc_next`, the access routine will return handles to nets and registers within the specified scope.

Order of objects returned

The routine `acc_next` returns objects in an arbitrary order.

Type of object	*Predefined integer constant*
integer variable	`accIntegerVar`
module	`accModule`
named event	`accNamedEvent`
net	`accNet`
primitive	`accPrimitive`
real variable	`accRealVar`
register	`accRegister`
time variable	`accTimeVar`

*Table 2-71: Types supported by **acc_next***

Net fulltype	*Predefined integer constant*
net of type `wire`	`accWire`
net of type `tri`	`accTri`
wired-AND nets	`accWand` `accTriand`
wired-OR nets	`accWor` `accTrior`
pulldown, pullup nets	`accTri0` `accTri1`
nets that model power supplies	`accSupply0` `accSupply1`
net that stores a value	`accTrireg`

Table 2-72: Net fulltypes

Module fulltype	*Predefined integer constant*
module instance	`accModuleInstance`
top-level module	`accTopModule`
cell instance	`accCellInstance`

Table 2-73: Module fulltypes

Primitive fulltype	*Predefined integer constant*

gates with one or more inputs, one output	`accAndGate`
	`accNandGate`
	`accNorGate`
	`accOrGate`
	`accXnorGate`
	`accXorGate`

gates with one input, one or more outputs	`accBufGate` `accNotGate`

gates that model tri-state drivers	`accBufif0`
	`accBufif1`
	`accNotif0`
	`accNotif1`

MOS gates	`accNmosGate`
	`accPmosGate`
	`accRnmosGate`
	`accRpmosGate`

CMOS gates	`accCmosGate` `accRcmosGate`

bidirectional pass gates	`accRtranGate`
	`accRtranif0Gate`
	`accRtranif1Gate`
	`accTranGate`
	`accTranif0Gate`
	`accTranif1Gate`

pulldown, pullup gates	`accPulldownGate` `accPullUpGate`
combinational user-defined primitive	`accCombPrim`
sequential user-defined primitive	`accSeqPrim`

Table 2-74: Primitive fulltypes

Usage example

The C-language routine in the following example uses `acc_next` to find all nets and registers in a module. The routine then displays the names of these nets and registers.

```
#include "acc_user.h"

void display_nets_and_registers()
{
  static   int   net_reg_list[3] = {accNet,accRegister,0};
  handle   mod_handle, obj_handle;

  /*reset environment for access routines*/
  acc_initialize();
  acc_configure( accDevelopmentVersion, "1.6" );

  /*get handle for module-first argument passed to*/
  /* user-defined system task associated with this routine*/
  mod_handle = acc_handle_tfarg( 1 );
  io_printf( "Module %s contains these nets and registers:\n",
             acc_fetch_fullname( mod_handle ) );

  /*display names of all nets and registers in the module*/
  obj_handle = null;
  while( obj_handle = acc_next(net_reg_list,mod_handle,obj_handle) )
     io_printf( "   %s\n", acc_fetch_name( obj_handle ) );
}
```

Figure 2-73: Displaying the names of all nets and registers in a module

2.15.49 acc_next_bit

acc_next_bit			
function	returns the handles of each bit in an expanded vector port or expanded vector net		
syntax	**acc_next_bit**(*port_or_net_handle, bit_handle);*		
arguments	name	type	description
	port_or_net_handle	*handle*	handle of a port
	bit_handle	*handle*	handle of a bit
related routines	Use ***acc_next_port*** to return the next port of a module Use ***acc_handle_port*** to return the handle for a module port		

The access routine `acc_next_bit` accesses all of the bits of a vector port or vector net. This routine retrieves the handles to each bit of a port or net. You can pass these handles to access routines that insert, replace, or return MIPD values.

Vector versus scalar objects

When the object associated with `port_handle` or `net_handle` is a *vector* object, the first call to `acc_next_bit` returns the handle to the most significant bit of the object. Subsequent calls return the handles to the remaining bits down to the least significant bit. The call after the return of the handle to the least significant bit returns null.

When the object associated with `port_handle` or `net_handle` is a *scalar* object, `acc_next_bit` treats the object like a vector of size 1. The first call returns the handle to the scalar object; the next call returns null.

Usage examples

The following example C-subroutine, `display_port_bits`, uses `acc_next_bit` to display the low hierarchical connection of each bit of a port.

```
#include "acc_user.h"
display_port_bits(module_handle, port_number)
handle   module_handle;
int      port_number;
{
  handle   port_handle, bit_handle;

  /* get handle for port */
  port_handle = acc_handle_port(module_handle, port_number);

  /* display port number and module instance name */
  io_printf("Port %d of module %s contains the following bits: \n",
            port_number, acc_fetch_fullname(module_handle));

  /* display low hierarchical connection of each bit */
  bit_handle = null;
  while( bit_handle = acc_next_bit(port_handle, bit_handle))
    io_printf( "  %s\n",acc_fetch_fullname(bit_handle) );
}
```

Figure 2-74: Displaying the low hierarchical connection of each bit of a port

The following example C subroutine, `monitor_bits`, requests that VCL monitor the logic value of each bit of an expanded vector net.

```
#include "acc_user.h"
void consumer_routine();
void monitor_bits()
{
  handle  bit_handle, net_handle, mod_handle;
  /*reset environment for access routines*/
  acc_initialize();
  acc_configure( accDevelopmentVersion, "1.5c" );

  /*get handle for module-first argument passed*/
  /* to user-defined system task associated with*/
  /* this routine*/
  mod_handle = acc_handle_tfarg( 1 );
  /*check all nets in the module*/
  net_handle = null;
  while( net_handle = acc_next_net( mod_handle, net_handle ) )
  {
     /*request that VCL to monitor each bit of expanded vector nets*/
     if( acc_object_of_type( net_handle,accExpandedVector ) )
     {
        bit_handle = null;
        while( bit_handle = acc_next_bit( net_handle, bit_handle ) )
           acc_vcl_add( bit_handle,consumer_routine,null,
                        vcl_verilog_logic);
     }
   }
}
```

Figure 2-75: Monitoring the logic value of each bit of an expanded vector net

2.15.50
acc_next_cell

acc_next_cell			
function	returns the next cell instance within the region that includes the entire hierarchy below a module		
syntax	*cell_handle* **= acc_next_cell(** *module_handle, cell_handle* **);**		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the starting module
	cell_handle	*handle*	handle of the cell instance found

What is a cell instance?

A cell instance is a module instance that has either of these characteristics:

- it is defined in a library (and you have *not* specified `+nolibcell`)
- its definition appears between the compiler directives `‘celldefine` and `‘endcelldefine`

Nested cells

The routine `acc_next_cell` does *not* find cells that are instantiated inside other cells.

Usage example

The following C-language routine, `list_cells`, uses `acc_next_cell` to find all cell instances beginning at the level defined by `module_handle`.

```
#include "acc_user.h"

list_cells()
{
  handle   module_handle;
  handle   cell_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  module_handle = acc_handle_tfarg(1);
  io_printf( "%s contains the following cells:\n",
               acc_fetch_fullname( module_handle ) );

  /*display names of all cells in the module*/
  cell_handle = null;
  while(cell_handle = acc_next_cell(module_handle,cell_handle))
     io_printf("   %s\n",acc_fetch_fullname(cell_handle));

  acc_close();
}
```

Figure 2-76: Finding all cell instances under a module

2.15.51 acc_next_cell_load

acc_next_cell_load			
function	returns the next load on a net inside a cell		
syntax	*load_handle* **= acc_next_cell_load(** *net_handle, load_handle* **);**		
arguments	name	type	description
inputs:	**net_handle**	*handle*	handle of the net
	load_handle	*handle*	handle of the primitive input terminal found
related routines	***acc_next_load***		

What is a cell load?

A ***cell load*** is a primitive input terminal that connects to one of the input or inout ports of a cell instance driven by a net.

The difference between acc_next_cell_load and acc_next_load

The routine `acc_next_cell_load` returns only one primitive input terminal per cell input or inout port driven by the net—chosen arbitrarily.

The routine `acc_next_load` returns every primitive input terminal driven by the net, whether it is inside a cell or a module instance.

Figure 2-77 illustrates the difference between these two access routines. It presents a sample circuit in which `net1` drives primitive gates in `cell1`, `cell2` and `cell3`. The diagram shows that for this circuit, calling `acc_next_cell_load` inside a ***while*** loop returns *three* primitive input terminals, while calling `acc_next_load` from inside a ***while*** loop returns *four* primitive input terminals.

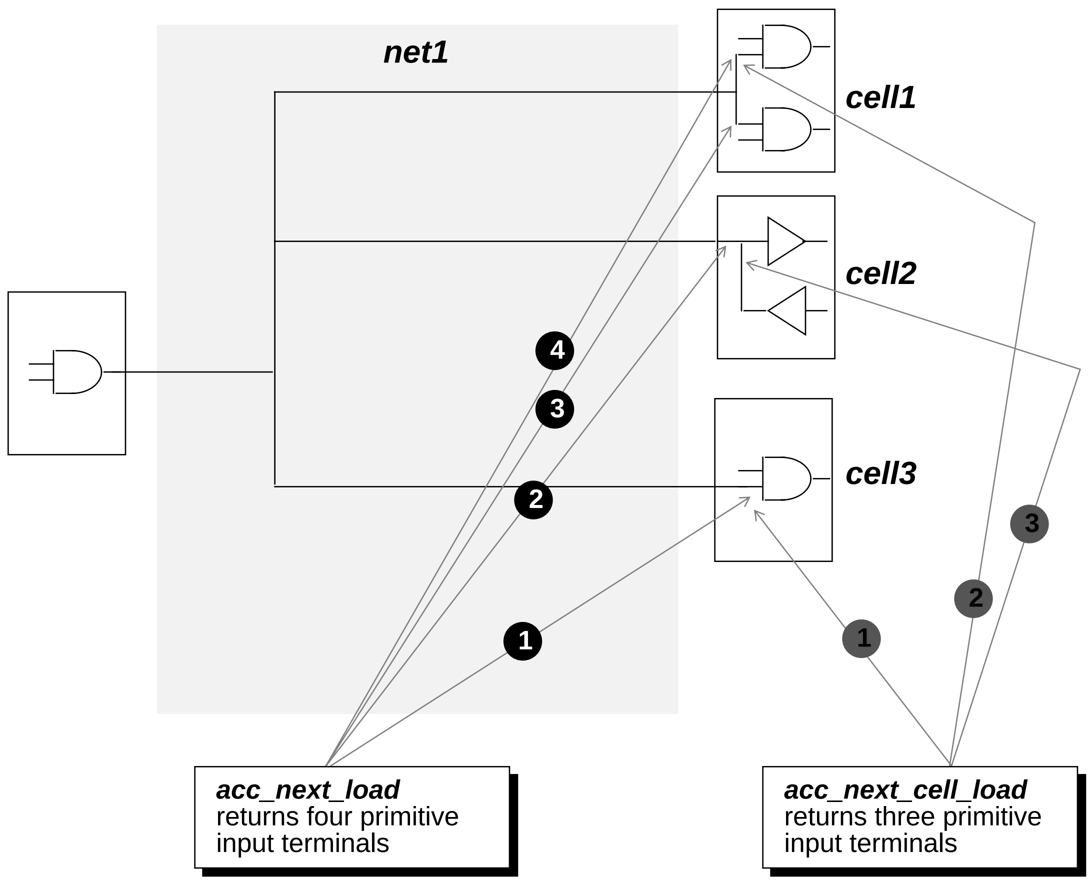


*Figure 2-77: The difference between **acc_next_cell_load** and **acc_next_load***

Usage example

The following C-language routine, `get_cell_loads`, uses `acc_next_cell_load` to find all cell loads on a net.

```
#include "acc_user.h"

get_cell_loads()
{
  handle   net_handle;
  handle   load_handle,load_net_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for net*/
  net_handle = acc_handle_tfarg(1);

  /*display names of all cell loads on the net*/
  load_handle = null;
  while(load_handle = acc_next_cell_load(net_handle,load_handle))
  {
     load_net_handle = acc_handle_conn(load_handle);
     io_printf("Cell load is connected to: %s\n",
                 acc_fetch_fullname(load_net_handle) );
  }
  acc_close();
}
```

Figure 2-78: Finding all cell loads on a net

2.15.52 acc_next_child

acc_next_child			
function	returns the next child of a module		
syntax	*child_handle* **= acc_next_child(** *module_handle, child_handle* **)**;		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	child_handle	*handle*	handle of a module instantiated inside the module associated with ***module_handle***

What is a child?

A ***child*** is a module instance that appears inside another module.

How the routine works

When:	*acc_next_child:*
the first argument, *module_handle*, is null	works exactly like *acc_next_topmod* to scan for top-level modules
the first argument, *module_handle*, is *not* null	scans for modules instantiated inside the module associated with *module_handle*

*Table 2-75: How **acc_next_child** works*

Collecting and counting top-level modules

The access routine `acc_next_topmod` does not work with `acc_collect` or `acc_count`. However, you can collect or count top-level modules by passing `acc_next_child` with a null reference object argument. Here is a sample call that collects top-level modules:

```
acc_collect( acc_next_child, null, &count );
```

Here is a sample call that counts top-level modules:

```
acc_count( acc_next_child, null );
```

Usage example

The following C-language routine, `print_children`, uses `acc_next_child` to display the names of all modules instantiated within the `module_handle` input argument.

```
#include "acc_user.h"

print_children(module_handle)
handle   module_handle;
{
  handle   child_handle;

  io_printf("Module %s contains the following module instances:\n",
             acc_fetch_fullname(module_handle) );

  /*display names of all children within the module*/
  child_handle = null;
  while(child_handle = acc_next_child(module_handle,child_handle))
     io_printf("    %s\n",acc_fetch_name(child_handle));
}
```

Figure 2-79: Displaying all children of a module

2.15.53 acc_next_driver

acc_next_driver			
function	returns the next primitive terminal that drives a net		
syntax	*driver_handle* **= acc_next_driver(***net_handle,driver_handle***)**;		
arguments	name	type	description
inputs:	**net_handle**	*handle*	handle of the net
	driver_handle	*handle*	handle of the driver found

Usage example

This example shows the C-language routine `print_drivers` that uses `acc_next_driver` to determine which primitives' terminals drive a net.

```
#include "acc_user.h"

print_drivers(net_handle)
handle   net_handle;
{
   handle   primitive_handle;
   handle   driver_handle;

   io_printf("Net %s is driven by the following primitives:\n",
             acc_fetch_fullname(net_handle) );

   /*get primitive that owns each terminal that drives the net*/
   driver_handle = null;
   while( driver_handle = acc_next_driver( net_handle, driver_handle ) )
   {
      primitive_handle = acc_handle_parent( driver_handle );
      io_printf( "   %s\n",
                 acc_fetch_fullname( primitive_handle ) );
   }
}
```

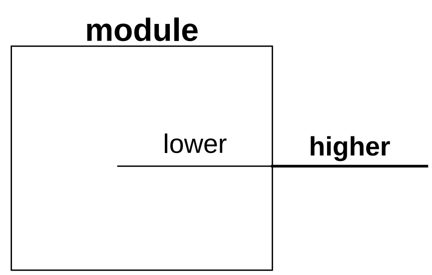
Figure 2-80: Determining which primitives' terminals drive a net

2.15.54 acc_next_hiconn

acc_next_hiconn			
function	returns the next hierarchically *higher* net connection to a port of a module		
syntax	*net_handle* **= acc_next_hiconn(***port_handle, net_handle***)**;		
arguments	name	type	description
inputs:	**port_handle**	*handle*	handle of the port
	net_handle	*handle*	handle of the net connection found
related routines	***acc_next_loconn***		

What is a hierarchically higher connection?

A **hierarchically higher connection** is the part of the net that appears outside the module, as shown in the diagram below:



Usage example

The following example presents a C-language routine, `display_connections`, that uses `acc_next_hiconn` to find and display the high net connections to a module port.

```
#include "acc_user.h"

display_connections(module_handle, port_handle)
handle module_handle, port_handle;
{
  handle   hiconn_net, loconn_net;

  /*get and display low connections*/
  io_printf("For module %s, port #%d low connections are:\n",
            acc_fetch_fullname(module_handle),
            acc_fetch_index(port_handle) );
  loconn_net = null;
  while ( loconn_net = acc_next_loconn( port_handle, loconn_net ) )
     io_printf("   %s\n", acc_fetch_fullname(loconn_net) );

  /*get and display high connections*/
  io_printf("For module %s, port #%d high connections are:\n",
            acc_fetch_fullname(module_handle),
            acc_fetch_index(port_handle) );
  hiconn_net = null;
  while ( hiconn_net = acc_next_hiconn( port_handle, hiconn_net ) )
     io_printf("   %s\n", acc_fetch_fullname(hiconn_net) );
}
```

Figure 2-81: Displaying the high net connections to a module port

2.15.55 acc_next_input

<table>
<tr><th colspan="4">acc_next_input</th></tr>
<tr><td>function</td><td colspan="3">returns a handle to the next input path terminal of the specified module path or datapath</td></tr>
<tr><td>syntax</td><td colspan="3">terminal_handle = acc_next_input (path_handle, terminal_handle);</td></tr>
<tr><td>arguments</td><td>name</td><td>type</td><td>description</td></tr>
<tr><td rowspan="2">input:</td><td>path_handle</td><td>handle</td><td>handle to a module path or data path</td></tr>
<tr><td>terminal_handle</td><td>handle</td><td>handle to an input path terminal</td></tr>
</table>

Description

The routine scans the inputs of a module path or sources of a data path and returns handles to the input path terminals. Routine `acc_handle_conn()` can then be applied to this path terminal to derive the net connected to the terminal.

