20. PLI overview

20.1 PLI purpose and history (informative)

Clause 20 through Clause 27 and Annex E through Annex G describe the C language procedural interface standard and interface mechanisms that are part of the Verilog HDL. This procedural interface, known as the Programming Language Interface, or PLI, provides a means for Verilog HDL users to access and modify data in an instantiated Verilog HDL data structure dynamically. An instantiated Verilog HDL data structure is the result of compiling Verilog HDL source descriptions and generating the hierarchy modeled by module instances, primitive instances, and other Verilog HDL constructs that represent scope. The PLI procedural interface provides a library of C language functions that can directly access data within an instantiated Verilog HDL data structure.

A few of the many possible applications for the PLI procedural interface are:

- C language delay calculators for Verilog model libraries that can dynamically scan the data structure of a Verilog software product and then dynamically modify the delays of each instance of models from the library
- C language applications that dynamically read test vectors or other data from a file and pass the data into a Verilog software product
- Custom graphical waveform and debugging environments for Verilog software products
- Source code decompilers that can generate Verilog HDL source code from the compiled data structure of a Verilog software product
- Simulation models written in the C language and dynamically linked into Verilog HDL simulations
- Interfaces to actual hardware, such as a hardware modeler, that dynamically interact with simulations

This document standardizes the Verilog PLI that has been in use since the mid-1980s. This standard comprises three primary generations of the Verilog PLI.

a) *Task/function* routines, called *TF* routines, make up the first generation of the PLI. These routines, most of which start with the characters **tf_**, are primarily used for operations involving user-defined task/function arguments, along with utility functions, such as setting up call-back mechanisms and writing data to output devices. The TF routines are sometimes referred to as *utility* routines

b) *Access* routines, called *ACC* routines, form the second generation of the PLI. These routines, which all start with the characters **acc_**, provide an object-oriented access directly into a Verilog HDL structural description. ACC routines are used to access and modify information, such as delay values and logic values on a wide variety of objects that exist in a Verilog HDL description. There is some overlap in functionality between ACC routines and TF routines.

c) *Verilog Procedural Interface* routines, called *VPI* routines, are the third generation of the PLI. These routines, all of which start with the characters **vpi_**, provide an object-oriented access for both Verilog HDL structural and behavioral objects. The VPI routines are a superset of the functionality of the TF routines and ACC routines.

20.2 User-defined system task or function names

A user-defined system task or function name is the name that will be used within a Verilog HDL source file to invoke specific PLI applications. The name shall adhere to the following rules:

- The first character of the name shall be the dollar sign character ($)
- The remaining characters shall be letters, digits, the underscore character (_) or the dollar character ($)
- Uppercase and lowercase letters shall be considered to be unique the name is case sensitive
- The name can be any size, and all characters are significant

Copyright © 2001 IEEE. All rights reserved.

20.3 User-defined system task or function types

The type of a user-defined system task or function determines how a PLI application is called from the Verilog HDL source code. The types are:

A user *task* can be used in the same places a Verilog HDL task can be used (refer to 10.2). A user-defined system task can read and modify the arguments of the task, but does not return any value.
A user *function* can be used in the same places a Verilog HDL function can be used (refer to 10.3). A user-defined system function can read and modify the arguments of the function, and shall return a scalar or vector value. The bit width of the return value shall be determined by a user-supplied *sizetf* application (see 21.1.1).
A user *real-function* can be used in the same places a Verilog HDL function can be used (refer to 10.3). A user-defined system real-function can read and modify the arguments of the function, and will return a double-precision floating point value.

20.4 Overriding built-in system task and function names

Clause 17 defines a number of built-in system tasks and functions that are part of the Verilog language. In addition, software products can include other built-in system tasks and functions specific to the product. These built-in system task and function names begin with the dollar sign character ($) just as user-defined system task and function names.

If a user-provided PLI application is associated with the same name as a built-in system task or function (using the PLI interface mechanism), the user-provided C application shall override the built-in system task/function, replacing its functionality with that of the user-provided C application. For example, a user could write a random number generator as a PLI application and then associate the application with the name **$random**, thereby overriding the built-in **$random** function with the user s application.

Verilog timing checks, such as **$setup**, are not system tasks, and cannot be overridden.

The system functions **$signed** and **$unsigned** can be overridden. These system functions are unique in the Verilog HDL, in that the return width is based on the width of their argument. If overridden, the PLI version shall have the same return width for all instances of the system function. The PLI return width is defined by the PLI *sizetf* routine.

20.5 User-supplied PLI applications

User-supplied PLI applications are C language functions that utilize the library of PLI C functions to access and interact dynamically with Verilog HDL software implementations as the Verilog HDL source code is executed.

These PLI applications are not independent C programs. They are C functions, which are linked into a software product, and become part of the product. This allows the PLI application to be called when the user-defined system task or function **$** name is compiled or executed in the Verilog HDL source code.

20.6 PLI interface mechanism

The PLI interface mechanism provides a means to have PLI applications called for various reasons when the associated system task or function **$** name is encountered in the Verilog HDL source description. For example, when a Verilog HDL simulator first compiles the Verilog HDL source description, a specific PLI application can be called that performs syntax checking to ensure the user-defined system task or function is being used correctly. Then, as simulation is executing, a different PLI application can be called to perform the operations required by the PLI application. Other PLI applications can be automatically called by the simulator for miscellaneous reasons, such as the end of a simulation time step or a logic value change on a specific signal.

 Copyright © 2001 IEEE. All rights reserved.

The PLI provides two interface mechanisms:

The TF and ACC interface mechanism is an older interface, which can be used to associate PLI applications which use routines from the ACC and TF function libraries. This interface mechanism is described in Clause 21.

The VPI interface mechanism is a newer interface, which can be used to associate PLI applications which use routines from the VPI function libraries. This interface mechanism is described in Clause 26.

Instances of system tasks and functions which are defined using the TF and ACC interface mechanism can only be accessed using the TF and ACC function libraries. Instances of system tasks and functions which are defined using the VPI interface mechanism can only be accessed using the VPI function library.

20.7 User-defined system task and function arguments

When a user-defined system task or function is used in a Verilog HDL source file, it can have arguments that can be used by the PLI applications associated with the system task or function. In the following example, the user-defined system task `$get_vector` has two arguments:

```
$get_vector("test_vector.pat", input_bus);
```

The arguments to a system task or function are referred to as *task/function arguments* (often abbreviated as *tfargs*). These arguments are not the same as C language arguments. When the PLI applications associated with a user-defined system task or function are called, the task/function arguments are not passed to the PLI application. Instead, a number of PLI routines are provided that allow the PLI applications to read and write to the task/function arguments. Refer to the sections on ACC routines, TF routines and VPI routines for information on specific routines that work with task/function arguments.

20.8 PLI include files

The libraries of PLI functions are defined in C include files, which are a normative part of the 1364 standard. These files also define constants, structures, and other data used by the library of PLI routines and the interface mechanisms. The files are `acc_user.h` (listed in Annex E), `veriuser.h` (listed in Annex F) and `vpi_user.h` (listed in Annex G).

PLI applications that use the TF routines shall include the file `veriuser.h`.
PLI applications that use the ACC routines shall include the file `acc_user.h`.
PLI applications that use the VPI routines shall include the file `vpi_user.h`.

20.9 PLI Memory Restrictions

Memory allocated by the PLI routines is not to be modified by the user, with the exception of the value storage returned by the PLI routines `tf_exprinfo()` and `tf_nodeinfo()`, as defined in 25.15 and 25.35.

Copyright © 2001 IEEE. All rights reserved.

21. PLI TF and ACC interface mechanism

The interface mechanism described in this section provides a means for users to link applications based on PLI task/function (TF) routines and access (ACC) routines to Verilog software products. Through the interface mechanism, a user can:

- Specify a user-defined system task or function name that can be included in Verilog HDL source descriptions; the user-defined system task or function name shall begin with a dollar sign ($), such as **$get_vector**
- Provide one or more PLI C applications to be called by a software product (such as a logic simulator)
- Define which PLI C applications are to be called and when the applications should be called when the user-defined system task or function name is encountered in the Verilog HDL source description
- Define whether the PLI applications should be treated as *function*s (which return a value) or *task*s (analogous to subroutines in other programming languages)
- Define a data argument to be passed to the PLI applications each time they are called

NOTE The PLI interface mechanism described in this section does not apply to applications that use the Verilog Procedural Interface (VPI) routines; these routines use the VPI registry mechanism described in Clause 26 and Clause 27.

21.1 User-supplied PLI applications

User-supplied PLI applications are C language functions that utilize the library of PLI C functions to access and interact dynamically with Verilog HDL software implementations as the Verilog HDL source code is executed.

These PLI applications are not independent C programs. They are C functions, which are linked into a software product, and become part of the product. This allows the PLI application to be called when the user-defined system task or function **$** name is compiled or executed in the Verilog HDL source code.

The PLI interface mechanism for TF and ACC routines provides five classes of user-supplied PLI applications: *checktf* applications, *sizetf* applications, *calltf* applications, *misctf* applications, and *consumer* applications. The *sizetf*, *checktf*, *calltf*, and *misctf* routines are called during specific periods during processing. The purpose of each of the PLI application classes is explained in the following subsections.

21.1.1 The sizetf class of PLI applications

A *sizetf* PLI application can be used in conjunction with user-defined system *functions*. A function shall return a value, and software products that execute the system function may need to determine how many bits wide that return value shall be. The sizetf application may be called early in the process, prior to a complete instantiation of the design. As a result, access to objects may be limited at this time. Each sizetf function shall be called at most once. It shall be called if its associated system function appears in the design. The value returned by the sizetf function shall be the number of bits that the calltf routine shall provide as the return value for the system function. If no sizetf application is specified for a user-defined system function, the function shall return 32-bits. The sizetf application shall not be called for user-defined system tasks or real-functions.

21.1.2 The checktf class of PLI applications

A *checktf* PLI application shall be called when the user-defined system task or function name is encountered during parsing or compiling the Verilog HDL source code. This application is typically used to check the correctness of any arguments used with the system task in the Verilog HDL source code. The checktf PLI application shall be called one time for each instance of a system task or function in the source description. Providing a checktf application is optional, but it is recommended that any arguments used with the system task or function be checked for correctness to avoid problems when the calltf or other PLI applications read and perform operations on the arguments. The checktf shall be called at the earliest possible time after all

 Copyright © 2001 IEEE. All rights reserved.

simulation data structures required by the PLI are available. Generally this means after the design is fully instantiated, but no simulation events have occurred. By the time the checktf application is called, all PLI routines can be used without concern for the state of the process, with the exception of setting the return value of user defined system functions. The return value of a user defined system function can only be set during a calltf application.

21.1.3 The calltf class of PLI applications

A *calltf* PLI application shall be called each time the associated user-defined system task or function is executed within the Verilog HDL source code. For example, the following Verilog loop would call the PLI calltf application that is associated with the $get_vector user-defined system task name 1024 times:

```
for (i = 1; i <= 1024; i = i + 1)
   @(posedge clk) $get_vector("test_vector.pat", input_bus);
```

In this example, the user-supplied PLI calltf application might read a test vector from a file called `test_vector.pat` (the first task/function argument), perhaps manipulate the vector to put it in a proper format for Verilog, and then assign the vector value to the second task/function argument called input_bus.

21.1.4 The misctf class of PLI applications

A *misctf* PLI application shall be called by a Verilog software product for miscellaneous reasons while the Verilog HDL source description is being executed. Among these reasons can be the end of a simulation time step, a logic value change on a user-defined system task/function argument, or the execution of the **$stop** and **$finish** built-in system tasks. When the software product calls the misctf PLI application, it shall pass in a reason argument, which can be used within the misctf application to determine why the application was called. The reason argument shall be a predefined integer constant. Table 87 and Table 88 list the reasons for which the misctf application can be called.

For most reasons, the misctf routine will not be called until the instance of the system task has been executed (at which point the calltf routine is called). The following reasons are exceptions, and will be called for each instance of the system task in the design regardless of whether or not it has been executed:

reason_endofcompile
reason_save
reason_startofsave
reason_restart
reason_endofreset
reason_reset

21.1.5 The consumer class of PLI applications

A *consumer* PLI application shall be called through a PLI callback mechanism referred to as the Value Change Link (VCL). Using the VCL, another PLI application, typically the calltf application, can place VCL flags on objects within the Verilog HDL data structure, such as a specific net. Whenever an object with a VCL flag changes value during a simulation, the consumer PLI application shall be called and passed information about the change.

21.2 Associating PLI applications to a class and system task/function name

Each user-provided PLI application is a standard C language function that makes use of the library of PLI functions. These user-provided PLI applications shall be associated with both the class of application (such as calltf or checktf) and the user-defined system task or function $ name. In addition, the user-defined name shall be declared as either a system task or a system function.

Copyright © 2001 IEEE. All rights reserved.

For the TF and ACC interface mechanism, the method of associating PLI applications with a class and system task/function name is not defined as part of this standard. Each software product vendor shall define an association mechanism specific to their product. Refer to the documentation provided by the vendor for instructions on associating PLI applications to classes and system task/function names and then linking the PLI applications into the software products of the vendor.

21.3 PLI application arguments

When the calltf, checktf, and sizetf PLI applications are called by a Verilog software implementation, they shall be passed two C arguments, *data* and *reason*, in that order. When the misctf application is called, it shall be passed three C arguments, *data*, *reason*, and *paramvc*, in that order. These arguments are defined in more detail in the following subsections.

21.3.1 The data C argument

The *data* C argument shall be an integer value. The value is defined by the user at the time the PLI applications are associated with a user-defined system task/function name. This value can be used to allow several different system task/function names to use the same calltf, checktf, sizetf, or misctf applications. To do this, each system task/function name would be associated with the same PLI applications, but each would have a different value for the user-defined data argument. When a PLI application is called, it can then check the value of the data argument to determine which system task/function name was used to call the application.

21.3.2 The reason C argument

The *reason* C argument shall be a predefined integer constant that is passed to the calltf, checktf, sizetf, and misctf applications each time the applications are called. Generally, the calltf, checktf, and sizetf applications do not need to check the reason argument, since these applications can only be called under specific circumstances. The misctf application, however, can be called for a wide variety of reasons, and therefore it should always examine the reason argument to determine why the application was called. The value for the reason argument is defined in the PLI include file veriuser.h. The reason constant that is passed is based on the class of the PLI application, as follows:

The calltf application is passed the reason constant `reason_calltf`.
The checktf application is passed the reason constant `reason_checktf`.
The sizetf application is passed the reason constant `reason_sizetf`.
The misctf application is passed one of the constants listed in Table 87. Software implementations can define additional reason constants to be passed to the misctf application. Table 88 lists some common reason constants that can be available in some software implementations.

Table 87—(normative) Predefined misctf reason constants

Integer constant	Reason
reason_endofcompile	end of Verilog source compilation/start of execution
reason_paramvc	a change of value on a user-defined system task or function argument
reason_synch	end of a time step flagged by **tf_synchronize()**
reason_rosynch	end of a time step flagged by **tf_rosynchronize()**
reason_reactiviate	a simulation event scheduled by **tf_setdelay()**
reason_finish	the **$finish()** built-in system task executed

 Copyright © 2001 IEEE. All rights reserved.

Table 88—(informative) Additional misctf reason constants

Integer constant	Reason
reason_paramdrc	a value change on the driver of a user-defined system task or function argument
reason_force	execution of a procedural force or procedural continuous assignment on any net, reg, integer variable, time variable or real variable
reason_release	execution of a procedural release or procedural deassign on any net, reg, integer variable, time variable or real variable
reason_disable	execution of a procedural disable statement
reason_interactive	execution of the **$stop()** built-in system task
reason_scope	execution of the **$scope()** built-in system task
reason_startofsave	start of execution of the **$save()** built-in system task
reason_save	completion of execution of the **$save()** built-in system task
reason_restart	execution of the **$restart()** built-in system task
reason_reset	start of execution of the **$reset()** built-in system task
reason_endofreset	completion of execution of the **$reset()** built-in system task

21.3.3 The paramvc C argument

The *paramvc* C argument shall be an integer value passed to the misctf application. The value of paramvc shall indicate which task/function argument changed value when the misctf application was called back after activating the utility routine **tf_asynchon()**. This routine shall cause the misctf application to be called with a reason argument of **reason_paramvc** or **reason_paramdrc**. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

Copyright © 2001 IEEE. All rights reserved.

22. Using ACC routines

This clause presents a general discussion of how and why to use PLI ACC routines. Clause 23 defines the ACC routine syntax, listed in alphabetical order.

22.1 ACC routine definition

ACC routines are C programming language functions that provide procedural access to information within the Verilog HDL.

ACC routines perform one of two functions:

a) Read data about particular objects in the Verilog HDL description directly from internal data structures.

b) Write new information about certain objects in the Verilog HDL description into the internal data structures.

ACC routines shall read information about the following objects:

Module instances
Module ports
Module or data paths
Intermodule paths
Top-level modules
Primitive instances
Primitive terminals
Nets
Reg variables
Parameters
Specparams
Timing checks
Named events
Integer, real, and time variables

ACC routines shall read and write information on the following objects:

Intermodule path delays
Module path delays
Module input port delays (MIPDs)
Primitive instance delays
Timing check limits
Reg variable logic values
Net logic values (force/release only)
Integer, real and time variable values
Sequential UDP logic values

22.2 The handle data type

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

 Copyright © 2001 IEEE. All rights reserved.

Most ACC routines require a handle argument to indicate the objects about which they need to read or write information. The PLI provides two categories of ACC 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 22.4.2 for a discussion of handle routines and 22.4.3 for more information about next routines.

Handles shall be passed to and from ACC routines through *handle variables*. To declare a handle variable, the keyword `handle` (all lowercase) shall be used, followed by the variable name, as in this example:

```
handle  net_handle;
```

After declaring a handle variable, it can be passed to any ACC routine that requires a handle argument or be used to receive a handle returned by an ACC routine. The following C language code fragment uses the variable `net_handle` to store the handle returned by the ACC routine **acc_handle_object()**:

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

22.3 Using ACC routines

22.3.1 Header files

The header file `acc_user.h` shall be included in any C language source file containing an application program that calls ACC routines. The `acc_user.h` file is listed in Annex E.

22.3.2 Initializing ACC routines

The ACC routine **acc_initialize()** shall initialize the environment for ACC routines and shall be called from the C language application program before the program invokes any other ACC routines.

22.3.3 Exiting ACC routines

Before exiting a C language application program that calls ACC routines, the ACC routine **acc_close()** should be called. This routine shall reset ACC routine configuration parameters back to their defaults, and it shall also free memory allocated by the ACC routines.

22.4 List of ACC routines by major category

The ACC routines are divided into the following major categories:

Fetch routines
Handle routines
Next routines
Modify routines
VCL routines
Miscellaneous routines

This section contains a summary list of each major category. The ACC routines sorted by the types of objects they work with are listed in 22.5. Clause 23 presents an alphabetical list of all ACC routines, with their functions, syntax, and usage.

Copyright © 2001 IEEE. All rights reserved.

22.4.1 Fetch routines

Fetch routines shall return a variety of information about different objects in the design hierarchy. 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 specparam.

Table 89—List of fetch routines

ACC routine	Description
acc_fetch_argc()	Get the number of invocation command line arguments
acc_fetch_argv()	Get the invocation command line arguments
acc_fetch_attribute()	Get the value of a Verilog parameter or specparam as a double
acc_fetch_attribute_int()	Get the value of a Verilog parameter or specparam as an integer
acc_fetch_attribute_str()	Get the value of a Verilog parameter or specparam as a string
acc_fetch_defname()	Get the definition name of a module or primitive
acc_fetch_delay_mode()	Get the delay mode of a module instance
acc_fetch_delays()	Get the existing delays for a primitive, module path, timing check, intermodule path, or module input port
acc_fetch_direction()	Get the direction of a module port or primitive terminal
acc_fetch_edge()	Get the edge specifier of a module path input terminal
acc_fetch_fullname()	Get the full hierarchical name of an object
acc_fetch_fulltype()	Get the full type description of an object as a predefined integer constant
acc_fetch_index()	Get the index number of a port or terminal
acc_fetch_itfarg()	Get the value of an instance of a system task/function argument as a double
acc_fetch_itfarg_int()	Get the value of an instance of a system task/function argument as an integer
acc_fetch_itfarg_str()	Get the value of an instance of a system task/function argument as a string
acc_fetch_location()	Get the location of an object in a Verilog source file
acc_fetch_name()	Get the local name of an object
acc_fetch_paramtype()	Get the data type of a parameter or specparam
acc_fetch_paramval()	Get the value of a parameter or specparam
acc_fetch_polarity()	Get the polarity of a module path or data path
acc_fetch_precision()	Get the simulation time precision
acc_fetch_pulsere()	Get the current pulse handling values of a module path, intermodule path or module input port
acc_fetch_range()	Get the range of a vector
acc_fetch_size()	Get the bit size of a vector or port
acc_fetch_tfarg()	Get the value of a system task/function argument as a double
acc_fetch_tfarg_int()	Get the value of a system task/function argument as an integer
acc_fetch_tfarg_str()	Get the value of a system task/function argument as a string
acc_fetch_timescale_info()	Get the timescale information for an object
acc_fetch_type()	Get the general type classification of an object as an integer constant
acc_fetch_type_str()	Get the string representation of a type or fulltype integer constant
acc_fetch_value()	Get the logic or strength value of a net, reg, integer variable, time variable or real variable

 Copyright © 2001 IEEE. All rights reserved.

22.4.2 Handle routines

Handle routines can return handles to a variety of objects in the design hierarchy. 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 a named object, while **acc_handle_conn()** retrieves a handle for a net connected to a particular terminal. Each handle routine shall return a handle to an object. This handle can, in turn, be passed as an argument to other ACC routines.

Table 90—List of handle routines

ACC routine	Description
acc_handle_by_name()	Get the handle to any named object
acc_handle_condition()	Get the handle to the condition of a module path, data path, or timing check
acc_handle_conn()	Get the handle to the net connected to a primitive, path, or timing check terminal
acc_handle_datapath()	Get the handle to a data path
acc_handle_hiconn()	Get the handle to the hierarchically higher net connected to a module port bit
acc_handle_interactive_scope()	Get the handle to the current simulation interactive scope
acc_handle_itfarg()	Get the handle to an argument of a specific system task/function instance
acc_handle_loconn()	Get the handle to the hierarchically lower net connected to a module port bit
acc_handle_modpath()	Get the handle to a module path
acc_handle_notifier()	Get the handle to the notifier argument of a timing check
acc_handle_object()	Get the handle to any named object
acc_handle_parent()	Get the handle to the parent of an object
acc_handle_path()	Get the handle to an intermodule path
acc_handle_pathin()	Get the handle to the first net connected to a module path source
acc_handle_pathout()	Get the handle to the first net connected to a module path destination
acc_handle_port()	Get the handle to a module port based on the port index
acc_handle_scope()	Get the handle to the scope containing an object
acc_handle_simulated_net()	Get the handle to the net associated with a collapsed net
acc_handle_tchk()	Get the handle to a timing check
acc_handle_tchkarg1()	Get the handle to the first argument of a timing check
acc_handle_tchkarg2()	Get the handle to the second argument of a timing check
acc_handle_terminal()	Get the handle to a terminal of a primitive based on the terminal index
acc_handle_tfarg()	Get the handle to the object named in a system task/function argument
acc_handle_tfinst()	Get the handle to the current instance of a system task/function

Copyright © 2001 IEEE. All rights reserved.

22.4.3 Next routines

When used inside a C loop construct, next routines shall find each object of a given type that is related to a particular reference object in the design hierarchy. The name of each routine begins with the prefix **acc_next_** and indicates the type of object desired, known as the target object. For example, **acc_next_net()** retrieves each net in a module, while **acc_next_driver()** retrieves each terminal driving a net. Each call to a next routine returns a handle to the object it finds.

Most next routines require two arguments:

The first argument shall be a handle to a *reference object*.
The second argument shall be a handle that indicates whether to retrieve the first or next *target object*.

The *reference object* shall indicate where the next routine shall look for the target object. The *target object* is the type of object to be returned by a next routine.

Table 91 summarizes how next routines shall find each target object associated with a given reference object.

Table 91—How next routines use the target object argument

When	A next routine shall return
the *target object* is `null`	a handle to the first *target object* related to the *reference object*
the *target object* is a handle to the last *target object* returned	a handle to the next *target object* related to the *reference object*
no *target objects* remain for the *reference object*	a `null` handle
no *target objects* are found initially for the *reference object*	a `null` handle
an error occurs	a `null` handle

Each call to a next routine shall return only one handle. Therefore, to retrieve all target objects for a particular reference object, the following process can be used:

a) Chose an appropriate ACC routine to retrieve the handle of the desired reference object.

b) Set the target object handle variable to `null`. When a next routine is called with a `null` target handle, it shall return the first target associated with the reference.

c) Call the next routine, assigning the return value to the same variable as the target object argument. This automatically updates the target object argument to point to the last object found.

d) Place the next routine call inside a C `while` loop that terminates when the loop control value is `null`. When a next routine cannot access any more target objects, it shall return a `null`.

NOTE Most next routines can return objects in an arbitrary order. However, certain next routines shall return objects in a defined order, as noted in the description of the routine in Clause 23.

The following example, `display_net_names`, uses a next routine to display the names of all nets in a module.

 Copyright © 2001 IEEE. All rights reserved.

```
#include "acc_user.h"
display_net_names()
{
   handle   module_handle;
   handle   net_handle;

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

   /*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();
}
```

Table 92—List of next routines

ACC routine	Description
acc_next()	Get handles to all objects of a set of types
acc_next_bit()	Get handles to all bits of a port or vector
acc_next_cell()	Get handles to all cell modules in the current hierarchy and below
acc_next_cell_load()	Get handles to all cell loads on a net
acc_next_child()	Get handles to all module instances within a module
acc_next_driver()	Get handles to all primitive terminals that drive a net
acc_next_hiconn()	Get handles to all nets connected hierarchically higher to a module port
acc_next_input()	Get handles to all input terminals of a module path or data path
acc_next_load()	Get handles to all primitive terminals driven by a net
acc_next_loconn()	Get handles to all nets connected hierarchically lower to a module port
acc_next_modpath()	Get handles to all paths in a module
acc_next_net()	Get handles to all nets in a module
acc_next_output()	Get handles to all output terminals of a module path or data path
acc_next_parameter()	Get handles to all parameters in a module
acc_next_port()	Get handles to all ports of a module or connected to a net
acc_next_portout()	Get handles to all output ports of a module
acc_next_primitive()	Get handles to all primitive instances in a module
acc_next_scope()	Get handles to all hierarchy scopes within a scope
acc_next_specparam()	Get handles to all specify block parameters in a module
acc_next_tchk()	Get handles to all timing checks in a module
acc_next_terminal()	Get handles to all terminals of a primitive
acc_next_topmod()	Get handles to all top-level modules

Copyright © 2001 IEEE. All rights reserved.

22.4.4 Modify routines

Modify routines shall alter the values of a variety of objects in the design hierarchy. Table 93 lists the types of values that shall be modified for particular objects.

Table 93—Values that can be modified

Modify routines alter	For these objects
Delay values	Primitives Module paths Intermodule paths Module input ports Timing checks
Logic values	Variable data types Net data types Sequential UDPs
Pulse handling values	Module paths Intermodule paths Module input ports

Table 94—List of modify routines

ACC routine	Description
acc_append_delays()	Add delays to existing delays on primitives, module paths, timing checks, intermodule paths, and module input ports
acc_append_pulsere()	Add to existing pulse control values of module paths, intermodule paths and module input ports
acc_replace_delays()	Replace existing delays on primitives, module paths, timing checks, intermodule paths and module input ports
acc_replace_pulsere()	Replace existing values on pulse control values of module paths, intermodule paths and module input ports
acc_set_pulsere()	Set the pulse control values for a module path, intermodule path or module input port as a percentage of the delay
acc_set_value()	Set and propagate a logic value onto a reg, integer variable, time variable, real variable or sequential UDP; continuously assign/deassign a reg or variable; force/release a net or reg or variable

More details on using the **acc_append_delays()** and **acc_replace_delays()** ACC routines are provided in 22.8.

22.4.5 Miscellaneous routines

Miscellaneous routines shall perform a variety of operations, such as initializing and configuring the ACC routine environment.

 Copyright © 2001 IEEE. All rights reserved.

Table 95—List of miscellaneous routines

ACC routine	Description
acc_close()	Close ACC routine environment
acc_collect()	Collect an array of handles for a reference object
acc_compare_handles()	Determine if two handles are for the same object
acc_configure()	Set the ACC routine environment parameters
acc_count()	Count the number of objects related to a reference object
acc_free()	Free up memory allocated by acc_collect()
acc_initialize()	Initialize the ACC routine environment
acc_object_in_typelist()	Determine if an object matches a set of types, fulltypes, or special properties
acc_object_of_type()	Determine if an object matches a specific type, fulltype, or special property
acc_product_type()	Get the type of software product being used
acc_product_version()	Get the version of software product being used
acc_release_object()	Release memory allocated by acc_next_input() or acc_next_output()
acc_reset_buffer()	Reset the string buffer
acc_set_interactive_scope()	Set the interactive scope of a software implementation
acc_set_scope()	Set the scope used by acc_handle_object()
acc_version()	Get the version of the ACC routines being used

22.4.6 VCL routines

The VCL shall allow a PLI application to monitor simulation value changes of selected objects. It consists of two ACC routines that instruct a Verilog simulator to start or stop informing an application when an object changes value. How the VCL routine is used is discussed in 22.10.

Table 96—List of VCL routines

ACC routine	Description
acc_vcl_add()	Add a value change callback on an object
acc_vcl_delete()	Remove a value change callback

22.5 Accessible objects

ACC routines shall access information about the following objects:

Module instances
Module ports
Individual bits of a port
Module or data paths
Intermodule paths

Copyright © 2001 IEEE. All rights reserved.

Top-level modules
Primitive instances
Primitive terminals
Nets (scalars, vectors, and bit- or part-selects of vectors)
Regs (scalars, vectors, and bit- or part-selects of vectors)
Integer variables (and bit- or part-selects of integers)
Real and time variables
Named events
Parameters
Specparams
Timing checks
Timing check terminals
User-Defined system task/function arguments

The following tables summarize the operations that can be performed for each of the above object types.

 Copyright © 2001 IEEE. All rights reserved.

22.5.1 ACC routines that operate on module instances

Table 97—Operations on module instances

To	Use
Obtain handles for module instances tagged as cells within a hierarchical scope and below	**acc_next_cell()**
Obtain handles for module instances within a particular module instance	**acc_next_child()**
Obtain a handle to the parent (the module that contains the instance)	**acc_handle_parent()**
Get the instance name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the module definition name	**acc_fetch_defname()**
Get the fulltype of a module instance (cell instance, module instance, or top-level module)	**acc_fetch_fulltype()**
Get the delay mode of a module instance (none, zero, unit, distributed, or path)	**acc_fetch_delay_mode()**
Get timescale information for a module instance	**acc_fetch_timescale_info()**

22.5.2 ACC routines that operate on module ports

Table 98—Operations on module ports

To	Use
Obtain handles for ports of a module instance	**acc_next_port()**
Obtain handles for output ports of a module instance	**acc_next_portout()**
Obtain a handle for a particular port	**acc_handle_port()**
Obtain a handle to the parent (the module instance that contains the port)	**acc_handle_parent()**
Obtain handles to hierarchically higher-connected nets	**acc_next_hiconn()**
Obtain handles to hierarchically lower-connected nets	**acc_next_loconn()**
Obtain a handle to the hierarchically higher-connected net of a scalar module port or bit of a vector port	**acc_handle_hiconn()**
Obtain a handle to the hierarchically lower-connected net of a scalar module port or bit of a vector port	**acc_handle_loconn()**
Get the instance name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the port direction	**acc_fetch_direction()**
Get the port index number	**acc_fetch_index()**
Get the fulltype of a module port	**acc_fetch_fulltype()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**
Read Module Input Port Delay (MIPD)	**acc_fetch_delays()**
Append to existing MIPD	**acc_append_delays()**
Replace existing MIPD	**acc_replace_delays()**

Copyright © 2001 IEEE. All rights reserved.

22.5.3 ACC routines that operate on bits of a port

Table 99—Operations on bits of a port

To	Use
Obtain handles for bits of a module port	**acc_next_bit()**
Obtain a handle to the port from a port bit	**acc_handle_parent()**
Get the port name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the fulltype of a port s bit	**acc_fetch_fulltype()**
Read Module Input Port Delay (MIPD)	**acc_fetch_delays()**
Append to existing MIPD	**acc_append_delays()**
Replace existing MIPD	**acc_replace_delays()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

22.5.4 ACC routines that operate on module paths or data paths

Table 100—Operations on module paths and data paths

To	Use
Obtain handles for module paths within a scope	**acc_next_modpath()**
Obtain a handle to the first connected net	**acc_handle_pathin()** **acc_handle_pathout()**
Obtain a handle to a module path	**acc_handle_modpath()**
Obtain a handle to a datapath	**acc_handle_datapath()**
Obtain a handle to a conditional expression for a path	**acc_handle_condition()**
Obtain handles for input terminals of a module path or data path	**acc_next_input()**
Obtain handles for output terminals of a module path or data path	**acc_next_output()**
Get the path name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the polarity of a path	**acc_fetch_polarity()**
Get the edge specified for a path terminal	**acc_fetch_edge()**
Read path delays	**acc_fetch_delays()**
Append to existing path delays	**acc_append_delays()**
Replace existing path delays	**acc_replace_delays()**
Read path pulse handling	**acc_fetch_pulsere()**
Append to existing path pulse control values	**acc_append_pulsere()**
Replace existing path pulse control values	**acc_replace_pulsere()**
Specify path pulse control values	**acc_set_pulsere()**
Free memory allocated by acc_next_input() or acc_next_output()	**acc_release_object()**

 Copyright © 2001 IEEE. All rights reserved.

22.5.5 ACC routines that operate on intermodule paths

Table 101—Operations on intermodule paths

To	Use
Obtain a handle for an intermodule path	**acc_handle_path()**
Get the fulltype of an intermodule path	**acc_fetch_fulltype()**
Read intermodule path delays	**acc_fetch_delays()**
Modify intermodule path delays	**acc_replace_delays()**
Read intermodule path pulse control values	**acc_fetch_pulsere()**
Append to existing intermodule path pulse control values	**acc_append_pulsere()**
Replace existing intermodule path pulse control values	**acc_replace_pulsere()**
Specify intermodule path pulse control values	**acc_set_pulsere()**

22.5.6 ACC routines that operate on top-level modules

Table 102—Operations on top-level modules

To	Use
Obtain handles for top-level modules in a design	**acc_next_topmod()** **acc_next_child()**
Get the module name	**acc_fetch_name()** **acc_fetch_fullname()** **acc_fetch_defname()**

22.5.7 ACC routines that operate on primitive instances

Table 103—Operations on primitive instances

To	Use
Obtain handles for primitive instances within a module instance	**acc_next_primitive()**
Obtain a handle to the parent (the module that contains the primitive)	**acc_handle_parent()**
Get the instance name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the definition name	**acc_fetch_defname()**
Get the primitive *fulltype*	**acc_fetch_fulltype()**
Read delays	**acc_fetch_delays()**
Append to existing primitive delays	**acc_append_delays()**
Replace existing primitive delays	**acc_replace_delays()**

Copyright © 2001 IEEE. All rights reserved.

22.5.8 ACC routines that operate on primitive terminals

Table 104—Operations on primitive terminals

To	Use
Obtain handles for terminals of a primitive instance	**acc_next_terminal()**
Obtain a handle to the net connected to the terminal	**acc_handle_conn()**
Obtain a handle to the parent (primitive instance containing the terminal)	**acc_handle_parent()**
Get the direction (input, output, inout)	**acc_fetch_direction()**
Get the terminal index number	**acc_fetch_index()**
Get the fulltype	**acc_fetch_fulltype()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

22.5.9 ACC routines that operate on nets

Table 105—Operations on nets

To	Use
Obtain handles for nets within a module instance	**acc_next_net()**
Obtain handles for nets within a module instance	**acc_next()**
Obtain a handle to the parent (the module instance that contains the net)	**acc_handle_parent()**
Determine if net is scalar, vector, collapsed, or expanded	**acc_object_of_type()**
Obtain handles to bits of a vector net	**acc_next_bit()**
Obtain handles to driving terminals of the net	**acc_next_driver()**
Obtain handles to load terminals of the net	**acc_next_load()**
Obtain handles to connected load terminals; only one per driven cell port	**acc_next_cell_load()**
Obtain a handle to the simulated net of a collapsed net	**acc_handle_simulated_net()**
Get the net name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the net vector size	**acc_fetch_size()**
Get the msb and lsb vector range	**acc_fetch_range()**
Get the net fulltype	**acc_fetch_fulltype()**
Get the net logic or strength value	**acc_fetch_value()**
Force or release the net logic value	**acc_set_value()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

 Copyright © 2001 IEEE. All rights reserved.

22.5.10 ACC routines that operate on reg types

Table 106—Operations on reg types

To	Use
Obtain handles to regs within a given scope	**acc_next()**
Obtain handles to bits of a vector reg	**acc_next_bit()**
Obtain a handle to the parent (module instance containing the reg)	**acc_handle_parent()**
Obtain handles to load terminals of the reg	**acc_next_load()**
Determine if reg is a scalar or a vector	**acc_object_of_type()**
Get the reg name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the reg size	**acc_fetch_size()**
Get the msb and lsb vector range	**acc_fetch_range()**
Get the reg value	**acc_fetch_value()**
Set the reg value	**acc_set_value()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

22.5.11 ACC routines that operate on integer, real, and time variables

Table 107—Operations on integer, real, and time variables

To	Use
Obtain handles to variables within a given scope	**acc_next()**
Obtain a handle to the parent (module instance containing the variable)	**acc_handle_parent()**
Get the variable name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the variable value	**acc_fetch_value()**
Set the variable value	**acc_set_value()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

22.5.12 ACC routines that operate on named events

Table 108—Operations on named events

To	Use
Obtain handles to named events within a given scope	**acc_next()**
Obtain a handle to the parent (module instance containing the named event)	**acc_handle_parent()**
Get the named-event name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Add VCL value change callback monitors	**acc_vcl_add()**
Delete VCL value change callback monitors	**acc_vcl_delete()**

Copyright © 2001 IEEE. All rights reserved.

22.5.13 ACC routines that operate on parameters and specparams

Table 109—Operations on parameters and specparams

To	Use
Obtain handles for parameters within a module instance	**acc_next_parameter()**
Obtain handles for specparams within a module instance	**acc_next_specparam()**
Obtain a handle to the parent (the module instance that contains the parameter)	**acc_handle_parent()**
Get the parameter or specparam name	**acc_fetch_name()**
Get the full hierarchical name	**acc_fetch_fullname()**
Get the parameter value data type (integer, floating point, string)	**acc_fetch_paramtype()** **acc_fetch_fulltype()**
Get the value of a parameter	**acc_fetch_paramval()**
Get the attribute value of a parameter defined with an attribute name	**acc_fetch_attribute()** **acc_fetch_attribute_int()** **acc_fetch_attribute_str()**

22.5.14 ACC routines that operate on timing checks

Table 110—Operations on timing checks

To	Use
Obtain handles for timing checks within a module instance	**acc_next_tchk()**
Obtain a handle to a specific timing check	**acc_handle_tchk()**
Obtain handles to all timing check terminals	**acc_next_input()**
Free memory allocated by acc_next_input()	**acc_release_object()**
Obtain a handle to a timing check terminal	**acc_handle_tchkarg1()** **acc_handle_tchkarg2()**
Get the timing check fulltype	**acc_fetch_fulltype()**
Get a timing check limit	**acc_fetch_delays()**
Append to an existing timing check limit	**acc_append_delays()**
Replace to an existing timing check limit	**acc_replace_delays()**

22.5.15 ACC routines that operate on timing check terminals

Table 111—Operations on timing check terminals

To	Use
Obtain a handle to the net attached to timing check terminals	**acc_handle_conn()**
Obtain a handle to the condition on a timing check terminal	**acc_handle_condition()**
Get edge information on a timing check terminal	**acc_fetch_edge()**

 Copyright © 2001 IEEE. All rights reserved.

22.5.16 ACC routines that operate on user-defined system task/function arguments

Table 112—Operations on user-defined system task/function arguments

To	Use
Obtain a handle for an object named in a task/function argument	**acc_handle_tfarg()** **acc_handle_itfarg()**
Get the value of a task/function argument as a double	**acc_fetch_tfarg()** **acc_fetch_itfarg()**
Get the value of a task/function argument as an integer	**acc_fetch_tfarg_int()** **acc_fetch_itfarg_int()**
Get the value of a task/function argument as a string pointer	**acc_fetch_tfarg_str()** **acc_fetch_itfarg_str()**

22.6 ACC routine types and fulltypes

Many objects in the Verilog HDL can have both a *type* and a *fulltype* associated with them. A type shall be a general classification of an object, whereas a fulltype shall be a specific classification. The type and fulltype for a given object can be different constants, or they can be the same constant. For example, an **and** logic gate has a type of **accPrimitive** and a fulltype of **accAndPrimitive**. The type and fulltype are predefined integer constants in the file `acc_user.h`. Several ACC routines either return a type or fulltype value, or use a type or fulltype value as an argument. Table 113 lists all type and fulltype constants that shall be supported by ACC routines, listed alphabetically by the type name.

Table 113—List of all predefined type and fulltype constants

type constant	fulltype constant	Description
accConstant	**accConstant**	Object is a constant
accDataPath	**accDataPath**	Object is a data path in a path delay
accFunction	**accFunction**	Object is a Verilog HDL function
accIntegerVar	**accIntegerVar**	Object is declared as an **integer** data type
accModPath	**accModPath**	Object is a module path
accModule	**accModuleInstance**	Object is a module instance
	accCellInstance	Object is a module instance that has been defined as a cell
	accTopModule	Object is a top-level module
accNamedEvent	**accNamedEvent**	Object is declared as an **event** data type

Copyright © 2001 IEEE. All rights reserved.

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accNet	**accSupply0**	Object is declared as a **supply0** net data type
	accSupply1	Object is declared as a **supply1** net data type
	accTri	Object is declared as a **tri** net data type
	accTriand	Object is declared as a **triand** net data type
	accTrior	Object is declared as a **trior** net data type
	accTrireg	Object is declared as a **trireg** net data type
	accTri0	Object is declared as a **tri0** net data type
	accTri1	Object is declared as a **tri1** net data type
	accWand	Object is declared as a **wand** net data type
	accWire	Object is declared as a **wire** net data type
	accWor	Object is declared as a **wor** net data type
accNetBit	**accNetBit**	Object is a bit-select of a net data type
accOperator	**accOperator**	Object is a Verilog HDL operator
accParameter	**accIntegerParam**	Object is a **parameter** with an integer value
	accRealParam	Object is a **parameter** with a real value
	accStringParam	Object is a **parameter** with a string value
accPartSelect	**accPartSelect**	Object is a part-select of a vector
accPathTerminal	**accPathInput**	Object is an input terminal of a module path
	accPathOutput	Object is an output terminal of a module path
accPort	**accConcatPort**	Object is a module port concatenation
	accScalarPort	Object is a scalar module port

	accBitSelectPort	Object is a bit-select of a module port (e.g.:
		`module (.a[1], .a[0], ...);`
		`input [1:0] a;`

	accPartSelectPort	Object is a part-select of a module port (e.g.:
		`module (.a[3:2], .a[1:0], ...);`
		`input [3:0] a;`

	accVectorPort	Object is a vector module port
accPortBit	**accPortBit**	Object is a bit of a module port

 Copyright © 2001 IEEE. All rights reserved.

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accPrimitive	**accAndGate**	Object is an **and** primitive
	accBufGate	Object is a **buf** primitive
	accBufif0Gate	Object is a **bufif0** primitive
	accBufif1Gate	Object is a **bufif1** primitive
	accCmosGate	Object is a **cmos** primitive
	accCombPrim	Object is a combinational logic UDP
	accNandGate	Object is a **nand** primitive
	accNmosGate	Object is an **nmos** primitive
	accNorGate	Object is a **nor** primitive
	accNotGate	Object is a **not** primitive
	accNotif0Gate	Object is a **notif0** primitive
	accNotif1Gate	Object is a **notif1** primitive
	accOrGate	Object is an **or** primitive
	accPmosGate	Object is a **pmos** primitive
	accPulldownGate	Object is a **pulldown** primitive
	accPullupGate	Object is a **pullup** primitive
	accRcmosGate	Object is an **rcmos** primitive
	accRnmosGate	Object is an **rnmos** primitive
	accRpmosGate	Object is an **rpmos** primitive
	accRtranGate	Object is an **rtran** primitive
	accRtranif0Gate	Object is an **rtranif0** primitive
	accRtranif1Gate	Object is an **rtranif1** primitive
	accSeqPrim	Object is a sequential logic UDP
	accTranGate	Object is a **tran** primitive
	accTranif0Gate	Object is a **tranif0** primitive
	accTranif1Gate	Object is a **tranif1** primitive
	accXnorGate	Object is an **xnor** primitive
	accXorGate	Object is an **xor** primitive
accRealVar	**accRealVar**	Object is declared as a **real** data type
accReg	**accReg**	Object is declared as a **reg** data type
accRegBit	**accRegBit**	Object is a bit-select of a **reg** data type
accSpecparam	**accIntegerParam**	Object is a **specparam** with an integer value
	accRealParam	Object is a **specparam** with a real value
	accStringParam	Object is a **specparam** with a string value

Copyright © 2001 IEEE. All rights reserved.

Table 113—List of all predefined type and fulltype constants (continued)

type constant	fulltype constant	Description
accStatement	**accStatement**	Object is a procedural statement
	accNamedBeginStat	Object is a named **begin** statement
	accNamedForkStat	Object is a named **fork** statement
accSystemTask	**accSystemTask**	Object is a built-in system task
accSystemFunction	**accSystemFunction**	Object is a built-in system function with a scalar or vector return
accSystemRealFunction	**accSystemRealFunction**	Object is a built-in system function with a real value return
accTask	**accTask**	Object is a Verilog HDL task
accTchk	**accHold**	Object is a **$hold** timing check
	accNochange	Object is a **$nochange** timing check
	accPeriod	Object is a **$period** timing check
	accRecovery	Object is a **$recovery** timing check
	accSetup	Object is a **$setup** timing check
	accSetuphold	Object is a **$setuphold** timing check
	accSkew	Object is a **$skew** timing check
	accWidth	Object is a **$width** timing check
accTchkTerminal	**accTchkTerminal**	Object is a timing check terminal
accTerminal	**accInputTerminal**	Object is a primitive input terminal
	accOutputTerminal	Object is a primitive output terminal
	accInoutTerminal	Object is a primitive inout terminal
accTimeVar	**accTimeVar**	Object is declared as a **time** data type
accUserTask	**accUserTask**	Object is a user-defined system task
accUserFunction	**accUserFunction**	Object is a user-defined system function with a scalar or vector return
accUserRealFunction	**accUserRealFunction**	Object is a user-defined system function with a real value return
accWirePath	**accIntermodPath**	Object is an intermodule path (from a module output to a module input)

22.7 Error handling

When an ACC routine detects an error, it shall perform the following operations:

a) Set the global error flag **acc_error_flag** to non-zero

b) Display an error message at run time to the output channel of the software product which invoked the PLI application

c) Return an exception value

 Copyright © 2001 IEEE. All rights reserved.

When an ACC routine is called, it automatically resets **acc_error_flag** to 0.

22.7.1 Suppressing error messages

By default, ACC routines shall display error messages. Error messages can be suppressed using the ACC routine **acc_configure()** to set the configuration parameter **accDisplayErrors** to "`false`".

22.7.2 Enabling warnings

By default, ACC routines shall not display warning messages. To enable warning messages, use the ACC routine **acc_configure()** to set the configuration parameter **accDisplayWarnings** to "`true`".

22.7.3 Testing for errors

If automatic error reporting is suppressed, error handling can be performed by checking the **acc_error_flag** explicitly after calling an ACC routine. This procedure is described in Figure 53.

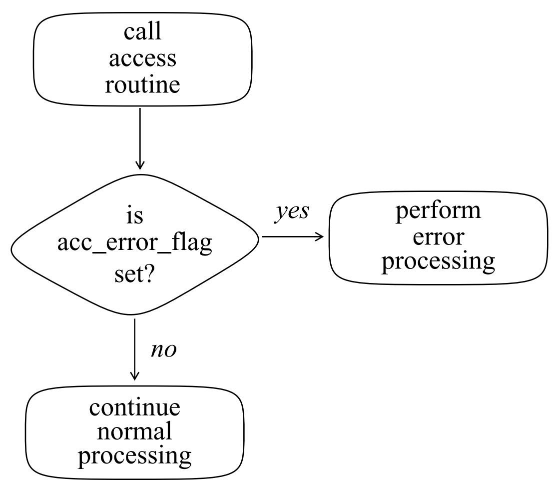


Figure 53—Using acc_error_flag to detect errors

22.7.4 Example

The following example shows a C language application that performs error checking for ACC routines. This example uses **acc_configure()** to suppress automatic error reporting. Instead, it checks **acc_error_flag** explicitly and displays its own specialized error message.

```
#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" );

   /* 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 is a primitive */
    if ( acc_fetch_type(gate_handle) != accPrimitive )
       tf_error("Invalid argument type:not a primitive\n");
   acc_close();
}
```

22.7.5 Exception values

ACC routines shall return one of three exception values when an error occurs, unless specified differently in the syntax of a specific ACC routine.

Table 114—Exception values returned by ACC routines on errors

When routine returns	The exception value shall be
PLI_INT32	0
double values	0.0
pointers or handles	null
bool (boolean) values	false

Because ACC 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.

22.8 Reading and writing delay values

This section explains how ACC routines that read and modify delays are used. The ACC routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** can read or modify delay values in a Verilog software implementation data structure. Refer to Clause 23 for the complete syntax of each of these routines.

 Copyright © 2001 IEEE. All rights reserved.

22.8.1 Number of delays for Verilog HDL objects

There are a variety of objects in a Verilog HDL source description that can model delays. These objects can have a single delay that represents all possible logic transitions, or multiple delays that represent different logic transitions. Table 115 lists the objects that can have delays and the number of different delays for each object.

Table 115—Number of possible delays for Verilog HDL objects

Verilog HDL Objects	Number of delays	Description
2-state primitives	**1**	One delay for: all transitions
	2	Separate delays for: rise, fall
3-state primitives	**1**	One delay for: all transitions
	2	Separate delays for: rise, fall
	3	Separate delays for: rise, fall, toZ
Module paths **Intermodule paths** **Module ports** **Module port bits**	**1**	One delay for: all transitions
	2	Separate delays for: rise, fall
	3	Separate delays for: rise, fall, toZ
	6	Separate delays for: `0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0`
	12	Separate delays for: `0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0, 0->X, X->1, 1->X, X->0, X->Z, Z->X`
Timing checks	**1**	One delay for: timing limit

In addition to the number of delays, each delay can be represented as a single delay for each transition or as a minimum:typical:maximum delay set for each transition. Thus, a module path, intermodule path and module input port with 1 delay might have one value or three values, and a module path, intermodule path and module input port with 12 delays can have 12 delay values or 36 delay values.

22.8.2 ACC routine configuration

The PLI shall use configuration parameters to set up the delay ACC routines to work with the variations of Verilog objects and the number of possible delays. These parameters shall be set using the routine **acc_configure()**. The parameters that configure the delay ACC routines are summarized in Table 115.

How these configuration parameters are used is presented in 22.8.3. Refer to 23.6 for details on using **acc_configure()**.

Copyright © 2001 IEEE. All rights reserved.

Table 116—Configuration parameters for delay ACC routines

Configuration parameter	Description
accMinTypMaxDelays	When "false", each delay shall be represented by one value. When "true", each delay shall be represented by three delay values, representing minimum, typical, maximum, respectively. The default shall be "false".
accToHiZDelay	When set to "average", "max" or "min", the delay modify ACC routines shall calculate the toZ delay for 3-state primitives, or for path and input port objects when accPathDelayCount is set to 2. When set to "from_user", the toZ delay shall not be calculated. The default is "from_user". This parameter shall be ignored when accMinTypMaxDelays is set to "true".
accPathDelayCount	Sets the number of delay arguments to be used by the ACC routines for module path, intermodule path and module input port delays. Shall be set to "1", "2", "3", "6", or "12". The default shall be "6".

22.8.3 Determining the number of arguments for ACC delay routines

The ACC routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require a different number of arguments based on

The type of object handle
The setting of configuration parameters

The following subsections discuss how these factors affect the number of arguments for delay ACC routines.

22.8.3.1 Single delay value mode

When the configuration parameter **accMinTypMaxDelays** is "false" (the default), a single value shall be used for each delay transition. In this mode, the routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require each delay value as a separate argument. For **acc_replace_delays()** and **acc_append_delays()**, the arguments shall be a literal value of type double or variables of type double. For **acc_fetch_delays()**, the arguments shall be pointers to variables of type double.

The number of arguments required is determined by the type of object handle passed to the delay ACC routine, as shown in Table 117.

Table 117—Number of delay arguments in single delay mode

Object handle type	Configuration parameters	Number and order of delay arguments
Timing check		**1** argument: timing check limit
2-state primitive		**2** arguments: rise, fall transitions
3-state primitive	**accToHiZDelay** set to "min", "max", or "average"	**2** arguments: rise, fall transitions (toZ delay is calculated; see Section 22.8.3.3)
	accToHiZDelay set to "from_user"	**3** arguments: rise, fall, toZ transitions

 Copyright © 2001 IEEE. All rights reserved.

Table 117—Number of delay arguments in single delay mode (continued)

Object handle type	Configuration parameters	Number and order of delay arguments
Module paths **Intermodule paths** **Module ports** **Module port bits**	**accPathDelayCount** set to "1"	**1** argument: all transitions
	accPathDelayCount set to "2"	**2** arguments: rise, fall transitions
	accPathDelayCount set to "3"	**3** arguments: rise, fall, toZ transitions
	accPathDelayCount set to "6"	**6** arguments: `0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0`

	accPathDelayCount set to "12"	**12** arguments:
		`0->1, 1->0, 0->Z, Z->1, 1->Z, Z->0`
		`0->X, X->1, 1->X, X->0, X->Z, Z->X`

22.8.3.2 Min:typ:max delay value mode

When the configuration parameter **accMinTypMaxDelays** is "`true`", a three-value set shall be used for each delay transition. In this mode, the routines **acc_fetch_delays()**, **acc_replace_delays()**, and **acc_append_delays()** shall require the delay argument to be a pointer of an array of variables of type double. The number of elements placed into or read from the array shall be determined by the type of object handle passed to the delay ACC routine, as shown in Table 118.

Table 118—Number of delay elements in min:typ:max delay mode

Object handle type	Configuration parameters	Size and order of the delay array

Timing check		**3** elements:
		`array[ 0] = min limit`
		`array[ 1] = typ limit`
		`array[ 2] = max limit`

2-state primitive
3-state primitive
		9 elements:
		`array[ 0] = min rise delay`
		`array[ 1] = typ rise delay`
		`array[ 2] = max rise delay`
		`array[ 3] = min fall delay`
		`array[ 4] = typ fall delay`
		`array[ 5] = max fall delay`
		`array[ 6] = min toZ delay`
		`array[ 7] = typ toZ delay`
		`array[ 8] = max toZ delay`
		(an array of at least 9 elements shall be declared, even if toZ delays are not used by the object)

Copyright © 2001 IEEE. All rights reserved.

Table 118—Number of delay elements in min:typ:max delay mode (continued)

Object handle type	`Configuration parameters`	Size and order of the delay array

Module path
Intermodule paths
Module ports
Module port bits
	accPathDelayCount set to `"1"`	**3** elements:
		`array[ 0] = min delay`
		`array[ 1] = typ delay`
		`array[ 2] = max delay`

	accPathDelayCount set to `"2"`	**6** elements:
		`array[ 0] = min rise delay`
		`array[ 1] = typ rise delay`
		`array[ 2] = max rise delay`
		`array[ 3] = min fall delay`
		`array[ 4] = typ fall delay`
		`array[ 5] = max fall delay`

	accPathDelayCount is set to `"3"`	**9** elements:
		`array[ 0] = min rise delay`
		`array[ 1] = typ rise delay`
		`array[ 2] = max rise delay`
		`array[ 3] = min fall delay`
		`array[ 4] = typ fall delay`
		`array[ 5] = max fall delay`
		`array[ 6] = min toZ delay`
		`array[ 7] = typ toZ delay`
		`array[ 8] = max toZ delay`

	accPathDelayCount set to `"6"`	**18** elements:
		`array[ 0] = min 0->1 delay`
		`array[ 1] = typ 0->1 delay`
		`array[ 2] = max 0->1 delay`
		`array[ 3] = min 1->0 delay`
		`array[ 4] = typ 1->0 delay`
		`array[ 5] = max 1->0 delay`
		`array[ 6] = min 0->Z delay`
		`array[ 7] = typ 0->Z delay`
		`array[ 8] = max 0->Z delay`
		`array[ 9] = min Z->1 delay`
		`array[ 10] = typ Z->1 delay`
		`array[ 11] = max Z->1 delay`
		`array[ 12] = min 1->Z delay`
		`array[ 13] = typ 1->Z delay`
		`array[ 14] = max 1->Z delay`
		`array[ 15] = min Z->0 delay`
		`array[ 16] = typ Z->0 delay`
		`array[ 17] = max Z->0 delay`

Copyright © 2001 IEEE. All rights reserved.

Table 118—Number of delay elements in min:typ:max delay mode (continued)

Object handle type	Configuration parameters	Size and order of the delay array
Module path *(continued)*	**accPathDelayCount** set to "12"	**36** elements: array[0] = min 0->1 delay array[1] = typ 0->1 delay array[2] = max 0->1 delay array[3] = min 1->0 delay array[4] = typ 1->0 delay array[5] = max 1->0 delay array[6] = min 0->Z delay array[7] = typ 0->Z delay array[8] = max 0->Z delay array[9] = min Z->1 delay array[10] = typ Z->1 delay array[11] = max Z->1 delay array[12] = min 1->Z delay array[13] = typ 1->Z delay array[14] = max 1->Z delay array[15] = min Z->0 delay array[16] = typ Z->0 delay array[17] = max Z->0 delay array[18] = min 0->X delay array[19] = typ 0->X delay array[20] = max 0->X delay array[21] = min X->1 delay array[22] = typ X->1 delay array[23] = max X->1 delay array[24] = min 1->X delay array[25] = typ 1->X delay array[26] = max 1->X delay array[27] = min X->0 delay array[28] = typ X->0 delay array[29] = max X->0 delay array[30] = min X->Z delay array[31] = typ X->Z delay array[32] = max X->Z delay array[33] = min Z->X delay array[34] = typ Z->X delay array[35] = max Z->X delay

22.8.3.3 Calculating turn-off delays from rise and fall delays

In single delay mode (**accMinTypMaxDelays** set to "false"), the routines **acc_replace_delays()** and **acc_append_delays()** can be instructed to calculate automatically the turn-off delays from rise and fall delays. How the calculation shall be performed is controlled by the configuration parameter **accToHiZDelay**, as shown in Table 119.

Copyright © 2001 IEEE. All rights reserved.

Table 119—Configuring accToHiZDelay to determine the toZ delay

Configuration of accToHiZDelay	Value of the toZ delay
`"average"`	The toZ turn-off delay shall be the average of the rise and fall delays.
`"min"`	The toZ turn-off delay shall be the smaller of the rise and fall delays.
`"max"`	The toZ turn-off delay shall be the larger of the rise and fall delays.
`"from_user"` (the default)	The toZ turn-off delay shall be set to the value passed as a user-supplied argument.

22.9 String handling

22.9.1 ACC routines share an internal string buffer

ACC routines that return pointers to strings can share an internal buffer to store string values. These routines shall return a pointer to the location in the buffer that contains the first character of the string, as illustrated in Figure 54. 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.

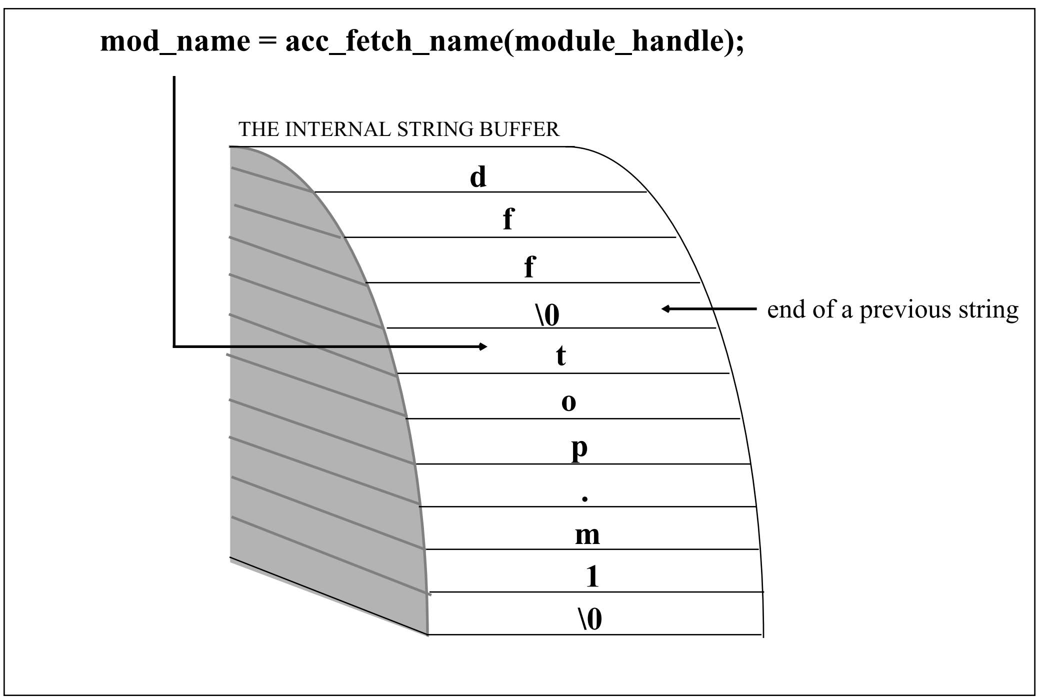


Figure 54—How ACC routines store strings in the internal buffer

 Copyright © 2001 IEEE. All rights reserved.

22.9.2 String buffer reset

ACC routines shall place strings at the next available sequential location in the string buffer, which stores at least 4096 characters. If there is not enough room to store an entire string starting at the next location, a condition known as *buffer reset* shall occur.

When buffer reset occurs, ACC routines shall place the next string starting at the beginning of the buffer, overwriting data already stored there. The result can be a loss of data, as illustrated in Figure 55.

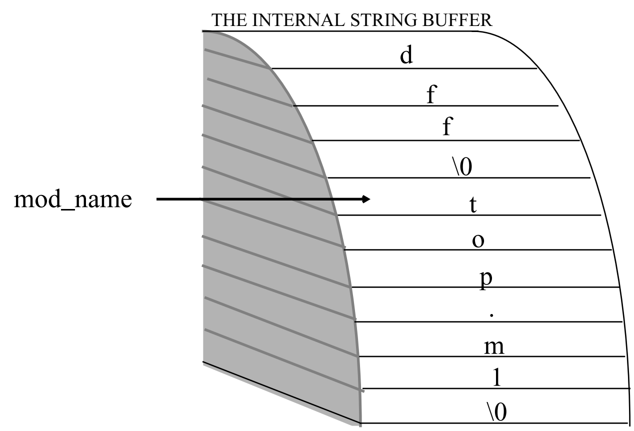
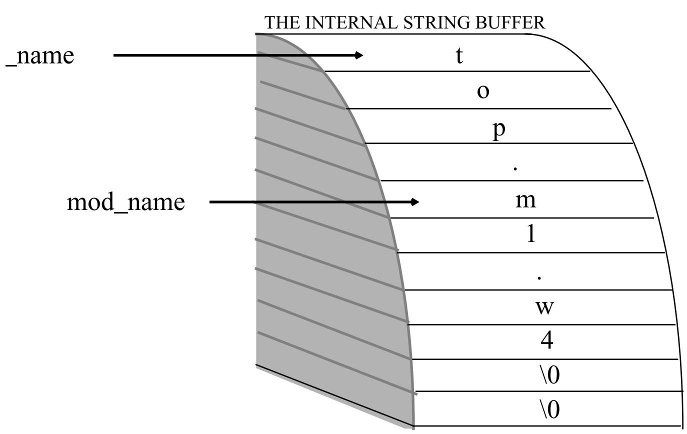
Action:	*Results:*
›d_name = acc_fetch_fullname(module_handle); 	mod_name points to the string `"top.m1"`. The string happens to be stored near the end of the buffer.
t_name = acc_fetch_fullname(net_handle); 	**acc_fetch_fullname()** cannot place the next string at the end of the buffer. Therefore, a buffer reset occurs. net_name points to the string `"top.m1.w4"` The data at the beginning of the buffer is overwritten; The old mod_name pointer now points to corrupted data, which in this example is `"m1.w4"`.

Figure 55—Buffer reset causes data in the string buffer to be overwritten

22.9.2.1 The buffer reset warning

ACC routines shall issue a warning whenever the internal string buffer resets. To view the warning message, the configuration parameter **accDisplayWarnings** shall be set to `"true"`, using the ACC routine **acc_configure()**.

Copyright © 2001 IEEE. All rights reserved.

22.9.3 Preserving string values

Applications that use strings immediately for example, to print names of objects do not need to be concerned about overwrites after a string buffer reset. Applications that have to preserve string values while calling other ACC routines that write to the string buffer should preserve the string value before it is overwritten. To preserve a string value, the C routine `strcpy` can be used to copy the string to a local character array.

22.9.4 Example of preserving string values

The following example code illustrates preserving string values. 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 an array called `mod_name`.

```
nclude "acc_user.h"

id  display_cells_in_module(mod)
ndle   mod;

 handle      cell;
 char       *mod_name;
 PLI_BYTE8  *temp;

/* save the module name in local buffer mod_name */
temp = acc_fetch_fullname(mod);
mod_name = (char*)malloc((strlen((char*)temp)+1) * sizeof(PLI_BYTE8));

strcpy(mod_name, (char *)temp);

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

 free(mod_name);
```

storage the size of the full module name is allocated

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

22.10 Using VCL ACC routines

The VCL routines add or delete value change monitors on a specified object. If a value change monitor is placed on an object, then whenever the object changes logic value or strength, a PLI consumer routine shall be called.

The ACC routine **acc_vcl_add()** adds a value change monitor on an object. The arguments for **acc_vcl_add()** specify

A handle to an object in the Verilog HDL structure
The name of a consumer routine
A user_data value
A VCL reason_flag

The following example illustrates the usage of **acc_vcl_add()**.

```
acc_vcl_add(net, netmon_consumer, net_name, vcl_verilog_logic);
```

 Copyright © 2001 IEEE. All rights reserved.

The purpose of each of these arguments is described in the following paragraphs. Refer to 23.97 for the full syntax and usage of **acc_vcl_add()** and its arguments.

The *handle* argument shall be a handle to any object type in the list in 22.10.1.

The *consumer routine* argument shall be the name of a C application that shall be called for the reasons specified by the *reason_flag*, such as a logic value change. When a consumer routine is called, it shall be passed a pointer to a C record, called `vc_record`. This record shall contain information about the object, including the simulation time of the change and the new logic value of the object. The `vc_record` is defined in the file `acc_user.h` and is listed in Figure 56.

The *user_data* argument shall be a PLI_BYTE8 pointer. The value of the user_data argument shall be passed to the consumer routine as part of the `vc_record`. The user_data argument can be used to pass a single value to the consumer routine, or it can be used to pass a pointer to information. For example, the name of the object could be stored in a global character string array, and a pointer to that array could be passed as the user_data argument. The consumer routine could then have access to the object name. Another example is to allocate memory for a user-defined structure with several values that need to be passed to the consumer routine. A pointer to the memory for the user-defined structure is then passed as the user_data argument. Note that the user_data argument is defined as a PLI_BYTE8 pointer; therefore, any other data type should be cast to a PLI_BYTE8 pointer.

The VCL *reason_flag* argument is one of two predefined constants that sets up the VCL callback mechanism to call the consumer routine under specific circumstances. The constant **vcl_verilog_logic** sets up the VCL to call the consumer routine whenever the monitored object changes logic value. The constant **vcl_verilog_strength** sets up the VCL to call the consumer routine when the monitored object changes logic value or logic strength.

An object can have any number of VCL monitors associated with it, as long as each monitor is unique in some way. VCL monitors can be deleted using the ACC routine **acc_vcl_delete()**.

22.10.1 VCL objects

The VCL shall monitor value changes for the following objects:

Scalar variables and bit-selects of vector variables
Scalar nets, unexpanded vector nets, and bit-selects of expanded vector nets
Integer, real, and time variables
Module ports
Primitive output or inout terminals
Named events

Note Adding a value change link to a module port is equivalent to adding a value change link to the loconn of the port. The vc_reason returned shall be based on the loconn of the port.

22.10.2 The VCL record definition

Each time a consumer routine is called, it shall be passed a pointer to a record structure called `vc_record`. This structure shall contain information about the most recent change that occurred on the monitored object. The `vc_record` structure is defined in `acc_user.h` and is listed in Figure 56.

Copyright © 2001 IEEE. All rights reserved.

```
typedef struct t_vc_record
{
  PLI_INT32 vc_reason;
  PLI_INT32 vc_hightime;
  PLI_INT32 vc_lowtime;
  PLI_BYTE8 *user_data;
  union
    {
      PLI_UBYTE8 logic_value;
      double real_value;
      handle vector_handle;
      s_strengths strengths_s;
    } out_value;
} s_vc_record, *p_vc_record;
```

Figure 56—The VCL s_vc_record structure

The *vc_reason* field of `vc_record` shall contain a predefined integer constant that shall describe what type of change occurred. The constants that can be passed in the vc_reason field are described in Table 120.

Table 120—Predefined vc_reason constants

Predefined vc_reason constant	Description
logic_value_change	A scalar net or bit-select of a vector net changed logic value.
strength_value_change	A scalar net or bit-select of a vector net changed logic value or strength.
vector_value_change	A vector net or part-select of a vector net changed logic value.
sregister_value_change	A scalar reg changed logic value.
vregister_value_change	A vector reg or part-select of a vector reg changed logic value.
integer_value_change	An integer variable changed value.
real_value_change	A real variable changed value.
time_value_change	A time variable changed value.
event_value_change	A named event occured.

The *vc_hightime* and *vc_lowtime* fields of `vc_record` shall be 32-bit integers that shall contain the simulation time in the simulator's time units during which the change occurred, as follows:

msb			**lsb**
vc_hightime		**vc_lowtime**	
64	**32**	**31**	**0**

The *user_data* field of `vc_record` shall be a PLI_BYTE8 pointer, and it shall contain the value specified as the user_data argument in the **acc_vcl_add()** ACC routine.

The *out_value* field of `vc_record` shall be a union of several data types. Only one data type shall be passed in the structure, based on the reason the callback occurred, as shown Table 121.

 Copyright © 2001 IEEE. All rights reserved.

Table 121—Predefined out_value constants

If vc_reason is	The out_value shall be a type of	Description
logic_value_change	PLI_UBYTE8	A predefined constant, from the following: **vcl0 vcl1 vclX vclx vclZ vclz**
strength_value_change	`s_strengths` structure	A structure with logic and strength, as shown in Figure 57
vector_value_change	handle	A handle to a vector net or part-select of a vector net
sregister_value_change	PLI_UBYTE8	A predefined constant, from the following: **vcl0 vcl1 vclX vclx vclZ vclz**
vregister_value_change	handle	A handle to a vector reg or part-select of a vector reg
integer_value_change	handle	A handle to an integer variable
real_value_change	double	The value of a real variable
time_value_change	handle	A handle to a time variable
event_value_change	none	Event types have no value

When the vc_reason field of the `vc_record` is **strength_value_change**, the s_strengths structure fields of the *out_value* field of `vc_record` shall contain the value. This structure shall contain three fields, as shown in Figure 57.

```
typedef struct t_strengths
{
  PLI_UBYTE8 logic_value;
  PLI_UBYTE8 strength1;
  PLI_UBYTE8 strength2;
} s_strengths, *p_strengths;
```

Figure 57—The VCL s_strengths structure

The values of the `s_strengths` structure fields are defined in Table 122.

Table 122—Predefined out_value constants

s_strengths field	C data type	Description
logic_value	PLI_UBYTE8	A predefined constant, from the following: **vcl0 vcl1 vclX vclx vclZ vclz**
strength1 **strength2**	PLI_UBYTE8	A predefined constant, from the following: **vclSupply vclWeak** **vclStrong vclMedium** **vclPull vclSmall** **vclLarge vclHighZ**

Copyright © 2001 IEEE. All rights reserved.

The *strength1* and *strength2* fields of the `s_strengths` structure can represent

a) A known strength when strength1 and strength2 contain the same value, the signal strength shall be that value.

b) An ambiguous strength with a known logic_value when strength1 and strength2 contain different values and the logic_value contains either **vcl0** or **vcl1**, the signal strength shall be an ambiguous strength, where the strength1 value shall be the maximum possible strength and strength2 shall be the minimum possible strength.

c) An ambiguous strength with an unknown logic_value when strength1 and strength2 contain different values and the logic_value contains **vclX**, the signal strength shall be an ambiguous strength, where the strength1 value shall be the logic 1 component and strength2 shall be the logic 0 component.

22.10.3 Effects of acc_initialize() and acc_close() on VCL consumer routines

The ACC routines **acc_initialize()** and **acc_close()** shall reset all configuration parameters set by the routine **acc_configure()** back to default values. Care should be taken to ensure that the VCL consumer routine does not depend on any configuration parameters, as these parameters might not have the same value when a VCL callback occurs. Refer to 23.6 on **acc_configure()** for a list of routines that are affected by configuration parameters.

22.10.4 An example of using VCL ACC routines

The following example contains three PLI routines: a checktf application, a calltf application, and a consumer routine. The example is based on the checktf and calltf applications both being associated with two user-defined system tasks, using the PLI interface mechanism described in Clause 21.

```
$net_monitor(<net_name>,<net_name>, ...);
$net_monitor_off(<net_name>,<net_name>, ...);
```

The checktf application, `netmon_checktf`, is shown below. This application performs syntax checking on instances of the user-defined system tasks to ensure there is at least one argument and that the arguments are valid net names.

```
PLI_INT32 netmon_checktf()
{
  int i;
  PLI_INT32 arg_cnt = tf_nump();

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

  /* check number and type of task/function arguments */
  if (arg_cnt == 0)
    tf_error("$net_monitor[_off] must have at least one argument");
  else
    for (i = 1; i <= arg_cnt; i++)
      if (acc_fetch_type(acc_handle_tfarg(i)) != accNet) {
        tf_error("$net_monitor[_off] arg %d is not a net type",i);
      }

  acc_close();
  return(0);
}
```

 Copyright © 2001 IEEE. All rights reserved.

The calltf application, `netmon_calltf`, follows. This application gets a handle to each task function argument and either adds or deletes a VCL monitor on the net. The application checks the data C argument associated with each system task name to determine whether the application was called by `$net_monitor` or `$net_monitor_off`.

```
PLI_INT32 netmon_calltf(data)
PLI_INT32 data;
{
  handle net;
  PLI_INT32  netmon_consumer();
  PLI_INT32  tfnum;

  #define ADD    0  /* data value associated with $net_monitor */
  #define DELETE 1  /* data value associated with $net_monitor_off */

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

  switch (data)  /* see which system task name called this application
  {
    case ADD:     /* called by $net_monitor */
      /* add a VCL flag to each net in the task/function argument list
      tfnum = 1;
      while ((net = acc_handle_tfarg(tfnum++)) != null)
      {
        /* add a VCL monitor; pass net pointer as user_data argument*/
        acc_vcl_add(net, netmon_consumer, (PLI_BYTE8*)net,
                    vcl_verilog_logic);
      }
      break;

    case DELETE:  /* called by $net_monitor_off */
      tfnum = 1;
      while ((net = acc_handle_tfarg(tfnum++)) != null)
      {
        /* delete the VCL monitor */
        acc_vcl_delete(net, netmon_consumer, (PLI_BYTE8*)net, vcl_veri
      }
      break;
  }
  acc_close();
}
```

Copyright © 2001 IEEE. All rights reserved.

The consumer routine, `netmon_consumer`, is shown in the following example. The consumer routine is called by the VCL callback mechanism. Since the checktf application only permits net data types to be used, the consumer routine only needs to check for scalar and vector net value changes when it is called. In this example, it is assumed that $net_monitor is associated with a data value of 0, and $net_monitor_off is associated with a data value of 1. Refer to 21.3.1 for a description of associating data values.

```
LI_INT32 netmon_consumer(vc_record)
_vc_record vc_record; /* record type passed to consumer routine */

 PLI_BYTE8 net_value;
 char      value;
 handle    vector_value;

 /* check reason VCL call-back occurred */
 switch (vc_record->vc_reason)
 {
   case logic_value_change :   /* scalar signal changed logic value */
   {
     net_value = vc_record->out_value.logic_value;
     /* convert logic value constant to a character for printing */
     switch (net_value)
     {
       case vcl0 : value = '0'; break;
       case vcl1 : value = '1'; break;
       case vclX : value = 'X'; break;
       case vclZ : value = 'Z'; break;
     }
     io_printf("%d : %s = %c\n",
               vc_record->vc_lowtime,
               acc_fetch_name((handle) vc_record->user_data),
               value);
     break;
   }
   case vector_value_change :/* vector signal changed logic value */
   {
     vector_value = vc_record->out_value.vector_handle;
     io_printf("%d : %s = %s\n",
               vc_record->vc_lowtime,
               acc_fetch_name((handle) vc_record->user_data),
               acc_fetch_value(vector_value, "%b",NULL) );
     break;
   }
 }
```

 Copyright © 2001 IEEE. All rights reserved.

23. ACC routine definitions

This clause describes the PLI access (ACC) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order.

The following conventions are used in the definitions of the PLI routines described in Clause 23, Clause 25, and Clause 27.

Synopsis: A brief description of the PLI routine functionality, intended to be used as a quick reference when searching for PLI routines to perform specific tasks.

Syntax: The exact name of the PLI routine and the order of the arguments passed to the routine.

Returns: The definition of the value returned when the PLI routine is called, along with a brief description of what the value represents. The return definition contains the fields

> **Type:** The data type of the C value that is returned. The data type is either a standard ANSI C type or a special type defined within the PLI.
> **Description:** A brief description of what the value represents.

Arguments: The definition of the arguments passed with a call to the PLI routine. The argument definition contains the fields

> **Type:** The data type of the C values that are passed as arguments. The data type is either a standard ANSI C type, or a special type defined within the PLI.
> **Name:** The name of the argument used in the Syntax definition.
> **Description:** A brief description of what the value represents.

All arguments shall be considered mandatory unless specifically noted in the definition of the PLI routine. Two tags are used to indicate arguments that may not be required:

> *Conditional:* Arguments tagged as conditional shall be required only if a previous argument is set to a specific value, or if a call to another PLI routine has configured the PLI to require the arguments. The PLI routine definition explains when conditional arguments are required.
> *Optional:* Arguments tagged as optional may have default values within the PLI, but they may be required if a previous argument is set to a specific value, or if a call to another PLI routine has configured the PLI to require the arguments. The PLI routine definition explains the default values and when optional arguments are required.

Related routines: A list of PLI routines that are typically used with, or provide similar functionality to, the PLI routine being defined. This list is provided as a convenience to facilitate finding information in this standard. It is not intended to be all-inclusive, and it does not imply that the related routines have to be used.

Copyright © 2001 IEEE. All rights reserved.