Example

The example on the following page accepts a handle to a scalar net or a net bit-select, and a module path. The routine returns true if the net is connected to the input of the path.

Related routines

`acc_handle_conn()` : returns nets connected to path terminals
`acc_release_object()` : frees allocated memory

Usage and efficiency hints

The first time you call `acc_next_input`, PLI allocates the memory for a handle to an input and returns the handle to you. Each subsequent time you call the routine, PLI changes the handle to point to the next input. When you have completed scanning all inputs or an error condition arises, PLI returns a *null* handle and deallocates the memory for the handle.

If you do not scan all inputs, memory for the handle remains allocated. Call `acc_release_object` to deallocate the memory.

For paths with only one input, such as a data path, you will most likely call `acc_next_input` once, outside a loop. In this case, you should also call `acc_release_object` because PLI cannot return a handle and deallocate the memory for that handle in the same call.

Failure to deallocate memory for input and output handles may result in a significant waste of memory in a large application.

```
bool is_net_on_path_input(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
   handle port_in, port_conn, bit;

   /* scan path intput terminals */
   port_in = null;
   while (port_in = acc_next_input(path,  port_in))
   {
      /* retrieve net connected to path terminal */
      port_conn = acc_handle_conn (port_in);

      bit = null;
      if (acc_object_of_type (port_conn, accExpandedVector))
      {
         bit = null;
         while (bit = acc_next_bit (port_conn, bit))
            if (acc_compare_handles (bit, net))
               return (true);
      }
      else
         if (acc_compare_handles (bit, net))
            return (true);
   }

   return (false);
}
```

Figure 2-82: Determine if a net is connected to the input of a path

2.15.56 acc_next_load

acc_next_load			
function	returns the next primitive terminal driven by a net		
syntax	*load_handle* **= acc_next_load(***net_handle, load_handle***);**		
arguments	name	type	description
inputs:	**net_handle**	*handle*	handle of the net
	load_handle	*handle*	handle of the terminal found
related routines	***acc_next_cell_load***		

The difference between acc_next_load and acc_next_cell_load

The routine `acc_next_load` returns every primitive input terminal driven by the net, whether it is inside a cell or a module instance.

The routine `acc_next_cell_load` returns only one primitive input terminal per cell input or inout port driven by the net—chosen arbitrarily.

Figure 2-83 illustrates the difference between these two access routines. It presents a sample circuit in which `net1` drives primitive gates in `cell1`, `cell2`, `module1`, and `cell3`. The diagram shows that for this circuit, calling `acc_next_cell_load` inside a ***while*** loop returns *three* primitive input terminals, while calling `acc_next_load` from inside a ***while*** loop returns *four* primitive input terminals.

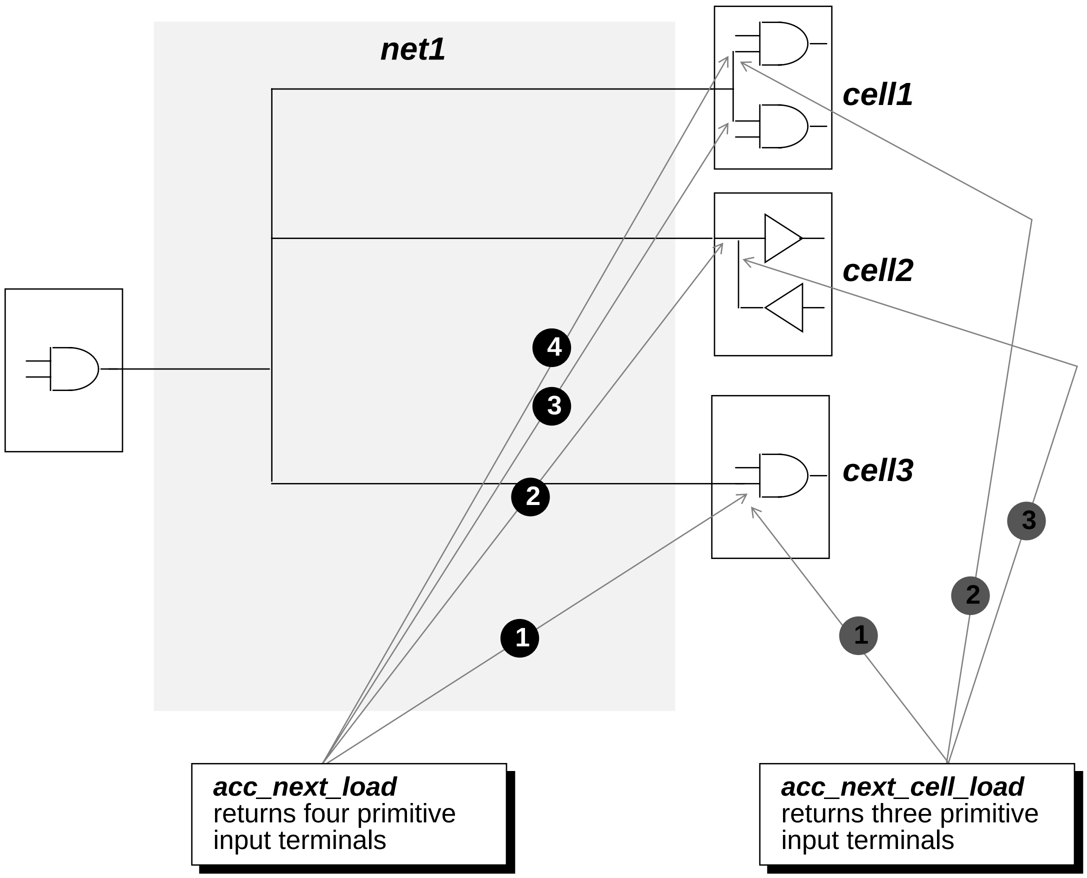


*Figure 2-83: The difference between **acc_next_cell_load** and **acc_next_load***

Usage example

This example shows how the C-language routine `get_loads` uses `acc_next_load` to find all terminals driven by a net.

```
#include "acc_user.h"

get_loads()
{
   handle   net_handle, load_handle, load_net_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for net*/
   net_handle = acc_handle_tfarg( 1 );
   io_printf( "Net %s is driven by:\n",acc_fetch_fullname( net_handle ) );

   /*get primitive that owns each terminal driven by the net*/
   load_handle = null;
   while( load_handle = acc_next_load( net_handle, load_handle ) )
   {
      load_net_handle = acc_handle_conn( load_handle );
      io_printf( "   %s ",
                 acc_fetch_fullname( load_net_handle ) );
   }
   acc_close();
}
```

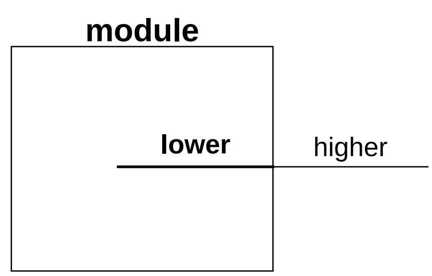
Figure 2-84: Finding all terminals driven by a net

2.15.57 acc_next_loconn

acc_next_loconn			
function	returns the next hierarchically *lower* net connection to a port of a module		
syntax	*net_handle* **= acc_next_loconn(***port_handle, net_handle***)***;*		
arguments	name	type	description
inputs:	**port_handle**	*handle*	handle of the port
	net_handle	*handle*	handle of the net connection found
related routines	***acc_next_hiconn***		

What is a hierarchically lower connection?

A **hierarchically lower connection** is the part of the net that appears inside the module, as shown in the diagram below:



Usage example

The following example presents a C-language routine, `display_connections`, that uses `acc_next_loconn` to find and display the low net connections to a module port.

```
#include "acc_user.h"

display_connections(module_handle, port_handle)
handle module_handle, port_handle;
{
  handle   hiconn_net, loconn_net;

  /*get and display low connections*/
  io_printf("For module %s, port #%d low connections are:\n",
             acc_fetch_fullname(module_handle),
             acc_fetch_index(port_handle) );
  loconn_net = null;
  while ( loconn_net = acc_next_loconn( port_handle, loconn_net ) )
     io_printf("   %s\n", acc_fetch_fullname(loconn_net) );

  /*get and display high connections*/
  io_printf("For module %s, port #%d high connections are:\n",
             acc_fetch_fullname(module_handle),
             acc_fetch_index(port_handle) );
  hiconn_net = null;
  while ( hiconn_net = acc_next_hiconn( port_handle, hiconn_net ) )
     io_printf("   %s\n", acc_fetch_fullname(hiconn_net) );
}
```

Figure 2-85: Displaying the low connections to a module port

2.15.58 acc_next_modpath

acc_next_modpath			
function	returns the next path of a module		
syntax	*load_handle* **= acc_next_modpath(***module_handle, path_handle***);**		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	path_handle	*handle*	handle of the path found

Usage example

The following example shows how the C-language routine `get_path_nets` uses `acc_next_modpath` to find the nets connected to the inputs and outputs of all paths across a module.

```
#include "acc_user.h"

get_path_nets(module_handle)
handle   module_handle;
{
  handle   pathin_handle, pathout_handle;
  handle   path_handle;

  /*scan all paths in the module and display nets connected to each*/
  /*  source and destination*/
  io_printf( "For module %s:\n",acc_fetch_fullname( module_handle ) );
  path_handle = null;
  while( path_handle = acc_next_modpath( module_handle, path_handle) )
  {
     io_printf("   path %s connections are:\n",
               acc_fetch_name( path_handle ) );
     pathin_handle = acc_handle_pathin( path_handle );
     pathout_handle = acc_handle_pathout( path_handle );
     io_printf( "net %s connected to input\n",
                acc_fetch_name( pathin_handle ) );
     io_printf( "net %s connected to output\n",
                acc_fetch_name( pathout_handle ) );
  }
}
```

Figure 2-86: Finding the nets connected to the inputs and outputs of all paths across a module

2.15.59 acc_next_net

acc_next_net			
function	returns the next net of a module		
syntax	*load_handle* **= acc_next_net(***module_handle, net_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	net_handle	*handle*	handle of the net found
related routines	Use ***acc_next_bit*** to return the next bit of a vector net		

Identifying vector nets

The access routine `acc_next_net` returns a handle to a vector net as a whole; it does not return a handle to each individual bit of a vector net. However, you can use `acc_next_bit` to retrieve a handle for each bit of an expanded vector net, as described in Section 2.15.49.

Usage example

The following C-language routine, `display_net_names`, uses `acc_next_net` to display the names of all nets in a module.

```
#include "acc_user.h"

display_net_names()
{
  handle   mod_handle, net_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*get handle for module*/
  mod_handle = acc_handle_tfarg( 1 );
  io_printf( "Module %s contains the following nets:\n",
             acc_fetch_fullname( mod_handle ) );

  /*display names of all nets in the module*/
  net_handle = null;
  while( net_handle = acc_next_net( mod_handle, net_handle ) )
     io_printf( "   %s\n", acc_fetch_name( net_handle ) );

  acc_close();
}
```

Figure 2-87: Displaying the names of all nets in a module

2.15.60
acc_next_output

acc_next_output			
function	returns a handle to the next output path terminal of the specified module path or datapath		
syntax	*terminal_handle* **= acc_next_output (** *path_handle, terminal_handle* **);**		
arguments	name	type	description
input:	**path_handle**	*handle*	handle to a module path or data path
	terminal_handle	*handle*	handle to an output path terminal

Description

The routine scans the outputs of a module path or sources of a data path and returns handles to the output path terminals. Routine `acc_handle_conn()` can then be applied to this path terminal to derive the net connected to the terminal.

Example

The example on the following page accepts a handle to a scalar net or a net bit-select, and a module path. The routine return true if the net is connected to the output of the path.

Related routines

`acc_handle_conn()` : returns nets connected to path terminals
`acc_release_object()` : frees allocated memory

Usage and efficiency hints

The first time you call `acc_next_output`, PLI allocates the memory for a handle to an output and returns the handle to you. Each subsequent time you call the routine, PLI changes the handle to point to the next output. When you have completed scanning all outputs or an error condition arises, PLI returns a *null* handle and deallocates the memory for the handle.

If you do not scan all outputs, memory for the handle remains allocated. Call `acc_release_object` to deallocate the memory.

For paths with only one output, such as a data path, you will most likely call `acc_next_output` once, outside a loop. In this case, you should also call `acc_release_object` because PLI cannot return a handle and deallocate the memory for that handle in the same call.

Failure to deallocate memory for output and output handles may result in a significant waste of memory in a large application.

```
bool is_net_on_path_output(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
   handle port_out, port_conn, bit;

   /* scan path output terminals */
   port_out = null;
   while (port_out = acc_next_output(path,  port_out))
   {
      /* retrieve net connected to path terminal */
      port_conn = acc_handle_conn (port_out);

      bit = null;
      if (acc_object_of_type (port_conn, accExpandedVector))
      {
         bit = null;
         while (bit = acc_next_bit (port_conn, bit))
            if (acc_compare_handles (bit, net))
               return (true);
      }
      else
         if (acc_compare_handles (bit, net))
            return (true);
   }

   return (false);
}
```

Figure 2-88: Determine if a net is connected to the input of a path

Related routines

`acc_release_object()`

Error conditions and detection

If the path is not a valid path of type `accModPath` or `accDataPath`, *null* is returned.

Usage and efficiency hints

The first time you call `acc_next_output`, PLI allocates the memory for a handle to an output and returns the handle to you. Each subsequent time you call the routine, PLI changes the handle to point to the next output. When you have completed scanning all outputs or an error condition arises, PLI returns a *null* handle and deallocates the memory for the handle.

2.15.61 acc_next_parameter

acc_next_parameter			
function	returns the next parameter within a module		
syntax	*parameter_handle* **= acc_next_parameter(***module_handle, parameter_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	parameter_handle	*handle*	handle of the parameter found

Usage example

The following C-language routine, `print_parameter_values`, uses `acc_next_parameter` to scan all parameters in a module.

```
#include "acc_user.h"

print_parameter_values(module_handle)
handle   module_handle;
{
   handle   param_handle;

   /*scan all parameters in the module and display their values*/
   /*  according to type*/
   param_handle = null;
   while( param_handle = acc_next_parameter( module_handle,param_handle))
   {
      io_printf( "Parameter %s = ",acc_fetch_fullname( param_handle ));
      switch( acc_fetch_paramtype( param_handle ) )
      {
        case accRealParam:
          io_printf( "%lf\n", acc_fetch_paramval( param_handle) );
          break;
        case accIntegerParam:
          io_printf( "%d\n", (int)acc_fetch_paramval( param_handle) );
          break;
        case accStringParam:
          io_printf("%s\n",(char*)(int)acc_fetch_paramval(param_handle) );
      }
   }
}
```

Figure 2-89: Displaying the values of all parameters in a module

2.15.62 acc_next_port

acc_next_port			
function	returns the next ***input***, ***output*** or ***inout*** port of a module in the order specified by the port list		
syntax	*load_handle* **= acc_next_port(***module_handle, port_handle***);**		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	port_handle	*handle*	handle of the ***input***, ***output*** or ***inout*** port found
related routines	***acc_next_portout***		

How acc_next_port differs from acc_next_portout

The routine `acc_next_port` returns *input*, *output*, and *inout* ports; `acc_next_portout` returns *output* and *inout* ports only, a task often required for delay calculation.

Accessing the next port connected to a hierarchically lower net

This routine returns the next port connected to a hierarchically lower net as well as the next port of a module. To get a port connected to a lower net, specify the net as the reference object argument for this routine.

Usage example

The following example presents a C-language routine, `display_inputs`, that uses `acc_next_port` to find and display the input ports of a module.

```
#include "acc_user.h"

display_inputs(module_handle)
handle   module_handle;
{
   handle   port_handle;
   int      direction;

   /*get handle for each module port*/
   port_handle = null;
   while (port_handle = acc_next_port( module_handle, port_handle) )
   {
      /*give the index of each input port*/
      if ( acc_fetch_direction( port_handle ) == accInput )
         io_printf( "Port #%d of %s is an input\n",
                        acc_fetch_index( port_handle ),
                        acc_fetch_fullname( module_handle ) );
   }
}
```

Figure 2-90: Displaying the input ports of a module

The next example presents a C-language routine, `display_port_connections`, that uses `acc_next_port` to find and display information about all of the ports connected to each bit of the port connected to the net input handle given as the argument.

```
display_port_connections()
{
   handle net = acc_handle_tfarg(1);
   handle port, bit;

   port = bit = null;
   while (port = acc_next_port (net, port))
      if (acc_object_of_type (port, accVectorPort))
         while (bit = acc_next_bit (port, bit))
            io_printf("PORTBIT: %s LOCONN: %s HICONN: %s/n",
                   acc_fetch_fullname(bit),
                   acc_fetch_fullname(acc_handle_loconn(bit)),
                   acc_fetch_fullname(acc_handle_hiconn(bit)));
}
```

Figure 2-91: Displaying the input ports of a hierarchically lower net

2.15.63 acc_next_portout

acc_next_portout			
function	returns the next ***output*** or ***inout*** port of a module in the order specified by the port list		
syntax	*load_handle* **= acc_next_portout(***module_handle, port_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	port_handle	*handle*	handle of the ***output*** or ***inout*** port found
related routines	***acc_next_port***		

How acc_next_portout differs from acc_next_port

The routine `acc_next_port` returns *input*, *output*, and *inout* ports; `acc_next_portout` returns *output* and *inout* ports only, a task often required for delay calculation.

Usage example

The following example presents a C-language routine, `display_outputs`, that uses `acc_next_portout` to find the output and inout ports of a module.

```
#include "acc_user.h"

display_outputs(module_handle)
handle   module_handle;
{
  handle   port_handle;

  /*get handle for each module port*/
  port_handle = null;
  while (port_handle = acc_next_portout( module_handle, port_handle) )
  {
     /*give the index of each output or inout port*/
     io_printf( "Port #%d of %s is an output or inout\n",
                  acc_fetch_index( port_handle ),
                  acc_fetch_fullname( module_handle ) );
  }
}
```

Figure 2-92: Displaying the output and inout ports of a module

2.15.64 acc_next_primitive

acc_next_primitive			
function	returns the next gate, switch or user-defined primitive (UDP) within a module		
syntax	*primitive_handle* **= acc_next_primitive(***module_handle, primitive_handle***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	primitive_handle	*handle*	handle of the gate, switch or UDP found

Usage example

The following example presents a C-language routine, `get_primitive_definitions`, that uses `acc_next_primitive` to display the defining names of all primitives in a module.

```
#include "acc_user.h"

get_primitive_definitions()
{
   handle   module_handle, prim_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   module_handle = acc_handle_tfarg( 1 );

   io_printf("Module %s contains the following types of primitives:\n",
              acc_fetch_fullname( module_handle ) );

   /*get and display defining names of all primitives in the module*/
   prim_handle = null;
   while( prim_handle = acc_next_primitive( module_handle,prim_handle))
      io_printf( "   %s\n",
                 acc_fetch_defname( prim_handle ) );
   acc_close();
}
```

Figure 2-93: Displaying the defining names of all primitives in a module

2.15.65 acc_next_specparam

acc_next_specparam			
function	returns the next specparam within a module		
syntax	*specparam_handle* **= acc_next_specparam(***module_handle, specparam_handle***)**;		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	specparam_handle	*handle*	handle of the specparam found

Usage example

The following C-language routine, `print_specparam_values`, uses `acc_next_specparam` to scan all specparams in a module.

```
#include "acc_user.h"

print_specparam_values(module_handle)
handle   module_handle;
{
   handle   sparam_handle;

   /*scan all parameters in the module and display their values*/
   /*  according to type*/
   sparam_handle = null;
   while(sparam_handle = acc_next_specparam(module_handle,sparam_handle))
   {
      io_printf( "Specparam %s = ", acc_fetch_fullname( sparam_handle ) );
      switch( acc_fetch_paramtype( sparam_handle ) )
      {
        case accRealParam:
         io_printf( "%lf\n", acc_fetch_paramval( sparam_handle) );
         break;
        case accIntegerParam:
         io_printf( "%d\n", (int)acc_fetch_paramval( sparam_handle) );
         break;
        case accStringParam:
         io_printf("%s\n",(char*)(int)acc_fetch_paramval(sparam_handle));
      }
   }
}
```

Figure 2-94: Displaying the values of all specparams in a module

2.15.66 acc_next_tchk

acc_next_tchk			
function	returns the next timing check within a module		
syntax	*timing_check_handle* **= acc_next_tchk(***module_handle, timing_check_handle***);**		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of the module
	timing_check_handle	*handle*	handle of the timing check found

Usage example

The following example presents a C-language routine, `show_setup_check_nets`, that uses `acc_next_tchk` to scan all cells below a module for setup timing checks.

```
#include "acc_user.h"

show_setup_check_nets()
{
   handle  mod_handle,cell_handle;
   handle  tchk_handle,tchkarg1_handle,tchkarg2_handle;
   int     tchk_type,counter;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*get handle for module*/
   mod_handle = acc_handle_tfarg( 1 );

   /*scan all cells in module for timing checks*/
   cell_handle = null;
   while ( cell_handle = acc_next_cell( mod_handle, cell_handle ) )
   {
      io_printf( "cell is: %s\n", acc_fetch_name( cell_handle ) );
      counter = 0;
      tchk_handle = null;
      while ( tchk_handle = acc_next_tchk( cell_handle, tchk_handle) )
      {
         /*get nets connected to timing check arguments*/
         tchk_type = acc_fetch_fulltype( tchk_handle );
         if ( tchk_type == accSetup )
         {
           counter++;
           io_printf("    for setup check #%d:\n",counter);
          tchkarg1_handle = acc_handle_tchkarg1( tchk_handle,mod_handle );
          tchkarg2_handle = acc_handle_tchkarg2( tchk_handle,mod_handle );
          io_printf("       1st net is %s\n        2nd net is %s\n",
                      acc_fetch_name( tchkarg1_handle ),
                      acc_fetch_name( tchkarg2_handle ) );
         }
      }
   }
   acc_close();
}
```

Figure 2-95: Scanning all cells below a module for setup timing checks

2.15.67 acc_next_terminal

acc_next_terminal			
function	returns the next terminal of a gate, switch or user-defined primitive (UDP)		
syntax	*terminal_handle* **= acc_next_terminal(***primitive_handle, terminal_handle***)***;*		
arguments	name	type	description
inputs:	**primitive_handle**	*handle*	handle of the gate, switch or UDP
	terminal_handle	*handle*	handle of the terminal found

Usage example

In the example in Figure 2-96, the routine `display_terminals` uses `acc_next_terminal` to retrieve all nets connected to a primitive.

```
#include "acc_user.h"

display_terminals()
{
     handle   prim_handle,term_handle;

     /*initialize environment for access routines*/
     acc_initialize();

     /*set development version*/
     acc_configure( accDevelopmentVersion, "1.6a" );

     /*get handle for primitive*/
     prim_handle = acc_handle_tfarg( 1 );

     io_printf("Connections to primitive %s:\n",
                acc_fetch_fullname(prim_handle) );
     /*scan all terminals of the primitive
     /*  and display their nets*/
     term_handle = null;
     while( term_handle = acc_next_terminal(prim_handle,term_handle) )
        io_printf("    %s\n",
                   acc_fetch_name( acc_handle_conn(term_handle) ) );
     acc_close();
}
```

Figure 2-96: Retrieving all nets connected to a primitive

2.15.68 acc_next_topmod

acc_next_topmod			
function	returns the next top-level module		
syntax	*module_handle* **= acc_next_topmod(***module_handle***)***;*		
arguments	name	type	description
inputs	**module_handle**	*handle*	handle of the top-level module found
related routines	Pass ***acc_next_child*** with null **module_handle** to ***acc_collect*** and ***acc_count*** to collect or count top-level modules		

Collecting and counting top-level modules

The access routine `acc_next_topmod` does not work with `acc_collect` or `acc_count`. However, you can collect or count top-level modules by passing `acc_next_child` with a null reference object argument. Here is a sample call that collects top-level modules:

```
acc_collect( acc_next_child, null, &count );
```

Here is a sample call that counts top-level modules:

```
acc_count( acc_next_child, null );
```

Usage example

The following example presents a C-language routine, `show_top_modules`, that uses `acc_next_topmod` to display the names of all top-level modules.

```
#include "acc_user.h"

show_top_modules()
{
   handle  module_handle;

   /*initialize environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*scan all top-level modules*/
   io_printf("The top-level modules are:\n");
   module_handle = null;
   while ( module_handle = acc_next_topmod( module_handle ) )

         /*display the instance name of each module*/
         io_printf("    %s\n",
                     acc_fetch_name(module_handle) );

   acc_close();
}
```

Figure 2-97: Displaying the names of all top-level modules

2.15.69 acc_object_in_typelist

acc_object_in_typelist			
function	determines whether an object fits a type or fulltype—or exhibits a property—specified in an input array		
syntax	*flag* **= acc_object_in_typelist(***object_handle, object_type_array***);**		
	name	type	description
return value	**flag**	*boolean*	**true** if object's type, fulltype or property matches one specified in the array **false** if there is no match
inputs	**object_handle**	*handle*	handle of an object
	object_type_array	*integer array*	array containing one or more predefined integer constants that represent the types and properties of objects desired; the last element must be 0

The access routine `acc_object_in_typelist` determines whether an object fits one of the types or fulltypes—or exhibits one of the properties—specified in a list. You pass this list as an array of predefined integer constants.

How to set up the array of object types

Declare the array of object types as a static, integer array inside the C routine that calls `acc_object_in_typelist`. The array can contain any number and combination of the predefined integer constants listed in Table 2-44 through Table 2-53 in Section 2.15.13, Table 2-60 in Section 2.15.23, and Table 2-76 in Section 2.15.70, and must be terminated by a `0`. These constants specify the types, fulltypes, and properties `acc_object_in_typelist` supports.

The following C-language statement shows how to declare an array of object types called `wired_nets`:

```
static int wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
```

If you pass this array to `acc_object_in_typelist`, the access routine will return true if its input object is a wired net.

Usage example

In the following example, the C-language routine `display_wired_nets` uses `acc_object_in_typelist` to determine if a net is a wired net. The routine then displays the name of each wired net found.

```
#include "acc_user.h"

display_wired_nets()
{
  static   int   wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
  handle   net_handle;

  /*reset environment for access routines*/
  acc_initialize();
  acc_configure( accDevelopmentVersion, "1.6" );

  /*get handle for net*/
  net_handle = acc_handle_tfarg( 1 );

  /*if a wired logic net, display its name*/
  if( acc_object_in_typelist(net_handle,wired_nets) )
     io_printf( "Net %s is a wired net\n",acc_fetch_name(net_handle) );
  else
     io_printf( "Net %s is not a wired net\n",acc_fetch_name(net_handle) );
}
```

Figure 2-98: Displaying the names of wired nets

2.15.70
acc_object_of_type

acc_object_of_type			
function	determines whether an object fits a specified type or fulltype, or exhibits a specified property		
syntax	*flag* **= acc_object_of_type(***object_handle, object_type***);**		
	name	type	description
return value	**flag**	*boolean*	**true** if object's type, fulltype, or property matches the one specified by *object_type* **false** if there is no match
inputs	**object_handle**	*handle*	handle of an object
	object_type	*integer*	a predefined integer constant that represents a type, fulltype or property

The access routine `acc_object_of_type` determines whether an object fits a specified type or fulltype or exhibits a particular property. The type, fulltype, or property can be any one of the predefined integer constants listed in Table 2-44 through Table 2-53 in Section 2.15.13, Table 2-60 in Section 2.15.23, and Table 2-76 below.

Property of object	*Predefined integer constant*
scalar	`accScalar`
vector	`accVector`
collapsed net	`accCollapsedNet`
expanded vector	`accExpandedVector`

Table 2-76: Properties

Usage example

In the following example, the routine `display_collapsed_nets` uses `acc_object_of_type` to determine whether nets are collapsed nets. The routine then displays each collapsed net, along with the simulated net.

```
#include "acc_user.h"

void  display_collapsed_nets()
{
   handle   mod_handle;
   handle   net_handle;
   handle   simulated_net_handle;

   /*reset environment for access routines*/
   acc_initialize();
   acc_configure( accDevelopmentVersion, "1.6" );

   /*get scope-first argument passed to user-defined system task*/
   /* associated with this routine*/
   mod_handle = acc_handle_tfarg(1);
   io_printf( "In module %s:\n",acc_fetch_fullname(mod_handle) );
   net_handle = null;

   /*display name of each collapsed net and its net of origin*/
   while( net_handle = acc_next_net(mod_handle,net_handle) )
   {
      if ( acc_object_of_type( net_handle,accCollapsedNet ) )
      {
         simulated_net_handle = acc_handle_simulated_net(net_handle);
         io_printf("   net %s was collapsed onto net %s\n",
                   acc_fetch_name( net_handle ),
                   acc_fetch_name( simulated_net_handle) );
      }
   }
}
```

Figure 2-99: Displaying collapsed nets in a particular scope

2.15.71 acc_product_version

acc_product_version	
function	returns a pointer to a character string that indicates what version of a Verilog simulator is linked to the access routines
syntax	*character_pointer* **= acc_product_version*()*;**
arguments	*none*

The output string

The routine `acc_product_version` produces a character string in the following format:

PRODUCT NAME **`Version`** ***VERSION NUMBER***

For example, if access routines are linked to version 1.6a of a Verilog simulator, `acc_product_version` returns a pointer to the following string:

`"Verilog Version 1.6a"`

Usage example

The following example presents a C-language routine, `show_versions`, that uses `acc_product_version` to identify the version of the Verilog simulator that is linked to access routines.

```
#include "acc_user.h"

show_versions()
{
 /*initialize environment for access routines*/
 acc_initialize();

 /*set development version*/
 acc_configure( accDevelopmentVersion, "1.6a" );

 /*show version of access routines*/
 /*  and version of Verilog that is linked to access routines*/
 io_printf("Running %s with %s\n",acc_version(),acc_product_version() );

 acc_close();
}
```

Figure 2-100: Identifying the version of a Verilog simulator that is linked to access routines

2.15.72 acc_release_object

acc_release_object			
function	deallocates memory associated with an input or output terminal path		
syntax	*integer_variable* **= acc_release_object (** *object_handle***)**;		
arguments	name	type	description
input:	**object_handle**	*handle*	handle to an input or output terminal path

Description

This routine deallocates memory for an input or output terminal path handle. You should call this routine after `acc_next_input` and `acc_next_output` under the following circumstances:

- You have not scanned all inputs or outputs.
- The input or output path had only one terminal.
- An error was returned.

Failure to deallocate memory for input and output handles may result in a significant waste of memory in a large application.

Examples

The following routine finds the data path corresponding to an input module path, and displays the source and destination port names for the data path. It uses `acc_next_input` and `acc_next_output` to get the first input and output, respectively, for a given path. Since it only calls `acc_next_input` and `acc_next_output` once, it must call `acc_release_object` to free the memory allocated for the input and output handles.

```
void display_datapath_terms(modpath)
handle modpath;
{
   handle datapath = acc_handle_datapath(modpath);
   handle pathin     = acc_next_input(datapath,  NULL);
   handle pathout   = acc_next_output(datapath,  NULL);
   /* there is only one input and output to a datapath */
   io_printf(“DATAPATH INPUT:     %s\n”, acc_fetch_fullname(pathin));
   io_printf(“DATAPATH OUTPUT:  %s\n”, acc_fetch_fullname(pathout));
   acc_release_object(pathin);
   acc_release_object(pathout);
}
```

Figure 2-101: Releasing the memory for a datapath’s input and output handles

In the following code fragment, there may be more than four inputs, but the code stops scanning at the fourth input.

```
pathin = NULL;
for (i = 0; i <= 4; i++)
   pathin = acc_next_input(path, pathin);
io_printf(“THE FOURTH INPUT IS: %s\n”,
         acc_fetch_name(pathin));
acc_release_object(pathin);
```

Figure 2-102: Releasing memory if scanning may have been incomplete

In the following code fragment, there may be more than one input, but only the first one is accessed.

```
first_pathout = acc_next_output(path, NULL);
do_something_with(first_pathout);
acc_release_object(first_pathout);
```

Figure 2-103: Releasing memory after one input

Related routines

`acc_next_input()`
`acc_next_output()`

2.15.73 acc_replace_delays

acc_replace_delays (single delay per transition)	
function	when accMinTypMaxDelays is "false", replaces delay values for primitives, module paths, timing checks or module input ports with delay values passed as arguments
syntax	
primitives:	**acc_replace_delays(** *primitive_handle, rise_delay, fall_delay, z_delay* **)**;
module paths:	**acc_replace_delays(** *path_handle, delay1, delay2, delay3***,** *delay4***,** *delay5, delay6* **)**;
timing checks:	**acc_replace_delays(** *timing_check_handle, limit* **)**;
ports:	**acc_replace_delays(** *port_handle, rise_delay, fall_delay, z_delay* **)**;
port's bits	**acc_replace_delays(** *bit_handle, rise_delay, fall_delay, z_delay* **)**;

arguments	name	type	description
inputs:	**object_handle**	*handle*	handle of a primitive, module path, timing check, module input port or bit of a module input port
primitives, ports, port's bits:	**rise_delay**	*double*	object's rise delay
	fall_delay	*double*	object's fall delay
	z_delay (depends on accToHiZDelay)	*double*	object's turn-off delay
module paths	**delay1**	*double*	module path's delay for transitions determined by accPathDelayCount
	delay2 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay3 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay4 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay5 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
	delay6 (depends on accPathDelayCount)	*double*	module path's delay for transitions determined by accPathDelayCount
timing checks	**limit**	*double*	timing check's limit

related routines	Use ***acc_configure(accMinTypMaxDelays, "false")*** for single delay per transition Use ***acc_configure(accPathDelayCount...)*** to set number of module path delays to replace Use ***acc_configure(accToHiZDelay...)*** to calculate turn-off delays from rise and fall delay values

acc_replace_delays (min:typ:max delays)	
function	when accMinTypMaxDelays is "true", replaces delay values for primitives, module paths, timing checks, module input ports or inter-module paths with min:typ:max delay values from an array
syntax	
primitives:	**acc_replace_delays(** *primitive_handle, array_ptr* **)**;
module paths:	**acc_replace_delays(** *path_handle, array_ptr* **)**;
timing checks:	**acc_replace_delays(** *timing_check_handle, array_ptr* **)**;
ports:	**acc_replace_delays(** *port_handle, array_ptr* **)**;
port's bits:	**acc_replace_delays(** *bit_handle, array_ptr* **)**;
inter-module paths	**acc_replace_delays(***intermod_path_handle, array_ptr***)**;

arguments	name	type	description
inputs:	**object_handle**	*handle*	handle of a primitive, module path, inter-module path, timing check, module input port or bit of a module input port
	array_ptr	*double address*	pointer to array of min:typ:max values (see Table 2-78)
related routines	Use ***acc_configure(accMinTypMaxDelays, "true")*** for min:typ:max delays for each transition		

The access routine `acc_replace_delays` works differently depending on how the configuration parameter `accMinTypMaxDelays` is set. When this parameter is set to `"false"`, `acc_replace_delays` assumes a single delay per transition and expects each delay to be passed as an individual argument. For this ***single delay mode***, the first syntax table in this section applies.