23.1 acc_append_delays()

acc_append_delays() for single delay values (accMinTypMaxDelays set to false)			
Synopsis:	Add delays to existing delay on primitives, module paths, intermodule paths, timing checks, and module input ports.		
Syntax:			
Primitives	`acc_append_delays(object_handle, rise_delay, fall_delay, z_delay)`		
Module paths Intermodule paths Ports or port bits	`acc_append_delays(object_handle, d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12)`		
Timing checks	`acc_append_delays(object_handle, limit)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	`handle`	object_handle	Handle of a primitive, module path, intermodule path, timing check, module input port or bit of a module input port
	`double`	rise_delay fall_delay	Rise and fall transition delay for 2-state primitives, 3-state primitives
Conditional	`double`	z_delay	If **accToHiZDelay** is set to from_user : turn-off (to Z) transition delay for 3-state primitives
	`double`	d1	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 1 : delay for all transitions If **accPathDelayCount** is set to 2 or 3 : rise transition delay If **accPathDelayCount** is set to 6 or 12 : **0**->**1** transition delay
Conditional	`double`	d2	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 2 or 3 : fall transition delay If **accPathDelayCount** is set to 6 or 12 : **1**->**0** transition delay
Conditional	`double`	d3	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 3 : turn-off transition delay If **accPathDelayCount** is set to 6 or 12 : **0**->**Z** transition delay
Conditional	`double`	d4 d5 d6	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 6 or 12 : `d4` is **Z**->**1** transition delay `d5` is **1**->**Z** transition delay `d6` is **Z**->**0** transition delay
Conditional	`double`	d7 d8 d9 d10 d11 d12	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 12 : `d7` is **0**->**X** transition delay `d8` is **X**->**1** transition delay `d9` is **1**->**X** transition delay `d10` is **X**->**0** transition delay `d11` is **X**->**Z** transition delay `d12` is **Z**->**X** transition delay
	`double`	limit	Limit of timing check

 Copyright © 2001 IEEE. All rights reserved.

acc_append_delays() for min:typ:max delays (accMinTypMaxDelays set to true)			
Synopsis:	Add min:typ:max delay values to existing delay values for primitives, module paths, intermodule paths, timing checks or module input ports; the delay values are contained in an array.		
Syntax:	`acc_append_delays(object_handle, array_ptr)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	`handle`	object_handle	Handle of a primitive, module path, intermodule path, timing check, module input port or bit of a module input port
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of **accPathDelayCount** (see 22.8)
Related routines:	Use acc_fetch_delays() to retrieve an object s delay values Use acc_replace_delays() to replace an object s delay values Use acc_configure() to set accPathDelayCount and accMinTypMaxDelays		

The ACC routine **acc_append_delays()** shall work differently depending on how the configuration parameter **accMinTypMaxDelays** is set. When this parameter is set to `false`, a single delay per transition shall be assumed, and delays shall be passed as individual arguments. For this single delay mode, the first syntax table in this section shall apply.

When **accMinTypMaxDelays** is set to `true`, **acc_append_delays()** shall pass one or more sets of minimum:typical:maximum delays contained 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 shall apply.

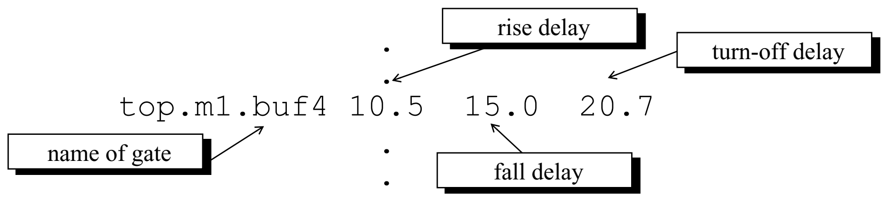
The number of delay values appended by **acc_append_delays()** shall be determined by the type of object and the setting of configuration parameters. Refer to 22.8 for a description of how the number of delay values are determined.

The acc_append_delays() routine shall write delays in the timescale of the module that contains the object_handle.

When altering the delay via **acc_append_delays()** the value of the reject/error region will not be affected unless they exceed the value of the delay. If the reject/error limits exceed the delay they will be truncated down to the new delay limit.

The example shown in Figure 58 is an example of backannotation. 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 below.

Copyright © 2001 IEEE. All rights reserved.




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

PLI_INT32 write_gate_delays()
{
   FILE      *infile;
   PLI_BYTE8  full_gate_name[NAME_SIZE];
   double     rise,fall,toz;
   handle     gate_handle;

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

   /*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 58—Using acc_append_delays() in single delay value mode

The example shown in Figure 59 shows how to append min:typ:max delays for a 2-state primitive (no high-impedance state). The C application follows these steps:

a) Declares an array of nine double-precision floating-point values to hold three sets of min:typ:max values, one set each for rising transitions, falling transitions, and transitions to Z.

b) Sets the configuration parameter **accMinTypMaxDelays** to `true` to instruct **acc_append_delays()** to write delays in min:typ:max format.

c) Calls **acc_append_delays()** with a valid primitive handle and the array pointer.

Since the primitive to be used in this example does not have a high-impedance state, **acc_append_delays()** automatically appends just the rise and fall delay value sets. The last three array elements for the toZ delay values are not used. However, even though the last three array elements are not used with a 2-state primitive, the syntax for using min:typ:max delays requires that the array contain all nine elements.

 Copyright © 2001 IEEE. All rights reserved.

For this example, the C application, `append_mintypmax_delays`, is associated through the ACC interface mechanism with a user-defined system task called `$appendprimdelays`. A primitive with no Z state and new delay values are passed as task/function arguments to `$appendprimdelays` as follows:

a 2-state primitive **typical fall delay** **maximum fall delay**

$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**

```
#include "acc_user.h"

PLI_INT32 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 has to be large enough to hold *nine* values to handle both 2-state primitives and 3-state primitives

Figure 59—Using acc_append_delays() in min:typ:max mode

Copyright © 2001 IEEE. All rights reserved.

23.2 acc_append_pulsere()

acc_append_pulsere()			
Synopsis:	Add delays to existing pulse handling *reject_limit* and *e_limit* for a module path, intermodule path or module input port.		
Syntax:	acc_append_pulsere(object,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	handle	object	Handle of module path, intermodule path or module input port
	double	r1...r12	*reject_limit* values; the number of arguments is determined by **accPathDelayCount**
	double	e1...e12	*e_limit* values; the number of arguments is determined by **accPathDelayCount**
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_append_pulsere()** shall add to an existing pulse handling reject_limit value and e_limit value for a module path, intermodule path and module input port. The reject_limit and e_limit values are used to control how pulses are propagated through paths.

A *pulse* is defined as two transitions that occur in a shorter period of time than the delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where

The reject_limit shall set a threshold for determining when to reject a pulse any pulse less than the reject_limit shall not propagate.
The e_limit shall set a threshold for determining when a pulse is considered to be an error any pulse less than the e_limit and greater than or equal to the reject_limit shall propagate a logic `x`.
A pulse that is greater than or equal to the e_limit shall propagate.

Table 123 illustrates the relationship between the reject_limit and the e_limit.

Table 123—Pulse control example

When	The pulse shall be
reject_limit = 10.5 *e_limit* = 22.6	Rejected if < 10.5 An error if >= 10.5 and < 22.6 Passed if >= 22.6

 Copyright © 2001 IEEE. All rights reserved.

The following rules shall apply when specifying pulse handling values:

a) The value of reject_limit shall be less than or equal to the value of e_limit.

b) The reject_limit and e_limit shall not be greater than the delay.

If any of the limits do not meet the above rules, they shall be truncated.

The number of pulse control values that **acc_append_pulsere()** sets shall be controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 124.

Table 124—How the value of accPathDelayCount affects acc_append_pulsere()

When accPathDelayCount is	acc_append_pulsere() shall write
1	One pair of reject_limit and e_limit values: one pair for all transitions, r1 and e1
2	Two pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
3	Three pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2 one pair for turn-off transitions, r3 and e3
6 (the default)	Six pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6
12	Twelve pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, X, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6 one pair for 0->X transitions, r7 and e7 one pair for X->1 transitions, r8 and e8 one pair for 1->X transitions, r9 and e9 one pair for X->0 transitions, r10 and e10 one pair for X->Z transitions, r11 and e11 one pair for Z->X transitions, r12 and e12

The minimum number of pairs of reject_limit and e_limit arguments to pass to **acc_append_pulsere()** has to equal the value of **accPathDelayCount**. Any unused reject_limit and e_limit argument pairs shall be ignored by **acc_append_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to six; therefore, six pairs of pulse reject_limit and e_limit arguments have to be passed when **acc_append_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_fetch_pulsere()**, and **acc_replace_pulsere()**.

Copyright © 2001 IEEE. All rights reserved.

Pulse control values shall be appended using the timescale of the module that contains the object handle.

23.3 acc_close()

acc_close()			
Synopsis:	Free internal memory used by ACC routines; reset all configuration parameters to default values.		
Syntax:	`acc_close()`		
Returns:	**Type**	Description	
	void	No return	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_initialize() to initialize the ACC routine environment		

The ACC routine **acc_close()** shall free internal memory used by ACC routines and reset all configuration parameters to default values. No other ACC routines should be called after calling **acc_close()**; in particular, ACC routines that are affected by **acc_configure()** should not be called.

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 all configuration parameters to their default values.

The example shown in Figure 60 presents a C language routine that calls **acc_close()** before exiting.

```
include "acc_user.h"

void show_versions()

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

  /*show version of ACC routines and simulator */
  io_printf("Running %s with %s\n",acc_version(),acc_product_version() )

  acc_close();
```

Figure 60—Using acc_close()

 Copyright © 2001 IEEE. All rights reserved.

23.4 acc_collect()

acc_collect()			
Synopsis:	Obtain an array of handles for all objects related to a particular reference object; get the number of objects collected.		
Syntax:	`acc_collect(acc_next_routine_name, object_handle, number_of_objects)`		
Returns:	**Type**	Description	
	handle array address	An address pointer to an array of handles of the objects collected	
Arguments:	**Type**	**Name**	**Description**
	pointer to acc_next_ routine	acc_next_routine_name	Actual name (unquoted) of the acc_next_ routine that finds the objects to be collected
	handle	object_handle	Handle of the reference object for acc_next_ routine
	PLI_INT32 *	number_of_objects	Integer pointer where the count of objects collected shall be written
Related routines:	All acc_next_ routines except acc_next_topmod() Use acc_free() to free memory allocated by acc_collect()		

The ACC routine **acc_collect()** shall scan through a reference object, such as a module, and collect handles to all occurrences of a specific target object. The collection of handles shall be stored in an array, which can then be used by other ACC routines.

The object associated with object_handle shall be a valid type of handle for the reference object required by the acc_next routine to be called.

The routine **acc_collect()** should be used in the following situations:

- To retrieve data that can be used 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 can be more efficient to use the an acc_next_ routine directly.

The routine **acc_collect()** shall allocate memory for the array of handles it returns. When the handles are no longer needed, the memory can be freed by calling the routine **acc_free()**.

The ACC routine **acc_next_topmod()** does not work with **acc_collect()**. However, top-level modules can be collected by passing **acc_next_child()** with a `null` reference object argument. For example:

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

The example shown in Figure 61 presents a C language routine that uses **acc_collect()** to collect and display all nets in a module.

Copyright © 2001 IEEE. All rights reserved.

```
#include "acc_user.h"

PLI_INT32 display_nets()
{
   handle    *list_of_nets, module_handle;
   PLI_INT32  net_count, i;

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

   /*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 61—Using acc_collect()

23.5 acc_compare_handles()

acc_compare_handles()			
Synopsis:	Determine if two handles refer to the same object.		
Syntax:	`acc_compare_handles(handle1, handle2)`		
Returns:	**Type**	Description	
	PLI_INT32	`true` if handles refer to the same object; `false` if different objects	
Arguments:	**Type**	**Name**	**Description**
	handle	handle1	Handle to any object
	handle	handle2	Handle to any object

The ACC routine **acc_compare_handles()** shall determine if two handles refer to the same object. In some cases, two different handles might reference the same object if each handle is retrieved in a different way for example, if an acc_next routine returns one handle and **acc_handle_object()** returns the other.

The C `==` operator cannot be used to determine if two handles reference the same object.

```
if (handle1 == handle2)   /* this does not work */
```

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 62 uses **acc_compare_handles()** to determine if a primitive drives the specified output of a scalar port of a module.

```
#include "acc_user.h"

PLI_INT32 prim_drives_scalar_port(prim, mod, port_num)
handle    prim, mod;
PLI_INT32 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 62—Using acc_compare_handles()

Copyright © 2001 IEEE. All rights reserved.

23.6 acc_configure()

acc_configure()			
Synopsis:	Set parameters that control the operation of various ACC routines.		
Syntax:	`acc_configure(configuration_parameter, configuration_value)`		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	integer constant	configuration_parameter	One of the following predefined constants: **accDefaultAttr0** **accDevelopmentVersion** **accDisplayErrors** **accDisplayWarnings** **accEnableArgs** **accMapToMipd** **accMinTypMaxDelays** **accPathDelayCount** **accPathDelimStr** **accToHiZDelay**
	quoted string	configuration_value	One of a fixed set of string values for the configuration_parameter
Related routines:	For **accDefaultAttr0** acc_fetch_attribute() acc_fetch_attribute_int() acc_fetch_attribute_str() For **accDisplayErrors** all ACC routines For **accDisplayWarnings** all ACC 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() acc_append_pulsere() acc_fetch_pulsere() acc_replace_pulsere()	For **accPathDelimStr** acc_fetch_attribute() acc_fetch_attribute_int() acc_fetch_attribute_str() acc_fetch_fullname() acc_fetch_name() For **accToHiZDelay** acc_append_delays() acc_replace_delays()

The ACC routine **acc_configure()** shall set parameters that control the operation of various ACC routines. Tables 125 through 134 describe each parameter and its set of values. Note that a call to either **acc_initialize()** or **acc_close()** shall set each configuration parameter back to its default value.

 Copyright © 2001 IEEE. All rights reserved.

Table 125—accDefaultAttr0 configuration parameter

	Set of values	Effect	Default
accDefaultAttr0	"true"	**acc_fetch_attribute()** shall return zero when it does not find the attribute requested and shall ignore the default_value argument	"false"
	"false"	**acc_fetch_attribute()** shall return the value passed as the default_value argument when it does not find the attribute requested	

Table 126—accDevelopmentVersion configuration parameter

	Set of values	Effect	Default
accDevelopmentVersion	Quoted string of letters, numbers, and the period character that form a valid PLI version, such as: IEEE 1364 PLI Software vendors can define version strings specific to their products	None (can be used to document which version of ACC routines was used to develop a PLI application)	Current version of ACC routines

Table 127—accDisplayErrors configuration parameter

	Set of values	Effect	Default
accDisplayErrors	"true"	ACC routines shall display error messages	"true"
	"false"	ACC routines shall not display error messages	

Table 128—accDisplayWarnings configuration parameter

	Set of values	Effect	Default
accDisplayWarnings	"true"	ACC routines shall display warning messages	"false"
	"false"	ACC routines shall not display warning messages	

Copyright © 2001 IEEE. All rights reserved.

Table 129—accEnableArgs configuration parameter

	Set of values	Effect	Default
accEnableArgs	acc_handle_modpath	**acc_handle_modpath()** shall recognize its optional arguments	no_acc_handle_modpath no_acc_handle_tchk no_acc_set_scope
	no_acc_handle_modpath	**acc_handle_modpath()** shall ignore its optional arguments	
	acc_handle_tchk	**acc_handle_tchk()** shall recognize its optional arguments	
	no_acc_handle_tchk	**acc_handle_tchk()** shall ignore its optional arguments	
	acc_set_scope	**acc_set_scope()** shall recognize its optional arguments	
	no_acc_set_scope	**acc_set_scope()** shall ignore its optional arguments	

Table 130—accMapToMipd configuration parameter

	Set of values	Effect	Default
accMapToMipd	`“max”`	**acc_replace_delays()** and **acc_append_delays()** shall map the longest intermodule path delay to the MIPD	`“max”`
	`“min”`	**acc_replace_delays()** and **acc_append_delays()** shall map the shortest intermodule path delay to the MIPD	
	`“latest”`	**acc_replace_delays()** and **acc_append_delays()** shall map the last intermodule path delay to the MIPD	

Table 131—accMinTypMaxDelays configuration parameter

	Set of values	Effect	Default
accMinTypMaxDelays	`“true”`	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere(), and acc_replace_pulsere()** shall use min:typ:max delay sets	`“false”`
	`“false”`	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere(), and acc_replace_pulsere()** shall use a single delay value	

 Copyright © 2001 IEEE. All rights reserved.

Table 132—accPathDelayCount configuration parameter

	Set of values	Effect	Default
accPathDelayCount	"1"	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere()**, **and acc_replace_pulsere()** shall use 1 delay value or value set	"6"
	"2"	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere()**, **and acc_replace_pulsere()** shall use 2 delay values or value sets	
	"3"	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere()**, **and acc_replace_pulsere()** shall use 3 delay values or value sets	
	"6"	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere()**, **and acc_replace_pulsere()** shall use 6 delay values or value sets	
	"12"	**acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere()**, **acc_fetch_pulsere()**, **and acc_replace_pulsere()** shall use 12 delay values or value sets	

Table 133—accPathDelimStr configuration parameter

	Set of values	Effect	Default
accPathDelimStr	Quoted string of letters, numbers, $ or _	**acc_fetch_name(), acc_fetch_fullname(), and acc_fetch_attribute()** shall use the string literal as the delimiter separating the source and destination in module path names	"$"

Copyright © 2001 IEEE. All rights reserved.

Table 134—accToHiZDelay configuration parameter

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

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 63 presents a C language application that obtains the load capacitance of all scalar nets connected to the ports in a module. This application uses **acc_configure()** to direct **acc_fetch_attribute()** to return 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"

PLI_INT32 display_load_capacitance()
{
   handle   module_handle, port_handle, net_handle;
   double   cap_val;

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

   /*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 63—Using acc_configure() to set accDefaultAttr0

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 64 presents a C language application that displays the name of a module path. It uses **acc_configure()** to set **accEnableArgs** and, therefore, forces **acc_handle_modpath()** to ignore its null name arguments and recognize its optional handle arguments, src_handle and dst_handle.

```
#include "acc_user.h"

PLI_INT32 get_path()
{
   handle   path_handle,mod_handle,src_handle,dst_handle;

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

   /*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 64—Using acc_configure() to set accEnableArgs

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 65 fetches the rise and fall delays of each path in a module and backannotates the maximum delay value as the delay for all transitions. The value of **accPathDelayCount** specifies the minimum number of arguments that have to be passed to routines that read or write delay values. 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.

```
#include "acc_user.h"

PLI_INT32 set_path_delays()
{
   handle   mod_handle;
   handle   path_handle;
   double   rise_delay,fall_delay,max_delay;

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

   /*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 65—Using acc_configure() to set accPathDelayCount

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 66 shows how **accToHiZDelay** is used to direct **acc_replace_delays()** to derive the turn-off delay for a Z-state primitive automatically as the smaller of its rise and fall delays.

```
#include "acc_user.h"

PLI_INT32 set_buf_delays()
{
   handle   primitive_handle;
   handle   path_handle;
   double   added_rise, added_fall;

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

   /*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 66—Using acc_configure() to set accToHiZDelay

23.7 acc_count()

acc_count()			
Synopsis:	Count the number of objects related to a particular reference object.		
Syntax:	`acc_count(acc_next_routine_name, object_handle)`		
	Type	Description	
Returns:	PLI_INT32	Number of objects	
	Type	**Name**	**Description**
Arguments:	pointer to an acc_next_ routine	acc_next_routine_name	Actual name (unquoted) of the acc_next_ routine that finds the objects to be counted
	handle	object_handle	Handle of the reference object for the acc_next_ routine
Related routines:	All acc_next_ routines except acc_next_topmod()		

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_count()** shall find the number of objects that exist for a specific acc_next_ routine with a given reference object. The object associated with object_handle shall be a valid reference object for the type acc_next_ routine to be called.

Note that the ACC routine **acc_next_topmod()** does not work with **acc_count()**. However, top-level modules can be counted using **acc_next_child()** with a `null` reference object argument. For example:

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

The example shown in Figure 67 uses **acc_count()** to count the number of nets in a module.

```
 ,
include "acc_user.h"
'LI_INT32 count_nets()

  handle      module_handle;
  PLI_INT32   number_of_nets;

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

  /*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 67—Using acc_count()

23.8 acc_fetch_argc()

acc_fetch_argc()			
Synopsis:	Get the number of command-line arguments supplied with a Verilog software tool invocation.		
Syntax:	`acc_fetch_argc()`		
Returns:	**Type**	Description	
	PLI_INT32	Number of command-line arguments	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_fetch_argv() to get a character string array of the invocation options		

The ACC routine **acc_fetch_argc()** shall obtain the number of command-line arguments given on a Verilog software product invocation command line.

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 68 uses **acc_fetch_argc()** to determine the number of invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

PLI_BYTE8* my_scan_plusargs(str)
 PLI_BYTE8 *str;

  PLI_INT32   i;
  int         length = strlen(str);
  PLI_BYTE8  *curStr;
  PLI_BYTE8 **argv = acc_fetch_argv();

  for (i = acc_fetch_argc()-1; i>0; i--)
  {
    curStr = argv[i];
    if ((curStr[0] == '+') && (!strncmp(curStr+1,str,length)))
    {
      PLI_BYTE8 *retVal;

      length = strlen(&(curStr[length]) + 1);
      retVal = (PLI_BYTE8 *)malloc(sizeof(PLI_BYTE8) * length);
      strcpy(retVal, &(curStr[length]));
      return(retVal);
    }
  }
  return(null);
```

Figure 68—Using acc_fetch_argc()

23.9 acc_fetch_argv()

acc_fetch_argv()			
Synopsis:	Get an array of character pointers that make up the command-line arguments for a Verilog software product invocation.		
Syntax:	acc_fetch_argv()		
Returns:	**Type**	Description	
	PLI_BYTE8 **	An array of character pointers that make up the command-line arguments	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_fetch_argc() to get a count of the number of invocation arguments		

The ACC routine **acc_fetch_argv()** shall obtain an array of character pointers that make up the command-line arguments.

 Copyright © 2001 IEEE. All rights reserved.

The format of the argv array is that each pointer in the array shall point to a NULL terminated character array which contains the string located on the tool’s invocation command line. There shall be argc entries in the argv array. The value in entry zero shall be the tool s name.

The argument following a `-f` argument shall contain a pointer to a NULL terminated array of pointers to characters. This new array shall contain the parsed contents of the file. The value in entry zero shall contain the name of the file. The remaining entries shall contain pointers to NULL terminated character arrays containing the different options in the file. The last entry in this array shall be a NULL. If one of the options is a `-f` then the next pointer shall behave the same as described above.

The example shown in Figure 69 uses **acc_fetch_argv()** to retrieve the invocation arguments used.

```
#include "acc_user.h"
#include <string.h> /* string.h is implementation dependent */

PLI_BYTE8* my_scan_plusargs(str)
 PLI_BYTE8 *str;

  PLI_INT32   i;
  int         length = strlen(str);
  PLI_BYTE8  *curStr;
  PLI_BYTE8 **argv = acc_fetch_argv();

  for (i = acc_fetch_argc()-1; i>0; i--)
  {
    curStr = argv[i];
    if ((curStr[0] == '+') && (!strncmp(curStr+1,str,length)))
    {
      PLI_BYTE8 *retVal;

      length = strlen(&(curStr[length]) + 1);
      retVal = (PLI_BYTE8 *)malloc(sizeof(PLI_BYTE8) * length);
      strcpy(retVal, &(curStr[length]));
      return(retVal);
    }
  }
  return(null);
```

Figure 69—Using acc_fetch_argv()

Copyright © 2001 IEEE. All rights reserved.

23.10 acc_fetch_attribute()

acc_fetch_attribute()			
Synopsis:	Get the value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	`acc_fetch_attribute(object_handle, attribute_string, default_value)`		
	Type	Description	
Returns:	double	Value of the parameter or specparam	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of a named object
	quoted string or PLI_BYTE8 *	attribute_string	Literal string or character string pointer with the ***attribute*** portion of the parameter or specparam declaration
Optional	double	default_value	Double-precision value to be returned if the attribute is not found (depends on ***accDefaultAttr0***)
Related routines:	Use acc_fetch_attribute_int() to get an attribute value as an integer Use acc_fetch_attribute_str() to get an attribute value as a string Use acc_configure(accDefaultAttr0...) to set default value returned when attribute is not found Use acc_fetch_paramtype() to get the data type of the parameter value Use acc_fetch_paramval() to get parameters or specparam values not declared in attribute/object format		

The ACC routine **acc_fetch_attribute()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as a double.

Any parameter or specparam can be an attribute by naming it in one of the following ways:

As a general attribute associated with more than one object in the module where the parameter or specparam attribute is declared

As a specific attribute associated with a particular object in the module where the parameter or specparam attribute is declared

Each of these methods uses its own naming convention, as described in Table 135. For either convention, *attribute_string* shall name the attribute and shall be passed as the second argument to **acc_fetch_attribute()**. The *object_name* shall be the actual name of a design object in a Verilog HDL source description.

Table 135—Naming conventions for attributes

For	Naming convention	Example
A general attribute	**attribute_string** A mnemonic name that describes the attribute	`specparam DriveStrength$ = 2.8;` *attribute_string* is `DriveStrength$`
A specific attribute associated with a particular object	**attribute_string object_name** Concatenate a mnemonic name that describes the attribute with the name of the object	`specparam DriveStrength$g1 = 2.8;` *attribute_string* is `DriveStrength$` *object_name* is `g1`

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_fetch_attribute()** shall identify module paths in terms of their sources and destinations in the following format:

source	path_delimiter	destination

The acc_fetch_attribute() routine shall look for module path names in this format, and **acc_fetch_name()** and **acc_fetch_fullname()** shall return names of module paths in this format. Therefore, the same naming convention should be used when associating an attribute with a module path. Note that names of module paths with multiple sources or destinations shall be derived from the first source or destination only.

By default, the *path_delimiter* used in path names is the `"$"` character. This default can be changed by using the ACC routine **acc_configure()** to set the delimiter parameter **accPathDelimStr** to another character string.

The examples in Table 136 show how to name module paths using different delimiter strings.

Table 136—Example module path names using delimiter strings

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

The following example shows an attribute name for a particular module path object:

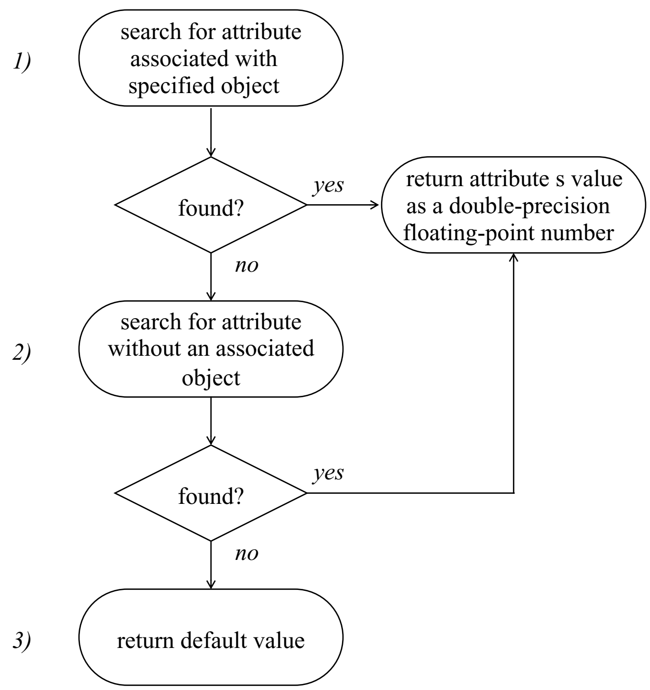
Given the module path: `(a => q) = 10;`

An attribute name is: `specparam RiseStrength$a$q = 20;`

In this example, the *attribute_string* is `RiseStrength$`, the *object_name* is `a$q`, and the *path_delimiter* is $ (the default path delimiter).

Copyright © 2001 IEEE. All rights reserved.

The following flowchart illustrates how **acc_fetch_attribute()** shall work:



This flowchart shows that when **acc_fetch_attribute()** finds the attribute requested, it returns the value of the attribute as a double-precision floating-point number.

1) The routine shall first look for the attribute name that concatenates *attribute_string* with the name associated with *object_handle*. For example, to find an attribute `InputLoad$` for a net `n1`, **acc_fetch_attribute()** would search for `InputLoad$n1`.

2) If **acc_fetch_attribute()** does not find the attribute associated with the object specified with *object_handle*, the routine shall then search for a name that matches *attribute_string*. Assume that, in the previous example, **acc_fetch_attribute()** does not find `InputLoad$n1`. It would then look for `InputLoad$`. Other variants of that name, such as `InputLoad$n3` or `InputLoad$n`, shall not be considered matches.

3) Failing both search attempts, the routine **acc_fetch_attribute()** shall return a default value. The default value is controlled by using the ACC routine **acc_configure()** to set or reset the configuration parameter **accDefaultAttr0** as shown in Table 137.

Table 137—Controlling the default value returned by acc_fetch_attribute()

When accDefaultAttr0 is	acc_fetch_attribute() shall return
true	Zero when the attribute is not found; the *default_value* argument can be dropped
false	The value passed as the *default_value* argument when the attribute is not found

The example shown in Figure 70 presents a C language application that uses **acc_fetch_attribute()** to obtain the load capacitance of all scalar nets connected to the ports in a module. Note that **acc_fetch_attribute()** does not require its third argument, *default_value*, because **acc_configure()** is used to set **accDefaultAttr0** to `true`.

 Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"
LI_INT32 display_load_capacitance()

  handle   module_handle, port_handle, net_handle;
  double   cap_val;

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

  /*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 70—Using acc_fetch_attribute()

Copyright © 2001 IEEE. All rights reserved.

23.11 acc_fetch_attribute_int()

acc_fetch_attribute_int()			
Synopsis:	Get the integer value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	`acc_fetch_attribute_int(object_handle, attribute_string, default_value)`		
	Type	Description	
Returns:	PLI_INT32	Value of the parameter or specparam	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of a named object
	quoted string or PLI_BYTE8 *	attribute_string	Literal string or character string pointer with the *attribute* portion of the parameter or specparam declaration
Optional	PLI_INT32	default_value	Integer value to be returned if the attribute is not found (depends on **accDefaultAttr0**)
Related routines:	Use acc_fetch_attribute() to get an attribute value as a double Use acc_fetch_attribute_str() to get an attribute value as a string Use acc_configure(accDefaultAttr0...) to set default value returned when attribute is not found Use acc_fetch_paramtype() to get the data type of the parameter value Use acc_fetch_paramval() to get parameters or specparam values not declared in attribute/object format		

The ACC routine **acc_fetch_attribute_int()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as an integer.

Any parameter or specparam can be an attribute. Refer to 23.10 for a description of attribute naming and how attribute values are fetched.

23.12 acc_fetch_attribute_str()

acc_fetch_attribute_str()			
Synopsis:	Get the value of a parameter or specparam named as an attribute in the Verilog source description.		
Syntax:	`acc_fetch_attribute_str(object_handle, attribute_string, default_value)`		
	Type	Description	
Returns:	PLI_BYTE8 *	Value of the parameter or specparam	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of a named object
	quoted string or PLI_BYTE8 *	attribute_string	Literal string or character string pointer with the *attribute* portion of the parameter or specparam declaration
Optional	quoted string or PLI_BYTE8 *	default_value	Character string value to be returned if the attribute is not found (depends on **accDefaultAttr0**)
Related routines:	Use acc_fetch_attribute() to get an attribute value as a double Use acc_fetch_attribute_int() to get an attribute value as an integer Use acc_configure(accDefaultAttr0...) to set default value returned when attribute is not found Use acc_fetch_paramtype() to get the data type of the parameter value Use acc_fetch_paramval() to get parameters or specparam values not declared in attribute/object format		

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_fetch_attribute_str()** shall obtain the value of a parameter or specparam that is declared as an attribute in the Verilog HDL source description. The value shall be returned as a pointer to a character string. The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

Any parameter or specparam can be an attribute. Refer to 23.10 for a description of attribute naming and how attribute values are fetched.

23.13 acc_fetch_defname()

acc_fetch_defname()			
Synopsis:	Get the definition name of a module instance or primitive instance.		
Syntax:	`acc_fetch_defname(object_handle)`		
	Type	Description	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the definition name	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of the module instance or primitive instance
Related routines	Use acc_fetch_name() to display the instance name of an object		

The ACC routine **acc_fetch_defname()** shall obtain the definition name of a module instance or primitive instance. The *definition name* is the declared name of the object as opposed to the *instance name* of the object. In the illustration shown below, the definition name is "`dff`", and the instance name is "`i15`".

definition name

```
dff i15 (q, clk, d);  //instance of a module or primitive
```

instance name

The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 71 presents a C language application that uses **acc_fetch_defname()** to display the definition names of all primitives in a module.

```
include "acc_user.h"

oid get_primitive_definitions(module_handle)
andle   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 71—Using acc_fetch_defname()

23.14 acc_fetch_delay_mode()

acc_fetch_delay_mode()			
Synopsis:	Get the delay mode of a module instance.		
Syntax:	`acc_fetch_delay_mode(module_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	A predefined integer constant representing the delay mode of the module instance: **accDelayModeNone** **accDelayModeZero** **accDelayModeUnit** **accDelayModePath** **accDelayModeDistrib** **accDelayModeMTM**	
Arguments:	**Type**	**Name**	**Description**
	`handle`	module_handle	Handle to a module instance

The ACC routine **acc_fetch_delay_mode()** shall return the delay mode of a module or cell instance. The delay mode determines how delays are stored for primitives and paths within the module or cell. The routine shall return one of the predefined constants given in Table 138.

Table 138—Predefined constants used by acc_fetch_delay_mode()

Predefined constant	Description
accDelayModeNone	No delay mode specified.
accDelayModeZero	All primitive delays are zero; all path delays are ignored.
accDelayModeUnit	All primitive delays are one; all path delays are ignored.
accDelayModeDistrib	If a logical path has both primitive delays and path delays specified, the primitive delays shall be used.

 Copyright © 2001 IEEE. All rights reserved.

Table 138—Predefined constants used by acc_fetch_delay_mode() (continued)

Predefined constant	Description
accDelayModePath	If a logical path has both primitive delays and path delays specified, the path delays shall be used.
accDelayModeMTM	If this property is true, Minimum:Typical:Maximum delay sets for each transition are being stored; if this property is false, a single delay for each transition is being stored.

Figure 72 uses **acc_fetch_delay_mode()** to retrieve the delay mode of all children of a specified module.

```
#include "acc_user.h"
PLI_INT32 display_delay_mode()
{
   handle   mod, child;

   /*reset environment for ACC routines*/
   acc_initialize();

   /*get module passed to user-defined system task*/
   mod = acc_handle_tfarg(1);

   /*find and display delay mode for each module instance*/
   child = null;
   while(child = acc_next_child(mod, child))
   {
      io_printf("Module %s set to: ",acc_fetch_fullname(child));
      switch(acc_fetch_delay_mode(child) )
      {
         case accDelayModePath:
            io_printf(" path delay mode\n");
            break;
         case accDelayModeDistrib:
            io_printf(" distributed delay mode\n");
            break;
         . . .
      }
   }
}
```

Figure 72—Using acc_fetch_delay_mode()

Copyright © 2001 IEEE. All rights reserved.

23.15 acc_fetch_delays()

acc_fetch_delays() for single delay values (accMinTypMaxDelays set to false)			
Synopsis:	Get existing delays for primitives, module paths, timing checks, module input ports, and intermodule paths.		
Syntax:			
Primitives	`acc_fetch_delays(object_handle, rise_delay, fall_delay, z_delay)`		
Module paths Intermodule paths Ports or port bits	`acc_fetch_delays(object_handle,` `d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12)`		
Timing checks	`acc_fetch_delays(object_check_handle, limit)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	`handle`	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	`double *`	rise_delay fall_delay	Rise and fall delay for 2-state primitive or 3-state primitive
Conditional	`double *`	z_delay	Turn-off (to Z) transition delay for 3-state primitives
	`double *`	d1	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 1 : delay for all transitions If **accPathDelayCount** is set to 2 or 3 : rise transition delay If **accPathDelayCount** is set to 6 or 12 : **0**->**1** transition delay
Conditional	`double *`	d2	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 2 or 3 : fall transition delay If **accPathDelayCount** is set to 6 or 12 : **1**->**0** transition delay
Conditional	`double *`	d3	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 3 : turn-off transition delay If **accPathDelayCount** is set to 6 or 12 : **0**->**Z** transition delay
Conditional	`double *`	d4 d5 d6	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 6 or 12 : `d4` is **Z**->**1** transition delay `d5` is **1**->**Z** transition delay `d6` is **Z**->**0** transition delay
Conditional	`double *`	d7 d8 d9 d10 d11 d12	For module/intermodule paths and input ports/port bits: If **accPathDelayCount** is set to 12 : `d7` is **0**->**X** transition delay `d8` is **X**->**1** transition delay `d9` is **1**->**X** transition delay `d10` is **X**->**0** transition delay `d11` is **X**->**Z** transition delay `d12` is **Z**->**X** transition delay
	`double *`	limit	Limit of timing check

Copyright © 2001 IEEE. All rights reserved.

acc_fetch_delays() for min:typ:max delays (accMinTypMaxDelays set to true)			
Synopsis:	Get existing delay values for primitives, module paths, timing checks, module input ports, or intermodule paths; the delay values are contained in an array.		
Syntax:	`acc_fetch_delays(object_handle, array_ptr),`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	`handle`	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of **accPathDelayCount** (see Section 22.8)

The ACC routine **acc_fetch_delays()** shall work differently depending on how the configuration parameter **accMinTypMaxDelays** is set. When this parameter is set to "`false`", a single delay per transition shall be assumed, and each delay shall be fetched into variables pointed to as individual arguments. For this *single delay mode*, the first syntax table in this section shall apply.

When **accMinTypMaxDelays** is set to "`true`", **acc_fetch_delays()** shall fetch one or more sets of minimum:typical:maximum delays into an array, rather than single delays fetched as individual arguments. For this min:typ:max delay mode, the second syntax table in this section shall apply.

The number of delay values that shall be fetched by **acc_fetch_delays()** is determined by the type of object and the setting of configuration parameters. Refer to 22.8 for a description of how the number of delay values is determined.

The ACC routine **acc_fetch_delays()** shall retrieve delays in the timescale of the module that contains the object_handle.

The example shown in Figure 73 presents a C language application that uses **acc_fetch_delays()** to retrieve the rise, fall, and turn-off delays of all paths through a module.

Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"

oid display_path_delays()

  handle   mod_handle;
  handle   path_handle;
  double   rise_delay,fall_delay,toz_delay;

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

  /*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 73—Using acc_fetch_delays() in single delay mode

The example shown in Figure 74 is a C language code fragment of an application that shows how to fetch min:typ:max delays for the intermodule paths. The example follows these steps:

a) Declares 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.

b) Sets the configuration parameter **accMinTypMaxDelays** to `"true"` to instruct **acc_fetch_delays()** to retrieve delays in min:typ:max format.

c) Calls **acc_fetch_delays()** with a valid intermodule path handle and the array pointer.

 Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"
oid fetch_mintypmax_delays(port_output, port_input)
andle 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 74—Using acc_fetch_delays() in min:typ:max delay mode

23.16 acc_fetch_direction()

acc_fetch_direction()			
Synopsis:	Get the direction of a port or terminal.		
Syntax:	`acc_fetch_direction(object_handle)`		
	Type	Description	
Returns:	PLI_INT32	A predefined integer constant representing the direction of a port or terminal **accInput** **accOutput** **accInout** **accMixedIo**	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of a port or terminal

Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_fetch_direction()** shall return a predefined integer constant indicating the direction of a module port or primitive terminal. The values returned are given in Table 139.

Table 139—The operation of acc_fetch_direction()

When direction is	acc_fetch_direction() shall return
Input only	**accInput**
Output only	**accOutput**
Bidirectional (input and output)	**accInout**
A concatenation of input ports and output ports	**accMixedIo**

The example shown in Figure 75 presents a C language application that uses **acc_fetch_direction()** to determine whether or not a port is an input.