When `accMinTypMaxDelays` is set to `"true"`, `acc_replace_delays` expects one or more sets of *min:typ:max* delays to be passed in an array, rather than single delays passed as individual arguments. For this ***min:typ:max*** delay mode, the second syntax table in this section applies.

Different delays for different objects

The routine `acc_replace_delays` writes different delay values for different objects, as summarized in Table 2-77 and Table 2-78. Table 2-77 applies when `acc_replace_delays` is set for *single delay mode* (`accMinTypMaxDelays` is `"false"`) and Table 2-78 applies when this routine is set for *min:typ:max delay mode* (`accMinTypMaxDelays` is `"true"`).

For:	***acc_replace_delays*** *writes:*
primitives with a Z state: ***bufif*** gates ***notif*** gates ***MOS*** switches	***two*** or ***three*** delays: depends on how you configure ***accToHiZDelay***
primitives with ***no*** Z state	***two*** delays: rise delay, ***rise_delay*** fall delay, ***fall_delay***
module paths	***one***, ***two***, ***three*** or ***six*** delays: depends on how you configure ***accPathDelayCount***
timing checks	***one*** delay: timing check limit, ***limit***
module input or inout ports (MIPDs)	***two*** or ***three*** delays: depends on how you configure ***accToHiZDelay***

*Table 2-77: How **acc_replace_delays** writes delays in single delay mode*

In *single delay mode*, it is up to the user to supply the correct number of delay arguments to `acc_replace_delays`, as follows:

- MIPDs and Z-state primitives require ***two*** delay arguments—`rise_delay` and `fall_delay`—if `accToHiZDelay` is set to "`average`", "`max`" or "`min`"
- MIPDs and Z-state primitives require ***three*** delay arguments—`rise_delay`, `fall_delay` and `Z_delay`—if `accToHiZDelay` is set to "`from_user`" (the default)
- Primitives with no Z state require ***two*** delay arguments—`rise_delay` and `fall_delay`.
- Timing checks require ***one*** limit argument—`limit`.
- Module paths require ***one***, ***two***, ***three*** or ***six*** delay arguments, depending on how you configure `accPathDelayCount`.

For more information on the configuration parameters `accToHiZDelay` and `accPathDelayCount`, refer to Table 2-79, Table 2-80 and Table 2-81.

Table 2-78 shows how `acc_replace_delays` writes delays in *min:typ:max delay mode*.

For:	***acc_replace_delays*:**	*The delay array must pass:*
module input ports primitives inter-module paths	writes *three* sets of *min:typ:max* delays: one set for rise delays one set for fall delays one set for turn-off delays	*nine* values: array[0] = minimum rise delay array[1] = typical rise delay array[2] = maximum rise delay array[3] = minimum fall delay array[4] = typical fall delay array[5] = maximum fall delay array[6] = minimum turn-off delay array[7] = typical turn-off delay array[8] = maximum turn-off delay ***(you must always declare an array of size 9, even if turn-off delays are not used)***
module paths	writes *one*, *two*, *three* or *six* sets of *min:typ:max* delays: depends on how you configure `accPathDelayCount` ***(see Table 2-80)***	*three*, *six*, *nine* or*18* values: ***(see Table 2-80)***
timing checks	writes *one* set of *min:typ:max* delays: timing check limit	*three* values: array[0] = minimum timing check limit array[1] = typical timing check limit array[2] = maximum timing check limit

*Table 2-78: How **acc_replace_delays** writes delays in **min:typ:max** delay mode*

There are a couple of points to note in Table 2-78:

- For module input ports, primitives and inter-module paths, always declare an array of size 9, even if the objects do not have a Z state.
- The configuration parameter `accPathDelayCount` affects the *min:typ:max* delays processed for module paths. Table 2-80 describes these effects in greater detail.

Replacing delays for module paths (single delay mode)

In *single delay mode*, you can control how many delays `acc_replace_delays` writes for module paths by using the access routine `acc_configure` to set the delay count parameter `accPathDelayCount` as shown in Table 2-79.

When ***accPathDelayCount*** *is:*	***acc_replace_delays*** *writes:*
1	***one*** delay value, the same for all transitions: ***delay1*** *(last five delay arguments may be dropped)*
2	***two*** delay values: one for rising transitions: ***delay1*** one for falling transitions***: delay2*** *(last four delay arguments may be dropped)*
3	***three*** delay values: one for rising transitions: ***delay1*** one for falling transitions: ***delay2*** one for transitions to z: ***delay3*** *(last three delay arguments may be dropped)*
6 (the default)	all ***six*** delay values, a different delay for each possible transition among 0, 1 and z: one for 01 transitions: ***delay1*** one for 10 transitions: ***delay2*** one for 0z transitions: ***delay3*** one for z1 transitions: ***delay4*** one for 1z transitions: ***delay5*** one for z0 transitions: ***delay6***

Table 2-79: How ***accPathDelayCount*** *affects module path delays in single delay mode*

The minimum number of delay arguments to pass to `acc_replace_delays` must equal the value of `accPathDelayCount`. You may drop any remaining arguments.

The following example shows how to set `accPathDelayCount` so that `acc_replace_delays` writes rise and fall delays for module paths:

```
acc_configure( accPathDelayCount, "2" );
```

If you do not set `accPathDelayCount` explicitly, it defaults to ***6***; in this case, you must pass all six delay arguments when you call `acc_replace_delays` in *single delay mode*.

Replacing delays for module paths (min:typ:max mode)

The following rules apply when you modify *min:typ:max* delays for module paths:

- the number of *sets* of *min:typ:max* delays must equal the value of `accPathDelayCount`
- the size of the delay array must be three times the value of `accPathDelayCount`

Table 2-80 summarizes how `accPathDelayCount` affects *min:typ:max* delays for module paths.

When ***accPathDelayCount*** *is:*	*The number of sets of min:typ:max path delays is:*	*The delay array must pass or retrieve:*
"1"	**one**: the same minimum, typical and maximum delay for all transitions	**three values:** `array[0]` = minimum delay `array[1]` = typical delay `array[2]` = maximum delay
"2"	**two**: one set for rising transitions one set for falling transitions	**six values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay
"3"	**three**: one set for rising transitions one set for falling transitions one set for transitions to z	**nine values:** `array[0]` = minimum rise delay `array[1]` = typical rise delay `array[2]` = maximum rise delay `array[3]` = minimum fall delay `array[4]` = typical fall delay `array[5]` = maximum fall delay `array[6]` = minimum turn-off delay `array[7]` = typical turn-off delay `array[8]` = maximum turn-off delay
"6" *(the default)*	**six:** one set for 01 transitions one set for 10 transitions one set for 0z transitions one set for z1 transitions one set for 1z transitions one set for z0 transitions	**18 values:** `array[0]` = minimum 01 delay `array[1]` = typical 01 delay `array[2]` = maximum 01 delay `array[3]` = minimum 10 delay `array[4]` = typical 10 delay `array[5]` = maximum 10 delay `array[6]` = minimum 0z delay `array[7]` = typical 0z delay `array[8]` = maximum 0z delay `array[9]` = minimum z1 delay `array[10]` = typical z1 delay `array[11]` = maximum z1 delay `array[12]` = minimum 1z delay `array[13]` = typical 1z delay `array[14]` = maximum 1z delay `array[15]` = minimum z0 delay `array[16]` = typical z0 delay `array[17]` = maximum z0 delay

Table 2-80: The relationship between ***accPathDelayCount*** *and* ***min:typ:max*** *delays for module paths*

Replacing delays for inter-module paths

An inter-module path is a wire path that connects an output or inout port of one module to an input or inout port of another module. You can use the access routines `acc_fetch_delays` and `acc_replace_delays` to fetch and modify delays for inter-module paths.

Please note: For inter-module paths, you *must* fetch or replace delays in *min:typ:max delay mode*. Therefore, set `accMinTypMax` to `"true"` before calling `acc_fetch_delays` or `acc_replace_delays` for inter-module paths.

Changing module input port delays (MIPDs)

Use `acc_replace_delays` to modify existing Module Input Port Delays (MIPDs). An MIPD is a delay associated with a module input port or inout port. The MIPD describes the delay between the module port and each of the loads in its fanout. In an MIPD you can specify rise, fall, and high impedance propagation delays.

You can write an MIPD for each individual bit of a vector port using `acc_replace_delays` in conjunction with `acc_next_bit`. For more information, see Section 2.15.49.

Declaring the array that holds min:typ:max values

Use Table 2-78 and Table 2-80 to decide how large to make the array that passes or holds *min:typ:max* values. The array must be able to store the correct number of delays that will be processed. Declaring an array that is too small will cause errors or unpredictable results.

Calculating turn-off delays from rise and fall delays

In *single delay mode*, you can instruct `acc_replace_delays` to automatically calculate turn-off delays from rise and fall delays by using the access routine `acc_configure` to set the parameter `accToHiZDelay` as follows:

*When **accToHiZDelay** is:*	***acc_replace_delays:***
"average"	calculates turn-off delay as the average of the rise and fall delays
"min"	calculates turn-off delay as the smaller of the rise and fall delays
"max"	calculates turn-off delay as the larger of the rise and fall delays
"from_user" (default)	sets turn-off delay to the value passed as a user-supplied argument: ***z_delay*** for primitives, ports and port's bits.

*Table 2-81: How the value of **accToHiZDelay** affects **acc_replace_delays***

The following example shows how to set `accToHiZDelay` so that `acc_replace_delays` calculates turn-off delays as the average of rise and fall delays for Z-state primitives:

```
acc_configure( accToHiZDelay, "average" );
```

Note that the value you assign to `accToHiZDelay` also influences how `acc_append_delays` derives turn-off delays.

Effect of timescales

The routine `acc_replace_delays` writes delay values in the timescale of the module that contains `primitive_handle`, `path_handle` or `timing_check_handle`.

Usage example: single delay mode

The following example presents a C-language routine, `write_path_delays`, that uses `acc_replace_delays` to replace the current delays on a path with new delay values read from a file called `pathdelay.dat`. The format of the file is shown in Figure 2-104; the example appears in Figure 2-105.

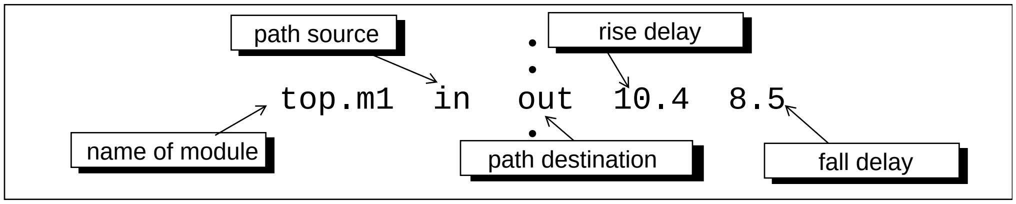


Figure 2-104: Format of file ***pathdelay.dat***

```
#include <stdio.h>
#include "acc_user.h"

write_path_delays()
{
   FILE     *infile;
   char     full_module_name[NAME_SIZE];
   char     pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
   double   rise,fall;
   handle   mod_handle, path_handle;

   /*initialize the environment for access routines*/
   acc_initialize();

   /*set development version*/
   acc_configure( accDevelopmentVersion, "1.6a" );

   /*set accPathDelayCount parameter to return rise and fall delays only*/
   acc_configure( accPathDelayCount, "2" );

   /*read delays from file - "r" means read only*/
   infile = fopen("pathdelay.dat","r");
   fscanf(infile, "%s %s %s %lf %lf",
          full_module_name,pathin_name,pathout_name,&rise,&fall);

   /*get handle for the module and the path*/
   mod_handle = acc_handle_object(full_module_name);
   path_handle = acc_handle_modpath(mod_handle,pathin_name,pathout_name);

   /*replace delays with new values*/
   acc_replace_delays( path_handle, rise, fall );

   acc_close();
}
```

Figure 2-105: Replacing delays on a module path

Note that the identifier `EOF` is a predefined constant that stands for ***end of file***. `NAME_SIZE` is a user-defined constant that represents the maximum number of characters allowed for any object name in an input file.

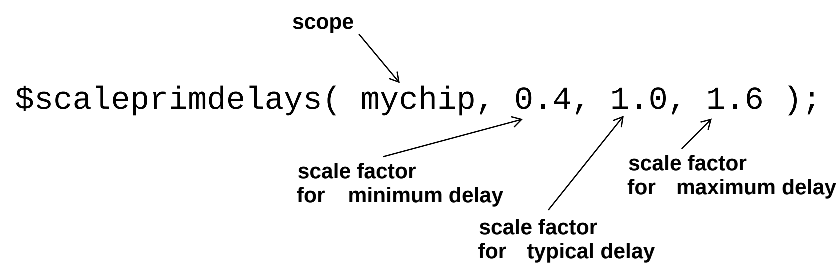
Usage example: scaling min:typ:max delays

One way to scale delays for min:typ:max is to write a C application routine that performs the following actions:

- fetches *min:typ:max* delays for an appropriate object
- multiplies each delay by a scale factor
- replaces *min:typ:max* delays for that object with the new, scaled values

The following C-language routine, `scale_prim_delays`, scales *min:typ:max* delays on all primitive delays inside cells within a given scope.

Assume for this example that `scale_prim_delays` is associated through the interface mechanism with a user-defined system task called `$scaleprimdelays`. The scope and scale factors are passed as arguments to `$scaleprimdelays` as follows:



Assume that the Verilog-HDL description contains only one delay per transition—in this case, the *typical* delay.

```
#include "acc_user.h"
#include "veriuser.h"

void scale_prim_delays()
{
 handle top, cell, prim;
 int i, count;
 double min_scale_factor, typ_scale_factor, max_scale_factor;
 double da[9];

 acc_initialize();
 acc_configure(accDevelopmentVersion,"1.5b.3");
 acc_configure(accMinTypMaxDelays,"true");

 top = acc_handle_tfarg(1);
 min_scale_factor = acc_fetch_tfarg(2);
 typ_scale_factor = acc_fetch_tfarg(3);
 max_scale_factor = acc_fetch_tfarg(4);


 io_printf("Scale min:typ:max delays for primitives in cells below %s\n",
            acc_fetch_fullname(top) );
 io_printf("Scaling factors-min:typ:max-%4.2f:%4.2f:%4.2f\n",
            min_scale_factor, typ_scale_factor, max_scale_factor );

 count = 0;
 cell = null;
 while (cell = acc_next_cell(top, cell))
    {
        prim = null;
        while (prim = acc_next_primitive(cell, prim))
        {
            acc_fetch_delays(prim,da);

            for (i=0; i<9; i+=3)
                da[i] = da[i]*min_scale_factor;
            for (i=1; i<9; i+=3)
                da[i] = da[i]*typ_scale_factor;
            for (i=2; i<9; i+=3)
                da[i] = da[i]*max_scale_factor;

            acc_replace_delays(prim,da);

            count++;
        }
    }

 io_printf("Completed scale of %d primitives\n", count);
}
```

array must hold three sets of *min:typ:max* values for rise, fall and turn-off delays

argument #1: scope
argument #2: scale factor for minimum delay
argument #3: scale factor for typical delay
argument #4: scale factor for maximum delay

fetch *min:typ:max* delays and store in array **da** as follows:

da[0] da[1] da[2] — typical rise delay
da[3] da[4] da[5] — typical fall delay
da[6] da[7] da[8] — typical turn-off delay

scale delays

replace *min:typ:max* delays with scaled values

Figure 2-106: Scaling **min:typ:max** *delays on primitives*

2.15.74 acc_set_scope

acc_set_scope			
function	sets a scope for ***acc_handle_object*** to use when searching in the design hierarchy		
syntax	**acc_set_scope(***module_handle, module_name***)***;*		
arguments	name	type	description
inputs:	**module_handle**	*handle*	handle of a module instance
	module_name (required if accEnableArgs is set and module_handle is null)	*character string pointer:* quoted string literal *or* character pointer variable	name of a module instance
related routine	***acc_handle_object*** Use ***acc_configure(accEnableArgs,"acc_set_scope")*** to use **module_name** argument		

How the routine works

If:	*acc_set_scope:*
you call: *acc_configure(accEnableArgs*, *"acc_set_scope")* and you pass: **module_handle** as a null pointer	sets scope to the level of ***module_name*** in the design hierarchy
you call: *acc_configure(accEnableArgs, "acc_set_scope")* and you pass: a valid **module_handle**	sets scope to the level of ***module_handle*** in the design hierarchy
you call: *acc_configure(accEnableArgs, "acc_set_scope")* and you pass: **module_handle** and **module_name** as null pointers	sets scope to the top-level module that appears first in the source description
you do *not* call: *acc_configure(accEnableArgs*, *"acc_set_scope")*	always ignores ***module_name*** and sets scope to the level of ***module_handle*** in the design hierarchy if ***module_handle*** is null, sets scope to the top-level module that appears first in the source description *(you may drop the optional argument* ***module_name****)*

Table 2-82: How ***acc_set_scope*** *works*

Optional arguments

When the optional argument is required for a particular call to `acc_set_scope`, you must set the configuration parameter `accEnableArgs` by calling `acc_configure` as follows:

```
acc_configure(accEnableArgs, "acc_set_scope");
```

If `accEnableArgs` is not set for `acc_set_scope`, the routine always ignores its optional argument.