```
include "acc_user.h"

nt      is_port_input(port_handle)
andle   port_handle;

  PLI_INT32  direction;

  direction = acc_fetch_direction(port_handle);
  if (direction == accInput || direction == accInout)
     return(true);
  else
     return(false);
```

Figure 75—Using acc_fetch_direction()

23.17 acc_fetch_edge()

acc_fetch_edge()			
Synopsis:	Get the edge specifier of a module path or timing check terminal.		
Syntax:	`acc_fetch_edge(pathio_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	A predefined integer constant representing the edge specifier of a path input or output terminal: **accNoedge** **accEdge01** **accEdgex1** **accPosedge** **accEdge10** **accEdge1x** **accNegedge** **accEdge0x** **accEdgex0**	
Arguments:	**Type**	**Name**	**Description**
	handle	pathio_handle	Handle to a module path input or output, or handle to a timing check terminal

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_fetch_edge()** shall return a value that is a masked integer representing the edge specifier for a module path or timing check terminal.

Table 140 lists the predefined edge specifiers as they are specified in `acc_user.h`.

Table 140—Edge specifiers constants

Edge type	Defined constant	Binary value
None	**accNoedge**	0
Positive edge (0→1,0→x,x→1)	**accPosedge**	00001101
Negative edge (1→0,1→x,x→0)	**accNegedge**	00110010
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

The integer mask returned by **acc_fetch_edge()** is usually either **accPosedge** or **accNegedge**. Occasionally, however, the mask is a hybrid mix of specifiers that is equal to neither. The example shown in Figure 76 illustrates how to check for these hybrid edge specifiers. The value **accNoEdge** is returned if no edge is found.

The 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.

Copyright © 2001 IEEE. All rights reserved.

This example first checks to see whether the returned mask is equal to **accPosedge** or **accNegedge**, which are the most likely cases. If it is not, the application 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"`.

```
PLI_BYTE8 *acc_fetch_edge_str(pathio)
handle pathio;

   PLI_INT32 edge = acc_fetch_edge(pathio);
   static PLI_BYTE8 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 76—Using acc_fetch_edge()

 Copyright © 2001 IEEE. All rights reserved.

23.18 acc_fetch_fullname()

acc_fetch_fullname()			
Synopsis:	Get the full hierarchical name of any named object or module path.		
Syntax:	`acc_fetch_fullname(object_handle)`		
	Type	Description	
Returns:	PLI_BYTE8 *	Character pointer to a string containing the full hierarchical name of the object	
	Type	**Name**	**Description**
Arguments:	handle	object_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		

The ACC routine **acc_fetch_fullname()** shall obtain the *full hierarchical name* of an object. The full hierarchical name is the name that uniquely identifies an object. In Figure 84, the top-level module, `top1`, contains module instance `mod3`, which contains net `w4`. In this example, the full hierarchical name of the net is `top1.mod3.w4`.

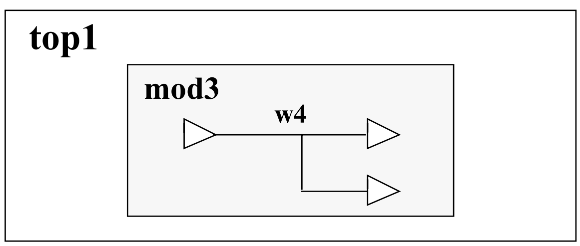


Figure 77—A design hierarchy; the fullname of net w4 is "top1.mod3.w4"

Table 141 lists the objects in a Verilog HDL description for which **acc_fetch_fullname()** shall return a name.

Table 141—Named objects supported by acc_fetch_fullname()

Modules	Integer, time and real variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs or Variables	Verilog HDL functions

Copyright © 2001 IEEE. All rights reserved.

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

source	path_delimiter	destination

By default, the *path_delimiter* shall be the character **$**. However, the delimiter can be changed by using the ACC 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 **$**. Note that names of module paths with multiple sources or destinations shall be derived from the first source and destination only.

Table 142—Module path names returned by acc_fetch_fullname()

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

If a Verilog software product creates default names for unnamed instances, **acc_fetch_fullname()** shall return the full hierarchical default name. Otherwise, the routine shall return `null` for unnamed instances.

Using **acc_fetch_fullname()** with a module port handle shall return the full hierarchical implicit name of the port.

The routine **acc_fetch_fullname()** shall store the returned string in a temporary buffer. To preserve the string for later use in an application, it should be copied to another variable (refer to 22.9).

In the example shown in Figure 78, the routine uses **acc_fetch_fullname()** to display the full hierarchical name of an object if the object is a net.

```
#include "acc_user.h"

PLI_INT32 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 78—Using acc_fetch_fullname()

 Copyright © 2001 IEEE. All rights reserved.

23.19 acc_fetch_fulltype()

acc_fetch_fulltype()			
Synopsis:	Get the fulltype of an object.		
Syntax:	`acc_fetch_fulltype(object_handle)`		
	Type	Description	
Returns:	PLI_INT32	A predefined integer constant from the list shown in 22.6	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of the object
Related routines:	Use acc_fetch_type() to get the general type classification of an object Use acc_fetch_type_str() to get the fulltype as a character string		

The ACC routine **acc_fetch_fulltype()** shall return the *fulltype* of an object. The fulltype is a specific classification of a Verilog HDL object, represented as a predefined constant (defined in `acc_user.h`). Table 113 lists all of the fulltype constants that can be returned by **acc_fetch_fulltype()**.

Many Verilog HDL objects have both a *type* and a *fulltype*. The type of an object is its general Verilog HDL type classification. The fulltype is the specific type of the object. The examples in Table 143 illustrate the difference between the type of an object and the fulltype of the same object for selected objects.

Table 143—The difference between the type and the fulltype of an object

For a handle to	acc_fetch_type() shall return	acc_fetch_fulltype() shall return
A setup timing check	**accTchk**	**accSetup**
An `and` gate primitive	**accPrimitive**	**accAndGate**
A sequential UDP	**accPrimitive**	**accSeqPrim**

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 79 uses **acc_fetch_fulltype()** to find and display the fulltypes of timing checks. This application is called by a higher-level application, `display_object_type`, presented as the usage example for **acc_fetch_type()**.

```
#include "acc_user.h"

PLI_INT32 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 79—Using acc_fetch_fulltype() to display the fulltypes of timing checks

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 80 uses **acc_fetch_fulltype()** to find and display the fulltypes of primitive objects passed as input arguments. This application is called by a higher-level application, `display_object_type`, presented as the usage example for **acc_fetch_type()**.

```
:include "acc_user.h"

'LI_INT32 display_primitive_type(primitive_handle)
ıandle            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 80—Using acc_fetch_fulltype() to display the fulltypes of primitives

Copyright © 2001 IEEE. All rights reserved.

23.20 acc_fetch_index()

acc_fetch_index()			
Synopsis:	Get the index number for a port or terminal.		
Syntax:	`acc_fetch_index(object_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	Integer index for a port or terminal, starting with zero	
Arguments:	**Type**	**Name**	**Description**
	`handle`	object_handle	Handle of the port or terminal

The ACC routine **acc_fetch_index()** shall return the index number for a module port or primitive terminal. Indices are integers that shall start at zero and increase from left to right.

The index of a *port* shall be its position in a module definition in the Verilog HDL source description.
The index of a *terminal* shall be its position in a gate, switch, or UDP instance.

Table 144 shows how indices shall be derived.

Table 144—Deriving indices

For	Indices are
Terminals: `nand g1(out, in1, in2);`	**0** for terminal `out` **1** for terminal `in1` **2** for terminal `in2`
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));`
`endmodule`
 	
`module 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`

The example shown in Figure 81 presents a C language application that uses **acc_fetch_index()** to find and display the input ports of a module.

 Copyright © 2001 IEEE. All rights reserved.

```
:include "acc_user.h"

'LI_INT32 display_inputs(module_handle)
ıandle module_handle;

  handle      port_handle;
  PLI_INT32   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 81—Using acc_fetch_index()

23.21 acc_fetch_location()

acc_fetch_location()			
Synopsis:	Get the location of an object in a Verilog-HDL source file.		
Syntax:	`acc_fetch_location(loc_p, object_handle)`		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	p_location	loc_p	Pointer to a predefined location structure
	handle	object_handle	Handle to an object

The ACC routine **acc_fetch_location()** shall return the *file name* and *line number* in the file for the specified object. The file name and line number shall be returned in an `s_location` data structure. This data structure is defined in `acc_user.h`, and listed in Figure 82.

```
typedef struct t_location
{
  PLI_INT32 line_no;
  PLI_BYTE8 *filename;
} s_location, *p_location;
```

Figure 82—s_location data structure

Copyright © 2001 IEEE. All rights reserved.

filename field is a character pointer.

line_no field is a nonzero positive integer.

Table 141 lists the objects that shall be supported by **acc_fetch_location()**.

Table 145—Objects supported by acc_fetch_location()

Object type	Location returned
Modules	Module instantiation line
Module ports	Module definition
Module paths	Module path line
Data paths	Module path line
Primitives	Instantiation line
Explicit nets	Definition line
Implicit nets	Line where first used
Reg variables	Definition line
Integer, time and real variables	Definition line
Named events	Definition line
Parameters	Definition line
Specparams	Definition line
Named blocks	Definition line
Verilog HDL tasks	Definition line
Verilog HDL functions	Definition line

The return value for *filename* is placed in the ACC internal string buffer. See 22.9 for an explanation of strings in ACC routines.

The example shown in Figure 83 uses **acc_fetch_location()** to print the file name and line number for an object.

```
LI_INT32 find_object_location (object)
 handle object;

 s_location s_loc;
 p_location loc_p = &s_loc;
 acc_fetch_location(loc_p, object); /*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 83—Using acc_fetch_location()

 Copyright © 2001 IEEE. All rights reserved.

23.22 acc_fetch_name()

acc_fetch_name()			
Synopsis:	Get the instance name of any named object or module path.		
Syntax:	`acc_fetch_name(object_handle)`		
Returns:	**Type**	Description	
	PLI_BYTE8 *	Character pointer to a string containing the instance name of the object	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle of the named object
Related routines:	Use acc_fetch_fullname() to get the full hierarchical name of the object Use acc_fetch_defname() to get the definition name of the object Use acc_configure(accPathDelimStr...) to set the naming convention for module paths		

The ACC routine **acc_fetch_name()** shall obtain the *name* of an object. The name of an object is its lowest-level name. In the following example, the top-level module, `top1`, contains module instance `mod3`, which contains net w4, as shown in Figure 84. In this example, the name of the net is `w4`.

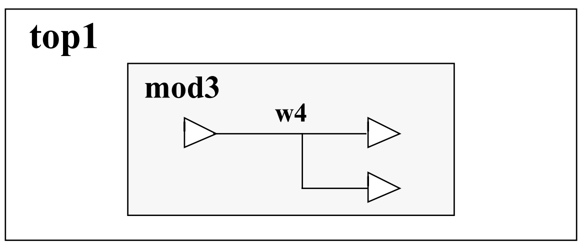


Figure 84—A design hierarchy; the name of net w4 is "w4"

The return value for this routine is placed in the ACC internal string buffer. See 22.9 for an explanation of strings in ACC routines.

Table 141 lists the objects in a Verilog HDL description for which **acc_fetch_name()** shall return a name.

Table 146—Named objects supported by acc_fetch_name()

Modules	Integer, time and real variables
Module ports	Named events
Module paths	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs or Variables	Verilog HDL functions

Copyright © 2001 IEEE. All rights reserved.

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

source	path_delimiter	destination

By default, the *path_delimiter* is the character **$**. However, the delimiter can be changed by using the ACC routine *acc_configure()* to set the delimiter parameter **accPathDelimStr** to another character string.

Table 147 shows names of paths within a top-level module `m3`, as returned by **acc_fetch_name()** when the *path_delimiter* is **$**. Note that names of module paths with multiple sources or destinations shall be derived from the first source and destination only.

Table 147—Module path names returned by acc_fetch_name()

For paths in module *m3*	acc_fetch_name() returns a pointer to
`(a => q) = 10;`	**a$q**
`(b *> q1,q2) = 8;`	**b$q1**
`(d,e,f *> r,s)= 8;`	**d$r**

If a Verilog software implementation creates default names for unnamed instances, **acc_fetch_name()** shall return the default name. Otherwise, the routine shall return `null` for unnamed instances.

Using **acc_fetch_name()** with a module port handle shall return the implicit name of the port.

The following example uses **acc_fetch_name()** to display the names of top-level modules.

```
include "acc_user.h"
LI_INT32 show_top_mods()

  handle   module_handle;

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

      /*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 85—Using acc_fetch_name()

 Copyright © 2001 IEEE. All rights reserved.

23.23 acc_fetch_paramtype()

acc_fetch_paramtype()			
Synopsis:	Get the data type of a parameter or specparam.		
Syntax:	`acc_fetch_paramtype(parameter_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	A predefined integer constant representing the data type of a parameter: **accIntParam** **accIntegerParam** **accRealParam** **accStringParam**	
Arguments:	**Type**	**Name**	**Description**
	handle	parameter_handle	Handle to a parameter or specparam
Related routines:	Use acc_next_parameter() to get all parameters within a module Use acc_next_specparam() to get all specparams within a module		

The ACC routine **acc_fetch_paramtype()** shall return an integer constant that represents the data type of a value that has been assigned to a parameter or specparam.

Figure 86 uses **acc_fetch_paramtype()** to display the values of all parameters within a module.

```
nclude "acc_user.h"
I_INT32 print_parameter_values()

andle module_handle, param_handle;

*initialize environment for ACC routines*/
cc_initialize();

odule_handle = acc_handle_tfarg(1);
aram_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;
  }
}
cc_close();
```

Figure 86—Using acc_fetch_paramtype()

Copyright © 2001 IEEE. All rights reserved.

23.24 acc_fetch_paramval()

acc_fetch_paramval()			
Synopsis:	Get the value of a parameter or specparam.		
Syntax:	`acc_fetch_paramval(parameter_handle)`		
Returns:	**Type**	Description	
	double	The value of a parameter or specparam	
Arguments:	**Type**	**Name**	**Description**
	handle	parameter_handle	Handle to a parameter or specparam
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		

The ACC routine **acc_fetch_paramtype()** shall return the value stored in a parameter or specparam. The value shall be returned as a double-precision floating-point number.

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

A double-precision floating-point number
An integer value
A string

Therefore, it can be necessary to call **acc_fetch_paramtype()** to determine the data type of the parameter value, as shown in the example in Figure 87.

The routine **acc_fetch_paramval()** returns values as type `double`. The values can be converted back to integers or character pointers using the C language *cast* mechanism, as shown in Table 148. Note that some C language compilers do not allow casting a double-precision value directly to a character pointer; it is therefore necessary to use a two-step cast to first convert the double value to an integer and then convert the integer to a character pointer.

If a character string is returned, it is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

Table 148—Casting acc_fetch_paramval() return values

To convert to	Follow these steps
Integer	Cast the return value to the integer data type using the C language cast operator **(int)**: `int_val=` **(int)** `acc_fetch_paramval(...);`
String	Cast the return value to a character pointer using the C language cast operators **(char*)(int)**: `str_ptr=` **(char*)(int)** `acc_fetch_paramval(...);`

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 87 presents a C language application, `print_parameter_values`, that uses **acc_fetch_paramtype()** to display the values of all parameters within a module.

```
nclude "acc_user.h"

I_INT32 print_parameter_values()

 handle   module_handle;
 handle   param_handle;

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

 /*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 87—Using acc_fetch_paramval()

Copyright © 2001 IEEE. All rights reserved.

23.25 acc_fetch_polarity()

acc_fetch_polarity()			
Synopsis:	Get the polarity of a path.		
Syntax:	`acc_fetch_polarity(path_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	A predefined integer constant representing the polarity of a path: **accPositive** **accNegative** **accUnknown**	
Arguments:	**Type**	**Name**	**Description**
	handle	path_handle	Handle to a module path or data path

The ACC routine **acc_fetch_polarity()** shall return an integer constant that represents the polarity of the specified path. The polarity of a path describes how a signal transition at its source propagates to its destination in the absence of logic simulation events. The return value shall be one of the predefined integer constant polarity types listed in Table 149.

Table 149—Polarity types returned by acc_fetch_polarity()

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.

The example shown in Figure 88 takes a path argument and returns the string corresponding to its polarity.

```
PLI_BYTE8 *fetch_polarity_str(path)
{
   switch (acc_fetch_polarity(path)) {
      case accPositive:  return(“accPositive”);
      case accNegative: return(“accNegative”);
      case accUnknown: return(“accUnknown”);
      default: return(null);
   }
}
```

Figure 88—Using acc_fetch_polarity()

 Copyright © 2001 IEEE. All rights reserved.

23.26 acc_fetch_precision()

acc_fetch_precision()			
Synopsis:	Get the smallest time precision argument specified in all `` `timescale `` compiler directives in a given design.		
Syntax:	`acc_fetch_precision()`		
	Type	Description	
Returns:	PLI_INT32	An integer value that represents a time precision	
	Type	**Name**	**Description**
Arguments:	None		
Related routines:	Use acc_fetch_timescale_info() to get the timescale and precision of a specific object		

The ACC routine **acc_fetch_precision()** shall return the smallest time precision argument specified in all `` `timescale `` compiler directives for a given design. The value returned shall be the order of magnitude of one second, as shown in Table 150.

Table 150—Value returned by acc_fetch_precision()

Integer value returned	Simulation time precision represented
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 s
-5	10 s
-6	1 s
-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

If there are no `` `timescale `` compiler directives specified for a design, **acc_fetch_precision()** shall return a value of 0 (1 s).

Copyright © 2001 IEEE. All rights reserved.

23.27 acc_fetch_pulsere()

acc_fetch_pulsere()			
Synopsis:	Get current pulse handling *reject_limit* and *e_limit* for a module path, intermodule path or module input port.		
Syntax:	`acc_fetch_pulsere(object,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)`		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	handle	object	Handle of module path, intermodule path or module input port
	double *	r1...r12	*reject_limit* values; the number of arguments is determined by **accPathDelayCount**
	double *	e1...e12	*e_limit* values; the number of arguments is determined by **accPathDelayCount**
Related routines:	Use acc_append_pulsere() to add to the existing pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_fetch_pulsere()** shall obtain the current values controlling how pulses are propagated through a module path, intermodule path or module input port.

A *pulse* is defined as two transitions that occur in a shorter period of time than the delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an *error*. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where

The reject_limit shall set a threshold for determining when to reject a pulse any pulse less than the reject_limit shall not propagate
The e_limit shall set a threshold for determining when a pulse is an error any pulse less than the e_limit and greater than or equal to the reject_limit shall propagate a logic x
A pulse that is greater than or equal to the e_limit shall propagate
Table 151 illustrates the relationship between the reject_limit and the e_limit.

Table 151—Pulse control example

When	The pulse shall be
reject_limit = 10.5 **e_limit** = 22.6	Rejected if < 10.5 An error if >= 10.5 and < 22.6 Passed if >= 22.6

The number of pulse control values that **acc_fetch_pulsere()** shall retrieve is controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 152.

 Copyright © 2001 IEEE. All rights reserved.

Table 152—How the accPathDelayCount affects acc_fetch_pulsere()

When accPathDelayCount is	acc_fetch_pulsere() shall retrieve
1	One pair of reject_limit and e_limit values: one pair for all transitions, r1 and e1
2	Two pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
3	Three pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2 one pair for turn-off transitions, r3 and e3
6 (the default)	Six pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6
12	Twelve pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, X, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6 one pair for 0->X transitions, r7 and e7 one pair for X->1 transitions, r8 and e8 one pair for 1->X transitions, r9 and e9 one pair for X->0 transitions, r10 and e10 one pair for X->Z transitions, r11 and e11 one pair for Z->X transitions, r12 and e12

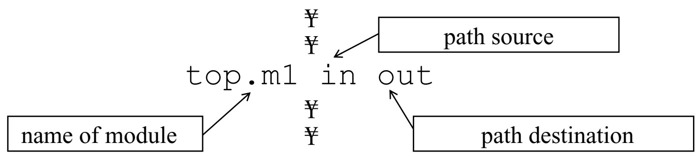
The minimum number of pairs of reject_limit and e_limit arguments to pass to **acc_fetch_pulsere()** shall equal the value of **accPathDelayCount**. Any unused reject_limit and e_limit argument pairs shall be ignored by **acc_fetch_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to 6, and therefore six pairs of pulse reject_limit and e_limit arguments have to be used when **acc_fetch_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays(), acc_fetch_delays(), acc_replace_delays(), acc_append_pulsere()**, and **acc_replace_pulsere()**.

Pulse control values shall be retrieved using the timescale of the module that contains the object handle.

The example shown in Figure 89 shows how an application, `get_pulsevals`, uses **acc_fetch_pulsere()** to retrieve rise and fall pulse handling values of paths listed in a file called `path.dat`. The format of the file is shown in the following diagram.

Copyright © 2001 IEEE. All rights reserved.




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

define NAME_SIZE 256
LI_INT32 get_pulsevals()

FILE       *infile;
PLI_BYTE8 mod_name[ NAME_SIZE] ;
PLI_BYTE8 pathin_name[ NAME_SIZE] , pathout_name[ NAME_SIZE] ;
handle    mod, path;
double    rise_reject_limit,rise_e_limit,fall_reject_limit,fall_e_limit;

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

  /*set accPathDelayCount to return two pairs of pulse handling values,*
  /*  one each for rise and fall transitions*/
  acc_configure(accPathDelayCount, "2");

  /*read all module path specifications from file "path.dat"*/
  infile = fopen("path.dat", "r");
  while(fscanf(infile, "%s %s %s"
           mod_name,pathin_name,pathout_name)!=EOF)
  {
     mod=acc_handle_object(mod_name);
     path=acc_handle_modpath(mod,pathin_name,pathout_name);
     if(acc_fetch_pulsere(path,
                         &rise_reject_limit,&rise_e_limit,
                         &fall_reject_limit, &fall_e_limit))
     {
        io_printf("rise reject limit = %lf, rise e limit = %lf\n",
                 rise_reject_limit, rise_e_limit);
        io_printf("fall reject limit = %lf, fall e limit = %lf\n",
                 fall_reject_limit, fall_e_limit);
  }
  }
  acc_close();
```

Figure 89—Using acc_fetch_pulsere()

 Copyright © 2001 IEEE. All rights reserved.

23.28 acc_fetch_range()

acc_fetch_range()			
Synopsis:	Get the most significant bit and least significant bit range values for a vector.		
Syntax:	`acc_fetch_range(vector_handle, msb, lsb)`		
	Type	Description	
Returns:	PLI_INT32	Zero if successful; nonzero upon error	
	Type	**Name**	**Description**
Arguments:	handle	vector_handle	Handle to a vector net or reg
	PLI_INT32 *	msb	Pointer to an integer variable to hold the most significant bit of vector_handle
	PLI_INT32 *	lsb	Pointer to an integer variable to hold the least significant bit of vector_handle
Related routines	Use acc_fetch_size() to get the number of bits in a vector		

The ACC routine **acc_fetch_range()** shall obtain the most significant bit (msb) and least significant bit (lsb) numbers of a vector.

The *msb* shall be the left range element, while the *lsb* shall be the right range element in the Verilog HDL source code.

The example shown in Figure 90 takes a handle to a module instance as its input. It then uses **acc_fetch_range()** to display the name and range of each vector net found in the module as: `<name>[<msb>:<lsb>]`.

```
PLI_INT32 display_vector_nets()
{
   handle     mod = acc_handle_tfarg(1);
   handle     net;
   PLI_INT32  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 90—Using acc_fetch_range()

Copyright © 2001 IEEE. All rights reserved.

23.29 acc_fetch_size()

acc_fetch_size()			
Synopsis:	Get the bit size of a net, reg, integer, time, real or port.		
Syntax:	acc_fetch_size(object_handle)		
	Type	Description	
Returns:	PLI_INT32	Number of bits in the net, reg, integer, time, real or port	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle to a net, reg, integer, time, real or port, or a bit-select or part select thereof

The ACC routine **acc_fetch_size()** shall return the number of bits of a net, reg, integer, time, real or port.

The example shown in Figure 91 uses **acc_fetch_size()** to display the size of a vector net.

```
include "acc_user.h"

LI_INT32 display_vector_size()

  handle      net_handle;
  PLI_INT32   size_in_bits;

  /* reset environment for ACC routines */
  acc_initialize();

  /*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 91—Using acc_fetch_size()

 Copyright © 2001 IEEE. All rights reserved.

23.30 acc_fetch_tfarg(), acc_fetch_itfarg()

acc_fetch_tfarg(), acc_fetch_itfarg()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as a double-precision number.		
Syntax:	`acc_fetch_tfarg(argument_number)` `acc_fetch_itfarg(argument_number, tfinst)`		
	Type	Description	
Returns:	double	The value of the task/function argument, returned as a double-precision number	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg_int() or acc_fetch_itfarg_int() to get the task/function argument value as an integer Use acc_fetch_tfarg_str() or acc_fetch_itfarg_str() to get the task/function argument value as a string Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg()** shall return the value of arguments passed to the current instance of a user-defined system task or function. The ACC routine **acc_fetch_itfarg()** shall return the value of arguments passed to a specific instance of a user-defined system task or function, using a handle to the task or function. The value is returned as a double-precision floating-point number.

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

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg()** and **acc_fetch_itfarg()** shall return a value of `0.0`, and generate a warning message if warnings are enabled. Note that the `acc_error_flag` is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg()** and **acc_fetch_itfarg()** shall return a value of `0.0` and generate a warning message if warnings are enabled. Literal numbers, nets, regs, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the `acc_error_flag` is not set when a nonvalued argument is referenced.

The routine **acc_fetch_tfarg()** returns values as type `double`. The routines **acc_fetch_tfarg_int()** and **acc_fetch_tfarg_str()** return values as integers or string pointers, respectively. The value returned by **acc_fetch_tfarg()** can also be converted to integers or character pointers using the C language *cast* mechanism, as shown in Table 153. Note that some C language compilers do not allow casting a double-precision value directly to a character pointer; it is therefore necessary to use a two-step cast to first convert the double value to an integer and then convert the integer to a character pointer. If a character string is returned, it is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

Copyright © 2001 IEEE. All rights reserved.

Table 153—Casting acc_fetch_tfarg() return values

To convert to	Follow these steps
Integer	Cast the return value to the integer data type using the C language cast operator **(PLI_INT32)**: `int_val=` **(PLI_INT32)** `acc_fetch_tfarg(...);`
String	Cast the return value to a character pointer using the C language cast operators **(char*)(int)**: `str_ptr=` **(char*)(int)** `acc_fetch_tfarg(...);`

The example shown in Figure 92 uses **acc_fetch_tfarg()**, **acc_fetch_tfarg_int()**, and **acc_fetch_tfarg_str()** to return the value of the first argument of a user-defined system task or function.

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

PLI_INT32 display_arg_value()

   PLI_INT32   arg_type;

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

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

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

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

returns value as an **integer number**

returns value as a **pointer to a character string**

Figure 92—Using acc_fetch_tfarg(), acc_fetch_tfarg_int(), and acc_fetch_tfarg_str()

 Copyright © 2001 IEEE. All rights reserved.

23.31 acc_fetch_tfarg_int(), acc_fetch_itfarg_int()

acc_fetch_tfarg_int(), acc_fetch_itfarg_int()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as an integer number.		
Syntax:	`acc_fetch_tfarg_int(argument_number)` `acc_fetch_itfarg_int(argument_number, tfinst)`		
	Type	Description	
Returns:	PLI_INT32	The value of the task/function argument, returned as an integer number	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg() or acc_fetch_itfarg() to get the task/function argument value as a double Use acc_fetch_tfarg_str() or acc_fetch_itfarg_str() to get the task/function argument value as a string Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg_int()** shall return the value of arguments passed to the current user-defined system task or function. The ACC routine **acc_fetch_itfarg_int()** shall return the value of arguments passed to a specific instance of a user-defined system task and function, using a handle to the task or function. The value is returned as an integer number.

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

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg_int()** and **acc_fetch_itfarg_int()** shall return a value of `0` and generate a warning message if warnings are enabled. Note that the `acc_error_flag` is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg_int()** and **acc_fetch_itfarg_int()** shall return a value of `0` and generate a warning message if warnings are enabled. Literal numbers, nets, regs, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the `acc_error_flag` is not set when a nonvalued argument is referenced.

If a user-defined task/function argument is a real value, the value is cast to a PLI_INT32 and returned as an integer. If the task/function argument is a string value, the string is copied into the ACC string buffer and the pointer to the string is cast to the type PLI_INT32 and returned as an integer.

Refer to Figure 92 for an example of using **acc_fetch_tfarg_int()**.

Copyright © 2001 IEEE. All rights reserved.

23.32 acc_fetch_tfarg_str(), acc_fetch_itfarg_str()

acc_fetch_tfarg_str(), acc_fetch_itfarg_str()			
Synopsis:	Get the value of the specified argument of the system task or function associated with the PLI application; the value is returned as a pointer to a character string.		
Syntax:	`acc_fetch_tfarg_str(argument_number)` `acc_fetch_itfarg_str(argument_number, tfinst)`		
	Type	Description	
Returns:	PLI_BYTE8 *	The value of the task/function argument, returned as a pointer to a character string	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	argument_number	Integer number that references the system task or function argument by its position in the argument list
	handle	tfinst	Handle to a specific instance of a user-defined system task or function
Related routines:	Use acc_fetch_tfarg() or acc_fetch_itfarg() to get the task/function argument value as a double Use acc_fetch_tfarg_int() or acc_fetch_itfarg_int() to get the task/function argument value as an integer Use acc_handle_tfinst() to get a handle to a specific instance of a user-defined system task or function		

The ACC routine **acc_fetch_tfarg_str()** shall return the value of arguments passed to the current instance of a user-defined system task or function. The ACC routine **acc_fetch_itfarg_str()** shall return the value of arguments passed to a specific instance of a user-defined system task or function, using a handle to the task or function. The value shall be returned as a pointer to a character string. The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

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

If an argument number is passed in that is out of range for the number of arguments in the user-defined system task/function call, **acc_fetch_tfarg_str()** and **acc_fetch_itfarg_str()** shall return a value of `null` and generate a warning message if warnings are enabled. Note that the `acc_error_flag` is not set for an out-of-range index number.

If a user-defined system task/function argument that does not represent a valued object is referenced, **acc_fetch_tfarg_str()** and **acc_fetch_itfarg_str()** shall return a value of `null` and generate a warning message if warnings are enabled. Literal numbers, nets, regs, integer variables, and real variables all have values. Objects such as module instance names do not have a value. Note that the `acc_error_flag` is not set when a nonvalued argument is referenced.

If a user-defined task/function argument is a value, each 8 bits of the value are converted into its equivalent ASCII character.

Refer to Figure 92 for an example of using **acc_fetch_tfarg_str()**.

 Copyright © 2001 IEEE. All rights reserved.

23.33 acc_fetch_timescale_info()

acc_fetch_timescale_info()			
Synopsis:	Get timescale information for an object or for an active $timeformat system task invocation.		
Syntax:	`acc_fetch_timescale_info(object_handle, timescale_p)`		
	Type	Description	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of a module instance, module definition, PLI user-defined system task/function call, or `null`
	p_timescale_info	timescale_p	Pointer to a variable defined as a `s_timescale_info` structure
Related routines:	Use acc_fetch_precision() to fetch the smallest timescale precision in a design		

The ACC routine **acc_fetch_timescale_info()** shall obtain the timescale information for an object or for an active `$timeformat` built-in system task invocation. The timescale returned shall be based on the type of object handle, as defined in Table 154.

Table 154—Return values from acc_fetch_timescale_info()

If the object_handle is	acc_fetch_timescale_info() shall return
A handle to a module instance or module definition	The timescale for the corresponding module definition
A handle to a user-defined system task or function	The timescale for the corresponding module definition that represents the parent module instance of the object
`null`	The timescale for an active `$timeformat` system task invocation

The routine **acc_fetch_timescale_info()** shall return a value to an `s_timescale_info` structure pointed to by the *timescale_p* argument. This structure is declared in the file `acc_user.h`, as shown in Figure 82.

```
typedef struct t_timescale_info
{
  PLI_INT16 unit;
  PLI_INT16 precision;
} s_timescale_info, *p_timescale_info;
```

Figure 93—s_timescale_info data structure

The term unit is a short integer that shall represent the timescale unit in all cases of *object*
The term precision is a short integer that shall represent the timescale precision. In the case of a `null` object handle, precision shall be the number of decimal points specified in the active `$timeformat` system task invocation.

Copyright © 2001 IEEE. All rights reserved.

The value returned for *unit* and *precision* shall be the order of magnitude of 1 s, as shown in Table 155.

Table 155—Value returned by acc_fetch_timescale_info()

Integer value returned	Time unit r
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 s
-5	10 s
-6	1 s
-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

For example, a call to

```
acc_fetch_timescale_info(obj, timescale_p)
```

Where `obj` is defined in a module that has `` `timescale 1us/1ns `` specified for its definition, shall return

```
timescale_p->unit: -6
timescale_p->precision: -9
```

23.34 acc_fetch_type()

acc_fetch_type()			
Synopsis:	Get the type of an object.		
Syntax:	`acc_fetch_type(object_handle)`		
Returns:	**Type**	Description	
	PLI_INT32	A predefined integer constant from the list shown in 22.6	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle of the object
Related routines:	Use acc_fetch_fulltype() to get the full type classification of an object Use acc_fetch_type_str() to get the type as a character string		

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_fetch_type()** shall return the type of an object. The *type* is a general classification of a Verilog HDL object, represented as a predefined constant (defined in `acc_user.h`). Refer to Table 113 for a list of all of the *type* constants that can be returned by **acc_fetch_type()**.

Many Verilog HDL objects can have a *type* and a *fulltype*. The type of an object is its general Verilog HDL type classification. The fulltype is the specific type of the object. Table 143 illustrates the difference between the type of an object and the fulltype of the same object.

Table 156—The difference between the type and the fulltype of an object

For a handle to	acc_fetch_type() shall return	acc_fetch_fulltype() shall return
A setup timing check	**accTchk**	**accSetup**
An `and` gate primitive	**accPrimitive**	**accAndGate**
A sequential UDP	**accPrimitive**	**accSeqPrim**

The example shown in Figure 94 uses **acc_fetch_type()** to identify the type of an object (the functions `display_primitive_type` and `display_timing_check_type` used in this example are presented in the usage examples in 23.19).

Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"

LI_INT32 display_object_type()

  handle object_handle;

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

  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 94—Using acc_fetch_type()

 Copyright © 2001 IEEE. All rights reserved.

23.35 acc_fetch_type_str()

acc_fetch_type_str()			
Synopsis:	Get a string that indicates the type of its argument.		
Syntax:	`acc_fetch_type_str(type)`		
Returns:	**Type**	Description	
	PLI_BYTE8 *	Pointer to a character string	
Arguments:	**Type**	**Name**	**Description**
	PLI_INT32	type	A predefined integer constant that stands for an object type or fulltype
Related routines:	Use acc_fetch_type() to get the type of an object as an integer constant Use acc_fetch_fulltype() to get the fulltype of an object as an integer constant		

The ACC routine **acc_fetch_type_str()** shall return the character string that specifies the type of its argument. The argument passed to **acc_fetch_type_str()** should be an integer value returned from either **acc_fetch_type()** or **acc_fetch_fulltype()**.

The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

In the example shown in Figure 95, a handle to an argument is passed to a C application. The application displays the name of the object and the type of the object.

```
#include "acc_user.h"
PLI_INT32 display_object_type(object)
handle object;
{
  PLI_INT32 type = acc_fetch_type(object);

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

Figure 95—Using acc_fetch_type_str()

In this example, if the application is passed a handle to an object named `top.param1`, the application shall produce the following output:

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

The output string, **accParameter**, is the name of the integer constant that represents the parameter type.

Copyright © 2001 IEEE. All rights reserved.

23.36 acc_fetch_value()

acc_fetch_value()			
Synopsis:	Get the logic or strength value of a net, reg, or variable.		
Syntax:	`acc_fetch_value(object_handle, format_string, value)`		
Returns:	**Type**	Description	
	PLI_BYTE8 *	Pointer to a character string	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle of the object
	quoted string or PLI_BYTE8 *	format_string	A literal string or character string pointer with one of the following specifiers for formatting the return value: **%b %d %h %o %v %%**
Optional	s_acc_value *	value	Pointer to a structure in which the value of the object is returned when the format string is **%%** (should be set to `null` when not used)
Related routines:	Use acc_fetch_size() to determine how many bits wide the object is Use acc_set_value() to put a logic value on the object		

The ACC routine **acc_fetch_value()** shall return *logic* simulation values for scalar or vector nets, reg, and integer, time and real variables; **acc_fetch_value()** shall return *strength* values for scalar nets and scalar regs only.

The routine **acc_fetch_value()** shall return the logic and strength values in one of two ways:

The value can be returned as a string
The value can be returned as an `aval/bval` pair in a predefined structure.

The return method used by **acc_fetch_value()** shall be controlled by the *format_string* argument, as shown in Table 157.

Table 157—How acc_fetch_value() returns values

format_specifier	Return format	Description
%b	binary	Value shall be retrieved as a string, and a character pointer to the string shall be returned
%d	decimal	
%h	hexadecimal	
%o	octal	
%v	strength	
%%	`s_acc_value` structure	Value shall be retrieved and placed in a structure variable pointed to by the optional *value* argument

The string value returned shall have the same form as output from the formatted built-in system task $display, in terms of value lengths and value characters used. The length shall be of arbitrary size, and unknown and high-impedance values shall be obtained. Note that strings are placed in a temporary buffer, and they should be preserved if not used immediately. Refer to 22.9 for details on preserving strings.

 Copyright © 2001 IEEE. All rights reserved.

The %v format shall return a three character string containing the strength code of a scalar net. Refer to 17.1.1.5 for the strength representations.

When a format_string of **%%** is specified, **acc_fetch_value()** shall retrieve the logic value and strength to a predefined structure, `s_acc_value`, which is defined in `acc_user.h` and is shown below [note that this structure definition is also used with the **acc_set_value()** routine].

.

```
typedef struct t_setval_value
{
  PLI_INT32 format;
  union
    {
      PLI_BYTE8    *str;
      PLI_INT32     scalar;
      PLI_INT32     integer;
      double        real;
      p_acc_vecval  vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 96—s_acc_value structure

To use the **%%** format_string to retrieve values to a structure requires the following steps:

a) A structure variable shall first be declared of type `s_acc_value`.

b) The format field of the structure has to be set to a predefined constant. The format controls which fields in the `s_acc_value` structure shall be used when **acc_fetch_value()** returns the value. The predefined constants for the format shall be one of the constants shown in Table 158.

c) The structure variable has to be passed as the third argument to **acc_fetch_value()**.

d) The function return value from **acc_fetch_value()** should be ignored.

Table 158—Format constants for the s_acc_value structure

Format constant	acc_fetch_value() shall return the value to the s_acc_value union field	Description
accBinStrVal	str	value is retrieved in the same format as %b
accOctStrVal	str	value is retrieved in the same format as %o
accDecStrVal	str	value is retrieved in the same format as %d
accHexStrVal	str	value is retrieved in the same format as %h
accStringVal	str	value is converted to a string, see Section 2.6 for a description of Verilog strings
accScalarVal	scalar	value is retrieved as one of the constants: **acc0**, **acc1**, **accZ** or **accX**
accIntVal	integer	value is retrieved as a C integer
accRealVal	real	value is retrieved as a C double
accVectorVal	vector	value is represented as aval/bval pairs stored in an array of `s_acc_vecval` structures

Copyright © 2001 IEEE. All rights reserved.

For example, calling **acc_fetch_value()** with the following setup would return a string in the `value.str` field. (This is essentially the same as using **acc_fetch_value()** with a **%b** format string.)

```
s_acc_value value;
value.format = accBinStrVal;
(void)acc_fetch_value(Net, "%%", &value);
```

If the format field for **acc_fetch_value()** is set to **accVectorVal**, then the value shall be placed in the record(s) pointed to by the value field. The value field shall be a pointer to an array of one or more `s_acc_vecval` structures. The `s_acc_vecval` structure is defined in the `acc_user.h` file and is listed in Figure 96. The structure shall contain two integers: *aval* and *bval*. Each `s_acc_vecval` record shall represent 32 bits of a vector. The encoding for each bit value is shown in Table 159.