When the optional argument is *not* required for a particular call to `acc_set_scope`, you can drop the argument.

However, when an optional argument precedes one or more required arguments, it must be supplied even if it is ignored by the access routine. In this case, you can specify the argument as a null value.

There is no relationship between acc_set_scope and $scope

The routine `acc_set_scope` defines a scope for `acc_handle_object` only. Access routines are *not* affected when you change scope by calling the system task `$scope` interactively.

Usage example

The following example shows how a C-language routine, `is_net_in_module`, uses `acc_set_scope` to set a scope for `acc_handle_object` to determine if a net is in a module.

```
#include "acc_user.h"

is_net_in_module(module_handle,net_name)
handle  module_handle;
char    *net_name;
{
  handle  net_handle;
  handle  load_handle, load_net_handle;

  /*set scope to module*/
  acc_set_scope( module_handle );

  /*get handle for net*/
  net_handle = acc_handle_object(net_name);

  if (net_handle)
      io_printf( "Net %s found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
  else
      io_printf( "Net %s not found in module %s\n",
                  net_name,
                  acc_fetch_fullname(module_handle) );
}
```

Figure 2-107: Setting a scope for ***acc_handle_object***

2.15.75 acc_set_value

acc_set_value			
function	set and propogate a value on a register or a sequential UDP		
syntax	*integer_variable* = acc_set_value *(object_handle, value_p, delay_p);*		
arguments	name	type	description
input:	**object_handle**	*handle*	handle to a register or sequential UDP
	value_p	*p_setval_value*	pointer to a structure containing value to be set
	delay_p	*p_setval_delay*	pointer to a structure containing delay before value is set

Description

This routine is used to set and propagate a value on a register or a sequential UDP.

The structure `s_setval_value` contains the value to be written. A value can be entered into this structure as a string, scalar, integer, or real. The 'format' field indicates the value type, while the 'value' union is set to the value to be written. Refer to the structure definition and associated defined values in file '`acc_user.h`' in Appendix A.

For registers, the structure `s_setval_delay` indicates if and what delay will take place before a register value assignment. A delay model is indicated with the 'model' field. The available delay models are specified using predefined integer constants. The predefined integer constant for delay models are listed in Table 2-83.

Predefined Integer Constant	Delay Model	Description
accInertialDelay	Inertial delay	All scheduled events on the object are removed before this event is scheduled.
accTransportDelay	Transport delay	All events scheduled for times later than this event are removed.
accPureTransportDelay	Pure transport delay	No events are removed.
accNoDelay	No delay	Sets the object to the indicated value with no delay.

Table 2-83: Delay models

For UDPs, the 'model' field must be accNoDelay, and the new value is assigned with no delay even if the UDP instance has a delay.

Refer to the file '`acc_user.h`' in Appendix A for more information about the `s_setval_delay` structure definition and associated defined values.

Example

The following example sets and propagates a value on a register.

```
int my_set_value()
{
  static s_setval_delay delay_s = {{accRealTime},
                                   accInertialDelay};
  static s_setval_value value_s = {accBinStrVal};

  handle reg = acc_handle_tfarg(1);

  value_s.value.str = acc_fetch_tfarg_str(2);

  delay_s.time.real = acc_fetch_tfarg(3);

  acc_set_value(reg, &value_s, &delay_s);
}
```

*Figure 2-108: Using **acc_set_value** to set and propagate a value on a register*

Error conditions and detection

The return value is 0 if OK, nonzero for an error, in which case an error message will be reported and 'acc_error_flag' is set to true. It is suggested that the error flag be utilized for error detection.

Usage and efficiency hints

If a value is to be set with no delay, it is more efficient to set the `s_setval_delay` model field to be accNoDelay than to simply set the delay to 0.

For 'tf_xputy' users

This routine can be used in place of the '`tf_put`' functions, including `tf_strdelputp`. It is more flexible than the '`tf_`' routines in that it operates on any writable object for which a handle is available instead of being limited to objects which are arguments to a system task. It is limited compared to those routines in that it cannot currently operate on expressions.

2.15.76 acc_vcl_add

acc_vcl_add			
function	tells the Verilog simulator to call a consumer routine with value change information whenever an object changes value		
syntax	**acc_vcl_add**(*object_handle, consumer_routine, user_data, vcl_flags);*		
arguments	name	type	description
inputs:	**object_handle**	*handle*	handle to an object you want to monitor (such as a register or net)
	consumer_routine	*C routine pointer*	C routine in an application that the Verilog simulator calls when the object changes value
	user_data	*character pointer*	user-defined data that the Verilog simulator passes back to the consumer routine when the object changes value
	vcl_flags	*integer constant*	flag that selects the type of information that the Verilog simulator reports to the consumer routine
related routine	***acc_vcl_delete***		

The VCL access routine `acc_vcl_add` tells the Verilog simulator to report logic value information or logic value and strength information to a consumer routine.

The `acc_vcl_add` access routine takes the four arguments that are defined in Table 2-84.

Argument	Definition
`object_handle`	The `object_handle` argument is a handle to the object to be monitored by an application. You obtain `object_handle` by calling PLI access routines. ***Please note:*** The object_handle passed to acc_vcl_add is not returned to the application when the Verilog simulator reports a value change. Therefore, the application should use the user_data argument to save any necessary information about the object prior to calling acc_vcl_add. (See the discussion on user_data below.)
`consumer_routine`	The `consumer_routine` argument is a pointer to a C routine. The Verilog simulator calls the routine whenever the object changes value. This routine processes VCL data. The Verilog simulator expects the consumer routine to have a specific calling interface.
`user_data`	The `user_data` argument is user-defined data (such as the object name, the last value, and the `object_handle` itself) that the Verilog simulator passes back to the application when the object changes value. This argument is typically a pointer to a data structure that contains information about the object.
`vcl_flags`	The `vcl_flags` argument tells theVerilog simulator the information to report. There are two kinds of information that the Verilog simulator reports, logic value or logic value and strength. The flag constants are defined in `acc_user.h` as `vcl_verilog_logic` and `vcl_verilog_strength`. These flags are defined in Table 2-85.

Table 2-84: Arguments to ***acc_vcl_add***

vcl_flags	*What it does*
vcl_verilog_logic	indicates the application needs logic value information from the Verilog simulator
vcl_verilog_strength	indicates the application needs logic value and strength information from the Verilog simulator

Table 2-85: ***vcl_flags*** *in* ***acc_vcl_add***

Multiple calls to acc_vcl_add

If an application calls `acc_vcl_add` with the same arguments more than once, the Verilog simulator calls the consumer routine only once when the object changes value.

Support for multiple applications

If multiple applications monitor the same object at the same time, each application receives a separate call whenever that object changes value.

Typically, multiple applications have distinct consumer routines and `user_data` pointers. These different consumer routines allow the value change information to be processed in different ways. Therefore, there are separate callbacks to different applications.

2.15.77 acc_vcl_delete

acc_vcl_delete			
function	tells the Verilog simulator to stop calling a consumer routine with value change information when an object changes value		
syntax	**acc_vcl_delete**(*object_handle, consumer_routine, user_data, vcl_flags);*		
arguments	name	type	description
inputs:	**object_handle**	*handle*	handle to an object you want to monitor (such as a register or net)
	consumer_routine	*C routine pointer*	C routine in an application that the Verilog simulator calls when the object changes value
	user_data	*character pointer*	user-defined data that the Verilog simulator passes back to the consumer routine when the object changes value
	vcl_flags	*integer constant*	flag that tells VCL to pass the delete request to the Verilog simulator
related routine	***acc_vcl_add***		

The VCL access routine `acc_vcl_delete` tells the Verilog simulator to stop reporting previously requested information to a consumer routine. The `acc_vcl_delete` access routine takes the four arguments that are defined in Table 2-86.

Argument	Definition
object_handle	The object_handle argument is a handle to the object to be monitored by an application. You obtain object_handle by calling PLI access routines.
consumer_routine	The consumer_routine argument is a pointer to a C routine. The Verilog simulator calls the routine whenever the object changes value. This routine processes VCL data. The Verilog simulator expects the consumer routine to have a specific calling interface.
user_data	The user_data argument is user-defined data (such as the object name, the last value, and the object_handle itself) that the Verilog simulator passes back to the application when the object changes value. This argument is typically a pointer to a data structure that contains information about the object.
vcl_flags	The vcl_flags argument is the constant vcl_verilog as defined in acc_user.h. This argument indicates that VCL should pass the delete request to the Verilog simulator. The constant vcl_verilog_flag is described in Table 2-87.

*Table 2-86: Arguments to **acc_vcl_delete***

vcl_flags	what it does
vcl_verilog	use this to stop monitoring either logic value or strength information from the Verilog simulator

*Table 2-87: **vcl_flags** in **acc_vcl_delete***

To stop monitoring either logic value or strength information, you select vcl_verilog.

Multiple calls

When multiple applications are monitoring the same object, acc_vcl_delete stops monitoring the object only for the application associated with a specific consumer_routine and user_data pointer.

Regardless of the number of calls an application makes to `acc_vcl_add` for the same `object_handle`, `user_data`, and `consumer_routine`, only one call to `acc_vcl_delete` is required to stop the Verilog simulator from notifying the consumer routine of value changes.

2.15.78 acc_version

acc_version	
function	returns a pointer to a character string that indicates version number of your access routine software
syntax	*character_pointer* **= acc_version*()*** *;*
arguments	*none*

The output string

The routine `acc_version` produces a character string in the following format:

`Access Routines Version` ***VERSION_NUMBER***

For example, if you run version 1.6a of access routines, `acc_version` returns a pointer to the following string:

`"Access Routines Version 1.6a"`

Usage example

The following C-language routine, `show_versions`, uses `acc_version` to identify the current version of access routines.

```
#include "acc_user.h"

show_versions()
{
  handle   module_handle;

  /*initialize environment for access routines*/
  acc_initialize();

  /*set development version*/
  acc_configure( accDevelopmentVersion, "1.6a" );

  /*show version of access routines*/
 /* and version of simulator that is linked to access routines*/
 io_printf("Running %s with %s\n",acc_version(),acc_product_version() );

  acc_close();
}
```

Figure 2-109: Identifying the current version of access routines

3

Interface Mechanism

The interface mechanism works as follows:

The user writes a routine that performs the desired operation, using the routines described in this manual to get data from the simulation environment and/or send data back to it. Information about this routine is placed in a table in the file `veriuser.c`. This information includes (at a minimum) the name of the user-supplied routine and the name by which it will be known to Verilog. There is one entry in the table for each task or function that the user wants to be able to invoke from a Verilog simulator. More than one task or function can call the same programming language routine, with the distinction between them made by a data value defined in the table and passed to the routine automatically by the Verilog simulator.

To invoke the new tasks and functions from the Verilog description, the user references them just as he would a built-in system task or function, such as `$display` or `$time`.

3.1 User-Supplied Routines

As described above, the user provides a routine that will be called when the new task or function is invoked. This routine is known as the `calltf` routine.

The user can also provide several other routines, which are described below.

A routine that will be called when the new task or function is encountered during compilation can be provided; it is optional. This routine will typically check the correctness of arguments passed from Verilog, but it can also perform other chores. This routine is known as the `checktf` routine.

Each routine that will be invoked as a function must have a routine that returns the width (in bits) of the function return value. This routine is known as the `sizetf` routine.

Another routine, known as the `misctf` routine, is optional. It is called for a variety of reasons, and a reason flag is passed to it so it can determine why it was called. The reasons are listed below, under Routine Arguments, and include input argument value changes and disable. The availability of this routine makes the interface very powerful, as it allows the user to set up tasks which are called asynchronously (just like `$monitor`), and to disable the activity associated with them.

3.2 Routine Arguments

The user-supplied routines are always passed two arguments: ‘data’ and ‘reason’. These are passed as the first and second arguments, respectively.

3.2.1 Data

The value that is passed as the ‘data’ argument is obtained from the table; it is defined by the user when the table is filled in. This value can be used to allow several different tasks and functions to use the same `checktf`, `calltf`, or `misctf` routines. To do this, the table entries for the various tasks and functions would contain the same routine names, but would have a different data value for each task or function.

3.2.2 Reason

The value for the ‘reason’ argument is defined as follows:

```
reason_checktf     - for checktf
reason_sizetf      - for sizetf
reason_calltf      - for calltf
reason_<xyz>       - for misctf, reasons defined in
                     veriuser.h
```

3.3 Supplied Files

Two files supplied with the Verilog system are used to link user program modules to the Verilog system:

- `veriuser.c`

  ‘C’ code template and examples

- `veriuser.h`

  ‘C’ header file

The data associated with the user-supplied routines is inserted into the external table 'veriusertfs' normally declared in `veriuser.c`. This table is an array having the structure as defined in the header file `veriuser.h` and described below. Each task or function that can be invoked in Verilog source description has an entry in the array.

The file `veriuser.h` is included by `veriuser.c` and should not be edited by the user.

3.3.1 Table of Tasks and Functions

A data type called `t_tfcell` is defined in `veriuser.h`. The fields of this structure define a task or function so that the Verilog system knows about it and can call the right routines upon invocation.

The file `veriuser.c` creates an array of `t_tfcell` structures and calls it veriusertfs. The fields of this array are filled in by editing the file `veriuser.c`.

WARNING

DO NOT ATTEMPT TO CHANGE THE CONTENTS OF THE DATA STRUCTURE BY ANY OTHER MEANS!

t_tfcell fields

The meaning of each field is defined below.

`type`

Defines entry to be a task or function by using the values of 'usertask,' 'userfunction,' or 'userrealfunction' as defined in `veriuser.h.`

`data`

The value given here is passed as the first parameter to 'checktf', 'sizetf', 'calltf', or 'misctf' (defined below) when each is called. This component can be useful when several user tasks or functions have just slightly differing purposes, so that corresponding routine pointers can point to the same routine definitions, and the task or function can be distinguished in the routines by the argument data value.

`checktf`

Pointer to the user-supplied routine for checking the parameters of the task or function statement. This routine is called once for each task or function reference in the source description, or in an interactive command, during the compilation stage. A value of 0 (null pointer) may be given if no routine is to be supplied; however, it is recommended that the parameters be checked for correctness.

`sizetf`

Pointer to the user-supplied routine that must return the number of bits of the function return value. For functions this routine is mandatory, and is ignored for tasks.

calltf

Pointer to the user-supplied routine that is to be called from Verilog whenever the task or function is executed during simulation.

misctf

Pointer to the user-supplied routine that is to be called from Verilog for various miscellaneous reasons. A value of 0 (null pointer) may be given if no routine is to be supplied. See above for complete list of reasons.

tfname

A literal string defining the name of the task or function. The first character of the name **must** be the dollar character ($). The remaining characters may be letters, digits, the underscore character (_), or the dollar character. Upper and lower case letters are considered to be different (unless the upper case compiler option `-u` has been used). The names may be of any size, and all characters are significant. It is possible to override and change the definition of a built-in system task or function by simply using the same name; for example, a user could build a different random number generator by supplying the name `$random` in the table.

forwref

Should be set to 1. If it is set to 0, then module or primitive instances may not be used as arguments to the system task or function.

All other components are for system usage and MUST NOT be assigned to.

Any of the routine pointers can be 0, in which case the calls from Verilog are not made. This means that you need not have a parameter checking routine, for example. In this case, fill in the corresponding field with the value 0.

3.4 A Simple Example

The following example illustrates how to use the interface. It is the simplest case possible: only one routine is associated with the task (the `calltf` routine) and no data is passed between Verilog and the routine.

3.4.1 User-Supplied Routine

The routine below is written in the C language. When it is called, it prints the string “hello world” to the terminal. It is simply a C language routine; nothing special has been done, and no special routines have been called.

```
hello()
{
    printf(" hello world \n");
}
```

3.4.2
Table Entry

The entire `veriuser.c` file for the example is provided below. The text associated with the examples normally provided with the Verilog system has been deleted for clarity.

```
/* Filename: veriuser.c */
/* Verilog user tasks example */
#include "veriuser.h"
hello()
{
    printf("hello world \n");
}
/
            * Template table for defining user tas
            ks and functions.
   See file "veriuser.h" for structure definition
*/
s_tfcell veriusertfs[] =
{
     {usertask, 0,
                0, 0, hello, 0,
                "$hello", 0},
    /* add extra entries here */
    {0} /* this line must always be last */
};
```

3.4.3
Task Invocation

Invocation of the new task is identical to invocation of any of the built-in system tasks, such as `$stop` or `$showscopes`. When this description is simulated, it will print the string “hello world”.

```
module test;
initial $hello;
endmodule
```

4

Utility Routines

Interaction between the Verilog system and the user’s routines is handled by a set of routines that are supplied with the Verilog system. These routines are called in the user-supplied routines to pass data in both directions across the Verilog/programming language boundary. The type, arguments, and operation of each routine are detailed below.

4.1 Call Instances

Most of these routines are in two forms: one dealing with the current call, or “instance,” and another dealing with an instance other than the current one and referenced by an instance pointer. The first routine described below gets the instance pointer of the current instance so that it can be saved for later use. This feature is useful when routines are related to each other; a typical example is a pair of routines where one is a setup routine and another is a display routine. This pairing concept has been used extensively in developing the graphical display mechanisms for Verilog (e.g., `$setup_waves`, `$display_waves`) and it is expected that users will find this very useful. Another example of the concept is a pair of routines where the first, a setup routine, prints a header, and the second, a display routine, writes the data. The instance mechanism is used to associate the list of strings passed to the setup routine with the list of variables passed to the display routine.