.

```
typedef struct t_acc_vecval
{
  PLI_INT32 aval;
  PLI_INT32 bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 97—s_acc_vecval structure

Table 159—Encoding of bits in the s_acc_vecval structure

aval	bval	Value
0	0	0
1	0	1
0	1	Z
1	1	X

The array of `s_acc_vecval` structures shall contain a record for every 32 bits of the vector, plus a record for any remaining bits. If a vector has N bits, then there shall be `((N-1)/32)+1` `s_acc_vecval` records. The routine **acc_fetch_size()** can be used to determine the value of N. The lsb of the vector shall be represented by the lsb of the first record of `s_acc_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the second record of the array, and so on. See Figure 99 for an example of `acc_fetch_value()` used in this way.

Note that when using aval/bval pairs, the `s_acc_value` record and the appropriately sized `s_acc_vecval` array shall first be declared. Setting the second parameter to `acc_fetch_value()` to **%%** and the third parameter to `null` shall be an error.

 Copyright © 2001 IEEE. All rights reserved.

The example application shown in Figure 98 uses **acc_fetch_value()** to retrieve the logic values of all nets in a module as strings.

```
include "acc_user.h"
LI_INT32 display_net_values()

 handle mod, net;

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

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

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

  acc_close();
```

Figure 98—Using acc_fetch_value() to retrieve the logic values as strings

The example in Figure 99 uses **acc_fetch_value()** to retrieve a value into a structure, and then prints the value. The example assumes the application, `my_fetch_value`, is called from the following user-defined system task:

```
$my_fetch_value(R);
```


Copyright © 2001 IEEE. All rights reserved.

```
.nclude "acc_user.h"

.I_INT32 my_fetch_value()

 handle             reg = acc_handle_tfarg(1);
 PLI_INT32          size = ((acc_fetch_size(reg) - 1) / 32) + 1;
 s_acc_value        value;
 int                index1, min_size;
 static PLI_BYTE8   table[4] = {'0','1','z','x'};
 static PLI_BYTE8   outString[33];

 io_printf("The value of %s is ",acc_fetch_name(reg));

 value.format = accVectorVal;
 value.value.vector = (p_acc_vecval)malloc(size * sizeof(s_acc_vecval));

 (void)acc_fetch_value(reg, "%%",&value);

 for (index1 = size - 1; index1 >= 0; index1--)
 {
    int index2;
    PLI_INT32 abits = value.value.vector[index1].aval;
    PLI_INT32 bbits = value.value.vector[index1].bval;

    if (index1 == size - 1)
    {
       min_size = (acc_fetch_size(reg) % 32);
       if (!min_size)
       min_size = 32;
    }
    else
       min_size = 32;
    outString[min_size] = '\0';
    min_size--;
    outString[min_size] = table[((bbits & 1) << 1) | (abits & 1)];
    abits >>= 1;

    for (index2 = min_size - 1; index2 >= 0; index2--)
    {
       outString[index2] = table[(bbits & 2) | (abits & 1)];
       abits >>= 1;
       bbits >>= 1;
    }
    io_printf("%s", outString);
 }
 io_printf("\n");
 return(0);
```

Figure 99—Using acc_fetch_value() to retrieve values into a data structure

 Copyright © 2001 IEEE. All rights reserved.

23.37 acc_free()

acc_free()			
Synopsis:	Frees memory allocated by **acc_collect()**.		
Syntax:	acc_free(handle_array_pointer)		
Returns:	**Type**	Description	
	void	No return	
Arguments:	**Type**	**Name**	**Description**
	handle *	handle_array_pointer	Pointer to the array of handles allocated by acc_collect()
Related routines:	Use acc_collect() to collect handles returned by acc_next_ routines		

The ACC routine **acc_free()** shall deallocate memory that was allocated by the routine **acc_collect()**.

The example shown in Figure 100 uses **acc_free()** to deallocate memory allocated by **acc_collect()** to collect handles to all nets in a module.

```
include "acc_user.h"

LI_INT32 display_nets()

   handle      *list_of_nets, module_handle;
   PLI_INT32    net_count, i;

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

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

   /*collect and display all nets in the module*/
   list_of_nets = acc_collect(acc_next_net, module_handle, &net_count);
   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 100—Using acc_free()

Copyright © 2001 IEEE. All rights reserved.

23.38 acc_handle_by_name()

acc_handle_by_name()			
Synopsis:	Get the handle to any named object based on its name and scope.		
Syntax:	`acc_handle_by_name(object_name, scope_handle)`		
Returns:	**Type**	Description	
	handle	A handle to the specified object	
Arguments:	**Type**	**Name**	**Description**
	quoted string or PLI_BYTE8 *	object_name	Literal name of an object or a character string pointer to the object name
	handle	scope_handle	Handle to scope, or `null`
Related Routines	Use acc_handle_object() to get a handle based on the local instance name of an object		

The ACC routine **acc_handle_by_name()** shall return the handle to any named object based on its specified name and scope. The routine can be used in two ways, as shown in Table 160.

Table 160—How acc_handle_by_name() works

When the *scope_handle* is	acc_handle_by_name() shall
A valid scope handle	Search for the *object_name* in the scope specified
`null`	Search for the object_name in the module containing the current system task or function

The routine **acc_handle_by_name()** combines the functionality of **acc_set_scope()** and **acc_handle_object()**, making it possible to obtain handles for objects that are not in the local scope without having to first change scopes. Object searching shall conform to rules in 12.4 on hierarchical name referencing.

Table 161 lists the objects in a Verilog HDL description for which **acc_handle_by_name()** shall return a handle.

Table 161—Named objects supported by acc_handle_by_name()

Modules	Parameters
Primitives	Specparams
Nets	Named blocks
Regs	Verilog HDL tasks
Integer, time and real variables	Verilog HDL functions
Named events	

 Copyright © 2001 IEEE. All rights reserved.

The routine **acc_handle_by_name()** does not return handles for module paths, intermodule paths, data paths, or ports. Use an appropriate acc_next_ or other ACC routines for these objects.

The example shown in Figure 101 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"

LI_INT32 is_net_in_module(module_handle, net_name)
andle module_handle;
LI_BYTE8 *net_name;

   handle 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 101—Using acc_handle_by_name()

Note that 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);
```

23.39 acc_handle_calling_mod_m

acc_handle_calling_mod_m			
Synopsis:	Get a handle to the module containing the instance of the user-defined system task or function that called the PLI application.		
Syntax:	`acc_handle_calling_mod_m()`		
	Type	Description	
Returns:	handle	Handle to a module	
	Type	**Name**	**Description**
Arguments:	None		

The ACC routine **acc_handle_calling_mod_m** shall return a handle to the module that contains the instance of the user-defined system task or function that called the PLI application.

23.40 acc_handle_condition()

acc_handle_condition()			
Synopsis:	Get a handle to the conditional expression of a module path, data path, or timing check terminal.		
Syntax:	`acc_handle_condition(path_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a conditional expression	
Arguments:	**Type**	**Name**	**Description**
	handle	path_handle	Handle to a module path, data path, or timing check terminal

The ACC routine **acc_handle_condition()** shall return a handle to a conditional expression for the specified module path, data path, or timing check terminal. The routine shall return `null` when

The module path, data path, or timing check terminal has no condition specified
The module path has an **ifnone** condition specified

To determine if a module path has an **ifnone** condition specified, use the ACC routine **acc_object_of_type()** to check for the property type of **accModPathHasIfnone**.

The example shown in Figure 102 provides functionality to see if a path is conditional, and, if it is, whether it is level-sensitive or edge-sensitive. The application assumes that the input is a valid handle to a module path.

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

int is_level_sensitive(path)
{
   int flag;
   handle path_in = acc_next_input(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 102—Using acc_handle_condition()

 Copyright © 2001 IEEE. All rights reserved.

23.41 acc_handle_conn()

acc_handle_conn()			
Synopsis:	Get the handle to the net connected to a primitive terminal, path terminal, or timing check terminal.		
Syntax:	`acc_handle_conn(terminal_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a net	
Arguments:	**Type**	**Name**	**Description**
	`handle`	terminal_handle	Handle of the primitive terminal, path terminal, or timing check terminal
Related routines:	Use acc_handle_terminal() or acc_next_terminal() to obtain a terminal_handle		

The ACC routine **acc_handle_conn()** shall return a handle to the net connected to a primitive terminal, path terminal, or timing check terminal. This handle can then be passed to other ACC routines to traverse a design hierarchy or to extract information about the design.

The example shown in Figure 103 displays the net connected to the output terminal of a gate.

```
include "acc_user.h"

LI_INT32 display_driven_net()

   handle  gate_handle, terminal_handle, net_handle;

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

  /*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 103—Using acc_handle_conn()

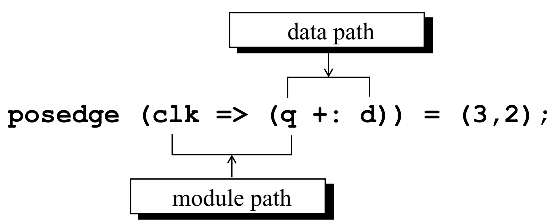
Copyright © 2001 IEEE. All rights reserved.

23.42 acc_handle_datapath()

acc_handle_datapath()			
Synopsis:	Get a handle to a data path for an edge-sensitive module path.		
Syntax:	`acc_handle_datapath(modpath_handle)`		
	Type	Description	
Returns:	handle	Handle of a data path	
	Type	**Name**	**Description**
Arguments:	handle	modpath_handle	Handle to a module path

The ACC routine **acc_next_datapath()** shall return a handle to the data path associated with an edge-sensitive module path. If there is no data path, `null` shall be returned.

A data path is part of the Verilog HDL description for edge-sensitive module paths, as illustrated below:



The example shown in Figure 104 uses **acc_handle_datapath()** to find the data path corresponding to the specified module path and displays the source and destination port names for the data path.

```
PLI_INT32 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 104—Using acc_handle_datapath()

 Copyright © 2001 IEEE. All rights reserved.

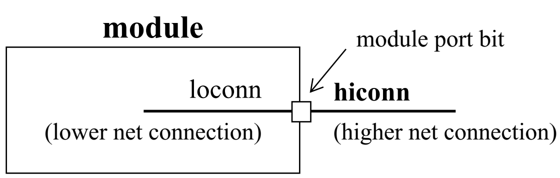
23.43 acc_handle_hiconn()

acc_handle_hiconn()			
Synopsis:	Get the hierarchically higher net connection to a scalar module port or a bit-select of a vector port.		
Syntax:	`acc_handle_hiconn(port_ref_handle)`		
	Type	Description	
Returns:	handle	Handle of a net	
	Type	**Name**	**Description**
Arguments:	handle	port_ref_handle	Handle to a scalar port or a bit-select of a vector port
Related routines:	Use acc_next_hiconn() to find all nets connected to a scalar port or bit-select of a port Use acc_handle_loconn() to get the hierarchically lower net connection of a port		

The ACC routine **acc_handle_hiconn()** shall return the hierarchically higher net connection for a scalar port or a bit-select of one of the following:

Vector port
Part-select of a port
Concatenation of scalar ports, vector ports, part-selects of ports, or other concatenations

The hiconn is the net connected one level above the hierarchical scope of a module port, as illustrated below:



The example shown in Figure 105 uses **acc_handle_hiconn()** and **acc_handle_loconn()** to display the higher and lower connections of a module port.

Copyright © 2001 IEEE. All rights reserved.

```
PLI_INT32 display_port_info(mod, index)
handle    mod;
PLI_INT32 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_bit);
         loconn = acc_handle_loconn (port_bit);
         io_printf ("    hi: %s  lo: %s\n",
            acc_fetch_fullname(hiconn), acc_fetch_fullname(loconn));
      }
   }
}
```

Figure 105—Using acc_handle_hiconn() and acc_handle_loconn()

23.44 acc_handle_interactive_scope()

acc_handle_interactive_scope()			
Synopsis:	Get a handle to the current interactive scope of the software tool.		
Syntax:	`acc_handle_interactive_scope()`		
	Type	Description	
Returns:	handle	Handle of a Verilog hierarchy scope	
	Type	**Name**	**Description**
Arguments:	None		
Related routines:	Use acc_fetch_type() or acc_fetch_fulltype() to determine the scope type returned Use acc_set_interactive_scope() to change the interactive scope		

The ACC routine **acc_handle_interactive_scope()** shall return a handle to the Verilog HDL design scope where the interactive mode of a software product is currently pointing.

A scope shall be

A top-level module
A module instance
A named begin-end block

 Copyright © 2001 IEEE. All rights reserved.

A named fork-join block
A Verilog HDL task
A Verilog HDL function

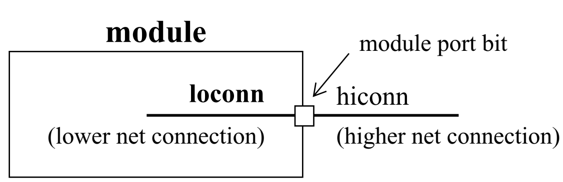
23.45 acc_handle_loconn()

acc_handle_loconn()			
Synopsis:	Gets the hierarchically lower net connection to a scalar module port or a bit-select of a vector port.		
Syntax:	`acc_handle_loconn(port_ref_handle)`		
	Type	Description	
Returns:	handle	Handle of a net	
	Type	**Name**	**Description**
Arguments:	handle	port_ref_handle	Handle to a scalar port or a bit-select of a vector port
Related routines:	Use acc_next_loconn() to find all nets connected to a scalar port or bit-select of a port Use acc_handle_hiconn() to get the hierarchically higher net connection of a port		

The ACC routine **acc_handle_loconn()** shall return the hierarchically lower net connection for a scalar port or a bit-select of one of the following:

Vector port
Part-select of a port
Concatenation of scalar ports, vector ports, part-selects of ports, or other concatenations

The loconn is the net connected within the hierarchical scope of a module port, as illustrated below:



Refer to the usage example in 23.43 for an example of using **acc_handle_loconn()**.

23.46 acc_handle_modpath()

acc_handle_modpath()			
Synopsis:	Gets a handle to a module path.		
Syntax:	`acc_handle_modpath(module_handle, source_name, destination_name, source_handle, destination_handle)`		
	Type	Description	
Returns:	handle	Handle of a module path	
	Type	**Name**	**Description**
Arguments:	handle	module_handle	Handle of the module
	quoted string or PLI_BYTE8 *	source_name	Literal string or character string pointer with the name of a net connected to a module path source
	quoted string or PLI_BYTE8 *	destination_name	Literal string or character string pointer with the name of a net connected to a module path destination
Optional	handle	source_handle	Handle of a net connected to a module path source (used when **accEnableArgs** is set and *source_name* is `null`)
Optional	handle	destination_handle	Handle of a net connected to a module path destination (used when **accEnableArgs** is set and *destination_name* is `null`)
Related routines:	Use acc_configure(accEnableArgs, acc_handle_modpath) to use the source_handle and destination_handle		

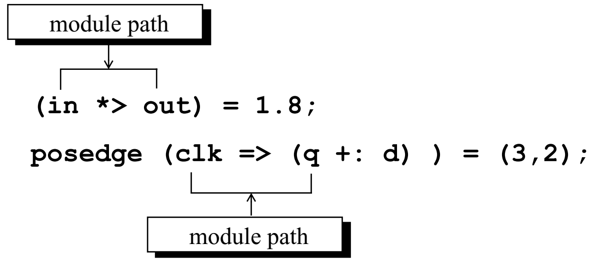
The ACC routine **acc_handle_modpath()** shall return a handle to a module path if one can be found. If a module path cannot be found the return value shall be **null**, the **acc_error_flag** will not be set. If any of the input args are improper a **null** shall be returned and the **acc_error_flag** will be set.

Table 162—How acc_handle_modpath() works

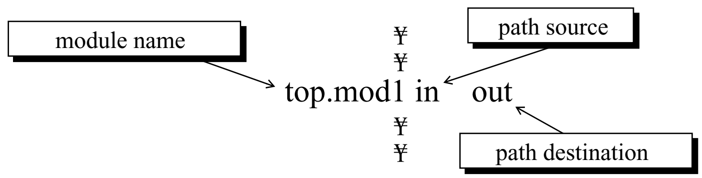
Setting of accEnableArgs	acc_handle_modpath() shall
`"no_acc_handle_modpath"` (the default setting)	Use the name arguments and ignore both handle arguments (the handle arguments can be dropped)
`"acc_handle_modpath"` and either *source_name* ***or*** *destination_name* is *null*	Use the handle argument of the `null` name argument; if the name argument is not `null`, the name shall be used and the associated handle argument ignored

 Copyright © 2001 IEEE. All rights reserved.

A module path is the specify block path for delays in the Verilog HDL description. For example:



The example shown in Figure 106 uses **acc_handle_modpath()** to obtain handles for paths that connect the sources and destinations listed in the file `pathconn.dat`. The format of `pathconn.dat` is shown below.




```
nclude <stdio.h>
nclude "acc_user.h"

efine NAME_SIZE 256

I_INT32 get_paths()

 FILE      *infile;
 PLI_BYTE8 mod_name[ NAME_SIZE] , src_name[ NAME_SIZE] , dest_name[ NAME_SIZE
 handle    path_handle, mod_handle;

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

 /* 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 (path_handle)
       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 106—Using acc_handle_modpath()

Copyright © 2001 IEEE. All rights reserved.

23.47 acc_handle_notifier()

acc_handle_notifier()			
Synopsis:	Get the notifier reg associated with a particular timing check.		
Syntax:	`acc_handle_notifier(tchk)`		
	Type	Description	
Returns:	handle	Handle to a timing check notifier	
	Type	**Name**	**Description**
Arguments:	handle	tchk	Handle of a timing check
Related routines:	Use acc_handle_tchk() to get a handle to a specific timing check Use acc_next_tchk() to get handles to all timing checks in a module		

The ACC routine **acc_handle_notifier()** shall return a handle to the notifier reg associated with a timing check.

The example shown in Figure 117 uses **acc_handle_notifier()** to display the name of a notifier associated with a timing check.

23.48 acc_handle_object()

acc_handle_object			
Synopsis:	Get a handle for any named object.		
Syntax:	`acc_handle_object(object_name)`		
	Type	Description	
Returns:	handle	Handle to an object	
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	object_name	Literal string or character string pointer with the full or relative hierarchical path name of an object
Related routines:	Use acc_set_scope() to set the scope when using relative path names for an object		

The ACC routine **acc_handle_object()** shall return a handle to a named object. The *object_name* argument shall be a quoted string or pointer to a string. The object_name can include a Verilog hierarchy path. The routine shall search for the object using the rules given in Table 163.

 Copyright © 2001 IEEE. All rights reserved.

Table 163—How acc_handle_object() works

If *object_name* contains	acc_handle_object() shall
A full hierarchical path name (a full hierarchical path begins with a top-level module)	Return a handle to the object; no search is performed
No path name or *a relative path name*	Search for object starting in the current PLI scope, following search rules defined in Section 12.6

The ACC routine **acc_handle_object()** shall use the current PLI scope as a basis for searching for objects. The PLI scope shall default to the Verilog scope of the system task/function that called the C application of the user, and it can be changed from within the application using **acc_set_scope()**.

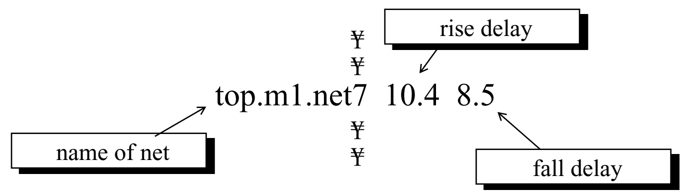
Table 141 lists the objects in a Verilog HDL description for which **acc_handle_object()** shall return a handle.

Table 164—Named objects

Modules	Named events
Module ports	Parameters
Data paths	Specparams
Primitives	Named blocks
Nets	Verilog HDL tasks
Regs	Verilog HDL functions
Integer, time and real variables	

The example shown in Figure 107 uses **acc_handle_object()** to retrieve handles for net names read from a file called `primdelay.dat`. The format of the file is shown below. Note that this example assumes that each net is driven by only one primitive.

Copyright © 2001 IEEE. All rights reserved.




```
include <stdio.h>
include "acc_user.h"
define NAME_SIZE 256

LI_INT32 write_prim_delays()

  FILE        *infile;
  PLI_BYTE8   full_net_name[ NAME_SIZE] ;
  double      rise, fall;
  handle      net_handle, driver_handle, prim_handle;

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

  /*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 107—Using acc_handle_object()

 Copyright © 2001 IEEE. All rights reserved.

23.49 acc_handle_parent()

acc_handle_parent()			
Synopsis:	Get a handle for the parent primitive instance or module instance of an object.		
Syntax:	`acc_handle_parent(object_handle)`		
	Type	Description	
Returns:	handle	Handle of a primitive, port or module	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle of an object

The ACC routine **acc_handle_parent()** shall return a handle to the parent of any object. A parent is an object that contains another object.

- The parent of a *terminal* shall be the *primitive* that contains the terminal.
- The parent of a port bit shall be the port that contains the bit.
- The parent of any other object (except a top-level module) shall be the *module instance* that contains the object.
- Top-level modules do not have parents. When a top-level module handle is passed to **acc_handle_parent()**, it shall return `null`.

The example shown in Figure 108 uses **acc_handle_parent()** to determine which terminals of a primitive drive a net.

```
include "acc_user.h"

LI_INT32 get_primitives(net_handle)
andle   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 108—Using acc_handle_parent()

Copyright © 2001 IEEE. All rights reserved.

23.50 acc_handle_path()

acc_handle_path()			
Synopsis:	Get a handle to an intermodule path that represents the connection from an output or inout port to an input or inout port.		
Syntax:	`acc_handle_path(port_output_handle, port_input_handle)`		
	Type	**Description**	
Returns:	handle	Handle of the intermodule path	
	Type	**Name**	**Description**
Arguments:	handle	port_output_handle	Handle to one of the following: ¥ A scalar output port ¥ A scalar bidirectional port ¥ 1 bit of a vector output port ¥ 1 bit of a vector bidirectional port
	handle	port_input_handle	Handle to one of the following: ¥ A scalar input port ¥ A scalar bidirectional port ¥ 1 bit of a vector input port ¥ 1 bit of a vector bidirectional port
Related routines:	Use acc_next_port() or 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		

The ACC routine **acc_handle_path()** shall return a handle to an *intermodule path*. An intermodule path shall be a net path that connects an output or inout port of one module to an input or inout port of another module.

The example shown in Figure 109 is a C code fragment that uses **acc_handle_path()** to fetch min:typ:max delays for the intermodule path referenced by `intermod_path`.

```
include "acc_user.h"
LI_INT32 fetch_mintypmax_delays(port_output, port_input)
andle   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 net path that represents the connection from an output or inout port to an input (or inout) port

Figure 109—Using acc_handle_path()

 Copyright © 2001 IEEE. All rights reserved.

23.51 acc_handle_pathin()

acc_handle_pathin()			
Synopsis:	Get a handle for the first net connected to a module path source.		
Syntax:	`acc_handle_pathin(path_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a net	
Arguments:	**Type**	**Name**	**Description**
	handle	path_handle	Handle of the module path
Related routines:	Use acc_next_modpath() or acc_handle_modpath() to get path_handle		

The ACC routine **acc_handle_pathin()** shall return a handle to the net connected to the first source in a module path. If a module path has more than one input source, only the handle to the net connected to the first source shall be returned. For example:

pathin

```
(posedge clk => (q +: d) ) = (3,2);

(a,b,c *> d,e,f) = 1.8;
```

pathin is first terminal

The example shown in Figure 110 uses **acc_handle_pathin()** to find the net connected to the input of a path.

```
include "acc_user.h"

LI_INT32 get_path_nets(path_handle)
andle 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 110—Using acc_handle_pathin()

Copyright © 2001 IEEE. All rights reserved.

23.52 acc_handle_pathout()

acc_handle_pathout()			
Synopsis:	Get a handle for the first net connected to a module path destination.		
Syntax:	`acc_handle_pathout(path_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a net	
Arguments:	**Type**	**Name**	**Description**
	handle	path_handle	Handle of the module path
Related routines:	Use acc_next_modpath() or acc_handle_modpath() to get path_handle		

The ACC routine **acc_handle_pathout()** shall return a handle to the net connected to the first destination in a module path. If a module path has more than one output destination, only the handle to the net connected to the first destination shall be returned. For example:

pathout

```
(posedge clk => (q +: d) ) = (3,2);

(a,b,c *> d,e,f) = 1.8;
```

pathout is first terminal

The example shown in Figure 111 uses **acc_handle_pathout()** to find the net connected to the output of a path.

```
include "acc_user.h"

LI_INT32 get_path_nets(path_handle)
andle 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 111—Using acc_handle_pathout()

 Copyright © 2001 IEEE. All rights reserved.

23.53 acc_handle_port()

<table>
<tr><th colspan="4">acc_handle_port()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Get a handle for a module port, based on the position of the port.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>acc_handle_port(module_handle, port_index)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>handle</td><td colspan="2">Handle to a module port</td></tr>
<tr><td rowspan="3">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>module_handle</td><td>Handle of a module</td></tr>
<tr><td>PLI_INT32</td><td>port_index</td><td>An integer index of the desired port</td></tr>
<tr><td>Related routines:</td><td colspan="3">Use acc_next_port() to get handles to all ports of a module</td></tr>
</table>

The ACC routine **acc_handle_port()** shall return a handle to a specific port of a module, based on the position of the port in the module declaration.

The index of a port shall be its position in a module definition in the source description. The indices shall be integers that start at **0** and increase from left to right. Table 165 shows how port indices are derived.

Table 165—Deriving port indices

<table>
<tr><th>For</th><th>Indices shall be</th></tr>
<tr><td>Implicit ports:
module A(q, a, b);</td><td>0 for port q
1 for port a
2 for port b</td></tr>
<tr><td>Explicit ports:
module top;
reg ra, rb;
wire wq;
explicit_port_mod epm1(.b(rb), .a(ra), .q(wq));
endmodule

module explicit_port_mod(q, a, b);
input a, b;
output q;
nand (q, a, b);
endmodule</td><td>0 for explicit port epm1.q
1 for explicit port epm1.a
2 for explicit port epm1.b</td></tr>
</table>

The example shown in Figure 112 uses **acc_handle_port()** to identify whether a particular module port is an output.

Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"

nt is_port_output(module_handle,port_index)
andle        module_handle;
LI_INT32     port_index;

  handle       port_handle;
  PLI_INT32    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 112—Using acc_handle_port()

23.54 acc_handle_scope()

acc_handle_scope()			
Synopsis:	Get a handle to the scope that contains an object.		
Syntax:	`acc_handle_scope(object_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a scope	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle to an object
Related routines:	Use acc_fetch_type() or acc_fetch_fulltype() to determine the scope type returned		

The ACC routine **acc_handle_scope()** shall return the handle to the scope of an object. A scope shall be

A top-level module
A module instance
A named begin-end block
A named fork-join block
A Verilog HDL task
A Verilog HDL function

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 113 uses **acc_handle_scope()** to display the scope that contains an object.

```
PLI_INT32 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);
}
```

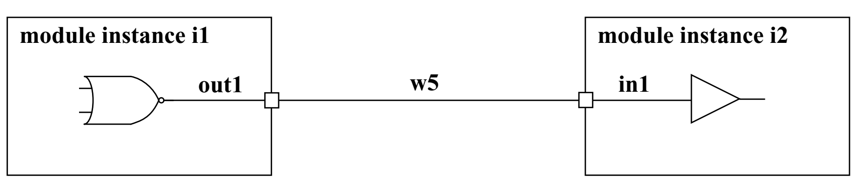
Figure 113—Using acc_handle_scope()

23.55 acc_handle_simulated_net()

acc_handle_simulated_net()			
Synopsis:	Get the simulated net associated with the collapsed net passed as an argument.		
Syntax:	`acc_handle_simulated_net(collapsed_net_handle)`		
Returns:	**Type**	Description	
	handle	Handle of the simulated net	
Arguments:	**Type**	**Name**	**Description**
	handle	collapsed_net_handle	Handle of a collapsed net
Related routines:	Use acc_object_of_type() to determine if a net has been collapsed		

The ACC routine **acc_handle_simulated_net()** shall return a handle to the simulated net that is associated with a specified collapsed net. If a handle to a net that is not collapsed is passed into the routine, a handle to that same net shall be returned.

When a Verilog HDL source description connects modules together, a chain of nets with different scopes and names are connected, as is illustrated in the following simple diagram:



In this small circuit, nets `out1`, `w5`, and `in1` are all tied together, effectively becoming the same net. Software products can collapse nets that are connected together within the data structure of the product. The resultant net after collapsing is referred to as the *simulated net*; the other nets are referred to as collapsed nets. The ACC routines can obtain a handle to any net, whether it is collapsed or not. The routine

Copyright © 2001 IEEE. All rights reserved.

acc_object_of_type() can be used to determine if a net has been collapsed, and the routine **acc_handle_simulated_net()** can be used to find the resultant net from the net collapsing process.

The example shown in Figure 114 uses **acc_handle_simulated_net()** to find all simulated nets within a particular scope. The application then displays each collapsed net, along with the simulated net. The ACC routine **acc_object_of_type()** is used with the property **accCollapsedNet** to determine whether a net has been collapsed onto another net.

```
include "acc_user.h"

LI_INT32 display_simulated_nets()

  handle   mod_handle;
  handle   simulated_net_handle;
  handle   net_handle;

  /*reset environment for ACC routines*/
  acc_initialize();

  /*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 114—Using acc_handle_simulated_net()

 Copyright © 2001 IEEE. All rights reserved.

23.56 acc_handle_tchk()

<table>
<tr><th colspan="4">acc_handle_tchk()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Get a handle for the specified timing check of a module (or cell).</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>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)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>handle</td><td colspan="2">Handle to a timing check</td></tr>
<tr><td rowspan="6">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>module_handle</td><td>Handle of the module</td></tr>
<tr><td>integer constant</td><td>timing_check_type</td><td>One of the following predefined constants:
accHold accSetup
accNochange accSkew
accPeriod accWidth
accRecovery</td></tr>
<tr><td>quoted string or PLI_BYTE8 *</td><td>first_arg_conn_name</td><td>Name of the net connected to first timing check argument</td></tr>
<tr><td>integer constant</td><td>first_arg_edge_type</td><td>Edge of the net connected to first timing check argument
One of the following predefined constants:
accNegedge accNoedge accPosedge
or a list of the following constants, separated by +:
accEdge01 accEdge0x accEdgex1
or a list of the following constants, separated by +:
accEdge10 accEdge1x accEdgex0</td></tr>
<tr><td colspan="3"></td></tr>
<tr><td>Conditional</td><td>quoted string or PLI_BYTE8 *</td><td>second_arg_conn_name</td><td>Name of the net connected to second timing check argument (depends on type of timing check)</td></tr>
<tr><td>Conditional</td><td>integer constant</td><td>second_arg_edge_type</td><td>Edge of the net connected to second timing check argument (depends on type of timing check)
Uses same constants as first_arg_edge_type</td></tr>
<tr><td>Optional</td><td>handle</td><td>first_arg_conn_handle</td><td>Handle of the net connected to first timing check argument (required if accEnableArgs is set and first_arg_conn_name is null)</td></tr>
<tr><td>Optional</td><td>handle</td><td>second_arg_conn_handle</td><td>Handle of the net connected to second timing check argument (required if accEnableArgs is set and second_arg_conn_name is null)</td></tr>
<tr><td>Related routines:</td><td colspan="3">Use acc_configure(accEnableArgs, acc_handle_tchk) to enable the optional first_arg_conn_handle and second_arg_conn_handle arguments</td></tr>
</table>

The ACC routine **acc_handle_tchk()** shall return a handle to a timing check based on arguments that describe the type of timing check, signals used, and edge qualifiers for the signals. The signals used to describe the timing check shall be passed as either signal names (passed as either a quoted string or a character string pointer) or signal handles. The number of signal arguments required by **acc_handle_tchk()** shall depend on the type of timing check.

Copyright © 2001 IEEE. All rights reserved.

Table 166 shows how the number of arguments for **acc_handle_tchk()** is determined.

Table 166—How acc_handle_tchk() works

If	acc_handle_tchk() shall
tchk_type is **accWidth** or **accPeriod**	ignore arguments: *second_arg_conn_name*, *second_arg_edge_type*, and optional *second_arg_conn_handle*
tchk_type is **accHold**, **accNochange**, **accRecovery**, **accSetup**, or **accSkew**	use arguments: *second_arg_conn_name*, *second_arg_edge_type*, and optional *second_arg_conn_handle*
Default mode, or **acc_configure(accEnableArgs, no_acc_handle_tchk)** has been called	Use the name arguments and ignore both optional handle arguments
The routine **acc_configure(accEnableArgs, acc_handle_tchk)** has been called, and either *first_arg_conn_name* or *second_arg_conn_name* is `null`	Use the associated handle argument of the `null` name argument if the name argument is not `null`, the name shall be used and the associated handle argument ignored

NOTE Unused arguments can be dropped if they do not precede any required arguments; otherwise, the unused arguments should be specified as `null`.

The routine **acc_handle_tchk()** shall use predefined edge group constants to represent groups of transitions among **0**, **1**, and **X** edge values, as described in Table 167. The routine shall treat transitions to or from a logic Z as transitions to or from a logic X.

Table 167—Edge group constants

Edge group constant	Description of edge trigger
accPosedge **accPosEdge**	Any positive transition**:** **0** to **1** **0** to **x** **x** to **1**
accNegedge **accNegEdge**	Any negative transition: **1** to **0** **1** to **x** **x** to **0**
accNoedge **accNoEdge**	Any transition: **0** to **1** **1** to **0** **0** to **x** **x** to **1** **1** to **x** **x** to **0**

The routine **acc_handle_tchk()** shall recognize predefined edge-specific constants that represent individual transitions among **0**, **1**, and **X** edge values that trigger timing checks, as described in Table 168.

 Copyright © 2001 IEEE. All rights reserved.

Table 168—Edge specific constants

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**

The Verilog HDL allows multiple edges to be specified for timing checks. The routine **acc_handle_tchk()** shall recognize multiple edges using *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 115 shows a call to **acc_handle_tchk()** that accesses a `$width` timing check containing edge-control specifiers.

This ACC routine call	Accesses this timing check
`cc_handle_tchk(cell_handle, accWidth, "clk", accEdge10+accEdgex0);`	`$width(edge[10,x0]clk, limit);`

edge sum
models
edge-control specifier

Figure 115—Edge sums model edge-control specifiers

The example shown in Figure 116 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**

Note that in this example:

a) Both calls to **acc_handle_tchk()** supply *names* for all relevant connections; therefore, the optional handle arguments are not supplied.

b) For `$period` timing checks, **acc_handle_tchk()** ignores the *second_arg_conn_name* and *second_arg_edge_type* arguments; therefore, these arguments are not supplied.

Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"

LI_INT32 get_ps_tchks()

  handle   module_handle, port_handle, net_handle, cell_handle;

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

  /*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 116—Using acc_handle_tchk()

23.57 acc_handle_tchkarg1()

acc_handle_tchkarg1()			
Synopsis:	Get a handle for the timing check terminal connected to the first argument of a timing check.		
Syntax:	`acc_handle_tchkarg1(tchk_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a timing check terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	tchk_handle	Handle of a timing check
Related routines:	Use acc_handle_conn() to get the net connected to a timing check terminal		

The ACC routine **acc_handle_tchkarg1()** shall return a handle to the timing check terminal associated with the first argument of a timing check.

 Copyright © 2001 IEEE. All rights reserved.

In order to trace a timing check terminal in the Verilog HDL description, or to display the name of the terminal, it is first necessary to obtain a handle to the net connected to the terminal. The routine **acc_handle_conn()** with the timing check terminal handle as the argument can be used to obtain the net handle.

The example shown in Figure 117 uses **acc_handle_tchkarg1()** and **acc_handle_tchkarg2()** to obtain the nets connected to the first and second arguments of each setup timing check in each cell under a module.

```
nclude "acc_user.h"

I_INT32 show_check_nets()
{
handle    module_handle, cell_handle;
handle    tchk_handle, tchkarg1_handle, tchkarg2_handle, notifier_handle
PLI_INT32 tchk_type, counter;

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

/* 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);
      io_printf("  data net is %s\n",
                acc_fetch_name(acc_handle_conn(tchkarg1_handle) );
     tchkarg2_handle = acc_handle_tchkarg2(tchk_handle);
      io_printf("  reference net is %s\n",
                acc_fetch_name(acc_handle_conn(tchkarg2_handle) );
     notifier_handle = acc_handle_notifier(tchk_handle);
      if (notifier_handle != null)
        io_printf("  notifier reg is %s\n",
                  acc_fetch_name(acc_handle_conn(notifier_handle) ) );
      else
        io_printf("  no notifier reg\n");
    }
  }
}
acc_close();
```

Figure 117—Using acc_handle_tchkarg1(), acc_handle_tchkarg2() and acc_handle_notifier()

Copyright © 2001 IEEE. All rights reserved.

23.58 acc_handle_tchkarg2()

acc_handle_tchkarg2()			
Synopsis:	Get a handle for the timing check terminal connected to the second argument of a timing check.		
Syntax:	`acc_handle_tchkarg2(tchk_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a timing check terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	tchk_handle	Handle of a timing check
Related routines:	Use acc_handle_conn() to get the net connected to a timing check terminal		

The ACC routine **acc_handle_tchkarg2()** shall return a handle to the timing check terminal associated with the second argument of a timing check.

In order to trace a timing check terminal in the Verilog HDL description, or to display the name of the terminal, it is first necessary to obtain a handle to the net connected to the terminal. The routine **acc_handle_conn()** with the timing check terminal handle as the argument can be used to obtain the net handle.

Refer to Figure 117 for an example of using **acc_handle_tchkarg2()**.

23.59 acc_handle_terminal()

acc_handle_terminal()			
Synopsis:	Get a handle for a primitive terminal based on the position of the primitive terminal.		
Syntax:	`acc_handle_terminal(primitive_handle, terminal_index)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	primitive_handle	Handle of a primitive
	PLI_INT32	terminal_index	Integer index of the desired terminal
Related routines	Use acc_handle_conn() to get the net connected to a primitive terminal		

The ACC routine **acc_handle_terminal()** shall return a handle of a primitive terminal based on the position of the terminal in the Verilog HDL source description.

 Copyright © 2001 IEEE. All rights reserved.

The index of a terminal shall be its position in a gate, switch, or UDP declaration. The indices shall be integers that start at zero and increase from left to right. Table 169 shows how terminal indices are derived.

Table 169—Deriving terminal indices

For	Indices shall be
`nand g1(out, in1, in2);`	**0** for terminal `out` **1** for terminal `in1` **2** for terminal `in2`

The example shown in Figure 118 uses **acc_handle_terminal()** to identify the name of a net connected to a primitive terminal.

```
include "acc_user.h"

'LI_INT32 print_terminal_net(gate_handle, term_index)
.andle      gate_handle;
'LI_INT32   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 118—Using acc_handle_terminal()

23.60 acc_handle_tfarg(), acc_handle_itfarg()

acc_handle_tfarg(), acc_handle_itfarg()			
Synopsis:	Get a handle for the specified argument of a user-defined system task or function.		
Syntax:	`acc_handle_tfarg(argument_number)` `acc_handle_itfarg(argument_number, instance_handle)`		
Returns:	**Type**	Description	
	handle	Handle to an object	
Arguments:	**Type**	**Name**	**Description**
	PLI_INT32	argument_number	Integer number that references an argument in the system task or function call by its position in the argument list
	handle	instance_handle	Handle to an instance of a system task/function
Related routines:	Use acc_fetch_tfarg() and related routines to get the value of a system task/function argument		

Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_handle_tfarg()** shall return a handle to an argument in the current instance of a user-defined system task/function. The ACC routine **acc_handle_itfarg()** shall return a handle to an argument in a specific instance of a user-defined system task/function.

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

`The system task/function argument can be:`

A module instance
A primitive instance
A net, reg, integer variable, time variable, or real variable
A legal bit select of a net, reg, integer variable or time variable

Table 170—How acc_handle_tfarg() operates

When	acc_handle_tfarg() shall
The system task or function argument is an unquoted Verilog HDL identifier	Return a handle to the object
The system task or function argument is a quoted string name of any object	Function similar to **acc_handle_object()** by searching for an object matching the string and, if found, returning a handle to the object. The object shall be searched for in the following order: a) The current PLI scope [as set by **acc_set_scope()**] b) The scope of the system task/function

The example shown in Figure 119 uses **acc_handle_tfarg()** in a C language application that has the following characteristics:

a) It changes the rise and fall delays of a gate.
b) It 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.
c) It associates through the PLI interface mechanism with a Verilog HDL system task called `$timing_task`.

To invoke the application, the system task `$timing_task` is called from the Verilog HDL source description, as in the following sample call:

```
$timing_task(top.g12, 8.4, 9.2);
```

When Verilog encounters this call, it executes `new_timing`. 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 **acc_fetch_tfarg()**.

 Copyright © 2001 IEEE. All rights reserved.

```
#include "acc_user.h"

PLI_INT32 new_timing()
{
   handle   gate_handle;
   double   new_rise, new_fall;

   /*initialize and configure ACC routines*/
   acc_initialize();
   acc_configure(accToHiZDelay, "max");

   /*get handle to gate*/
   gate_handle = acc_handle_tfarg( 1 );          top.g12

   /* get new delay values */                    8.4
   new_rise = acc_fetch_tfarg( 2 );
   new_fall = acc_fetch_tfarg( 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 119—Using acc_handle_tfarg()

23.61 acc_handle_tfinst()

acc_handle_tfinst()			
Synopsis:	Get a handle to the current user-defined system task or function call.		
Syntax:	`acc_handle_tfinst()`		
Returns:	**Type**	Description	
	handle	Handle of a user-defined system task or function	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_fetch_type() or acc_fetch_fulltype() to determine the type of the handle returned		

The ACC routine **acc_handle_tfinst()** is used to obtain a handle of the user-defined system task/function call that invoked the current PLI application.

Copyright © 2001 IEEE. All rights reserved.

23.62 acc_initialize()

acc_initialize()			
Synopsis:	Initializes the environment for ACC routines.		
Syntax:	`acc_initialize()`		
Returns:	**Type**	Description	
	PLI_INT32	1 if successful; 0 if an error is encountered	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_configure() to set configuration parameter after calling acc_initialize() Use acc_close() at the end of a routine that called acc_initialize()		

The ACC routine **acc_initialize()** shall perform the following functions:

Initialize all configuration parameters to their default values
Allocate memory for string handling and other internal uses

The routine **acc_initialize()** should be called in a C language application before invoking any other ACC routines. 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.

The example shown in Figure 120 uses **acc_initialize()** to initialize the environment for ACC routines.

```
#include "acc_user.h"
PLI_INT32 append_mintypmax_delays()

   handle   prim;
   double   delay_array[ 9] ;
   int      i;

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

   /* configure ACC routine environment */
   acc_configure(accMinTypMaxDelays, "true");

   /* append delays for primitive as specified in task/function args */
   prim = acc_handle_tfarg(1);
   for (i = 0; i < 9; i++)
      delay_array[ i]  = acc_fetch_tfarg(i+2);
   acc_append_delays(prim, delay_array);

   /* close the environment for ACC routines */
   acc_close();
```

Figure 120—Using acc_initialize()

Copyright © 2001 IEEE. All rights reserved.

23.63 acc_next()

acc_next()			
Synopsis:	Get handles to objects of each type specified in an array within the reference scope.		
Syntax:	`acc_next(object_type_array, reference_handle, object_handle)`		
	Type	Description	
Returns:	handle	Handle of the object found	
	Type	**Name**	**Description**
Arguments:	static PLI_INT32 array	object_type_array	Static integer array containing one or more predefined integer constants that represent the types of objects desired; the last element has to be **0**
	handle	reference_handle	Handle of a scope
	handle	object_handle	Handle of the previous object found; initially `null`

The ACC routine **acc_next()** shall scan for and return handles to one or more types of objects within a scope. This routine performs a more general function than the object-specific *acc_next_* routines, such as **acc_next_net()** and **acc_next_primitive()**, which scan only one type of object within a scope.

The objects for which **acc_next()** is to scan shall be listed as an array of object *types or fulltypes* in a static integer array. The array shall contain any number and combination of the predefined integer constants listed in Table 171. The array list shall be terminated by a **0**. The routine **acc_next()** can return objects in an arbitrary order.

The following C language statement is an example of declaring an array of object types called `net_reg_list`:

```
static PLI_INT32 net_reg_list[3] = {accNet,accRegister,0};
```

When this array is passed to **acc_next()**, the ACC routine shall return handles to nets and regs within the reference object.

Note that a Verilog HDL function contains an object with the same name, size, and type as the function. If the function is scanned for objects of the type of the function, a handle to this object shall be returned.

The objects for which **acc_next()** shall obtain handles are listed in Table 171.

Copyright © 2001 IEEE. All rights reserved.

Table 171—Type and fulltype constants supported by acc_next()

Description		Predefined integer constant
General object *types*	Integer variable	**accIntegerVar**
	Module	**accModule**
	Named event	**accNamedEvent**
	Net	**accNet**
	Primitive	**accPrimitive**
	Real variable	**accRealVar**
	Reg	**accRegister**
	Time variable	**accTimeVar**
	Parameter	**accParameter**
Module *fulltypes*	Top-level module	**accTopModule**
	Module instance	**accModuleInstance**
	Cell module instance	**accCellInstance**
Net *fulltypes*	Wire nets	**accWire** **accTri**
	Wired-AND nets	**accWand** **accTriand**
	Wired-OR nets	**accWor** **accTrior**
	Pulldown, pullup nets	**accTri0** **accTri1**
	Supply nets	**accSupply0** **accSupply1**
	Storage nets	**accTrireg**
Parameter *fulltypes*	Integer parameters	**accIntegerParam**
	Real parameters	**accRealParam**
	String parameters	**accStringParam**

Copyright © 2001 IEEE. All rights reserved.

Table 171—Type and fulltype constants supported by acc_next() (continued)

Description		Predefined integer constant
Primitive *fulltypes*	N-input, 1-output gates	**accAndGate** **accNandGate** **accNorGate** **accOrGate** **accXnorGate** **accXorGate**
	1-input, N-output gates	**accBufGate** **accNotGate**
	Tri-state gates	**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 UDP	**accCombPrim**
	Sequential UDP	**accSeqPrim**

The example shown in Figure 121 uses **acc_next()** to find all nets and regs in a module. The application then displays the names of these nets and reg.

Copyright © 2001 IEEE. All rights reserved.

```
#include "acc_user.h"

PLI_INT32 display_nets_and_registers()

   static PLI_INT32  net_reg_list[3] = {accNet,accRegister,0};
   handle            mod_handle, obj_handle;

   /*reset environment for ACC routines*/
   acc_initialize();

   /*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) );

   acc_close();
```

Figure 121—Using acc_next()

23.64 acc_next_bit()

acc_next_bit()			
Synopsis:	Get handles to bits in a port or expanded vector.		
Syntax:	`acc_next_bit(reference_handle, bit_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a port bit, vector bit or path terminal bit	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a port, expanded vector or path terminal
	handle	bit_handle	Handle of the previous bit found; initially `null`
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 Use acc_object_of_type() to determine if a vector is expanded		

The ACC routine **acc_next_bit()** shall obtain handles to the bits of a vector port, an expanded vector, or a path terminal.

An *expanded vector* is a vector for which a software product shall permit access to the discrete bits of the vector. The routine **acc_object_of_type()** can be used to determine if a vector reference handle is expanded before calling **acc_next_bit()** with the vector handle. For example:

 Copyright © 2001 IEEE. All rights reserved.

```
if (acc_object_of_type(vector_handle, accExpandedVector) )
  while (bit_handle = acc_next_bit(vector_handle, bit_handle) )
  ...
```

When the *reference_handle* object is a vector, the first call to **acc_next_bit()** shall return the handle to the msb (leftmost bit) of the object. Subsequent calls shall return the handles to the remaining bits down to the lsb (rightmost bit). The call after the return of the handle to the lsb returns `null`. When the *reference_handle* is scalar, **acc_next_bit()** shall treat the object as a 1-bit vector.

The example shown in Figure 122 uses **acc_next_bit()** to display the lower connection of each bit of a port.

```
include "acc_user.h"
LI_INT32 display_port_bits(module_handle, port_number)
andle      module_handle;
LI_INT32   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 lower 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 122—Using acc_next_bit() with module ports

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 123 uses **acc_next_bit()** to assign a VCL monitor flag to each bit of a vector net.

```
nclude "acc_user.h"
I_INT32 monitor_bits()

  handle   bit_handle, net_handle, mod_handle;

  /* reset environment for ACC routines */
  acc_initialize();

  /* get handle for system task argument associated with this routine */
  mod_handle = acc_handle_tfarg(1);

  /* get handles to all nets in the module */
  net_handle = null;
  while (net_handle = acc_next_net(mod_handle, net_handle) )
  {
     /* add VCL 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, net_consumer, null, vcl_verilog_logic)
     }
  }
```

Figure 123—Using acc_next_bit() with a vector net

23.65 acc_next_cell()

acc_next_cell()			
Synopsis:	Get handles to cell instances within a region that includes the entire hierarchy below a module.		
Syntax:	`acc_next_cell(reference_handle, cell_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a cell module	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	cell_handle	Handle of the previous cell found; initially `null`

The ACC routine **acc_next_cell()** shall return handles to the cell module instances in the reference scope and all module instance scopes below the reference scope. The routine shall not find cells that are instantiated inside other cells.

 Copyright © 2001 IEEE. All rights reserved.

A cell instance shall be a module instance that has either of these characteristics:

The module definition appears between the compiler directives ‘ `celldefine` and ‘`endcelldefine`.

The module definition is in a model library, where a library is a collection of module definitions in a file or directory that are read by library invocation options.

The example shown in Figure 124 uses **acc_next_cell()** to list all cell instances at or below a given hierarchy scope.

```
#include "acc_user.h"
PLI_INT32 list_cells()

   handle   module_handle, cell_handle;

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

   /*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 124—Using acc_next_cell()

23.66 acc_next_cell_load()

acc_next_cell_load()			
Synopsis:	Get handles for cell loads on a net.		
Syntax:	`acc_next_cell_load(reference_handle, load_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive input terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a scalar net or bit select of a vector net
	handle	load_handle	Handle of the previous load found; initially `null`
Related routines:	Use acc_next_load() to get a handle to all primitive input terminal loads		

The ACC routine **acc_next_cell_load()** shall return handles to the *cell module instances* that are driven by a net. The handle for a cell load shall be a primitive input terminal connected to an input or inout port of the cell load instance.

The routines **acc_next_load()** and **acc_next_cell_load()** have different functionalities. The routine **acc_next_load()** shall return every primitive input terminal driven by a net, whether it is inside a cell or a module instance. The routine **acc_next_cell_load()** shall return only one primitive input terminal per cell input or inout port driven by a net. Figure 125 illustrates the difference, using a circuit in which `net1` drives primitive gates in `cell1`, `cell2`, and `module1`. For this circuit, **acc_next_load()** returns four primitive input terminals as loads on `net1`, while **acc_next_cell_load()** returns two primitive input terminals as loads on `net1`.

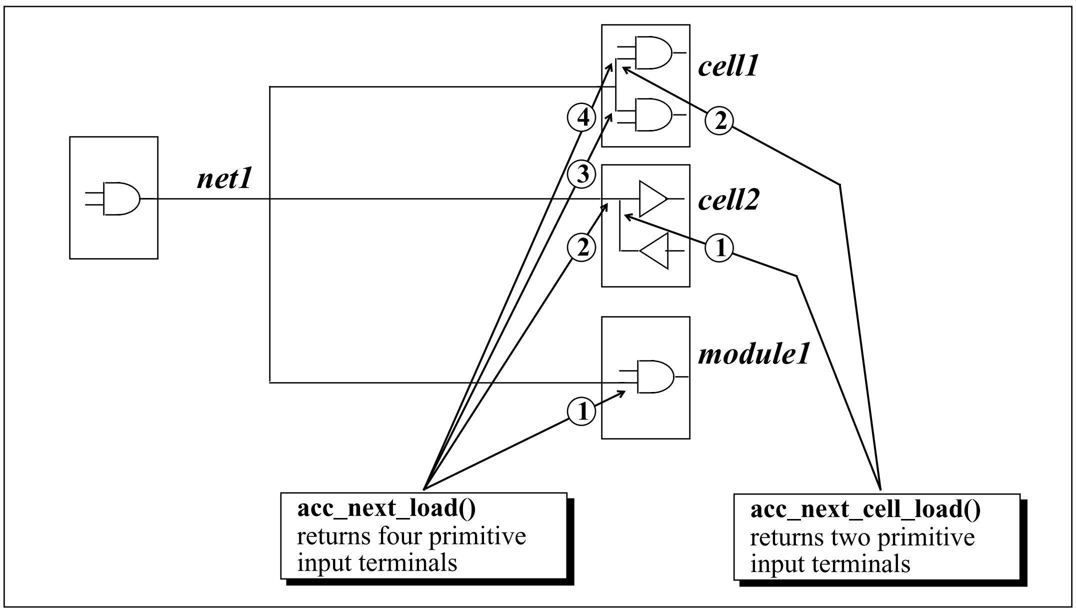


Figure 125—The difference between acc_next_load() and acc_next_cell_load()

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 126 uses **acc_next_cell_load()** to find all cell loads on a net.

```
#include "acc_user.h"

PLI_INT32 get_cell_loads()

  handle   net_handle;
  handle   load_handle, load_net_handle;

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

  /*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 126—Using acc_next_cell_load()

23.67 acc_next_child()

acc_next_child()			
Synopsis:	Get handles for children of a module.		
Syntax:	`acc_next_child(reference_handle, child_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a module instance	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	child_handle	Handle of the previous child found; initially `null`

The ACC routine **acc_next_child()** shall return handles to the module instances (children) within the reference module. The routine shall also return handles to top-level modules, as shown in Table 172.

Copyright © 2001 IEEE. All rights reserved.

Table 172—How acc_next_child() works

When	acc_next_child() shall
The *reference_handle* is not `null`	Scan for modules instantiated inside the module associated with *reference_handle*
The *reference_handle* is `null`	Scan for top-level modules (same as **acc_next_topmod()**)

The ACC routine **acc_next_topmod()** does not work with **acc_collect()** or **acc_count()**, but **acc_next_child()** with a `null` reference handle argument can be used in place of **acc_next_topmod()**. For example:

```
   acc_count(acc_next_child, null); /* counts top-level modules */

   acc_collect(acc_next_child, null, &count); /* collect top-level
modules */
```

Figure 127 shows the use of **acc_next_child()** to display the names of all modules instantiated within a module.

```
#include "acc_user.h"
PLI_INT32 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) );
   child_handle = null;
   while(child_handle = acc_next_child(module_handle, child_handle) )
      io_printf("    %s\n",acc_fetch_name(child_handle) );
```

Figure 127—Using acc_next_child()

23.68 acc_next_driver()

acc_next_driver()			
Synopsis:	Get handles to primitive terminals that drive a net.		
Syntax:	`acc_next_driver(reference_handle, driver_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a scalar net or bit select of a vector net
	handle	driver_handle	Handle of the previous driver found; initially `null`

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_next_driver()** shall return handles to the primitive output or inout terminals that drive a net.

The example shown in Figure 128 uses **acc_next_driver()** to determine which terminals of a primitive drive a net.

```
include "acc_user.h"

LI_INT32 print_drivers(net_handle)
andle 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) );
  }
```

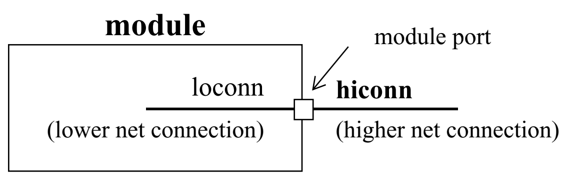
Figure 128—Using acc_next_driver()

23.69 acc_next_hiconn()

acc_next_hiconn()			
Synopsis:	Get handles for hierarchically higher net connections to a module port.		
Syntax:	`acc_next_hiconn(reference_handle, net_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a net	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a port
	handle	net_handle	Handle of the previous net found; initially `null`
Related routines:	Use acc_handle_hiconn() to get a handle to hierarchically higher connection of a specific port bit Use acc_next_loconn() to get handles to the hierarchically lower connection		

Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_next_hiconn()** shall return handles to the hierarchically higher net connections to a module port. A hierarchically higher connection shall be the part of the net that appears outside the module, as shown in the following diagram:



When the reference handle passed to **acc_next_hiconn()** is a vector port, the routine shall return the hiconn nets bit-by-bit, starting with the msb (leftmost bit) and ending with the lsb (rightmost bit).

The example shown in Figure 129 uses **acc_next_hiconn()** and **acc_next_loconn()** to find and display all net connections made externally (hiconn) and internally (loconn) to a module port.

```
include "acc_user.h"

LI_INT32 display_connections(module_handle, port_handle)
andle module_handle, port_handle;

  handle   hiconn_net, loconn_net;

  /*get and display low connections*/
  io_printf("For module %s, port #%d internal 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 external 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 129—Using acc_next_hiconn() and acc_next_loconn()

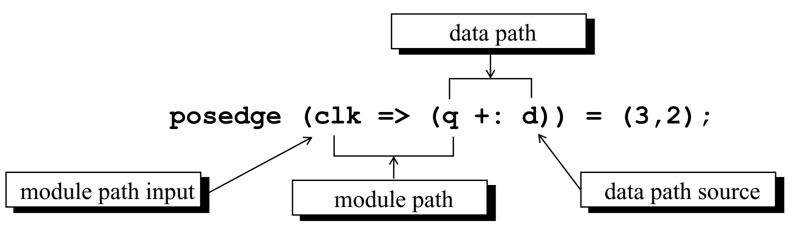
 Copyright © 2001 IEEE. All rights reserved.

23.70 acc_next_input()

acc_next_input()			
Synopsis:	Get handles to input path terminals of a module path, source terminals of a data path, or the terminals of a timing check.		
Syntax:	`acc_next_input (reference_handle, terminal_handle)`		
	Type	Description	
Returns:	handle	Handle of a module path terminal, a data path terminal, or a timing check terminal	
	Type	**Name**	**Description**
Arguments:	handle	reference_handle	Handle to a module path, data path or timing check
	handle	terminal_handle	Handle of the previous terminal found; initially `null`
Related routines:	Use acc_handle_conn() to get the net attached to the path terminal Use acc_release_object() to free memory allocated by acc_next_input()		

The ACC routine **acc_next_input()** shall return handles to the input path terminals of a module path, the source terminals of a data path or the timing check terminals of a timing check. The routine **acc_handle_conn()** can be passed the input path terminal handle to derive the net connected to the terminal.

A *module path* is the specify block path for delays in the Verilog HDL description. A *data path* is part of the Verilog HDL description for edge-sensitive module paths, as shown in the following diagram:



Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 130 uses **acc_next_input()**. It accepts a handle to a scalar net or a net bit-select, and a module path. The application returns `true` if the net is connected to the input of the path.

```
int is_net_on_path_input(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
   handle pterm_in, pterm_conn, bit;

   /* scan path input terminals */
   pterm_in = null;
   while (pterm_in = acc_next_input(path, pterm_in) )
   {
      /* retrieve net connected to path terminal */
      pterm_conn = acc_handle_conn (pterm_in);

      bit = null;
      if (acc_object_of_type (pterm_conn, accExpandedVector) )
      {
         bit = null;
         while (bit = acc_next_bit (pterm_conn, bit) )
            if (acc_compare_handles (bit, net) )
               return (true);
      }
      else
         if (acc_compare_handles(bit, net) )
            return (true);
   }

   return (false);
}
```

Figure 130—Using acc_next_input()

23.71 acc_next_load()

acc_next_load()			
Synopsis:	Get handles to primitive terminals driven by a net.		
Syntax:	`acc_next_load(reference_handle, load_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a scalar net or bit select of a vector net
	handle	load_handle	Handle of the previous load found; initially `null`
Related routines:	Use acc_next_cell_load() to get cell module loads		

 Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_next_load()** shall return handles to the primitive loads that are being driven by a net. The handle for a load shall be a primitive input terminal.

The routines **acc_next_load()** and **acc_next_cell_load()** have different functionalities. The routine **acc_next_load()** shall return every primitive input terminal driven by a net, whether it is inside a cell or a module instance. The routine **acc_next_cell_load()** shall return only one primitive input terminal per cell input or inout port driven by a net. Figure 131 illustrates the difference, using a circuit in which `net1` drives primitive gates in `cell1`, `cell2`, and `module1`. For this circuit, **acc_next_load()** returns four primitive input terminals as loads on `net1`, while **acc_next_cell_load()** returns two primitive input terminals as loads on `net1`.

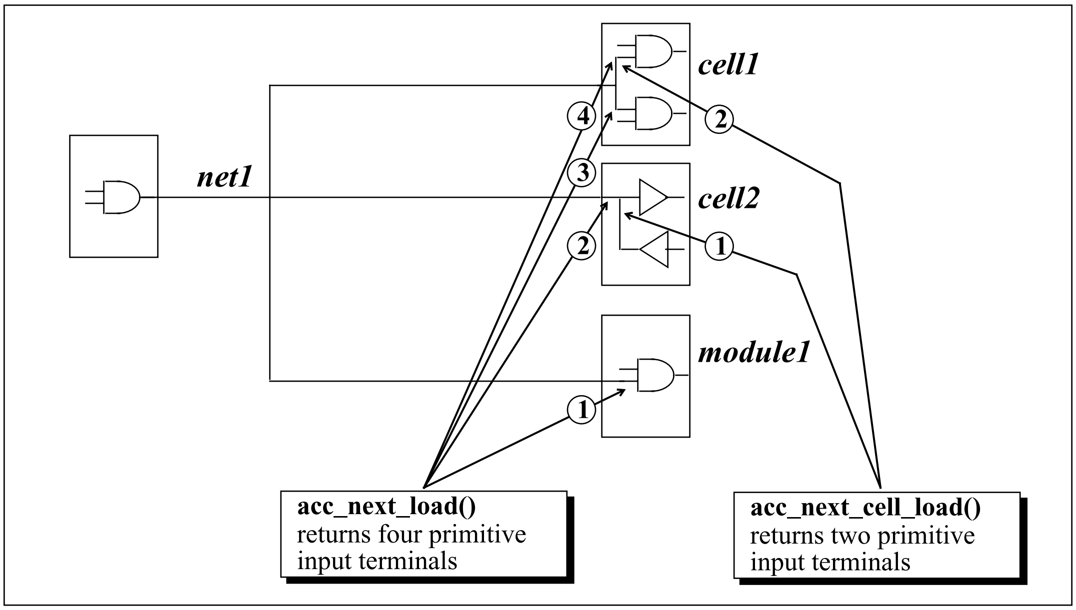


Figure 131—The difference between acc_next_load() and acc_next_cell_load()

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 132 uses **acc_next_load()** to find all terminals driven by a net.

```
include "acc_user.h"

LI_INT32 get_loads()

  handle   net_handle, load_handle, load_net_handle;

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

  /*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();
```

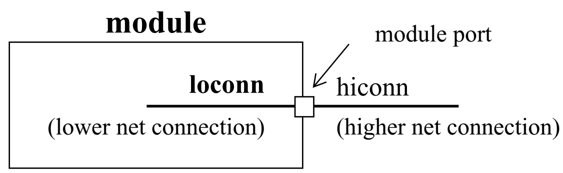
Figure 132—Using acc_next_load()

23.72 acc_next_loconn()

acc_next_loconn()			
Synopsis:	Get handles to hierarchically lower net connections to a port of a module.		
Syntax:	`acc_next_loconn(reference_handle, net_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a net	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a port
	handle	net_handle	Handle of the previous net found; initially `null`
Related routines:	Use acc_handle_loconn() to get a handle to hierarchically lower connection of a specific port bit Use acc_next_hiconn() to get handles to the hierarchically higher connection		

The ACC routine **acc_next_loconn()** shall return handles to the hierarchically lower net connections to a module port. A hierarchically lower connection shall be the part of the net that appears inside the module, as shown in the following diagram:

 Copyright © 2001 IEEE. All rights reserved.



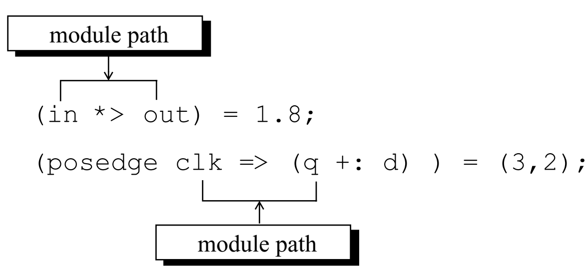
When the reference handle passed to **acc_next_loconn()** is a vector port, the routine shall return the loconn nets bit-by-bit, starting with the msb (leftmost bit) and ending with the lsb (rightmost bit).

Refer to Figure 129 for an example of using **acc_next_loconn()**.

23.73 acc_next_modpath()

acc_next_modpath()			
Synopsis:	Get handles to module paths of a module.		
Syntax:	`acc_next_modpath(reference_handle, path_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a module path	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	path_handle	Handle of the previous path found; initially `null`

The ACC routine **acc_next_modpath()** shall return handles to the module paths in a module. A module path is the specify block path for delays in the Verilog HDL description. For example:



The example in Figure 133 uses **acc_next_modpath()** to list the nets connected to all module paths in a module.

Copyright © 2001 IEEE. All rights reserved.

```
.nclude "acc_user.h"
,I_INT32 get_path_nets(module_handle)
.ndle module_handle;

.andle  path_handle, pathin_handle, pathout_handle;

'*scan all paths in the module */
.o_printf("For module %s:\n",acc_fetch_fullname(module_handle) );
.ath_handle = null;
/hile (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 133—Using acc_next_modpath()

23.74 acc_next_net()

acc_next_net()			
Synopsis:	Get handles to nets in a module.		
Syntax:	`acc_next_net(reference_handle, net_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a net	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	net_handle	Handle of the previous net found; initially `null`
Related routines:	Use acc_object_of_type() to determine if a net is scalar or vector, expanded or unexpanded Use acc_next_bit() to get handles to all bits of an expanded vector net		

The ACC routine **acc_next_net()** shall return handles to the nets within a module scope. The routine shall return a handle to a vector net as a whole; it does not return a handle to each individual bit of a vector net. The routine **acc_object_of_type()** can be used to determine if a net is vector or scalar and if it is expanded or unexpanded. The routine **acc_next_bit()** can be used to retrieve a handle for each bit of an expanded vector net.

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 134 uses **acc_next_net()** to display the names of all nets in a module.

```
#include “acc_user.h”

PLI_INT32 display_net_names()

   handle   mod_handle, net_handle;

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

   /*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 134—Using acc_next_net()

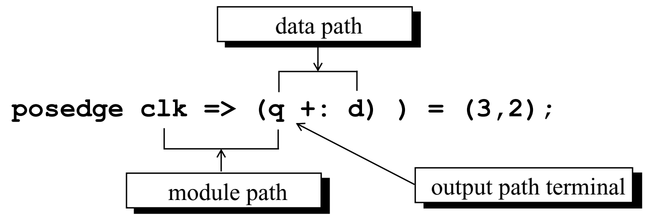
23.75 acc_next_output()

acc_next_output()			
Synopsis:	Get handles to output path terminals of a module path or data path.		
Syntax:	`acc_next_output(reference_handle, terminal_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a module path terminal or data path terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle to a module path or data path
	handle	terminal_handle	Handle of the previous terminal found; initially `null`
Related routines:	Use acc_handle_conn() to get the net attached to the path terminal Use acc_release_object() to free memory allocated by *acc_next_output()*		

The ACC routine **acc_next_output()** shall return handles to the output path terminals of a module path or a data path. The routine **acc_handle_conn()** can be passed the output path terminal handle to derive the net connected to the terminal.

A *module path* is the specify block path for delays in the Verilog HDL description. A *data path* is part of the Verilog HDL description for edge-sensitive module paths, as shown in the following illustration:

Copyright © 2001 IEEE. All rights reserved.



The example shown in Figure 135 uses **acc_next_output()**. It accepts a handle to a scalar net or a net bit-select, and a module path. The application returns `true` if the net is connected to the output of the path.

```
int is_net_on_path_output(net, path)
handle net; /* scalar net or bit-select of vector net */
handle path;
{
   handle pterm_out, pterm_conn, bit;

   /* scan path output terminals */
   pterm_out = null;
   while (pterm_out = acc_next_output(path,  pterm_out) )
   {
      /* retrieve net connected to path terminal */
      pterm_conn = acc_handle_conn (pterm_out);

      if (acc_object_of_type (pterm_conn, accExpandedVector) )
      {
         bit = null;
         while (bit = acc_next_bit (pterm_conn, bit) )
            if (acc_compare_handles (bit, net) )
               return (true);
      }
      else
         if (acc_compare_handles (pterm_conn, net) )
            return (true);
   }

   return (false);
}
```

Figure 135—Using acc_next_output()

 Copyright © 2001 IEEE. All rights reserved.

23.76 acc_next_parameter()

acc_next_parameter()			
Synopsis:	Get handles to parameters within a module.		
Syntax:	`acc_next_parameter(reference_handle, parameter_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a parameter	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a scope
	handle	parameter_handle	Handle of the previous parameter found; initially `null`
Related routines:	Use acc_fetch_paramtype() to determine the parameter data type Use acc_fetch_paramval() to retrieve the parameter value Use acc_next_specparam() to get handles to specify block parameters		

The ACC routine **acc_next_parameter()** shall return handles to the parameters in a scope. This handle can be passed to **acc_fetch_paramtype()** and **acc_fetch_paramval()** to retrieve the data type and value of the parameter. A scope is a module, task, function, or named block.

The example shown in Figure 136 uses **acc_next_parameter()** to scan for all parameters in a module.

```
nclude "acc_user.h"
I_INT32 print_parameter_values(module_handle)
ndle module_handle;

 handle   param_handle;
 /*scan all parameters in the module and display 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", (PLI_INT32)acc_fetch_paramval(param_handle) );
      break;
    case accStringParam:
      io_printf("%s\n",
           (char*)(int)acc_fetch_paramval(param_handle) );
   }
 }
```

Figure 136—Using acc_next_parameter()

Copyright © 2001 IEEE. All rights reserved.

23.77 acc_next_port()

acc_next_port()			
Synopsis:	Gets handles to the ports of a module, or to ports which are connected to a given net or reg.		
Syntax:	`acc_next_port(reference, port_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a module port	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module, net, reg or variable
	handle	object_handle	Handle of the previous port found; initially `null`
Related routines:	Use acc_fetch_direction() to determine the direction of a port Use acc_next_portout() to get handles to just output and inout ports		

The ACC routine **acc_next_port()** shall return handles to the input, output, and inout ports of a module. The handles shall be returned in the order specified by the port list in the module declaration, working from left to right.

The routine **acc_next_port()** shall be used two ways, as shown in Table 173.

Table 173—How acc_next_port() works

If the reference handle is	acc_next_port() shall return
A handle to a module	All ports of the module
A handle to a net, reg or variable	All ports connected to the net, reg or variable within the scope of the net, reg or variable

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 137 uses **acc_next_port()** to find and display the input ports of a module.

```
include "acc_user.h"

LI_INT32 display_inputs(module_handle)
andle module_handle;

  handle      port_handle;
  PLI_INT32   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 137—Using acc_next_port() with a module handle

The example shown in Figure 138 uses **acc_next_port()** to find the port that is connected to a net, and then to display information about other nets connected to each bit of the same port.

```
LI_INT32 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 138—Using acc_next_port() with a net handle

Copyright © 2001 IEEE. All rights reserved.

23.78 acc_next_portout()

acc_next_portout()			
Synopsis:	Get handles to output or inout ports of a module.		
Syntax:	`acc_next_portout(reference_handle, port_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a module port	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	port_handle	Handle of the previous port found; initially `null`
Related routines:	Use acc_fetch_direction() to determine the direction of a port Use acc_next_port() to get handles to input, output, and inout ports		

The ACC routine **acc_next_portout()** shall return handles to the output and inout ports of a module. The handles shall be returned in the order specified by the port list in the module declaration, working from left to right.

The example shown in Figure 139 uses **acc_next_portout()** to find the output and inout ports of a module.

```
include "acc_user.h"

LI_INT32 display_outputs(module_handle)
andle 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 139—Using acc_next_portout()

 Copyright © 2001 IEEE. All rights reserved.

23.79 acc_next_primitive()

acc_next_primitive()			
Synopsis:	Get handles to gates, switches, or user-defined primitives (UDPs) within a module.		
Syntax:	`acc_next_primitive(reference_handle, primitive_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	primitive_handle	Handle of the previous primitive found; initially `null`

The ACC routine **acc_next_primitive()** shall return handles to the built-in and user-defined primitives within a module.

The example shown in Figure 140 uses **acc_next_primitive()** to display the definition names of all primitives in a module.

```
include "acc_user.h"

LI_INT32 get_primitive_definitions()

  handle module_handle, prim_handle;

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

  /*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 140—Using acc_next_primitive()

Copyright © 2001 IEEE. All rights reserved.

23.80 acc_next_scope()

acc_next_scope()			
Synopsis:	Get handles to hierarchy scopes within a scope.		
Syntax:	`acc_next_scope(reference_handle, scope_handle)`		
Returns:	**Type**	Description	
	handle	Handle to a hierarchy scope	
Arguments:	**Type**	**Name**	**Description**
	`handle`	reference_handle	Handle of a scope
	`handle`	scope_handle	Handle of the previous scope found; initially `null`
Related routines:	Use acc_fetch_type() and acc_fetch_fulltype() to determine the type of scope object found Use acc_next_topmod() to get handles to top-module scopes		

The ACC routine **acc_next_scope()** shall return the handles to the internal scopes within a given scope. Internal scopes shall be the immediate children of the *reference_handle*. The reference scope and the internal scopes shall be one of the following:

A top-level module
A module instance
A named begin-end block
A named fork-join block
A Verilog HDL task
A Verilog HDL function

23.81 acc_next_specparam()

acc_next_specparam()			
Synopsis:	Get handles to specify block parameters within a module.		
Syntax:	`acc_next_specparam(reference_handle, specparam_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a specparam	
Arguments:	**Type**	**Name**	**Description**
	handle	module_handle	Handle of a module
	handle	specparam_handle	Handle of the previous specparam found; initially `null`
Related routines:	Use acc_fetch_paramtype() to determine the parameter data type Use acc_fetch_paramval() to retrieve the parameter value Use acc_next_parameter() to get handles to module parameters		

The ACC routine **acc_next_specparam()** shall return handles to the specify block parameters in a module. This handle can be passed to **acc_fetch_paramtype()** and **acc_fetch_paramval()** to retrieve the data type and value.

 Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 141 uses **acc_next_specparam()** to scan for all specparams in a module.

```
nclude "acc_user.h"
I_INT32 print_specparam_values(module_handle)
ndle module_handle;

 handle   sparam_handle;
 /*scan all parameters in the module and display 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 141—Using acc_next_specparam()

23.82 acc_next_tchk()

acc_next_tchk()			
Synopsis:	Get handles to timing checks within a module.		
Syntax:	`acc_next_tchk(reference_handle, timing_check_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a timing check	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a module
	handle	timing_check_handle	Handle of the previous timing check found; initially `null`
Related routines:	Use acc_handle_tchk() to get a timing check handle using the timing check description Use acc_handle_tchkarg1() and acc_handle_tchkarg2() to get handles of the timing check arguments Use acc_handle_notifier() to get a handle to the timing check notifier reg Use acc_fetch_delays(), acc_append_delays(), and acc_replace_delays() to read or modify timing check values		

Copyright © 2001 IEEE. All rights reserved.

The ACC routine **acc_next_tchk()** shall return handles to the timing checks within a module. The handles can be passed to other ACC routines to get the nets or notifier in the timing check, and to read or modify timing check values.

The example shown in Figure 142 uses **acc_next_tchk()** to display information about setup timing checks.

```
nclude "acc_user.h"
I_INT32 show_setup_check_nets()

    handle          mod_handle, cell_handle;
    handle          tchk_handle, tchkarg1_handle, tchkarg2_handle;
    PLI_INT32       tchk_type, counter;

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

 /*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);
        tchkarg2_handle = acc_handle_tchkarg2(tchk_handle);
        io_printf("       1st net is %s\n        2nd net is %s\n",
                   acc_fetch_name(acc_handle_conn(tchkarg1_handle) ),
                   acc_fetch_name(acc_handle_conn(tchkarg2_handle) ) );
       }
    }
 }
 acc_close();
```

Figure 142—Using acc_next_tchk()

 Copyright © 2001 IEEE. All rights reserved.

23.83 acc_next_terminal()

acc_next_terminal()			
Synopsis:	Get handles to terminals of a gate, switch, or user-defined primitive (UDP).		
Syntax:	`acc_next_terminal(reference_handle, terminal_handle)`		
Returns:	**Type**	Description	
	handle	Handle of a primitive terminal	
Arguments:	**Type**	**Name**	**Description**
	handle	reference_handle	Handle of a gate, switch or UDP
	handle	terminal_handle	Handle of the previous terminal found; initially `null`

The ACC routine **acc_next_terminal()** shall return handles to the terminals on a primitive. The handles shall be returned in the order of the primitive instance statement, starting at terminal 0 (the leftmost terminal).

The example shown in Figure 143 uses **acc_next_terminal()** together with **acc_handle_conn()** to retrieve all nets connected to a primitive.

```
include "acc_user.h"

LI_INT32 display_terminals()

     handle   prim_handle,term_handle;

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

     /*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 143—Using acc_next_terminal()

Copyright © 2001 IEEE. All rights reserved.

23.84 acc_next_topmod()

<table>
<tr><th colspan="4">acc_next_topmod()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Get handles to top-level modules.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>acc_next_topmod(module_handle)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>handle</td><td colspan="2">Handle of a top-level module</td></tr>
<tr><td rowspan="2">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>module_handle</td><td>Handle of the previous top-level module found; initially <code>null</code></td></tr>
<tr><td>Related routines:</td><td colspan="3">Use acc_next_child() with a null reference_handle to collect or count top-level modules with acc_collect() and acc_count()</td></tr>
</table>

The ACC routine **acc_next_topmod()** shall return handles to the top-level modules in a design.

The ACC routine **acc_next_topmod()** does not work with **acc_collect()** or **acc_count()**, but **acc_next_child()** with a `null` reference handle argument can be used in place of **acc_next_topmod()**. For example:

```
   acc_count(acc_next_child, null); /* counts top-level modules */

   acc_collect(acc_next_child, null, &count); /* collect top-level
modules */
```

The example shown in Figure 144 uses **acc_next_topmod()** to display the names of all top-level modules.

```
include "acc_user.h"

'LI_INT32 show_top_modules()

     handle         module_handle;

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

     /*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 144—Using acc_next_topmod()

 Copyright © 2001 IEEE. All rights reserved.

23.85 acc_object_in_typelist()

<table>
<tr><th colspan="4">acc_object_in_typelist()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Determine whether an object fits a type or fulltype, or special property, as specified in an input array.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>acc_object_in_typelist(object_handle, object_type_array)</code></td></tr>
<tr><td rowspan="2">Returns:</td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>PLI_INT32</td><td colspan="2"><code>true</code> if the type, fulltype, or property of an object matches one specified in the array; <code>false</code> if there is no match</td></tr>
<tr><td rowspan="3">Arguments:</td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>handle</td><td>object_handle</td><td>Handle of an object</td></tr>
<tr><td>static integer array</td><td>object_type_array</td><td>Static integer array containing one or more predefined integer constants that represent the types and properties of objects desired; the last element shall be 0</td></tr>
<tr><td>Related routines:</td><td colspan="3">Use acc_object_of_type() to check for a match to a single predefined constant</td></tr>
</table>

The ACC routine **acc_object_in_typelist()** shall determine whether an object fits one of a list of types, fulltypes, or special properties. The properties for which **acc_object_in_typelist()** is to check shall be listed as an array of constants in a static integer array. The array can contain any number and combination of the predefined integer constants, and it shall be terminated by a **0**.

The following C language statement shows an example of how to declare an array of object types called `wired_nets`:

```
static PLI_INT32
wired_nets[ 5] ={ accWand,accWor,accTriand,accTrior,0} ;
```

When this array is passed to **acc_object_in_typelist()**, the ACC routine shall return `true` if its object_handle argument is a wired net.

All type and fulltype constants shall be supported by **acc_object_in_typelist()**. These constants are listed in Table 113.

The special property constants supported by **acc_object_in_typelist()** are listed in Table 174.

The example shown in Figure 145 uses **acc_object_in_typelist()** to determine if a net is a wired net. The application then displays the name of each wired net found.

Copyright © 2001 IEEE. All rights reserved.