The steps are as follows:

1. In the setup routine, save instance pointer by using the `tf_getinstance` routine.
2. In the display routine, use `tf_igetcstringp` to get the string associated with each parameter.
3. Use this data just as if it had been passed in to the display routine.

This mechanism eliminates the need to pass the same information to all the routines that need it—later calls can use the data passed to earlier ones.

4.2 64-Bit Integer and Real Number Values

Some routines are provided in multiple forms:

- one form to deal with 32-bit parameter values that can be passed in one integer
- a second form to deal with 64-bit values, which must be passed in two integers
- and sometimes a third form to deal with real number values, which must be passed as double-precision floating-point numbers

The routines that deal with 64-bit integer values are named using the name of the corresponding 32-bit routine with the word `long` inserted—for example, `tf_gettime` and `tf_getlongtime`. Similarly, routines that deal with real number values are named using the name of the corresponding 32-bit routine with the word `real` inserted—as in `tf_setdelay` and `tf_setrealdelay`.

4.3 Effect of Timescales on Utility Routines

By default, each of the following routines expresses its delay or time in the timescale unit of the system task or function that calls the utility:

tf_setdelay
tf_isetdelay
tf_setlongdelay
tf_isetlongdelay
tf_strdelputp
tf_istrdelputp
tf_strlongdelputp
tf_istrlongdelputp
tf_strrealdelputp
tf_istrrealdelputp
tf_gettime
tf_getlongtime
tf_getnextlongtime

If, for example, a system task in a module with a timescale of nanoseconds calls `tf_setdelay(10)`, but the simulator is operating in picoseconds, `tf_setdelay` automatically scales the delay by 1000 to 10,000 picoseconds.

4.4 Routine Definitions

This section describes each utility routine.

4.4.1 tf_getinstance

Purpose

get current instance pointer

Synopsis

```
char *tf_getinstance()
```

This routine returns a pointer that identifies the current instance of the task or function that is being acted upon by the Verilog simulator. In the following sections, the argument `instance_p` will always refer to this instance pointer. `tf_getinstance` can be called during any of the reasons, saved by the user, and used later in other calls to refer to the current instance. Most of the utility routines are in two forms, one dealing with the current task or function instance, and the other dealing with information about another instance, where the instance pointer was obtained during a call to that other task or function. For example, the call `tf_inump(tf_getinstance())` is equivalent to the call `tf_nump()`.

4.4.2 tf_nump

Purpose

get number of task or function parameters

Synopsis

```
int tf_nump()
int tf_inump(instance_p)
    char *instance_p;
```

The routine `tf_nump` returns the number of parameters specified in the task or function statement in the Verilog source description. The number returned is greater than or equal to zero. `tf_inump` will return the number of parameters for a given instance.

4.4.3 tf_typep

Purpose

get parameter type

Synopsis

```
int tf_typep(nparam)
    int nparam;
int tf_itypep(nparam, instance_p)
    int nparam;
    char *instance_p;
```

The routine tf_typep is used for determining a given parameter's type. The value nparam must be a number specifying the parameter position, with 1 for the first parameter, 2 for the second, etc. tf_itypep returns a parameter type for a given instance.

For example, the statement

```
int itype=tf_typep(3)
```

examines the third parameter in the argument list, and sets itype to one of the values shown below, depending on parameter type. If the third argument in the list is an integer value which cannot be written to, the value tf_readonly is returned. These return values, shown in the following list, are defined constants.

tf_nullparam
: If the parameter is the null expression, i.e., where no text has been given as the parameter, or if 'nparam' is out of range.

tf_string
: If the parameter is a literal string.

tf_readonly
: If the parameter is a value returning expression that cannot be written to (i.e., an expression that would be illegal as a left-hand-side construct in a procedural assignment).

tf_readwrite
: If the parameter is a value returning expression that can be written to. Writable expressions are the left-hand-side constructs in procedural assignments.

tf_readonlyreal
: Same as tf_readonly except that the specified parameter is a real value.

tf_readwritereal
: Same as tf_readwrite except that the specified

parameter is a real value.

4.4.4
tf_getp/tf_putp

Purpose

get parameter value
put parameter value

Synopsis

```
int tf_getp(nparam)
      int nparam;
int tf_igetp(nparam, instance_p)
      int nparam;
      char *instance_p;
double tf_getrealp(nparam)
      int nparam;
double tf_igetrealp(nparam, instance_p)
      int nparam;
      char *instance_p;
tf_putp(nparam, value)
      int nparam;
      int value;
tf_iputp(nparam, value, instance_p)
      int nparam;
      int value;
      char instance_p;
tf_putrealp(nparam, value)
      int nparam;
      double value;
tf_iputrealp(nparam, value, instance_p)
      int nparam;
      double value;
      char *instance_p;
int tf_getlongp(aof_highvalue, nparam)
      int *aof_highvalue;
      int nparam;
int tf_igetlongp(aof_highvalue, nparam, instance_p)
      int *aof_highvalue;
      int nparam;
      char *instance_p;
int tf_putlongp(nparam, lowvalue, highvalue)
      int nparam;
      int lowvalue, highvalue;
int tf_iputlongp(nparam, lowvalue, highvalue, instance_p)
      int nparam;
      int lowvalue, highvalue;
      char *instance_p;
```

The routines `tf_getp` and `tf_getrealp` return a value of the parameter specified by `nparam`. The routines `tf_igetp` and `tf_igetrealp` return a parameter value for a given instance. The difference between the real and ‘non-real’ routines is in the value returned, as shown by the following example.

Assume that the fourth parameter in the argument list has a value of 9.6 (a real value). Then,

```
int ivalue = tf_getp(4)
     would set ivalue to 10

double dvalue = tf_getrealp(4)
     would set dvalue to 9.6
```

In the first example, note that the int conversion rounds off the value of 9.6 to 10 (rather than truncates to 9). In the second example, note that the real value must be declared as a ‘double’ (not as a ‘float’).

If the parameter is a literal string, then the `tf_getp` routine returns a pointer to the ‘C’ type string (string terminated by a ‘\0’ character). If `nparam` is out of range or the parameter is null, zero is returned.

The routine `tf_getrealp` cannot handle literal strings. Therefore, before calling `tf_getrealp`, make sure the argument is not a literal string by calling `tf_typep` to check its type.

The routine `tf_putp` writes the integer value to the parameter specified by `nparam`. Similarly, the routine `tf_putrealp` writes the real value to the parameter specified by `nparam`. The routines `tf_iputp` and `tf_iputrealp` write a parameter value for a given instance. The parameter must be one that can be written. To check if it is, use the `tf_typep` routine. You can write back the function return value by making `nparam` equal to zero. If `nparam` is out of range or the parameter cannot be written to, then nothing happens.

The routines `tf_getlongp` and `tf_putlongp` operate on 64-bit integer parameter values. The `tf_getlongp()` routine returns both the least significant bits of the value in the parameter lowvalue and the most significant bits in the parameter `highvalue`.

Please note: When using the `tf_putp` and `tf_putrealp` routines to write values to parameters, make sure that the data types of the arguments passed are consistent with the data types required by the function called. There is no inherent data type checking in C.

The following examples illustrate what cautions should be taken.

If the second parameter is of type `tf_readwritereal`, the following 'put' routines are valid:

```
int i = 5;
tf_putp(2, i);
```

(Sets the second argument to 5.0)

```
double d = 5.7;
tf_putrealp(2, d);
```

(Sets the second argument to 5.7)

The following routines, however, are invalid for the reasons given below:

```
int i = 5;
tf_putrealp(2, i);
```

(The statement "int i = 5" passes a 32-bit integer to `tf_putrealp`(), which is looking for a 64-bit double. Since there is no data type checking, `tf_putreal`() reads 32 bits of undefined data and tries to use it as if it were valid data. No error message is generated.)

```
float f = 5;
tf_putrealp(2, f);
```

(Similar to the example above. The float statement passes a 32-bit float to `tf_putrealp()` which is looking for a 64-bit double. No error message is generated.)

```
double d = 5.7;
tf_putp(2, d);
```

(The `tf_putp()` routine takes only 32-bits of the 64-bit double passed to it by the statement double d = 5.7.)

4.4.5 tf_strgetp

Purpose

get formatted parameter values

Synopsis

```
char *tf_strgetp(nparam, format_char)
    int nparam;
    char format_char;
char *tf_istrgetp(nparam, format_char, instance_p)
    int nparam;
    char format_char;
    char *instance_p;
```

The routines `tf_strgetp` and `tf_istrgetp` return a pointer to a string which contains the value of the parameter expression in the format specified by `format_char`. For the routine `tf_strgetp`, `nparam` specifies the parameter in the current instance while `tf_igetstrp` looks at the instance pointed to by `instance_p` to locate the parameter specified by `nparam`.

`format_char` can be one of:

```
'b' or 'B' for binary
'o' or 'O' for octal
'd' or 'D' for decimal
'h' or 'H' for hexadecimal
```

The string value returned will have the same form as output from the formatted output task `$display` in terms of value lengths and value characters used. The length is of arbitrary size (i.e. not limited to 32 bits as with the `tf_getp` routine), and unknown and high-impedance values can be obtained.

The referenced parameter can be a string, in which case a pointer to the string will be returned (`format_char` is ignored in this case). A 0 pointer is returned for errors.

4.4.6
tf_exprinfo / tf_nodeinfo

Purpose

get expression information
get node information

Synopsis

```
struct t_tfexprinfo *tf_exprinfo(nparam, exprinfo_p)
    int nparam;
    struct t_tfexprinfo *exprinfo_p;
struct t_tfexprinfo *tf_iexprinfo(nparam, exprinfo_p,
    instance_p)
    int nparam;
    struct t_tfexprinfo *exprinfo_p;
    char *instance_p;
struct t_tfnodeinfo *tf_nodeinfo(nparam, nodeinfo_p)
    int nparam;
    struct t_tfnodeinfo *nodeinfo_p;
struct t_tfnodeinfo *tf_inodeinfo(nparam, nodeinfo_p,
    instance_p)
    int nparam;
    struct t_tfnodeinfo *nodeinfo_p;
    char *instance_p;
```

These routines are for obtaining general information about a parameter indicated by `nparam`. `tf_exprinfo` and `tf_iexprinfo` give information related to the parameter expression where the information is stored in the C structure `t_tfexprinfo` as defined in the file `veriuser.h`. `tf_nodeinfo` and `tf_inodeinfo` give information related to a node (relevant only when a parameter is writable), where the information is stored in the `t_tfnodeinfo` structure.

For all these routines the user must allocate space to hold the information before making the call. For example the user could allocate the necessary space in the following way:

```
{
    s_tfexprinfo info;
    tf_exprinfo(n, &info);
...
}
```

All four routines return the second argument, i.e. the pointer to the information structure. If `nparam` is out of range, or some other error is found, then 0 is returned.

For every parameter there always exists expression information. For parameters that are writable, node information also exists. Memory and strength manipulations can be done only through node information. The following defines the components passed in the expression information.

expr_type
One of the following values as defined in `veriuser.h`:

```
tf_nullparam
tf_string
tf_readonly(for 'int')
tf_readonlyreal(for 'real')
tf_readwrite(for 'int')
tf_readwritereal(for 'real')
tf_rwbitselect(bit-select)
tf_rwpartselect(part-select)
tf_rwmemselect(memory-select)
```

expr_value_p
If the expression type is not real, `expr_value_p` is a pointer to an array of `t_vecval` structures that gives the resultant value of the expression. See below for the definition of this structure.

real_value
If the expression type is real (`tf_readonlyreal` or `tf_readwritereal`), `real_value` will contain the value.

expr_string
If the expression is of type `tf_string`, `expr_string` points to the string.

expr_ngroups
If the expression type is not real, `expr_ngroups` indicates the number of groups for the parameter expression value and determines the array size of the `expr_value_p` value structure. If the expression type is real, `expr_ngroups` is zero.

expr_vec_size
If the expression type is not real, `expr_vec_size` indicates the total number of bits in the array of `expr_value_p` value structures. If the expression type is real, `expr_vec_size` is zero.

expr_sign
The sign of the expression: zero for unsigned, non-zero for signed.

The following defines the components passed in the node information:

node_type

A `node_type` is one of the following values as defined in `veriuser`.h:

```
tf_null_node - not a writable parameter
tf_reg_node - parameter references a register variable
tf_integer_node - parameter references an integer variable
tf_real_node - parameter references a real variable
tf_time_node - parameter references a time variable
tf_netvector_node - parameter references a vector net
tf_netscalar_node - parameter references a scalar net
tf_memory_node - parameter references a memory
```

node_value

A `node_value` is a union of pointers to value structures defining the current value on the node referenced by the parameter. The union member accessed depends on `node_type`. The union members are:

```
vecval_p - for tf_reg_node, tf_integer_node,tf_time_node,
      and tf_netvector_node.
real_value_p - for tf_real_node
strengthval_p - for tf_netscalar_node
memoryval_p - for tf_memory_node
```

node_symbol

A `node_symbol` is a string pointer to the parameter's identifier.

node_ngroups

If the node type is not real, `node_ngroups` indicates the number of groups for the parameter `nodevalue` and is used to determine the array size of the `node_value.vecval_p` value structure. If the node type is real, `node_ngroups` is zero.

node_vec_size

If the node type is not real, `node_vec_size` indicates the total number of bits in the array of `node_value.vecval_p` structure. If the expression type is real, `node_vec_size` is zero.

node_sign

The sign of the node: zero for unsigned, non-zero for signed.

node_mem_size

Number of elements in `node_value.memoryval_p` structure.

The usual data structure for representing vector values is an array of the following structure.

If the number of bits in the vector (defined by `expr_vec_size` or `node_vec_size`) is less than or equal to 32, then there is only one group in the array. For 33 to 64 bits, two groups are in the array, and so on. The number of groups is also given by the value of `expr_ngroups` and `node_ngroups`. The components `avalbits` and `bvalbits` hold the bit patterns making up the parameter's value. The least significant bit in the value is represented by the least significant bits in the `avalbits` and `bvalbits` components, and so on up. The bit coding is as follows:

```
ab value
00 logic 0
10 logic 1
11 unknown
01 high impedance
```

For node information only, when `node_type` is `tf_netscalar_node` then `node_value.strengthval_p` points to a net strength data structure of the following form:

```
typedef struct t_strengthval
{
    int strength0;
    int strength1;
} s_strengthval, *p_strengthval;
```

Where `strength0` gives the 0-strength bit pattern for the value, and `strength1` gives the 1-strength bit pattern. See the section on logic strength modeling in the reference manual for details about these bit patterns.

For node information only, when `node_type` is `tf_memory_node` then `node_value.memoryval_p` points to a structure giving the total contents of the memory. The structure is organized as follows

```
struct
{
    char avalbits[node_ngroups];
    char bvalbits[node_ngroups];
} memval[node_mem_size];
```

Note that this data structure cannot be represented in C, thus `node_value.memoryval_p` is declared as a pointer to a char. The memory element addressed by the left-hand-side index given in the memory declaration is located in the first group of bytes, i.e. the byte groups represented by `memval[0]`.

4.4.7
tf_asynchon / tf_asynchoff

Purpose

enable asynchronous calling
disable asynchronous calling

Synopsis

```
tf_asyncon()
tf_iasynchon(instance_p)
    char *instance_p;
tf_asynchoff()
tf_iasynchoff(instance_p)
    char *instance_p;
```

These routines enable a user routine to be called asynchronously whenever a parameter value changes. The routines `tf_asynchon` and `tf_iasynchon` enable this feature. After enabling, the routine specified by `misctf` in the veriusertfs table is called with a reason of `reason_paramvc` each time a parameter changes value. The parameter index number of the parameter that changed value is passed to the `misctf` routine as the third parameter.

The routines `tf_asynchoff` and `tf_iasynchoff` disable further calling of the `misctf` routine for reason `paramvc`.

Any number of enables and disables can be made during simulation.

4.4.8 tf_synchronize

Purpose

synchronize to end of simulation time unit

Synopsis

```
tf_synchronize()
tf_isynchronize(instance_p)
    char *instance_p;
```

The routines `tf_synchronize` and `tf_isynchronize` allow the processing of parameters to be delayed until the end of the current simulation time slot. This is useful when the user wants to synchronize all parameter value changes and process them after all that will change at a particular simulation time have changed.

The routine specified by `misctf` in the veriusertfs table is called with a reason of `reason_synch` at the end of the current simulation time slot if `tf_synchronize` or `tf_isynchronize` is called during the current time.

It does not matter if these routines are called more than once at a particular time slot. Multiple calls at a time slot will result in only one activation of the `misctf` routine at the end of the time slot. They must be called at each time slot in which synchronization is desired.

4.4.9 tf_rosynchronize

Purpose

synchronize to end of simulation time slot

Synopsis

```
tf_rosynchronize()

tf_irosynchronize(instance_p)
    char *instance_p;
```

These routines allow certain processing to be delayed until the end of the current simulation time. The routine specified by `misctf` in the veriusertfs table is called with a reason of `reason_rosynch` at the end of the current simulation time.

These routines are very similar to the `tf_synchronize` and `tf_isynchronize` calls. The difference is that these calls guarantee that no new events at the current simulation time will be created after processing routines called due to reason `reason_rosynch`. Consequently, it is not possible to write output values or generate new events during the `misctf` call with reason `reason_rosynch`. This means that calls to routines such as `tf_strdelputp` and `tf_setdelay` are illegal during processing of the `misctf` routine with reason `reason_rosynch`.