```
nclude "acc_user.h"

.I_INT32 display_wired_nets()

  static PLI_INT32  wired_nets[5]={accWand,accWor,accTriand,accTrior,0};
  handle            net_handle;

  /*reset environment for ACC routines*/
  acc_initialize();

  /*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) )

  acc_close();
```

Figure 145—Using acc_object_in_typelist()

23.86 acc_object_of_type()

acc_object_of_type()			
Synopsis:	Determine whether an object fits a specified type or fulltype, or special property.		
Syntax:	`acc_object_of_type(object_handle, object_type)`		
Returns:	**Type**	Description	
	PLI_INT32	`true` if the type, fulltype, or property of an object matches the object_type argument `false` if there is no match	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle of an object
	PLI_INT32	object_type	An integer constant that represents a type, fulltype, or special property
Related routines:	Use acc_object_in_typelist() to check for a match to any of several predefined constants		

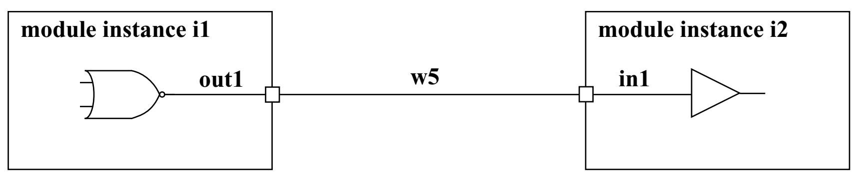
The ACC routine **acc_object_of_type()** shall determine whether an object fits a specified type, fulltype, or special property. The type, fulltype, or property is an integer constant, defined in `acc_user.h`. All type and fulltype constants shall be supported by **acc_object_of_type()**. These constants are listed in Table 113. The special property constants supported by **acc_object_of_type()** are listed in Table 174.

 Copyright © 2001 IEEE. All rights reserved.

Table 174—Special object properties

Property of object	Predefined integer constant
Scalar	**accScalar**
Vector	**accVector**
Collapsed net	**accCollapsedNet**
Expanded vector	**accExpandedVector**
Unexpanded vector	**accUnExpandedVector**
Hierarchy scope	**accScope**
Module path with `ifnone` condition	**accModPathHasIfnone**

Simulated nets and *collapsed* nets are defined as follows. When a Verilog HDL source description connects modules together, a chain of nets with different scopes and names are connected, as is illustrated in the following simple diagram:



In this small circuit, nets `out1`, `w5`, and `in1` are all tied together, effectively becoming the same net. Software products can collapse nets that are connected together within the data structure of the product. The resultant net after collapsing is referred to as a simulated net; the other nets are referred to as collapsed nets. The ACC routines can obtain a handle to any net, whether it is collapsed or not. The routine **acc_object_of_type()** can be used to determine if a net has been collapsed. The routine **acc_handle_simulated_net()** can be used to find the resultant net from the net collapsing process.

Expanded and *unexpanded* vectors determine if ACC routines can access a vector as a whole or access the bits within a vector. If a vector has the property **accExpandedVector**, then access to the discrete bits of the vector shall be permitted. This property has to be `true` in order for certain ACC routines, such as **acc_next_bit()**, to access each bit of a vector. If a vector has the property **accUnExpandedVector**, then access to the vector as a whole shall be permitted. This property has to be `true` in order for certain ACC routines to access the complete vector. A vector object can have just one of these properties `true`, or both can be `true`.

acc_object_of_type() with an **accScope** type constant will return true if the reference object is a Verilog scope. A scope is a module, task, function or named block.

acc_object_of_type() with an **accModPathHasIfnone** type constant will return true if the reference object is a Verilog module path, and there is an `ifnone` condition specified for the path.

The example shown in Figure 146 uses **acc_object_of_type()** to determine whether nets are collapsed nets. The application then displays each collapsed net, along with the simulated net.

Copyright © 2001 IEEE. All rights reserved.

```
include "acc_user.h"

PLI_INT32 display_collapsed_nets()

  handle   mod_handle;
  handle   net_handle;
  handle   simulated_net_handle;

  /*reset environment for ACC routines*/
  acc_initialize();

  /*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 146—Using acc_object_of_type()

23.87 acc_product_type()

acc_product_type()			
Synopsis:	Get the software product type that is calling the PLI application.		
Syntax:	`acc_product_type()`		
	Type	Description	
Returns:	PLI_INT32	A predefined integer constant representing the software product type	
	Type	**Name**	**Description**
Arguments:	None		

The ACC routine **acc_product_type()** shall return a predefined integer constant that identifies the class of software product that is calling the PLI application. This information can be useful when a PLI application needs to customize the routine to specific types of software implementations. For example, a delay calculator might use typical delays for logic simulation and min:typ:max delays for timing analysis.

 Copyright © 2001 IEEE. All rights reserved.

The integer constant values returned by **acc_product_type()** are listed in Table 175.

Table 175—Product types returned by acc_product_type()

If the product is	acc_product_type() returns
A logic simulator	**accSimulator**
A timing analyzer	**accTimingAnalyzer**
A fault simulator	**accFaultSimulator**
Some other product	**accOther**

NOTE Software product vendors can define additional integer constants specific to their products.

The example shown in Figure 147 uses **acc_product_type()** to identify and display the product type being used.

```
nclude "acc_user.h"
I_INT32 show_application()

* reset environment for ACC routines */
cc_initialize();

* show application type and ACC routine version */
witch (acc_product_type() )

case accSimulator:
 io_printf("Running logic simulation with PLI version %s\n",acc_version()
 break;
case accTimingAnalyzer:
 io_printf("Running timing analysis with PLI version %s\n",acc_version()
 break;
case accFaultSimulator:
 io_printf("Running fault simulation with PLI version %s\n",acc_version()
 break;
default:
 io_printf("Running other product with PLI version %s\n",acc_version());
}
cc_close();
```

Figure 147—Using acc_product_type()

Copyright © 2001 IEEE. All rights reserved.

23.88 acc_product_version()

acc_product_version()			
Synopsis:	Get the version of the software product that is linked to the ACC routines.		
Syntax:	`acc_product_version()`		
Returns:	**Type**	Description	
	PLI_BYTE8 *	Pointer to a character string	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_product_type() to get the type of software product Use acc_version() to get the version of PLI ACC routines		

The ACC routine **acc_product_version()** shall return a pointer to a character string that indicates the version of the software product that called the PLI application. The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

The character string shall be in the following format:

```
<product_name> Version <version_number>
```

For example:

```
"Verilog Simulator Version OVIsim 1.0"
```

The string returned by **acc_product_version()** shall be defined by the software tool vendor.

The example shown in Figure 148 uses **acc_product_version()** to identify the version of the software product that is linked to ACC routines.

```
include "acc_user.h"

LI_INT32 show_versions()

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

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

  acc_close();
```

Figure 148—Using acc_product_version()

 Copyright © 2001 IEEE. All rights reserved.

23.89 acc_release_object()

acc_release_object()			
Synopsis:	Deallocate memory allocated by calls to acc_next_input() and acc_next_output().		
Syntax:	`acc_release_object(object_handle)`		
	Type	Description	
Returns:	PLI_INT32	0 if successful; 1 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle to an input or output terminal path
Related routines:	Use acc_next_input() to get handles to module path inputs and data path inputs Use acc_next_output() to get handles to module path outputs and data path outputs		

The ACC routine **acc_release_object()** shall deallocate memory that was allocated by a call to **acc_next_input()** or **acc_next_output()**. The routine should be called after using these ACC routines under the following circumstances:

Not all inputs or outputs were scanned.
The input or output path had only one terminal.
An error was returned.

The example shown in Figure 149 finds the data path corresponding to an input module path, and it displays the source and destination port names for the data path. The example calls **acc_next_input()** and **acc_next_output()** to get the first input and output, respectively, for a given path. Since these routines are only called once, **acc_release_object()** is called to free the memory allocated for the input and output handles.

```
PLI_INT32 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 data path */
   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 149—Using acc_release_object()

Copyright © 2001 IEEE. All rights reserved.

23.90 acc_replace_delays()

<table>
<tr><th colspan="4">acc_replace_delays() for single delay values (accMinTypMaxDelays set to false)</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Replace existing delays for primitives, module paths, timing checks, module input ports, and inter-module paths.</td></tr>
<tr><td>Syntax:
Primitives</td><td colspan="3"><code>acc_replace_delays(object_handle, rise_delay, fall_delay, z_delay)</code></td></tr>
<tr><td>Module paths
Intermodule paths
Ports or port bits</td><td colspan="3"><code>acc_replace_delays(object_handle,
d1,d2,d3,d4,d5,d6,d7,d8,d9,d10,d11,d12)</code></td></tr>
<tr><td>Timing checks</td><td colspan="3"><code>acc_replace_delays(object_check_handle, limit)</code></td></tr>
<tr><td></td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>Returns:</td><td>PLI_INT32</td><td colspan="2">1 if successful; 0 if an error occurred</td></tr>
<tr><td></td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>Arguments:</td><td><code>handle</code></td><td>object_handle</td><td>Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path</td></tr>
<tr><td></td><td><code>double</code></td><td>rise_delay
fall_delay</td><td>Rise and fall delay for 2-state primitives or 3-state primitives</td></tr>
<tr><td>Conditional</td><td><code>double</code></td><td>z_delay</td><td>If accToHiZDelay is set to from_user :
turn-off (to Z) transition delay for 3-state primitives</td></tr>
<tr><td></td><td><code>double</code></td><td>d1</td><td>For module/intermodule paths and input ports/port bits:
If accPathDelayCount is set to 1 :
delay for all transitions
If accPathDelayCount is set to 2 or 3 :
rise transition delay
If accPathDelayCount is set to 6 or 12 :
0->1 transition delay</td></tr>
<tr><td>Conditional</td><td><code>double</code></td><td>d2</td><td>For module/intermodule paths and input ports/port bits:
If accPathDelayCount is set to 2 or 3 :
fall transition delay
If accPathDelayCount is set to 6 or 12 :
1->0 transition delay</td></tr>
<tr><td>Conditional</td><td><code>double</code></td><td>d3</td><td>For module/intermodule paths and input ports/port bits:
If accPathDelayCount is set to 3 :
turn-off transition delay
If accPathDelayCount is set to 6 or 12 :
0->Z transition delay</td></tr>
<tr><td>Conditional</td><td><code>double</code></td><td>d4
d5
d6</td><td>For module/intermodule paths and input ports/port bits:
If accPathDelayCount is set to 6 or 12 :
<code>d4</code> is Z->1 transition delay
<code>d5</code> is 1->Z transition delay
<code>d6</code> is Z->0 transition delay</td></tr>
<tr><td>Conditional</td><td><code>double</code></td><td>d7
d8
d9
d10
d11
d12</td><td>For module/intermodule paths and input ports/port bits:
If accPathDelayCount is set to 12 :
<code>d7</code> is 0->X transition delay
<code>d8</code> is X->1 transition delay
<code>d9</code> is 1->X transition delay
<code>d10</code> is X->0 transition delay
<code>d11</code> is X->Z transition delay
<code>d12</code> is Z->X transition delay</td></tr>
<tr><td></td><td><code>double</code></td><td>limit</td><td>Limit of timing check</td></tr>
</table>

 Copyright © 2001 IEEE. All rights reserved.

acc_replace_delays() for min:typ:max delays (accMinTypMaxDelays set to true)			
Synopsis:	Replace min:typ:max delay values for primitives, module paths, timing checks, module input ports, or intermodule paths; the delay values are contained in an array.		
Syntax:	`acc_append_delays(object_handle, array_ptr)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	`handle`	object_handle	Handle of a primitive, module path, timing check, module input port, bit of a module input port, or intermodule path
	double address	array_ptr	Pointer to array of min:typ:max delay values; the size of the array depends on the type of object and the setting of **accPathDelayCount** (see Section 22.8)

The ACC routine **acc_replace_delays()** shall work differently depending on how the configuration parameter **accMinTypMaxDelays** is set. When this parameter is set to `false`, a single delay per transition shall be assumed, and delays shall be passed as individual arguments. For this single delay mode, the first syntax table in this section shall apply.

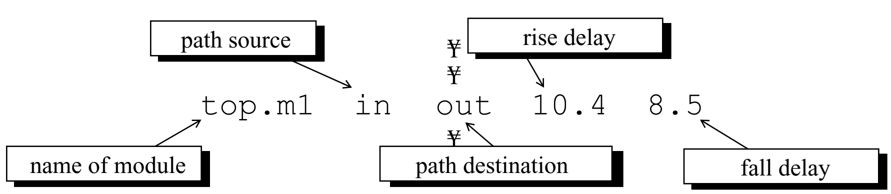
When **accMinTypMaxDelays** is set to `true`, **acc_replace_delays()** shall pass one or more sets of minimum:typical:maximum delays contained in an array, rather than single delays passed as individual arguments. For this min:typ:max delay mode, the second syntax table in this section shall apply.

The number of delay values replaced by **acc_replace_delays()** shall be determined by the type of object and the setting of configuration parameters. Refer to 22.8 for a description of how the number of delay values are determined.

The routine **acc_replace_delays()** shall write delays in the timescale of the module that contains the object_handle.

When altering the delay via **acc_replace_delays()**, the value of the reject/error region will not be affected unless the limits exceed the value of the delay. If the reject/error limits exceed the delay they will be truncated down to the new delay limit.

The example shown in Figure 150 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 the following diagram:



Copyright © 2001 IEEE. All rights reserved.

```
nclude <stdio.h>
nclude "acc_user.h"

efine NAME_SIZE 256
I_INT32 write_path_delays()

 FILE        *infile;
 PLI_BYTE8   full_module_name[NAME_SIZE];
 PLI_BYTE8   pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
 double      rise, fall;
 handle      mod_handle, path_handle;

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

 /*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 150—Using acc_replace_delays() in single delay mode

The example shown in Figure 151 uses **acc_replace_delays()** to scale the min:typ:max delays on all primitive delays inside cells within a given scope. The application fetches the existing delays for an object, multiplies the delays by a scale factor, and replaces the delays with the new, scaled values. This example assumes that the user application is associated through the PLI interface mechanism with a user-defined system task called `$scaleprimdelays`. The scope and scale factors are passed as arguments as follows:

```
$scaleprimdelays( mychip, 0.4, 1.0, 1.6 );
```

scope

scale factor for typical delay

scale factor for minimum delay

scale factor for maximum delay

 Copyright © 2001 IEEE. All rights reserved.

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

LI_INT32 scale_prim_delays()

handle top, cell, prim;
int    i;
double da[ 9] ;
double min_scale_factor, typ_scale_factor, max_scale_factor;

acc_initialize();
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);
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);
      }
  }

acc_close();
```

array has to 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 151—Using acc_replace_delays() in min:typ:max delays mode

Copyright © 2001 IEEE. All rights reserved.

23.91 acc_replace_pulsere()

acc_replace_pulsere()			
Synopsis:	Replace existing pulse handling *reject_limit* and *e_limit* for a module path, intermodule path or module input port.		
Syntax:	`acc_replace_pulsere(object,r1,e1, r2,e2, r3,e3, r4,e4, r5,e5, r6,e6, r7,e7, r8,e8, r9,e9, r10,e10, r11,e11, r12,e12)`		
	Type	Description	
Returns:	PLI_INT32	1 if successful; 0 if an error is encountered	
	Type	**Name**	**Description**
Arguments:	handle	object	Handle of module path, intermodule path or module input port
	double	r1...r12	reject_limit values; the number of arguments is determined by **accPathDelayCount**
	double	e1...e12	e_limit values; the number of arguments is determined by **accPathDelayCount**
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_append_pulsere() to append existing pulse handling values Use acc_set_pulsere() to set pulse handling values as a percentage of the path delay Use acc_configure() to set accPathDelayCount		

The ACC routine **acc_replace_pulsere()** shall replace existing pulse handling *reject_limit* and *e_limit* values for a module path, intermodule path and module input port. The reject_limit and e_limit values are used to control how *pulses* are propagated through paths.

A pulse is defined as two transitions that occur in a shorter period of time than the delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_limit* and an *e_limit* pair of values, where:

The reject_limit shall set a threshold for determining when to reject a pulse any pulse less than the reject_limit shall not propagate to the output
The e_limit shall set a threshold for determining when a pulse is considered to be an error any pulse less than the e_limit and greater than or equal to the reject_limit shall propagate a logic `x`
A pulse that is greater than or equal to the e_limit shall propagate
The example in Table 176 illustrates the relationship between the reject_limit and the e_limit.

Table 176—Pulse control example

When	The pulse shall be
reject_limit = 10.5 *e_limit* = 22.6	Rejected if < 10.5 An error if >= 10.5 and < 22.6 Passed if >= 22.6

 Copyright © 2001 IEEE. All rights reserved.

The following rules shall apply when specifying pulse handling values:

a) The value of reject_limit shall be less than or equal to the value of e_limit.

b) The reject_limit and e_limit shall not be greater than the delay.

If any of the limits do not meet the above rules, they will be truncated.

The number of pulse control values that **acc_replace_pulsere()** sets shall be controlled using the ACC routine **acc_configure()** to set the delay count configuration parameter **accPathDelayCount**, as shown in Table 177.

Table 177—How the value of accPathDelayCount affects acc_replace_pulsere()

When accPathDelayCount is	acc_replace_pulsere() shall write
1	One pair of reject_limit and e_limit values: one pair for all transitions, r1 and e1
2	Two pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2
3	Three pairs of reject_limit and e_limit values: one pair for rise transitions, r1 and e1 one pair for fall transitions, r2 and e2 one pair for turn-off transitions, r3 and e3
6 (the default)	Six pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6
12	Twelve pairs of reject_limit and e_limit values a different pair for each possible transition among 0, 1, X and Z: one pair for 0->1 transitions, r1 and e1 one pair for 1->0 transitions, r2 and e2 one pair for 0->Z transitions, r3 and e3 one pair for Z->1 transitions, r4 and e4 one pair for 1->Z transitions, r5 and e5 one pair for Z->0 transitions, r6 and e6 one pair for 0->X transitions, r7 and e7 one pair for X->1 transitions, r8 and e8 one pair for 1->X transitions, r9 and e9 one pair for X->0 transitions, r10 and e10 one pair for X->Z transitions, r11 and e11 one pair for Z->X transitions, r12 and e12

The minimum number of pairs of reject_limit and e_limit arguments to pass to **acc_replace_pulsere()** shall equal the value of **accPathDelayCount**. Any unused reject_limit and e_limit argument pairs shall be ignored by **acc_replace_pulsere()** and can be dropped from the argument list.

If **accPathDelayCount** is not set explicitly, it shall default to `6`, and therefore six pairs of pulse reject_limit and e_limit arguments have to be passed when **acc_replace_pulsere()** is called. Note that the value assigned to **accPathDelayCount** also affects **acc_append_delays()**, **acc_fetch_delays()**, **acc_replace_delays()**, **acc_append_pulsere(),** and **acc_fetch_pulsere()**.

Copyright © 2001 IEEE. All rights reserved.

Pulse control values shall be replaced using the timescale of the module that contains the object handle.

The example shown in Figure 152 uses **acc_replace_pulsere()** to replace rise and fall pulse handling values of paths listed in a file `path.dat`.

```
nclude <stdio.h>
nclude "acc_user.h"

efine NAME_SIZE 256

I_INT32 replace_halfpulsevals()

ILE        *infile;
LI_BYTE8   mod_name[NAME_SIZE];
LI_BYTE8   pathin_name[NAME_SIZE], pathout_name[NAME_SIZE];
andle      mod, path;
ouble      rise_reject_limit=0.0, rise_e_limit=0.0,
           fall_reject_limit=0.0, fall_e_limit=0.0;

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

/*set accPathDelayCount to return two pairs of pulse handling values;*/
/* one each for rise and fall transitions*/
 acc_configure(accPathDelayCount, "2");

/*read all module path specifications from file "path.dat"*/
 infile = fopen("path.dat", "r");
 while(fscanf(infile,"%s %s %s",mod_name,pathin_name,pathout_name)!=EOF)
 {
    mod=acc_handle_object(mod_name);
    path=acc_handle_modpath(mod,pathin_name,pathout_name);
    rise_reject_limit = .05;
    if(acc_replace_pulsere(path,  &rise_reject_limit, &rise_e_limit,
                                  &fall_reject_limit, &fall_e_limit) )
    {
       io_printf("rise reject limit = %lf, rise e limit = %lf\n",
                  rise_reject_limit, rise_e_limit);
       io_printf("fall reject limit = %lf, fall e limit = %lf\n",
                  fall_reject_limit, fall_e_limit);
    }
 }
 acc_close();
```

Figure 152—Using acc_replace_pulsere()

 Copyright © 2001 IEEE. All rights reserved.

23.92 acc_reset_buffer()

acc_reset_buffer()			
Synopsis:	Reset the string buffer to the beginning.		
Syntax:	`acc_reset_buffer()`		
	Type	Description	
Returns:	void		
	Type	**Name**	**Description**
Arguments	None		
Related routines:	All ACC routines that return a pointer to a character string		

The ACC routine **acc_reset_buffer()** shall reset the string buffer to its beginning. The string buffer shall be used as temporary storage by other ACC routines that return a pointer to a character string. Refer to 22.9 for more information on the character string buffer.

23.93 acc_set_interactive_scope()

acc_set_interactive_scope()			
Synopsis:	Set the interactive scope of a software tool.		
Syntax:	`acc_set_interactive_scope(scope, callback_flag)`		
	Type	Description	
Returns:	handle	Handle of a Verilog hierarchy scope	
	Type	**Name**	**Description**
Arguments:	handle	scope	Handle to the scope which will be the new interactive scope
	PLI_INT32	callback_flag	If set to TRUE, then the misctf routines shall be called with reason reason_scope immediately. If set to FALSE, then the misctf routines are not called
Related routines:	Use acc_handle_interactive_scope() to get a handle for the current interactive scope		

The ACC routine **acc_set_interactive_scope()** shall set the Verilog HDL design scope where the interactive mode of the software product is operating.

A scope shall be

A top-level module
A module instance
A named begin-end block
A named fork-join block
A Verilog HDL task
A Verilog HDL function

Copyright © 2001 IEEE. All rights reserved.

23.94 acc_set_pulsere()

acc_set_pulsere()			
Synopsis:	Set the pulse handling values for a module path, intermodule path or module input port as a percentage of the delay.		
Syntax:	`acc_set_pulsere(object, reject_percentage, e_percentage)`		
Returns:	**Type**	Description	
	PLI_INT32	Always returns 0	
Arguments:	**Type**	**Name**	**Description**
	handle	object	Handle of a module path, intermodule path or module input port
	double	reject_percentage	Multiplier of the delay value that forms the upper limit for rejecting a path output pulse
	double	e_percentage	Multiplier of the delay value that forms the upper limit for setting a path output pulse to `x`.
Related routines:	Use acc_fetch_pulsere() to get current pulse handling values Use acc_append_pulsere() to append existing pulse handling values Use acc_replace_pulsere() to replace existing pulse handling values		

The ACC routine **acc_set_pulsere()** shall set the pulse handling values *reject_percentage* and *e_percentage* for a module path, intermodule path or module input port, specified as a percentage multiplier of the delay.

A pulse is defined as two transitions that occur in a shorter period of time than the delay. Pulse control values determine whether a pulse should be rejected, propagated through to the output, or considered an error. The pulse control values consist of a *reject_percentage* and an *e_percentage* pair of values, where

The reject_percentage shall set a threshold for determining when to reject a pulse any pulse less than the reject_percentage shall not propagate

The e_percentage shall set a threshold for determining when a pulse is considered to be an error any pulse less than the e_percentage and greater than or equal to the reject_percentage shall propagate a logic `x`

A pulse that is greater than or equal to the e_percentage shall propagate

The example in Table 178 illustrates the relationship between the reject_percentage and the e_percentage.

Table 178—Pulse control example

Given a path with a delay of 5.0	
When	**A pulse shall be**
reject_percentage = 0.5 *e_percentage* = 1.0	Rejected if < 2.5 (50% of path delay) An error if >= 2.5 and < 5.0 (between 50% and 100% of path delay) Passed if >= 5.0 (greater than or equal to 100% of path delay)

 Copyright © 2001 IEEE. All rights reserved.

The following rules shall apply when specifying pulse handling values:

a) The reject_percentage and e_percentage shall be greater than or equal to 0.0 and less than or equal to 1.0.

b) The value of reject_percentage shall be less than or equal to the value of e_percentage.

The example shown in Figure 153 uses **acc_set_pulsere()** to set pulse control values for each path in a module such that all pulses between 0 and the path delay generate an X at the path output.

```
include "acc_user.h"

LI_INT32 set_pulse_control_e(module)
andle module;

  handle   path;

  /*set pulse control values for all paths in the module*/
  path = null;
  while (path = acc_next_modpath(module, path) )
     acc_set_pulsere(path, 0.0, 1.0);
```

Figure 153—Using acc_set_pulsere()

23.95 acc_set_scope()

acc_set_scope()			
Synopsis:	Set a scope for acc_handle_object() to use when searching in the design hierarchy.		
Syntax:	`acc_set_scope(module_handle, module_name)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string containing the full hierarchical name of the scope set; *null* if an error occurred	
	Type	**Name**	**Description**
Arguments:	handle	module_handle	A handle to a module
Optional	quoted string or PLI_BYTE8 *	module_name	Quoted string or pointer to a character string with the name of a module instance (optional: used when **accEnableArgs** is set and module_handle is `null`)
Related routines:	Use acc_handle_object() to get a handle to any named object Use acc_configure(accEnableArgs, acc_set_scope) to use the module_name argument Use acc_set_interactive_scope() to set the interactive scope		

The ACC routine **acc_set_scope()** shall set the scope and search rules for the routine **acc_handle_object()**. The way that **acc_set_scope()** functions shall be dependent on the setting of configuration parameters as shown in Table 179.

Copyright © 2001 IEEE. All rights reserved.

Table 179—How acc_set_scope() works

If	acc_set_scope() shall
Default mode, or **acc_configure(accEnableArgs, no_acc_set_scope)** is called, and *module_handle* is a valid handle	Set the scope to the level of *module_handle* in the design hierarchy and ignore the optional *module_name* argument
Default mode, or **acc_configure(accEnableArgs, no_acc_set_scope)** is called, and *module_handle* is `null`	Set the scope to the top-level module that appears first in the source description
The routine **acc_configure(accEnableArgs, acc_set_scope)** has been called, and *module_handle* is a `null`	Set scope to the level of *module_name* in the design hierarchy
The routine **acc_configure(accEnableArgs, acc_set_scope)** has been called, and *module_handle* is a valid handle	Set scope to the level of *module_handle* in the design hierarchy and ignore the optional *module_name* argument
The routine **acc_configure(accEnableArgs, acc_set_scope)** has been called, and *module_handle* and *module_name* are both `null`	Set scope to the top-level module that appears first in the source description

To use the optional module_name argument, the configuration parameter **accEnableArgs** first has to be set by calling **acc_configure()** as follows:

```
acc_configure(accEnableArgs, “acc_set_scope”);
```

If **accEnableArgs** is not set for **acc_set_scope()**, the routine shall ignore its optional argument. When the optional argument is not required for a call to **acc_set_scope()**, the argument can be dropped.

The example shown in Figure 154 uses **acc_set_scope()** to set a scope for the ACC routine **acc_handle_object()** to determine if a net is in a module.

 Copyright © 2001 IEEE. All rights reserved.

```
include “acc_user.h”

LI_INT32 is_net_in_module(module_handle,net_name)
andle      module_handle;
LI_BYTE8   *net_name;

  handle   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 154—Using acc_set_scope()

23.96 acc_set_value()

acc_set_value()			
Synopsis:	Set and propagate a value on a reg, variable, user-defined system function or a sequential UDP; procedurally assign a reg or variable; force a reg, variable, or net.		
Syntax:	`acc_set_value(object_handle, value_p, delay_p)`		
Returns:	**Type**	Description	
	PLI_INT32	Zero if no errors; nonzero if an error occurred	
Arguments:	**Type**	**Name**	**Description**
	handle	object_handle	Handle to a reg, variable, net, user-defined system function, or sequential UDP
	p_setval_value	value_p	Pointer to a structure containing value to be set
	p_setval_delay	delay_p	Pointer to a structure containing delay before value is set
Related routines:	Use acc_fetch_value() to retrieve a logic value Use acc_fetch_size() to get the number of bits in a vector		

The ACC routine **acc_set_value()** shall set and propagate a value onto a reg, integer variable, time variable, real variable, or a sequential UDP. The routine shall also perform procedural assign/deassign or procedural force/release functions.

Copyright © 2001 IEEE. All rights reserved.

The **acc_set_value()** routine shall also return the value of a system function by passing a handle to the user-defined system function as the object handle. This should only occur during execution of the calltf routine for the system function. Attempts to use **acc_set_value()** with a handle to the system function when the calltf routine is not active shall be ignored. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

The logic value and propagation delay information shall be placed in separate structures. To use **acc_set_value()** to propagate a value, follow these basic steps:

a) Allocate memory for the structures `s_setval_value`, `s_setval_delay`, and if using vectors, `s_acc_vecval`.

b) Set the appropriate fields in each structure to the desired values.

c) Call **acc_set_value()** with an object handle and pointers to the `s_setval_value` and `s_setval_delay` structures.

The structure `s_setval_value` shall contain the value to be written. A value can be entered into this structure as a string, scalar, integer, real, or as an *aval/bval* pair. The `s_setval_value` structure is defined in `acc_user.h` and listed in Figure 155 (note that this structure is also used with the **acc_fetch_value()** routine).

The *format* field in the `s_setval_value` structure shall indicate the value type. The format shall be a predefined integer constant, listed in Table 180.

The *value* union in the `s_setval_value` structure shall be the value to be written. The value is placed in the appropriate field within the union for the format selected.

```
typedef struct t_setval_value
{
  PLI_INT32 format;
  union
    {
      PLI_BYTE8    *str;
      PLI_INT32     scalar;
      PLI_INT32     integer;
      double        real;
      p_acc_vecval  vector;
    } value;
} s_setval_value, *p_setval_value, s_acc_value, *p_acc_value;
```

Figure 155—The s_setval_value structure used by acc_set_value()

Table 180—Predefined constants for the format field of s_setval_value

Value format	Definition
accScalarVal	One of: **acc0**, **acc1**, **accZ**, **accX**
accVectorVal	*aval* and *bval* bit groups, with each group being an integer quantity
accIntVal	An integer quantity

 Copyright © 2001 IEEE. All rights reserved.

Table 180—Predefined constants for the format field of s_setval_value (continued)

Value format	Definition
accRealVal	A real-valued quantity
accStringVal	For integers and appropriately sized regs, any ASCII string; for real-valued objects, any string that represents a real number
accBinStrVal	A base 2 representation as a string
accOctStrVal	A base 8 representation as a string
accDecStrVal	A base 10 representation as a string
accHexStrVal	A base 16 representation as a string

When the *format* field of the `s_acc_vecval` structure is set to **accVectorVal**, the *value* union field used shall be *vector*. The *vector* field is set to a pointer or an array of `s_acc_vecval` structures that contain *aval/ bval* pairs for each bit of the vector. The `s_acc_vecval` structure is listed in Figure 156.

```
typedef struct t_acc_vecval
{
  PLI_INT32 aval;
  PLI_INT32 bval;
} s_acc_vecval, *p_acc_vecval;
```

Figure 156—s_acc_vecval structure

The array of `s_acc_vecval` structures shall contain a record for every 32 bits of the vector, plus a record for any remaining bits. Memory has to be allocated by the user for the array of `s_acc_vecval` structures. If a vector has N bits, the size of the array shall be `((N-1)/32)+1` *s_acc_vecval* records. The routine **acc_fetch_size()** can be used to determine the value of N.

The lsb of the vector shall be represented by the lsb of the first record of `s_acc_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the second record of the array, and so on. Each bit of the vector shall be encoded as an *aval/bval* pair. The encoding for each bit is shown in Table 181.

Table 181—Encoding of bits in the s_acc_vecval structure

aval	bval	Value
0	0	0
1	0	1
0	1	Z
1	1	X

The structure `s_setval_delay` shall control how values are to be propagated into the Verilog HDL data structure. The structure is defined in `acc_user.h` and is listed in Figure 157.

The *time* field in the `s_setval_delay` structure shall indicate the delay that shall take place before a reg value assignment. The time field shall be of type `s_acc_time` structure, as shown in Figure 158.

Copyright © 2001 IEEE. All rights reserved.

The *model* field in the `s_setval_delay` structure shall determine how the delay shall be applied, and how other simulation events scheduled for the same object shall be affected. The delay *model* shall be specified using predefined integer constants, listed in Table 182 and Table 184.

```
typedef struct t_setval_delay
{
  s_acc_time time;
  PLI_INT32  model;
} s_setval_delay, *p_setval_delay;
```

Figure 157—The s_setval_delay structure for acc_set_value()

Table 182—Predefined delay constants for the model field of s_setval_delay

Integer constant	Delay model	Description
accNoDelay	No delay	Sets a reg, variable or sequential UDP to the indicated value with no delay; other events scheduled for the object are not affected
accInertialDelay	Inertial delay	Sets a reg or variable to the indicated value after the specified delay; all scheduled events on the object are removed before this event is scheduled
accTransportDelay	Modified transport delay	Sets a reg or variable to the indicated value after the specified delay; all scheduled events on the object for times later than this event are removed
accPureTransportDelay	Pure transport delay	Sets a reg or variable to the indicated value after the specified delay; no scheduled events on the object are removed

When setting the value of a sequential UDP, the *model* field shall be **accNoDelay**, and the new value shall be assigned with no delay even if the UDP instance has a delay.

The `s_acc_time` structure shall hold the delay value that shall be used by **acc_set_value()**. The `s_acc_time` structure is defined in `acc_user.h` and is listed in Figure 158.

The *type* field in the `s_acc_time` structure shall indicate the data type of the delay that shall be stored in the structure. The type shall be specified using predefined integer constants, listed in Table 183.

The *low* field shall be an integer that represents the lower 32 bits of a 64-bit delay value.

The *high* field shall be an integer that represents the upper 32 bits of a 64-bit delay value.

The *real* field shall be a double that represents the delay as a real number value.

 Copyright © 2001 IEEE. All rights reserved.

```
typedef struct t_acc_time
{
  PLI_INT32 type;
  PLI_INT32 low,
            high;
  double    real;
} s_acc_time, *p_acc_time;
```

Figure 158—The s_acc_time structure for acc_set_value()

Table 183—Predefined time constants for the type field of s_acc_time

Integer constant	Description
accTime	Delay is a 64-bit integer; time shall be scaled to the timescale in effect for the module containing the object.
accSimTime	Delay is a 64-bit integer; time shall be scaled to the time units being used by the simulator
accRealTime	Delay is a real number; time shall be scaled to the timescale in effect for the module containing the object.

The routine **acc_set_value()** shall be used to perform a procedural continuous assignment of a value to a reg or variable or to deassign the reg or variable. This shall be the same functionality as the procedural **assign** and **deassign** keywords in the Verilog HDL.

The routine **acc_set_value()** shall also be used to perform a procedural force of a value onto a reg, variable or net, or to release the reg, variable or net. This shall be the same functionality as the procedural **force** and **release** keywords in the Verilog HDL.

When an object is deassigned or released using **acc_set_value()**, the current value of the object shall be returned to the `s_setval_value` structure.

To assign, deassign, force, or release an object using **acc_set_value()**, the `s_setval_value` and `s_setval_delay` structures shall be allocated and the fields shall be set to the appropriate values. For the *model* field of the `s_setval_delay` structure, one of the predefined constants listed in Table 184 shall be used.

Table 184—Predefined assign/force constants for the model field of s_setval_delay

Integer constant	Description
accAssignFlag	Assigns a reg or variable to the indicated value with no delay; other events scheduled for the object are overridden. Same functionality as the Verilog HDL procedural **assign** keyword.
accDeassignFlag	Deassigns an assigned reg or variable; other events scheduled for the object are no longer overridden. Same functionality as the Verilog HDL procedural **deassign** keyword.
accForceFlag	Forces a value onto a reg, variable or net; other events scheduled for the object are overridden. Same functionality as the Verilog HDL procedural **force** keyword.
accReleaseFlag	Releases a forced reg, variable or net; other events scheduled for the object are no longer overridden, and nets immediately return to the current driven value. Same functionality as the Verilog HDL procedural **release** keyword.

Copyright © 2001 IEEE. All rights reserved.