Please note: It is essential that these routines be used instead of the `tf_synchronize` and `tf_isynchronize` calls when the `tf_getnextlongtime` routine is being used.

4.4.10 tf_gettime

Purpose

get current simulation time

Synopsis

```
int tf_gettime()

int tf_getlongtime(aof_hightime)
    int *aof_hightime;
```

The routine `tf_gettime` returns the current simulation time as an integer.

The routine `tf_getlongtime` returns double simulation time. The high 32 bits of simulation time is assigned to the `aof_hightime` argument, and the low 32 bits of time is returned.

Each routine expresses its time in the timescale unit of the system task or function that calls the utility (see Section 4.3).

4.4.11
tf_strgettime

Purpose

get current simulation time as a string

Synopsis

```
char *tf_strgettime()
```

Returns a pointer to a string which is the ASCII representation of the current simulation time.

4.4.12 tf_gettimeunit

Purpose

get the timescale units of a module

Synopsis

```
int tf_gettimeunit()

int tf_igettimeunit(instance_p)
    char *instance_p;
```

The routines `tf_gettimeunit` and `tf_igettimeunit` fetch the timescale units specified for the module containing the current system task call or the call referenced by the instance pointer. Each routine returns an integer code representing the unit of time, as shown in Table 4-1.

Please note: If `instance_p` is null, these routines return the smallest time precision specified in the design, which is equivalent to the simulation time unit.

Integer Code	*Unit of Time*
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 us
-5	10 us
-6	1 us
-7	100 ns
-8	10 ns
-9	1 ns
-10	100 ps
-11	10 ps
-12	1 ps
-13	100 fs
-14	10 fs
-15	1 fs

Table 4-1: Integer codes that represent units of time or precision

4.4.13 tf_gettimeprecision

Purpose

get the timescale precision of a module

Synopsis

```
int tf_gettimeprecision()

int tf_igettimeprecision(instance_p)
    char *instance_p;
```

The routines `tf_gettimeprecision` and `tf_igettimeprecision` fetch the timescale precision for the module that contains the current system task call or the call referenced by the instance pointer. Each routine returns an integer code representing the time precision, as shown in Table 4-1.

Please note: If `instance_p` is null, these routines return the smallest time precision specified in the design, which is equivalent to the simulation time unit.

4.4.14
io_printf

Purpose

formatted print to standard output and log file

Synopsis

```
io_printf(format, arg1, ...)
     char *format;
```

The routine `io_printf` places output to the standard output stream and to log file output. It works in the same way as the standard C routine `printf`, except that a maximum of 12 arguments are allowed.

4.4.15 tf_error

Purpose

report error

Synopsis

```
tf_error(format, arg1, ...)
```

`tf_error` provides an error reporting mechanism compatible with the Verilog compiler. The arguments work in the same way as with the `io_printf` routine, except that a maximum of five arguments are allowed. Statement location information (file name and line number of source statement) is appended to the message.

If `tf_error` is called during the checking of parameters, then Verilog will either abort compilation, or, if the user task was called on the interactive command line, the command will be abandoned.

4.4.16 tf_warning

Purpose

report warning

Synopsis

```
tf_warning(format, arg1, ...)
```

`tf_warning` gives a warning message compatible with the Verilog compiler. The arguments work in the same way as with the `io_printf` routine, except that a maximum of five arguments are allowed. Statement location information (file name and line number of source statement) and the text “warning!” are appended to the message.

4.4.17 tf_message

Purpose

report error

Synopsis

```
tf_message(level, facility, code, message,
           arg1,...arg5)
```

The `tf_message` utility routine allows an application to tell the Verilog simulator to display error message information. This utility routine contains the following required message string arguments: `level`, `facility`, `code`, and `message`.

The `level` field gives the class of information. There are five classes: `ERR_ERROR`, `ERR_MESSAGE`, `ERR_INTERNAL`, `ERR_SYSTEM`, and `ERR_WARNING`.

`Facility` and `code` are string arguments that you specify. These string arguments are printed in the Verilog simulator message syntax and should be kept to 6-10 characters in length. `Message` is the PLI message you want to display.

Your application can use up to a maximum of five variable arguments. These variable arguments display information such as the variable name, file name, or operating system name to the software tool that issues the message. There is no limit to the length of the variable argument.

Whenever possible, the Verilog simulator displays the file name and line number of the statement that triggers the error along with the statement itself before it displays the information contained in these arguments.

If your application calls `tf_message` during an execution of the `checktf` routine with the level set to `ERR_ERROR`, `ERR_SYSTEM`, or `ERR_INTERNAL`, the Verilog simulator stops compiling your source description.

If you interactively enter a user-defined system task that invokes this routine with the level set to `ERR_ERROR`, `ERR_SYSTEM`, or `ERR_INTERNAL`, the Verilog simulator terminates that system task.

You do not need to include formatting characters such as `\n`, `\t`, `\b`, `\f`, or `\r` when creating a message. The error handling unit automatically formats each message.

An example of `tf_message` follows in Section 4.4.18, *tf_text*.

4.4.18
tf_text

Purpose

stores error information

Synopsis

`tf_text(message, arg1,...arg5)`

The `tf_text` utility routine stores information about an error in a buffer. It allows your application to store information about an error before it calls the `tf_message` utility routine. Your application can call `tf_text` any number of times before it calls `tf_message`.

When your application calls `tf_message`, the Verilog simulator displays the information stored by `tf_text` before it displays the information in an application's call to `tf_message`. Each call to `tf_message` clears the buffer where `tf_text` stores its information.

Your application can use a single message string argument and up to a maximum of five variable arguments to store error information. These arguments can be any kind of information you want to display in your message, such as the variable name, file name, or operating system name. An example of this is `argnum` in Example 4-1. There is no limit to the length of a message argument.

```
    tf_text (“Argument number %d”, argnum);
    tf_message (ERR_ERROR, "User", "TFARG",
                “ is illegal in task %s", task_name);
```

Example 4-1: An example of tf_test and tf_message in an application

The error message created by the code in Example 4-1 is shown in Example 4-2.

```
Error!     Argument number 2 is illegal in task         [User-TFARG]
           $usertask
```

Example 4-2: The printed message for the example in Example 4-1

4.4.19
tf_dostop

Purpose

enable interactive mode

Synopsis

```
tf_dostop()
```

The routine `tf_dostop` stops the simulation and puts the Verilog simulator into the interactive mode exactly as if a `$stop` was encountered in a Verilog source description.

4.4.20 tf_dofinish

Purpose

finish simulation

Synopsis

```
tf_dofinish()
```

The routine `tf_dofinish` finishes the simulation exactly as if a `$finish` was encountered in a Verilog source description.

4.4.21 tf_getcstringp

Purpose

get parameter value as a pointer to a C language string

Synopsis

```
char *tf_getcstringp(nparam)
    int nparam;

char *tf_igetcstringp(nparam, instance_p)
    int nparam;
    char *instance_p;
```

Returns a pointer to a C language string. Returns null if the argument identified by `nparam` is null or if `nparam` is out of range. If the argument identified by `nparam` is a literal string, a variable, or an expression, then `tf_getcstringp` will convert its value to a C language ASCII string by: eliminating leading zeros, converting each group of eight bits to an ASCII character, and adding a null character to the end.

4.4.22 tf_setdelay

Purpose

activate the `misctf` routine for the user task at a particular simulation time

Synopsis

```
int tf_setdelay(delay)
    int delay;

int tf_isetdelay(delay, instance_p)
    int delay;
    char *instance_p;

int tf_setlongdelay(lowdelay, highdelay)
    int lowdelay, highdelay;

int tf_isetlongdelay(lowdelay, highdelay,
                     instance_p)
    int lowdelay, highdelay;
    char *instance_p;

int tf_setrealdelay(realdelay)
    double realdelay;

int tf_isetrealdelay(realdelay, instance_p)
    double realdelay;
    char *instance_p;
```

Causes the `misctf` routine to be called at a future simulation time by setting a "reactivation" time. The `misctf` routine is called at the reactivation time, with a reason of `reason_reactivate`. Single-precision integer, double-precision integer, and double-precision floating-point forms are available. Delay must be greater than or equal to 0. The reactivation time is the current simulation time plus the delay specified. Each delay assumes the timescale units specified for the module containing the current system task call or the call referenced by the instance pointer (see Section 4.3). Each routine can be called several times with different delays, and several reactivations will be scheduled.

Returns 0 if an error is detected, 1 if not.

4.4.23 tf_clearalldelays

Purpose

clear all scheduled reactivations

Synopsis

```
tf_clearalldelays()

tf_iclearalldelays(instance_p)
    char *instance_p;
```

Clear all reactivation delays. Remove the effect of all previous `tf_setdelay` calls for the given instance.

4.4.24 tf_strdelputp

Purpose

write value to parameter from string value specification

Synopsis

```
int tf_strdelputp(nparam, bitlength, format_char, value_p,
      delay, delaytype)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      int delay;
      int delaytype;

int tf_istrdelputp(nparam, bitlength, format_char,
      value_p, delay, delaytype, instance_p)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      int delay;
      int delaytype;
      char *instance_p;

int tf_strlongdelputp(nparam, bitlength, format_char,
      value_p, lowdelay, highdelay, delaytype)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      int lowdelay, highdelay;
      int delaytype;

int tf_istrlongdelputp(nparam, bitlength, format_char,
      value_p, lowdelay, highdelay, delaytype, instance_p)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      int lowdelay, highdelay;
      int delaytype;
      char *instance_p;
```

```
int tf_strrealdelputp(nparam, bitlength, format_char,
      value_p, realdelay, delaytype)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      double realdelay;
      int delaytype;

int tf_istrrealdelputp(nparam, bitlength, format_char,
      value_p, realdelay, delaytype, instance_p)
      int nparam;
      int bitlength;
      int format_char;
      char *value_p;
      double realdelay;
      int delaytype;
      char *instance_p;
```

Write a string value to the specified parameter with delay. Schedules an event on the parameter in the Verilog model at a future simulation time. Both single and double integer and real forms are available. The parameter `bitlength` defines the value size in bits. `Format_char` defines the format of the value specified by `value_p` and must be one of:

’b’, ’B’, ’o’, ’O’, ’d’, ’D’, ’h’, ’H’.

`Delay` must be greater than or equal to 0. The `delaytype` parameter is one of:

```
0 - for inertial delay
1 - for transport delay
2 - for pure transport delay
```

Inertial delays cause all scheduled events on the output parameter in the Verilog model to be removed before scheduling a new event. Transport delays cause removal of events that are scheduled for times later than the new event. Pure transport delays do not cause any events to be removed before the new event is scheduled. Caution should be used when using pure delays because the last event to be scheduled is not necessarily the last one to occur.

Each delay assumes the timescale units specified for the module containing the current system task call or the call referenced by the instance pointer (see Section 4.3).

Returns 0 if an error is detected, 1 if not.

4.4.25
tf_scale_longdelay / tf_scale_realdelay

Purpose

convert a long integer delay or double-precision floating-point delay to internal simulation time units

Synopsis

```
void tf_scale_longdelay(instance_p, delay_lo,
     delay_hi, aof_delay_lo, aof_delay_hi)
     char *instance_p;
     int delay_lo;
     int delay_hi;
     int *aof_delay_lo;
     int *aof_delay_hi;

void tf_scale_realdelay(instance_p, realdelay,
     aof_realdelay)
     char *instance_p;
     double realdelay;
     double *aof_realdelay;
```

These two routines convert a long integer delay or a double-precision floating-point delay into internal simulation time units. The delay parameters assume the timescale units of the module that contains the system task or function referenced by the instance pointer. The parameters begining with `aof` contain the address of the converted delay returned by the routine; that is, integer parameters should be passed to this routine using ‘&’.

4.4.26
tf_unscale_longdelay / tf_unscale_realdelay

Purpose

convert a delay expressed in internal simulation time units to the timescale of a particular module

Synopsis

```
void tf_unscale_longdelay(instance_p, delay_lo,
     delay_hi, aof_delay_lo, aof_delay_hi)
     char *instance_p;
     int delay_lo;
     int delay_hi;
     int *aof_delay_lo;
     int *aof_delay_hi;

void tf_unscale_realdelay(instance_p, realdelay,
     aof_realdelay)
     char *instance_p;
     double realdelay;
     double *aof_realdelay;
```

These two routines take a long integer delay or a double-precision floating-point delay expressed in internal simulation time units and convert it into the timescale units of the module that contains the system task or function referenced by the instance pointer. The parameters beginning with `aof` contain the address of the converted delay returned by the routine; that is, integer parameters should be passed to this routine using ‘&’.

4.4.27 Parameter Value Change Flags

Parameter Value Change (pvc) flags are used to indicate whether a particular parameter has changed value. Each parameter has two pvc flags: an old pvc flag which is set by Verilog when the change occurs, and a saved pvc flag which is controlled by the user.

Typical usage of these flags is for the user to move the old flags to the saved flags by the call `tf_movepvc_flag(-1)`, before anything else is done. The saved flags are then available for later use.

Routines that use the pvc flags are:

`tf_copypvc_flag(), tf_icopypvc_flag()`
`tf_movepvc_flag(), tf_imovepvc_flag()`
`tf_testpvc_flag(), tf_itestpvc_flag()`
`tf_getpchange(), tf_igetpchange()`

Please note: pvc flags will not be set until `tf_asynchon` is called.

4.4.28
tf_copypvc_flag

Purpose

copy parameter value change flag

Synopsis

```
int tf_copypvc_flag(nparam)
    int nparam;

int tf_icopypvc_flag(nparam, instance_p)
    int nparam;
    char *instance_p;
```

Copy pvc flag to saved flag. Returns flag that was copied. If `nparam` is -1, then all parameters are copied and the logical OR of all saved flags is returned.

4.4.29 tf_movepvc_flag

Purpose

move parameter value change flag

Synopsis

```
int tf_movepvc_flag(nparam)
    int nparam;

int tf_imovepvc_flag(nparam, instance_p)
    int nparam;
    char *instance_p;
```

Moves pvc flag to saved flag and clears old flag. Returns flag that was moved. If `nparam` is -1, then all parameters are moved and the logical OR of all moved flags is returned.

4.4.30 tf_testpvc_flag

Purpose

test parameter value change flag

Synopsis

```
int tf_testpvc_flag(nparam)
    int nparam;

int tf_itestpvc_flag(nparam, instance_p)
    int nparam;
    char *instance_p;
```

Returns saved pvc flag for a given instance. If `nparam` is -1, then all parameters are tested and the logical OR of all saved flags is returned.

4.4.31
tf_getpchange

Purpose

get number of parameter that changed value

Synopsis

```
int tf_getpchange(nparam)
    int nparam;

int tf_igetpchange(nparam, instance_p)
    int nparam;
    char *instance_p;
```

Gets the number of the next parameter, with number greater than `nparam`, that changed value. The `nparam` argument must be 0 the first time this routine is called within a given user routine invocation. Returns the parameter number if there is a change in a parameter with a number greater than `nparam`. Returns 0 if there are no changes in parameters greater than `nparam` or if an error is detected. This routine uses the saved pvc flags, so it is necessary to execute `tf_movepvc_flag(-1)` first.

4.4.32 Work Areas

Work areas allow storage of data associated with specific instances of user task invocations. Routines are provided to store and retrieve pointers. These pointers should point to the data structures which represent data associated with the user task. It is up to the user to allocate the memory required and save the pointers from one invocation to the next.

4.4.33 tf_setworkarea

Purpose

store work area pointer

Synopsis

```
void tf_setworkarea(workarea)
    char *workarea;

void tf_isetworkarea(workarea, instance_p)
    char *workarea;
    char *instance_p;
```

Store a given work area pointer into cell. The parameter `workarea_p` is any memory pointer.

4.4.34 tf_getworkarea

Purpose

get work area pointer

Synopsis

```
char *tf_getworkarea()

char *tf_igetworkarea(instance_p)
    char *instance_p;
```

Get the stored work area pointer.

4.4.35 tf_setroutine

Purpose

store routine pointer

Synopsis

```
tf_setroutine(routine)
    char (*routine)();

tf_isetroutine(routine, instance_p)
    char (*routine)();
    char *instance_p;
```

Store a given routine pointer into cell.

4.4.36 tf_getroutine

Purpose

get routine pointer

Synopsis

```
char *tf_getroutine()

char *tf_igetroutine(instance_p)
    char *instance_p;
```

Get the routine pointer stored using `tf_setroutine`.

4.4.37 tf_settflist

Purpose

store user task/function instance pointer

Synopsis

```
tf_settflist(tflist)
    char *tflist;

tf_isettflist(other_instance_p, instance_p)
    char *other_instance_p;
    char *instance_p;
```

Store a given task or function instance pointer into cell.

4.4.38
tf_gettflist

Purpose

get user task/function instance pointer

Synopsis

```
char *tf_gettflist()

char *tf_igettflist(instance_p)
    char *instance_p;
```

Get a task or function instance pointer that was stored using `tf_settflist`.

4.4.39 tf_mipname

Purpose

get module instance path name as a string

Synopsis

```
char *tf_mipname()

char *tf_imipname(instance_p)
    char *instance_p;
```

Gets a module instance path name. Returns a string that is the Verilog-HDL hierarchical path name to the module containing the call to the user task or function.

Please note: If this string is needed across multiple calls to the user task, the string value should be stored or `tf_mipname` should be called in each invocation.

4.4.40
tf_ispname

Purpose

get scope path name as a string

Synopsis

```
char *tf_spname()

char *tf_ispname(instance_p)
    char *instance_p;
```

Gets a scope path name. Returns a string which is the Verilog-HDL hierarchical path name to the scope containing the call to the user task or function.

4.4.41 tf_sizep

Purpose

get bit length of a parameter

Synopsis

```
int tf_sizep(nparam)
    int nparam;

int tf_isizep(nparam, instance_p)
    int nparam;
    char *instance_p;
```

If the parameter is not real, `tf_sizep` (or `tf_isizep`) returns the value size of the parameter. The return value is an integer which indicates the length of the parameter value in bits.

If the parameter is a literal string, `tf_sizep` returns the string length.

If the parameter is real or if an error is detected, `tf_sizep` (or `tf_isizep`) returns a zero.

4.4.42
io_mcdprintf

Purpose

write to multiple-channel descriptor files

Synopsis

```
io_mcdprintf(multi_chann_desc, format, arg1, ...)
    int multi_chann_desc;
    char *format;
```

Similar to `io_printf` but writes to the files opened using `$fopen`. The files must have been previously opened from a Verilog description. The parameter `multi_chann_desc` is an integer with the same meaning as the multi-channel descriptor used with `$fdisplay` and related system tasks.

4.4.43 mc_scan_plusargs

Purpose

scan command line plus (+) options

Synopsis

```
char *mc_scan_plusargs(startarg)
    char *startarg;
```

Scans all command arguments and matches given string to a plus argument. Returns null if `startarg` is not found. If `startarg` is found, then returns remaining part of command argument (e.g., if the argument string is “`+siz64`”, and `startarg` is “`siz`”, then “`64`” is returned). If there is no remaining part of a found plus argument, then a pointer to the C string null terminator is returned. The match is case sensitive.

4.4.44 tf_getnextlongtime

Purpose

get next time at which a simulation event is scheduled

Synopsis

```
int tf_getnextlongtime(aof_lowtime, aof_hightime)
    int *aof_lowtime, *aof_hightime;
```

Assigns the double time of the next simulation event to `aof_lowtime` and `aof_hightime`. If not called in read-only synchronize mode (`misctf` call with `reason_rosynch`), then the current time is always assigned. The time is expressed in the timescale units of the system task or function that calls the utility (see Section 4.3).

The routine returns one of the following:

0 for ok
1 for no more future events to execute - assigning 0 time.
2 for not in read-only synchronize mode - assigning current time.

Please note: case 2 overrides both case 0 and 1.

Appendix A
The Contents of acc_user.h

```
/*** File acc_user.c ***/
/*** This file is to be included in files which call access routines ***/

#define ACCUSERH 1

/*********************************************************************/
/*** General constant definitions ***/

#ifndef ACCH

typedef int *HANDLE;
typedef int *handle;

#define bool int
#define true 1
#define TRUE 1
#define false 0
#define FALSE 0

#define global extern
#define exfunc
#define local static
#define null 0L

extern bool acc_error_flag;

#endif

/*********************************************************************/
/*** Type and configuration constant definitions ***/

#define accModule 20
#define accScope 21
#define accNet 25
#define accReg 30
#define accRegister 30
#define accPort 35
#define accTerminal 45
#define accInputTerminal 46
#define accOutputTerminal 47
#define accInoutTerminal 48
#define accCombPrim 140
#define accSeqPrim 142
```

```
#define accAndGate 144
#define accNandGate 146
#define accNorGate 148
#define accOrGate 150
#define accXorGate 152
#define accXnorGate 154
#define accBufGate 156
#define accNotGate 158
#define accBufif0Gate 160
#define accBufif1Gate 162
#define accNotif0Gate 164
#define accNotif1Gate 166
#define accNmosGate 168
#define accPmosGate 170
#define accCmosGate 172
#define accRnmosGate 174
#define accRpmosGate 176
#define accRcmosGate 178
#define accRtranGate 180
#define accRtranif0Gate 182
#define accRtranif1Gate 184
#define accTranGate 186
#define accTranif0Gate 188
#define accTranif1Gate 190
#define accPullupGate 192
#define accPulldownGate 194
#define accIntegerParam 200
#define accIntParam 200
#define accRealParam 202
#define accStringParam 204
#define accPath 206
#define accTchk 208
#define accPrimitive 210
#define accBit 212
#define accPortBit 214
#define accNetBit 216
#define accRegBit 218
#define accParameter 220
#define accSpecparam 222
#define accTopModule 224
#define accModuleInstance 226
#define accCellInstance 228
#define accModPath 230
#define accPrimPath 232
#define accWirePath 234
#define accModNetPath 236 /*alias for accInterModPath*/
#define accInterModPath 236
#define accTermPath 238
#define accModTermPath 240
#define accTermModPath 242
#define accScalarPort 250
```

```
#define accBitSelectPort 252
#define accPartSelectPort 254
#define accVectorPort 256
#define accConcatPort 258
#define accWire 260
#define accWand 261
#define accWor 262
#define accTri 263
#define accTriand 264
#define accTrior 265
#define accTri0 266
#define accTri1 267
#define accTrireg 268
#define accSupply0 269
#define accSupply1 270
#define accNamedEvent 280
#define accEventVar 280
#define accIntegerVar 281
#define accIntVar 281
#define accRealVar 282
#define accTimeVar 283
#define accScalar 300
#define accVector 302
#define accCollapsedNet 304
#define accExpandedVector 306
#define accProtected 308
#define accVlogSimPath 310
#define accExpandedPath 312
#define accSwXlInvisibleNet 314
#define accAcceleratedNet 316
#define accSetup 366
#define accHold 367
#define accWidth 368
#define accPeriod 369
#define accRecovery 370
#define accSkew 371
#define accNochange 376
#define accNoChange 376
#define accSetuphold 377
#define accInput 402
#define accOutput 404
#define accInout 406
#define accPositive 408
#define accNegative 410
#define accUnknown 412
#define accPathTerminal 420
#define accPathInput 422
#define accPathOutput 424
#define accDataPath 426
#define accTchkTerminal 428
#define accBitSelect 500
```

```
#define accPartSelect 502
#define accTask 504
#define accFunction 506
#define accStatement 508
#define accTaskCall 510
#define accFunctionCall 512
#define accSystemTask 514
#define accSystemFunction 516
#define accSystemRealFunction 518
#define accUserTask 520
#define accUserFunction 522
#define accUserRealFunction 524
#define accAssignmentStat 526
#define accContAssignStat 527
#define accNullStat 528
#define accDelayStat 530
#define accAssignDelayStat 532
#define accRtlDelayStat 534
#define accAssignEventStat 536
#define accAssignMultiStat 537
#define accRtlEventStat 538
#define accRtlMultiStat 539
#define accGenEventStat 540
#define accDisableStat 542
#define accAssignStat 544
#define accDeassignStat 546
#define accForceStat 548
#define accReleaseStat 550
#define accInitialStat 552
#define accAlwaysStat 554
#define accAtEventStat 556
#define accUnnamedBeginStat 558
#define accNamedBeginStat 560
#define accUnnamedForkStat 562
#define accNamedForkStat 564
#define accIfStat 566
#define accCaseStat 568
#define accCaseZStat 570
#define accCaseXStat 572
#define accForeverStat 574
#define accRepeatStat 576
#define accWhileStat 578
#define accForStat 580
#define accWaitStat 582
#define accConstant 600
#define accConcat 610
#define accOperator 620

/* acc_configure() parameters */
#define accPathDelayCount 1
#define accPathDelimStr 2
```

```
#define accDisplayErrors 3
#define accDefaultAttr0 4
#define accToHiZDelay 5
#define accEnableArgs 6
#define accSpecitemScope 7
#define accDisplayWarnings 8
#define accWarnNestedLoconn 9
#define accWarnNestedHiconn 10
#define accDevelopmentVersion 11
#define accMinMultiplier 12
#define accTypMultiplier 13
#define accMaxMultiplier 14
#define accAttrDelimStr 15
#define accDelayCount 16
#define accMapToMipd 17
#define accDelayArrays 18
#define accMinTypMaxDelays 19
#define accUserErrorString 20

/* Edge information used by acc_handle_tchk, etc. */
#define accNoedge 0
#define accNoEdge 0
#define accEdge01 1
#define accEdge10 2
#define accEdge0x 4
#define accEdgex1 8
#define accEdge1x 16
#define accEdgex0 32
#define accPosedge 13 /* accEdge01 & accEdge0x & accEdgex1 */
#define accPosEdge 13 /* accEdge01 & accEdge0x & accEdgex1 */
#define accNegedge 50 /* accEdge10 & accEdge1x & accEdgex0 */
#define accNegEdge 50 /* accEdge10 & accEdge1x & accEdgex0 */

/* Product types */
#define accVerilog 1

/* Version defines */
#define accVersion15Beta 1
#define accVersion15a 2
#define accVersion15b 3
#define accVersion15b1 4
#define accVersion15b2Beta 5
#define accVersion15b2 6
#define accVersion15b3 7
#define accVersion15b4 8
#define accVersion15b5 9
#define accVersion15cBeta 12
#define accVersion15c 16
#define accVersion15c03 20
#define accVersion15c04 21
#define accVersion15c10 24
```

```
#define accVersion15c30 28
#define accVersion15c40 32
#define accVersion15c41 33
#define accVersion16Beta 36
#define accVersion16Beta2 37
#define accVersion16Beta3 38
#define accVersion16Beta4 39
#define accVersion16 40
#define accVersion161 41
#define accVersion16aBeta 42
#define accVersion16a 44
#define accVersion16b 48
#define accVersionLatest accVersion16aBeta

/* Delay modes */
#define accDelayModeNone 0
#define accDelayModePath 1
#define accDelayModeDistrib 2
#define accDelayModeUnit 3
#define accDelayModeZero 4


/*****************************
** typedefs for time structure*/

typedef struct t_acc_time {
 int type; /* one of accTime accSimTime accRealTime */
 int low, high; /* for accTime and accSimTime */
 double real; /* for accRealTime */
} s_acc_time, *p_acc_time;

/* t_acc_time types */
#define accTime 1 /* timescaled time */
#define accSimTime 2 /* internal simulation time */
#define accRealTime 3 /* timescaled real time */

/*****************************************
** typedefs and defines for acc_set_value()*/

typedef struct t_setval_delay {
 s_acc_time time;
 int model;
 /* accNoDelay */
 /* accInertialDelay */
 /* accTransportDelay */
 /* accPureTransportDelay */
} s_setval_delay, *p_setval_delay;


/* t_setval_delay types */
#define accNoDelay 0
```

```
#define accInertialDelay 1
#define accTransportDelay 2
#define accPureTransportDelay 3


typedef struct t_setval_value {
 int format; /* acc[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real]Val */
 union {
 char *str;
 int scalar; /* acc[0,1,X,Z] */
 int integer;
 double real;
 } value;
} s_setval_value, *p_setval_value,
 s_acc_value, *p_acc_value;

/* t_setval_value fromats */
#define accBinStrVal 1
#define accOctStrVal 2
#define accDecStrVal 3
#define accHexStrVal 4
#define accScalarVal 5
#define accIntVal 6
#define accRealVal 7
#define accStringVal 8
#define accCompactVal 9

/* scalar values */
#define acc0 0
#define acc1 1
#define accX 2
#define accZ 3


/**********************************************************************/
/*
 * includes for Value Change Link
 */

#define logic_value_change 1
#define strength_value_change 2
#define real_value_change 3
#define vector_value_change 4
#define event_value_change 5
#define integer_value_change 6
#define time_value_change 7
#define sregister_value_change 8
#define vregister_value_change 9
#define realtime_value_change 10
#define compact_value_change 11
```

```
typedef void (*consumer_function) ();

/* structure that stores strengths */
typedef struct t_strengths {
 unsigned char logic_value;
 unsigned char strength1;
 unsigned char strength2;
} s_strengths, *p_strengths;


typedef struct t_vc_record{
 int vc_reason;
 int vc_hightime;
 int vc_lowtime;
 char *user_data;
 union {
 unsigned char logic_value;
 double real_value;
 handle vector_handle;
 s_strengths strengths_s;
 } out_value;
} s_vc_record, *p_vc_record;

/* logic values */
#define vcl0 acc0
#define vcl1 acc1
#define vclX accX
#define vclZ accZ

/* VCL strength values */
#define vclSupply 7
#define vclStrong 6
#define vclPull 5
#define vclLarge 4
#define vclWeak 3
#define vclMedium 2
#define vclSmall 1
#define vclHighZ 0

/* vcl bit flag definitions */
#define vcl_strength_flag 1
#define vcl_verilog_flag 2
#define vcl_compact_flag 8


/* flags used with acc_vcl_add */
#define vcl_verilog_logic (vcl_verilog_flag)
#define VCL_VERILOG_LOGIC (vcl_verilog_flag)

#define vcl_verilog_strength (vcl_verilog_flag + vcl_strength_flag)
```

```
#define VCL_VERILOG_STRENGTH (vcl_verilog_flag + vcl_strength_flag)

/* flags used with acc_vcl_delete */
#define vcl_verilog (vcl_verilog_flag)
#define VCL_VERILOG (vcl_verilog_flag)

/* test whether strength information is requested for vcl */
#define vcl_setstr_m(flags_) ( flags_ |= vcl_strength_flag )
#define vcl_clearstr_m(flags_) ( flags_ &= ~vcl_strength_flag )
#define vcl_isstr_m(flags_) ( flags_ & vcl_strength_flag )

/* test whether Verilog information is requested for vcl */
#define vcl_setvl_m(flags_) ( flags_ |= vcl_verilog_flag )
#define vcl_clearvl_m(flags_) ( flags_ &= ~vcl_verilog_flag )
#define vcl_isvl_m(flags_) ( flags_ & vcl_verilog_flag )

/* test whether vcl trigger is compact or normal */
#define vcl_setcompact_m(flags_) ( flags_ |= vcl_compact_flag )
#define vcl_clearcompact_m(flags_) ( flags_ &= ~vcl_compact_flag )
#define vcl_iscompact_m(flags_) ( flags_ & vcl_compact_flag )

/**********************************************************************/
/*** includes for the location structure ***/
/* structure that stores location */
typedef struct t_location {
 int line_no;
 char *filename;
} s_location, *p_location;

/**********************************************************************/
/*** includes for the time callbacks ***/
#define reason_begin_of_simtime 1
#define reason_end_of_simtime 2

/**********************************************************************/
/*
 * include information for stability checks
 */

/* defines for positions */
#define acc_taskfunc_stable 0x0001
#define acc_systf_stable 0x0002
#define acc_primitive_stable 0x0004
#define acc_contassign_stable 0x0008
#define acc_behav_stable 0x0010
#define acc_netreg_stable 0x0020

/* for setting stability integer */
#define acc_setstabflags_m(flags_,pos) (flags_ |= pos)
#define acc_clearstabflags_m(flags_,pos) (flags_ &= ~pos)
#define acc_isstabflags_m(flags_,pos) (flags_ & pos)
```

```
/*********************************************************************/
/*** Routine declarations ***/

#ifndef ACCH

/* Handle routines */
handle acc_handle_object();
handle acc_handle_port();
handle acc_handle_terminal();
handle acc_handle_parent();
handle acc_handle_conn();
handle acc_handle_tchk();
handle acc_handle_pathout();
handle acc_handle_pathin();
handle acc_handle_tchkarg1();
handle acc_handle_tchkarg2();
handle acc_handle_modpath();
handle acc_handle_path();
handle acc_handle_loconn();
handle acc_handle_hiconn();
handle acc_handle_datapath();
handle acc_handle_condition();
handle acc_handle_by_name();
handle acc_handle_scope();

/* Nexts routines */
handle acc_next_terminal();
handle acc_next_port();
handle acc_next_child();
handle acc_next_driver();
handle acc_next_load();
handle acc_next_primitive();
handle acc_next_net();
handle acc_next_topmod();
handle acc_next_loconn();
handle acc_next_hiconn();
handle acc_next_modpath();
handle acc_next_path(); /* alias for acc_next_modpath */
handle acc_next_cell();
handle acc_next_cell_load();
handle acc_next_tchk();
handle acc_next_parameter();
handle acc_next_specparam();
handle acc_next_portout();
handle acc_next_bit();
handle acc_next();
handle acc_next_input();
handle acc_next_output();
```

```
/* Fetch routines */
char *acc_fetch_value();
char *acc_fetch_name();
char *acc_fetch_fullname();
char *acc_fetch_defname();
int acc_fetch_type();
int acc_fetch_fulltype();
char *acc_fetch_type_str();
int acc_fetch_index();
int acc_fetch_direction();
bool acc_fetch_delays();
double acc_fetch_paramval();
double acc_fetch_attribute();
int acc_fetch_polarity();
int acc_fetch_paramtype();
int acc_fetch_size();
int acc_fetch_range();
int acc_fetch_edge();
void acc_fetch_location();

/* Modify routines */
bool acc_replace_delays();
bool acc_append_delays();

/* Utility routines */
bool acc_initialize();
void acc_close();
bool acc_configure();
int acc_count();
handle *acc_collect();
char *acc_version();
void acc_free();
handle acc_handle_tfarg();
double acc_fetch_tfarg();
bool acc_compare_handles();
char *acc_set_scope();
int acc_release_object();
bool acc_object_of_type();
bool acc_object_in_typelist();
int acc_set_value();

/* Value Change Link routines */
void acc_vcl_add();
void acc_vcl_delete();



#endif
/**********************************************************************/
```

Master Index

U

V

W