The example shown in Figure 159 uses **acc_set_value()** to set and propagate a value onto a reg. This example assumes the application is linked to a user-defined system task (using the PLI interface mechanism) called `$my_set_value()`, which has the following usage for a four bit reg, `r1`:

```
$my_set_value(r1, "x011", 2.4);
```

```
PLI_INT32 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 159—Using acc_set_value()

23.97 acc_vcl_add()

acc_vcl_add()			
Synopsis:	Set a callback to a consumer routine with value change information whenever an object changes value.		
Syntax:	`acc_vcl_add(object_handle,consumer_routine,user_data, vcl_flag)`		
	Type	Description	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle to an object to be monitored (such as a reg or net)
	C routine pointer	consumer_routine	Unquoted name of the C routine to be called when the object changes value
	PLI_BYTE8 *	user_data	User-defined data that is passed back to the consumer routine when the object changes value
	PLI_INT32	vcl_flag	Predefined integer constant that selects the type of change information reported to the consumer routine
Related routines:	Use acc_vcl_delete() to remove a VCL callback monitor		

The ACC routine **acc_vcl_add()** shall set up a callback monitor on an object that shall call a user-defined consumer routine when the object changes value. The consumer routine shall be passed logic value information or logic value and strength information about the object.

 Copyright © 2001 IEEE. All rights reserved.

The **acc_vcl_add()** routine requires four arguments, as described in the following paragraphs.

The *object_handle* argument is a handle to the object to be monitored by an application. The VCL shall monitor value changes for the following objects:

Scalar regs and bit-selects of vector regs
Scalar nets, unexpanded vector nets, and bit-selects of expanded vector nets
Integer, real and time variables
Module ports
Primitive output or inout terminals
Named events

NOTE Adding a value change link to a module port is equivalent to adding a value change link to the loconn of the port. The vc_reason returned shall be based on the loconn of the port.

The object_handle passed to **acc_vcl_add()** is not returned when the consumer routine is called. However, the handle can be passed using the user_data argument.

The *consumer_routine* argument is a pointer to a C application. This application shall be called whenever the object changes value. When a value change callback occurs, the consumer_routine shall be passed the user_data argument and a pointer to a `vc_record` structure, which shall contain information about the change.

Refer to 22.10 for a full description of consumer routines and the `vc_record` structure.

The *user_data* argument is user-defined data, such as the object name, the object handle, the object value, or a pointer to a data structure. The value of the user_data argument shall be passed to the consumer routine each time a callback occurs. Note that the user_data argument is defined as character string pointer, and therefore any other type should be cast to a `PLI_BYTE8*`.

The *vcl_flag* argument shall set the type of information the callback mechanism shall report. There are two types of flags, as shown in Table 185.

Table 185—vcl_flag constants used in acc_vcl_add()

vcl_flag	What it does
vcl_verilog_logic	Indicates the VCL callback mechanism shall report information on logic value changes
vcl_verilog_strength	Indicates the VCL callback mechanism shall report information on logic value and strength changes

If an application calls **acc_vcl_add()** with the same arguments more than once, the VCL callback mechanism shall only call the consumer routine once when the object changes value. If any of the VCL arguments, including the user_data, are different, the VCL callback mechanism shall call the consumer routine multiple times, once for each unique **acc_vcl_add()**.

NOTE It is not recommended that multiple VCL flags be added with the same object, consumer and user_data. If multiple flags with the same values are added, then each call to acc_vcl_delete() with those values shall delete one flag; the order of deletion is indeterminate.

If multiple PLI applications monitor the same object at the same time, each application shall receive 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.

Copyright © 2001 IEEE. All rights reserved.

Refer to 22.10 for an example of using **acc_vcl_add()**.

23.98 acc_vcl_delete()

acc_vcl_delete()			
Synopsis:	Removes a VCL callback monitor.		
Syntax:	`acc_vcl_delete(object_handle, consumer_routine, user_data, vcl_flag)`		
	Type	Description	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	handle	object_handle	Handle to the object to be monitored specified in the call to acc_vcl_add()
	C routine pointer	consumer_routine	Unquoted name of the C routine specified in the call to acc_vcl_add()
	PLI_BYTE8 *	user_data	User-defined data specified in the call to acc_vcl_add()
	PLI_INT32	vcl_flag	Predefined integer constant; **vcl_verilog**
Related routines:	Use acc_vcl_add() to place a VCL callback monitor on an object		

The ACC routine **acc_vcl_delete()** shall remove a VCL callback monitor previously requested with a call to **acc_vcl_add()**. The **acc_vcl_delete()** routine requires four arguments, as described in the following paragraphs. When multiple PLI applications are monitoring the same object, **acc_vcl_delete()** shall stop monitoring the object only for the application associated with a specific **acc_vcl_add()** call.

The *object_handle* argument is a handle to the object for which the VCL callback monitor is to be removed. This has to be a handle to the same object that was used when **acc_vcl_add()** was called.

The *consumer_routine* argument is the unquoted name of the C application called by the VCL callback monitor. This has to be the same C application that was specified when **acc_vcl_add()** was called.

The *user_data* argument is user-defined data that is passed to the consumer routine each time the object changes value. This has to be the same value that was specified when **acc_vcl_add()** was called.

The *vcl_flag* argument is a predefined integer constant and has to be **vcl_verilog**. This constant shall be used in place of the vcl_flag values used with **acc_vcl_add()**.

Refer to 22.10 for an example of using **acc_vcl_delete()**.

 Copyright © 2001 IEEE. All rights reserved.

23.99 acc_version()

acc_version()			
Synopsis:	Get a pointer to a character string that indicates version number of the ACC routine software.		
Syntax:	`acc_version()`		
Returns:	**Type**	Description	
	PLI_BYTE8 *	Character string pointer	
Arguments:	**Type**	**Name**	**Description**
	None		
Related routines:	Use acc_product_version() to get the version of the software product in use Use acc_product_type() to get the type of software product in use		

The ACC routine **acc_version()** shall return a pointer to a character string that indicates the version of the ACC routines used in the software product that called the PLI application. The return value for this routine is placed in the ACC internal string buffer. See 22.9 for explanation of strings in ACC routines.

The character string shall be in the following format:

```
Access routines Version <version_number>
```

For example, if the software product is using the IEEE Std 1364 PLI version of ACC routines, **acc_version()** might return a pointer to the following string:

```
"Access routines Version IEEE 1364 PLI"
```

NOTE The string returned by **acc_version()** shall be defined by the software product vendor.

The example shown in Figure 160 uses **acc_version()** to identify the version of ACC routines linked to the application.

```
include "acc_user.h"

LI_INT32 show_versions()

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

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

  acc_close();
```

Figure 160—Using acc_version()

24. Using TF routines

This clause provides an overview of the types of operations that are done with the PLI task/function (TF) routines. Detailed descriptions of the routines are provided in the next section.

24.1 TF routine definition

The PLI TF routines, sometimes referred to as *utility routines*, provide a mechanism to manipulate the arguments of user-defined system tasks and functions and to synchronize interaction between a task and the simulator. Appropriate applications include stimulus generation, error checking, and interfaces to C models.

24.2 TF routine system task/function arguments

The number of arguments passed to a system task shall be returned by **tf_nump()**. A type for each argument shall be returned by **tf_typep()** and is primarily used to determine if an argument is writable.

An argument shall be considered *read-only* if, in the Verilog HDL source description, the argument cannot be used on the left-hand side of a procedural assignment statement. Signals declared as one of the net data types or the event data type, or bit-selects, part-selects, or concatenations of net data types, shall be read-only. A module instance name or a primitive instance name shall also be read-only.

Arguments shall be considered *writable* from the PLI if the arguments can be used on the left-hand side of procedural assignment in the Verilog HDL source description. Signals declared as reg, integer, time, or real shall be writable, as well as bit-selects, part-selects, and concatenations of these data types.

24.3 Reading and writing system task/function argument values

User-defined system task and function argument values can be determined and altered in a number of ways with the TF routines, depending on factors such as value type, data size, and desired format.

24.3.1 Reading and writing 2-state parameter argument values

To access the 2-state (logic 0 and 1) value of a system task/function argument of size less than or equal to 32 bits, the routine **tf_getp()** can be used. To set the 2-state value of an argument of size less than or equal to 32 bits, **tf_putp()** can be used. If the argument is 33—64 bits, **tf_getlongp()** and **tf_putlongp()** can be used. For arguments of type real, **tf_getrealp()** and **tf_putrealp()** can be used. Logic X and Z bits in the argument value shall be interpreted as 0.

24.3.2 Reading and writing 4-state values

If 4-states (logic 0, 1, X, and Z) are required and a string representation of the value is appropriate, **tf_strgetp()** can be used to access the value. The routines **tf_strdelputp()**, **tf_strlongdelputp()**, and **tf_strrealdelputp()** can be used to write 4-state values to writable arguments. For applications with a high frequency of PLI calls, the overhead of these string-based routines can be excessive. The following paragraph describes an alternative.

4-state values can also be accessed with the routine **tf_exprinfo()**. This routine shall create a persistent structure that contains the 4-state value of an argument encoded in an `s_vecval` structure. After **tf_exprinfo()** has been called once for an argument, the pointer to the `s_vecval` structure can be saved. The argument value can be changed using that structure along with routines **tf_propagatep()** to send the value in the structure into a simulation and **tf_evaluatep()** to update the value in the structure to the current simulation value.

 Copyright © 2001 IEEE. All rights reserved.

24.3.3 Reading and writing strength values

Strength values on scalar net arguments can be accessed with the routine **tf_nodeinfo()**.

24.3.4 Reading and writing to memories

Memory array values can be accessed with the routine **tf_nodeinfo()**. This routine returns a pointer to a `memval` structure that represents the array in the Verilog HDL software product. Setting a value in the `memval` structure shall make it available for the software tool access, but this does not automatically cause the value to be propagated to any right-hand-side memory references.

24.3.5 Reading and writing string values

The routine **tf_getcstringp()** shall return the string representation of a string constant or a vector argument. There is no direct method to write string values using TF routines, but it can be accomplished by writing 8-bit ASCII character values to 8-bit reg elements in a vector reg using the **tf_exprinfo()** value structure.

24.3.6 Writing return values of user-defined functions

2-state values can be set as the return value of a user-defined function using **tf_putp()**, **tf_putlongp()** and **tf_putrealp()** with an argument value of **0**. It is illegal to schedule the return value of a system function at a future simulation time. The routines **tf_strdelputp()**, **tf_strlongdelputp()**, and **tf_strrealdelputp()** cannot be used to return the value of a system function. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

24.3.7 Writing the correct C data types

It is important to ensure that the data type of the argument to any of the tf_put routines is consistent with the data type required by the routine and specified argument.

The following examples illustrate what cautions should be taken.

If the second argument of a system task/function instance is of type **tf_readwritereal**, meaning the argument is declared as a real variable in the Verilog HDL source description, the following tf_put routines shall produce valid results:

```
PLI_INT32 i = 5;
tf_putp(2, i); /* write an integer value to 2nd argument */
```

This example sets the second task/function argument to 5.0 assigning an integer value to a real variable is legal in the Verilog HDL.

```
double d = 5.7;
tf_putrealp(2, d); /* write a real value to 2nd argument */
```

This example sets the second task/function argument to 5.7.

The following routines, however, shall produce invalid results for the following reasons:

```
PLI_INT32 i = 5;
tf_putrealp(2, i);  /* invalid result */
```

The statement `PLI_INT32 i = 5` passes a 32-bit integer to **tf_putrealp()**, which expects a 64-bit double value type. Since there is no data type checking, **tf_putrealp()** shall read 32 bits of undefined data and try to use it as if it were valid data. The result is unpredictable.

Copyright © 2001 IEEE. All rights reserved.

```
float f = 5;
tf_putrealp(2, f);  /* invalid result */
```

The float statement passes a 32-bit float to **tf_putrealp()**, which is expecting a 64-bit double value type. The result is unpredictable.

```
double d = 5.7;
tf_putp(2, d);  /* invalid result */
```

The **tf_putp()** routine shall take only the lower 32 bits of the 64-bit double passed to it by the statement `double d = 5.7`.

24.4 Value change detection

Value changes on system task/function arguments can be detected by enabling asynchronous callbacks with **tf_asynchon()**. The callbacks can be disabled with **tf_asynchoff()**. When argument change callbacks are enabled with **tf_asynchon()**, whenever an argument changes value, the misctf application associated with the user-defined system task/function shall be called back with three integer arguments: *data*, *reason*, and *paramvc*. Argument *reason* shall be **reason_paramvc**. The value change can be examined immediately, or a second callback can be requested later in the same time step (as described in 24.6). By setting a second callback at the end of the time step, an application can process all argument value changes within in a time step at once. The routines **tf_copypvc_flag()**, **tf_movepvc_flag()**, **tf_testpvc_flag()**, and **tf_getpchange()** can be used to determine all the arguments that changed in a time step.

24.5 Simulation time

TF routines are provided to read simulation time and to scale delays to simulation time scales.

The routines **tf_gettime()** and **tf_getlongtime()** shall return the current simulation time in unsigned format. These times shall be scaled to the timescale of the module where the system task or function is invoked. The routine **tf_strgettime()** shall return unscaled simulation time in a string format.

PLI TF routines that involve time shall automatically scale delay values to the timescale of the module containing the instance of the user-defined task or function.

The routines **tf_gettimeunit()** and **tf_gettimeprecision()** can be used to obtain the timescale unit and precision of a module. These routines can also be used to obtain the internal simulation time unit, which is the smallest precision of all modules within a simulation. The routines **tf_scale_longdelay()**, **tf_scale_realdelay()**, **tf_unscale_longdelay()**, and **tf_unscale_realdelay()** can be used to convert between scaled delays and internal simulation time.

24.6 Simulation synchronization

There are TF routines that allow synchronized calling of the misctf application associated with a user-defined system task or function. The misctf application can be called at the end of the current time step or at some future time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** shall cause the misctf application associated with a user-defined system task to be called back in the current simulation time step.

The **tf_synchronize()** routine shall place a callback at the end of the inactive event queue for the current time step. The misctf application shall be called with **reason_synch**. It is possible for subsequent events to be added to the current time step after the **tf_synchronize()** callback (for this reason, when the callback occurs, the next scheduled time step cannot be determined). The misctf application can propagate new values in **reason_synch** mode.

 Copyright © 2001 IEEE. All rights reserved.

The **tf_rosynchronize()** callback shall occur after all active, inactive, and nonblocking assign events for a time step have been processed. The misctf application shall be called with **reason_rosynch**. With **reason_rosynch**, it is possible to determine the time of the next scheduled time step using **tf_getnextlongtime()**. Values cannot be written to system task/function arguments during a **reason_rosynch** callback (the *'ro'* indicates read-only). Placing a callback for tf_rosynchronize() during a callback for reason reason_rosynch will result in another reason_rosynch callback occurring during the same time slice.

The routine **tf_setdelay()** and its variations shall schedule the misctf application to be called back at a specified time with reason argument **reason_reactivate**. The routine **tf_clearalldelays()** shall remove any previously scheduled callbacks of this type.

24.7 Instances of user-defined tasks or functions

The routine **tf_getinstance()** shall return a unique identifier for each instance of a user-defined system task or function in the Verilog HDL source description. This value can then be used as the *instance_p* argument to all the *tf_i** routines so that the arguments of one instance can be manipulated from another task or function instance.

24.8 Module and scope instance names

The full hierarchical path name of the module that contains an instance shall be returned by the routine **tf_mipname()**. The full name of the containing scope, which can be a Verilog HDL task or function, a named block, or a module instance, shall be returned by **tf_spname()**.

24.9 Saving information from one system TF call to the next

The TF routines **tf_setworkarea()** and **tf_getworkarea()** provide a special storage *work area* that can be used for:

- Saving data during one call to a PLI application that can be retrieved in a subsequent call to the application.
- Passing data from one type of PLI application to another, such as from a checktf application to a calltf application.

24.10 Displaying output messages

The routine **io_printf()** can be used in place of the C **printf()** statement. This routine has essentially the same syntax and semantics as **printf()**, but it displays the output message to both the output channel of the software product which invoked the PLI application and to the log file of the software product.

The routine **io_mcdprintf()** is also similar to the C **printf()**, but permits writing information to files that were opened within the Verilog HDL source description using the **$fopen()** built-in system function.

The routines **tf_warning()**, **tf_error()**, **tf_message()**, and **tf_text()** can be used to display warning and error messages that are automatically formatted to a similar format as the warning and error messages for the software product. The routines **tf_error()** and **tf_message()** shall also provide control for aborting the software product execution when an error is detected.

24.11 Stopping and finishing

The routines **tf_dostop()** and **tf_dofinish()** are the PLI equivalents to the built-in system tasks **$stop** and **$finish**.

Copyright © 2001 IEEE. All rights reserved.

25. TF routine definitions

This clause defines the PLI TF routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Clause 23 for conventions that are used in the definitions of the PLI routines.

25.1 io_mcdprintf()

io_mcdprintf()			
Synopsis:	Write a formatted message to one or more files.		
Syntax:	`io_mcdprintf(mcd, format, arg1,...arg12)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_INT32	mcd	An integer multi-channel descriptor value representing one or more open files
	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
(optional)		arg1...arg12	1 to 12 optional arguments of the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use io_printf() to write messages to the output channel of the software product which invoked the PLI application and to the Verilog product log file		

The TF routine **io_mcdprintf()** shall write a formatted message to one or more open files, as described by the multi-channel descriptor mcd. This routine uses the descriptors created by the **$fopen** system task or the VPI routine **vpi_mcd_open()**. See 17.2.1 for the functional description of $fopen, and 27.25 for the description of vpi_mcd_open().

The format strings shall use the same format as the C routine fprintf().

The maximum number of arguments that can be used in the format control string is 12.

 Copyright © 2001 IEEE. All rights reserved.

25.2 io_printf()

io_printf()			
Synopsis:	Print a formatted message to the output channel of the software product which invoked the PLI application and to the log file of the product.		
Syntax:	`io_printf(format, arg1,...arg12)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
(optional)		arg1...arg12	1 to 12 optional arguments of the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use io_mcdprintf() to write a formatted message to one or more open files Use tf_message(), tf_error(), or tf_warning() to write error or warning messages		

The TF routine **io_printf()** shall write a formatted message as text output. The functionality is similar to the C printf() function. However, **io_printf()** differs from printf() because it ensures the message is written to both the output channel of the software product which invoked the PLI application and the output log file of the product.

The *format* control string uses the same formatting controls as the C printf() function (for example, %d).

The maximum number of arguments that can be used in the format control string is 12.

25.3 mc_scan_plusargs()

mc_scan_plusargs()			
Synopsis:	Scan software product invocation command line for plus (+) options.		
Syntax:	`mc_scan_plusargs(startarg)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a string with the result of the search	
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	startarg	A quoted string or pointer to a character string with the first part of the invocation option to search for

The TF routine **mc_scan_plusargs()** shall scan all software product invocation command options and match a given string to a plus argument. The match is case sensitive.

Copyright © 2001 IEEE. All rights reserved.

The routine **mc_scan_plusargs()** shall

Return `null` if *startarg* is not found

Return the remaining part of the command argument if *startarg* is found (e.g., if the invocation option string is "`+siz64`", and *startarg* is "`siz`", then "`64`" is returned)

Return a pointer to a C string with a `null` terminator if there is no remaining part of a found plus argument

25.4 tf_add_long()

tf_add_long()			
Synopsis:	Add two 64-bit integers.		
Syntax:	`tf_add_long(aof_low1, aof_high1, low2, high2)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	high2	Most significant 32 bits of second operand
Related routines:	Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_add_long()** shall add two 64-bit values. After calling **tf_add_long()**, the variables used to pass the first operand shall contain the results of the addition. Figure 161 shows the high and low 32 bits of two 64-bit integers and how **tf_add_long()** shall add them.

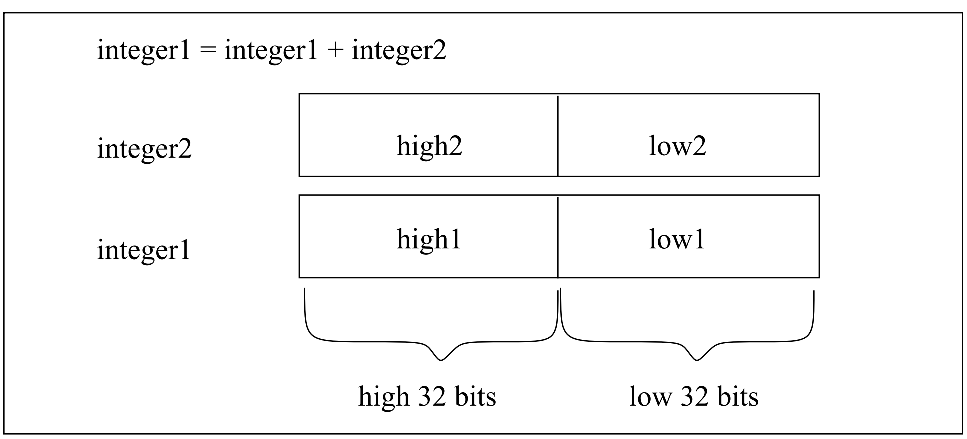


Figure 161—Adding with tf_add_long()

 Copyright © 2001 IEEE. All rights reserved.

25.5 tf_asynchoff(), tf_iasynchoff()

tf_asynchoff(), tf_iasynchoff()			
Synopsis:	Disable asynchronous calling of the misctf application.		
Syntax:	`tf_asynchoff()` `tf_iasynchoff(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable asynchronous calling of the misctf application Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_asynchoff()** and **tf_iasynchoff()** shall disable further calling of the misctf application for **reason_paramvc** for the current instance or a specific instance of a user-defined system task or function.

Asynchronous calling is first enabled by the routines **tf_asynchon()** or **tf_iasynchon()**.

25.6 tf_asynchon(), tf_iasynchon()

tf_asynchon(), tf_iasynchon()			
Synopsis:	Enable asynchronous calling of the misctf application for system task/function argument value changes.		
Syntax:	`tf_asynchon()` `tf_iasynchon(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchoff() or tf_iasynchoff() to disable asynchronous calling of the misctf application Use tf_getpchange() or tf_igetpchange() to get the index number of the argument that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy pvc flags Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_asynchon()** and **tf_iasynchon()** shall enable a misctf user application to be called asynchronously whenever a system task/function argument value changes in the current instance or in a specific instance of a user-defined system task or function. After enabling, the routine specified by misctf in the PLI interface mechanism shall be called with a reason of **reason_paramvc** each time any task/function argument changes value or strength. The index number of the argument that changed is passed to the misctf application as a third C argument, **paramvc**.

Copyright © 2001 IEEE. All rights reserved.

The value change can be examined immediately, or a second callback can be requested later in the same time step (as described in Section 24.6). By setting a second callback at the end of the time step, an application can process all argument value changes within a time step at once. The routines **tf_copypvc_flag()**, **tf_movepvc_flag()**, **tf_testpvc_flag()**, and **tf_getpchange()** can be used to determine all the arguments that changed in a time step.

Task/function argument index numbering shall proceed from left to right, and the left-most argument shall be number 1.

25.7 tf_clearalldelays(), tf_iclearalldelays()

tf_clearalldelays(), tf_iclearalldelays()			
Synopsis:	Clear all scheduled reactivations by tf_setdelay() or tf_isetdelay().		
Syntax:	`tf_clearalldelays()` `tf_iclearalldelays(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 1	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() to schedule a reactivation Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_clearalldelays()** and **tf_iclearalldelays()** shall clear all reactivation delays, which shall remove the effect of all previous **tf_setdelay()** or **tf_isetdelay()** calls for the current instance or specific instance of a user-defined system task or function.

25.8 tf_compare_long()

tf_compare_long()			
Synopsis:	Compare two 64-bit integer values.		
Syntax:	`tf_compare_long(low1, high1, low2, high2)`		
	Type	**Description**	
Returns:	PLI_INT32	An integer flag indicating the result of the comparison	
	Type	**Name**	**Description**
Arguments:	PLI_UINT32	low1	Least significant 32 bits of first operand
	PLI_UINT32	high1	Most significant 32 bits of first operand
	PLI_UINT32	low2	Least significant 32 bits of second operand
	PLI_UINT32	high2	Most significant 32 bits of second operand

 Copyright © 2001 IEEE. All rights reserved.

tf_compare_long()	
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers

The TF routine **tf_compare_long()** shall compare two 64-bit integers and return one of the values given in Table 186.

Table 186—Return values for tf_compare_long()

When	tf_compare_long() shall return
operand1 < operand2	**-1**
operand1 = operand2	**0**
operand1 > operand 2	**1**

25.9 tf_copypvc_flag(), tf_icopypvc_flag()

tf_copypvc_flag(), tf_icopypvc_flag()			
Synopsis:	Copy system task/function argument value change flags.		
Syntax:	`tf_copypvc_flag(narg)` `tf_icopypvc_flag(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	The value of the pvc flag	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument, or **-1**
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the argument that changed Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_copypvc_flag()** and **tf_icopypvc_flag()** shall copy the current pvc flag to the saved pvc flag and return the value of the flag that was copied. The argument *narg* is the index number of an argument in the current instance or a specific instance of a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument pvc flags shall be copied and the logical OR of all saved flags returned.

Copyright © 2001 IEEE. All rights reserved.

Argument Value Change (pvc) flags shall be used to indicate whether a particular user-defined system task or function argument has changed value. Each argument shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

25.10 tf_divide_long()

tf_divide_long()			
Synopsis:	Divide two 64-bit integers.		
Syntax:	`tf_divide_long(aof_low1, aof_high1, low2, high2)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_divide_long()** shall divide two 64-bit values. After calling **tf_divide_long()**, the variables used to pass the first operand shall contain the result of the division.

The operands shall be assumed to be in two s complement form. Figure 162 shows the high and low 32 bits of two 64-bit integers and how **tf_divide_long()** shall divide them.

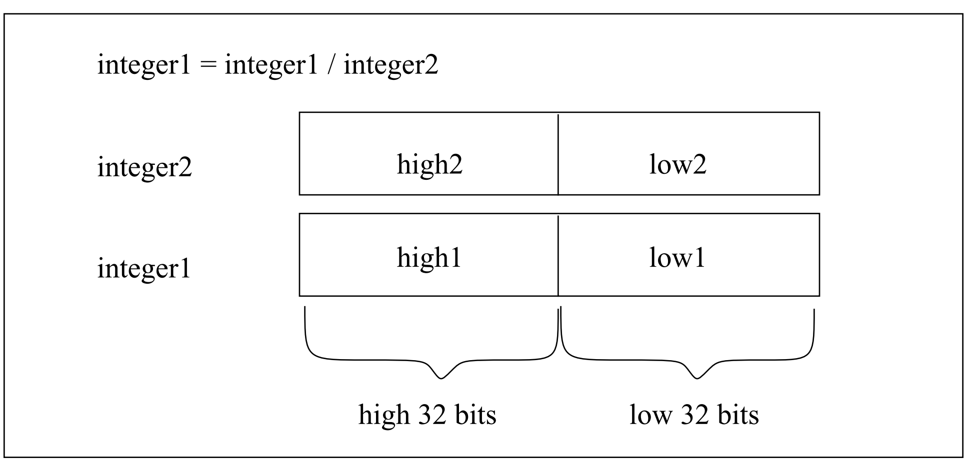


Figure 162—Dividing with tf_divide_long()

 Copyright © 2001 IEEE. All rights reserved.

25.11 tf_dofinish()

tf_dofinish()			
Synopsis:	Exit software product execution.		
Syntax:	`tf_dofinish()`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	None		
Related routines:	Use tf_dostop() to cause a product to enter interactive mode		

The TF routine **tf_dofinish()** shall finish the software product execution the same as if a **$finish()** built-in system task had been executed in the Verilog HDL source description.

25.12 tf_dostop()

tf_dostop()			
Synopsis:	Cause software product to enter interactive mode.		
Syntax:	`tf_dostop()`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	None		
Related routines:	Use tf_dofinish() exit software product execution		

The TF routine **tf_dostop()** shall cause a software product to enter into its interactive mode as if a **$stop()** built-in system task had been executed in the Verilog HDL source description.

Copyright © 2001 IEEE. All rights reserved.

25.13 tf_error()

tf_error()			
Synopsis:	Report an error message.		
Syntax:	`tf_error(format, arg1,...arg5)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
(optional)		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use tf_message() to write error messages with additional format control Use tf_warning() to write a warning message Use io_printf() or io_mcdprintf() to write a formatted message		

The TF routine **tf_error()** shall provide an error reporting mechanism compatible with error messages generated by the software product.

The *format* control string uses the same formatting controls as the C `printf()` function (for example, %d).

The maximum number of arguments that can be used in the format control string is five.

The location information (file name and line number) of the current instance of the user-defined system task or function is appended to the message using a format compatible with error messages generated by the software product.

The *message* is written to both the output channel of the software product which invoked the PLI application and the output log file of the product.

If **tf_error()** is called by the checktf application associated with the user-defined system task or function, the following rules shall apply:

If the checktf application is called when the Verilog HDL source code was being parsed or compiled, parsing or compilation shall be aborted after the error is reported.

If the checktf application is called when the user-defined task or function was invoked on the interactive command line, the interactive command shall be aborted.

 Copyright © 2001 IEEE. All rights reserved.

25.14 tf_evaluatep(), tf_ievaluatep()

tf_evaluatep(), tf_ievaluatep()			
Synopsis:	Evaluate a system task/function argument expression.		
Syntax:	`tf_evaluatep(narg)` `tf_ievaluatep(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() to get a pointer to the s_tfexprinfo structure Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_evaluatep()** and **tf_ievaluatep()** shall evaluate the current value of the specified argument in the current instance or a specific instance of a user-defined system task or function. The current value shall be returned to the value cell in the `tf_exprinfo` structure returned from a previous call to the routine **tf_exprinfo()** or **tf_iexprinfo()**. This can be a more efficient way to obtain the current value of an expression than to call **tf_exprinfo()** or **tf_iexprinfo()** repeatedly.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.15 tf_exprinfo(), tf_iexprinfo()

tf_exprinfo(), tf_iexprinfo()			
Synopsis:	Get system task/function argument expression information.		
Syntax:	`tf_exprinfo(narg, exprinfo_p)` `tf_iexprinfo(narg, exprinfo_p, instance_p)`		
	Type	**Description**	
Returns:	struct t_tfexprinfo *	Pointer to a structure containing the value of the second argument if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	struct t_tfexprinfo *	exprinfo_p	Pointer to a variable declared as a `t_tfexprinfo` structure type
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_nodeinfo() or tf_inodeinfo() for additional information on writable arguments Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_exprinfo()** and **tf_iexprinfo()** shall return a pointer to a structure containing general information about the specified argument in the current instance or a specific instance of a user-defined system task or function. The information shall be stored in the C structure `s_tfexprinfo`.

Memory space shall first be allocated to hold the information before calling **tf_exprinfo()** or **tf_iexprinfo()**. For example:

```
   {
   s_tfexprinfo info;       /* declare a variable of the structure type
*/
   tf_exprinfo(n, &info);   /* pass tf_exprinfo a pointer to the variable
*/
   ...
   }
```

This routine shall return the second argument, which is the pointer to the information structure. If *narg* is out of range, or if some other error is found, then **0** shall be returned. The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The `s_tfexprinfo` structure is defined in `veriuser.h` and is listed in Figure 163.

```
typedef struct t_tfexprinfo
{
  PLI_INT16 expr_type;
  PLI_INT16 padding;
  struct t_vecval *expr_value_p;
  double    real_value;
  PLI_BYTE8 *expr_string;
  PLI_INT32 expr_ngroups;
  PLI_INT32 expr_vec_size;
  PLI_INT32 expr_sign;
  PLI_INT32 expr_lhs_select;
  PLI_INT32 expr_rhs_select;
} s_tfexprinfo, *p_tfexprinfo;
```

Figure 163—The s_tfexprinfo structure definition

The *expr_type* of the `s_tfexprinfo` structure shall indicate the Verilog HDL data type of the argument, and it shall be one of the predefined constants as given in Table 187 and defined in `veriuser.h`.

Table 187—Predefined constants used with tf_exprinfo()

Predefined constant	Description
tf_nullparam	For null or non-existent arguments
tf_string	For string arguments
tf_readonly	For net, net bit, net part select and constant integer arguments
tf_readonlyreal	For constant real number arguments
tf_readwrite	For reg, integer and time variable arguments

 Copyright © 2001 IEEE. All rights reserved.

Table 187—Predefined constants used with tf_exprinfo() (continued)

Predefined constant	Description
tf_readwritereal	For real variable arguments
tf_rwbitselect	For bit-select of reg, integer and time variable arguments
tf_rwpartselect	For part-select of reg, integer and time variable arguments
tf_rwmemselect	For memory word arguments

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbitselect**, **tf_rwpartselect**, or **tf_rwmemselect**, the *expr_value_p* of the `s_tfexprinfo` structure shall be a pointer to an array of `s_vecval` structures that shall contain the resultant value of the expression. The `s_vecval` structure for representing vector values is defined in `veriuser.h` and is listed in Figure 164.

```
typedef struct t_vecval
{
  PLI_INT32 avalbits;
  PLI_INT32 bvalbits;
} s_vecval, *p_vecval;
```

Figure 164—The s_vecval structure definition

If the number of bits in the vector (defined by the *expr_vec_size* field of the `s_tfexprinfo` structure) is less than or equal to 32, then there shall only be one `s_vecval` group in the `expr_value_p` array. For 33 bits to 64 bits, there shall be two groups in the array, and so on. The number of groups shall also be given by the value of the *expr_ngroups* field of the `s_tfexprinfo` structure. The components *avalbits* and *bvalbits* of the `s_vecval` structure shall hold the bit patterns making up the value of the argument. The lsb in the value shall be represented by the lsb s in the *avalbits* and *bvalbits* components, and so on. The bit coding shall be as given in Table 188.

Table 188—avalbits/bvalbits encoding

aval / bval	Logic value
00	0
10	1
01	High impedance
11	Unknown

If the expression type is **tf_readonlyreal** or **tf_readwritereal**, the *real_value* field of the `s_tfexprinfo` structure shall contain the value.

If the expression is of type **tf_string**, the *expr_string* field of the `s_tfexprinfo` structure shall point to the string.

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbitselect**, **tf_rwpartselect**, or **tf_rwmemselect**, the *expr_ngroups* of the `s_tfexprinfo` structure shall indicate the number of groups for the argument expression value and determine the array size of the *expr_value_p* value structure pointer. If the expression type is **tf_readonlyreal** or **tf_readwritereal**, *expr_ngroups* shall be 0.

Copyright © 2001 IEEE. All rights reserved.

If the expression type is **tf_readonly**, **tf_readwrite**, **tf_rwbitselect**, **tf_rwpartselect**, or **tf_rwmemselect**, the *expr_vec_size* field of the `s_tfexprinfo` structure shall indicate the total number of bits in the array of `expr_value_p` value structures. If the expression type is **tf_readonlyreal** or **tf_readwritereal**, *expr_vec_size* shall be 0.

The *expr_sign* field of the `s_tfexprinfo` structure shall indicate the sign type of the expression. It shall be 0 for unsigned or nonzero for signed.

The expr_lhs_select and expr_rhs_select fields shall contain the select information about the object if it is a reg bit-select, net bit-select, part-select, variable array word-select, or memory word-select.

25.16 tf_getcstringp(), tf_igetcstringp()

tf_getcstringp(), tf_igetcstringp()			
Synopsis:	Get system task/function argument value as a string.		
Syntax:	`tf_getcstringp(narg)` `tf_igetcstringp(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_strgetp() or tf_istrgetp() to get an argument value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getcstringp()** and **tf_igetcstringp()** shall return a character string representing the value of the specified argument in the current instance or a specific instance of a user-defined system task or function. If the argument identified by *narg* is a literal string, reg, integer variable, time variable, or an expression, then **tf_getcstringp()** or **tf_igetcstringp()** shall convert its value to a C language ASCII string by

a) Eliminating leading zeros

b) Converting each group of 8 bits to an ASCII character

c) Adding a \0 string termination character to the end

If the argument identified by *narg* is `null` or if *narg* is out of range, then a `null` shall be returned. If the argument identified by *narg* is a real variable or an expression that evaluates to a real value, then **tf_getcstringp()** and **tf_igetcstringp()** shall return `NULL`.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

 Copyright © 2001 IEEE. All rights reserved.

25.17 tf_getinstance()

tf_getinstance()			
Synopsis:	Get a pointer to the current instance of a user-defined system task or function.		
Syntax:	`tf_getinstance()`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a system task or function instance	
	Type	**Name**	**Description**
Arguments:	None		

The TF routine **tf_getinstance()** shall return a pointer that identifies the current instance of the user-defined task or function in the Verilog HDL source code. The pointer returned by **tf_getinstance()** can be used later in other TF routine calls to refer to this instance of the task or function. Many of the TF routines are in two forms. One deals with the current task or function instance. The other deals with some other instance of the task or function, where the instance pointer for the other instance was previously obtained using **tf_getinstance()** during a call to a user routine initiated by that instance.

25.18 tf_getlongp(), tf_igetlongp()

tf_getlongp(), tf_igetlongp()			
Synopsis:	Get system task/function argument value as a 64-bit integer.		
Syntax:	`tf_getlongp(aof_highvalue, narg)` `tf_igetlongp(aof_highvalue, narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Least significant (right-most) 32 bits of the argument value	
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_highvalue	Pointer to most significant (left-most) 32 bits of the argument value
	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get an argument value as a 32-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_strgetp() or tf_istrgetp() to get an argument value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getlongp()** and **tf_igetlongp()** shall return a 64-bit integer value for the argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If *narg* is out of range or the argument is `null`, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

Copyright © 2001 IEEE. All rights reserved.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.19 tf_getlongtime(), tf_igetlongtime()

tf_getlongtime(), tf_igetlongtime()			
Synopsis:	Get current simulation time as a 64-bit integer.		
Syntax:	`tf_getlongtime(aof_hightime)` `tf_igetlongtime(aof_hightime, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Least significant (right-most) 32 bits of simulation time	
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_hightime	Pointer to most significant (left-most) 32 bits of simulation time
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_gettime() to get the simulation time as a 32-bit integer Use tf_strgettime() to get the simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getlongtime()** and **tf_igetlongtime()** shall return the simulation time as a 64-bit integer. The high 32 bits of simulation time shall be assigned to the *aof_hightime* argument, and the low 32 bits of time shall be returned.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of the user-defined system task or function.

25.20 tf_getnextlongtime()

tf_getnextlongtime()			
Synopsis:	Get next time at which a simulation event is scheduled.		
Syntax:	`tf_getnextlongtime(aof_lowtime, aof_hightime)`		
	Type	**Description**	
Returns:	PLI_INT32	Integer value representing the meaning of the next event time obtained	
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_lowtime	Pointer to least significant (right-most) 32 bits of simulation time
	PLI_INT32 *	aof_hightime	Pointer to most significant (left-most) 32 bits of simulation time

 Copyright © 2001 IEEE. All rights reserved.

The TF routine **tf_getnextlongtime()** shall assign the 64-bit time of the next simulation event to *aof_lowtime* and *aof_hightime*, and it shall return an integer value that indicates the meaning of the time assigned. The time shall be expressed in the timescale units of the module containing the current user-defined system task or function instance.

The **tf_getnextlongtime()** routine shall only return the time for the next simulation event when it is called in a *read-only synchronize mode*. A read-only synchronize mode occurs when the misctf user application has been called with **reason_rosynch**. If **tf_getnextlongtime()** is not called in read-only synchronize mode, then the current simulation time shall be assigned.

Table 189 summarizes the functions of **tf_getnextlongtime()**.

Table 189—Return values for tf_getnextlongtime()

When	tf_getnextlongtime() shall return	tf_getnextlongtime() shall assign to aof_lowtime and aof_hightime
tf_getnextlongtime() was called from a misctf application that was called with **reason_rosynch**	**0**	The next simulation time for which an event is scheduled
There are no more future events scheduled	**1**	0
tf_getnextlongtime() was not called from a misctf application that was called with **reason_rosynch**	**2**	The current simulation time

NOTE Case 2 shall take precedence over case 1.

25.21 tf_getp(), tf_igetp()

tf_getp(), tf_igetp()			
Synopsis:	Get a system task/function argument value as an integer or character string pointer.		
Syntax:	`tf_getp(narg)` `tf_igetp(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Integer value of an argument or character string pointer of argument string value	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getlongp() or tf_igetlongp() to Get an argument value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_strgetp() or tf_istrgetp() to get an argument value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_getp()** and **tf_igetp()** shall return a value of the argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If the value of the argument is an integer or a real number, the routines shall return an integer value. If the argument is a literal string, then the routines shall return a pointer to a C type string (a string terminated by a \0 character). If *narg* is out of range or the argument is `null`, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The routines **tf_getp()** and **tf_getrealp()** differ in the value returned, as shown by the following example.

If the fourth argument in the user-defined system task or function has a value of 9.6 (a real value), then

```
PLI_INT32 ivalue = tf_getp(4)
```

would set `ivalue` to 10, whereas

```
double dvalue = tf_getrealp(4)
```

would set `dvalue` to 9.6.

In the first example, note that the PLI_INT32 conversion rounds off the value of 9.6 to 10 (rather than truncating it to 9). In the second example, note that the real value has to be declared as a double (not as a float). Rounding is performed following the Verilog HDL rules.

25.22 tf_getpchange(), tf_igetpchange()

tf_getpchange(), tf_igetpchange()			
Synopsis:	Get the index number of the next system task/function argument that changed value.		
Syntax:	`tf_getpchange(narg)` `tf_igetpchange(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Index number of the argument that changed	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_imovepvc_flag(-1) to save pvc flags before calling tf_getpchange() Use tf_copypvc_flag() or tf_icopypvc_flag() to copy pvc flags Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getpchange()** and **tf_igetpchange()** shall return the number of the next argument with a number greater than *narg* that changed value for the current instance or for a specific instance of a user-defined system task or function. The *narg* argument shall be 0 the first time this routine is called within a

 Copyright © 2001 IEEE. All rights reserved.

given user routine invocation. The routines shall return the argument number if there is a change in an argument with a number greater than *narg*, and they shall return 0 if there are no changes in arguments greater than *narg* or if an error is detected. The routine shall use the saved pvc flags, so it is necessary to execute **tf_movepvc_flag(-1)** prior to calling the routine.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

PVC flags shall indicate whether a particular user-defined system task or function argument has changed value. Each argument shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

25.23 tf_getrealp(), tf_igetrealp()

tf_getrealp(), tf_igetrealp()			
Synopsis:	Get a system task/function argument value as a double-precision value.		
Syntax:	`tf_getrealp(narg)` `tf_igetrealp(narg, instance_p)`		
	Type	**Description**	
Returns:	double	Double-precision value of an argument	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_strgetp() or tf_istrgetp() to get an argument value as a formatted string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getrealp()** and **tf_igetrealp()** shall return a double-precision value of the argument specified by *narg* in the current instance or a specific instance of a user-defined system task or function. If *narg* is out of range or the argument is null, then 0 shall be returned. Logic X and Z bits in the argument value shall be interpreted as 0.

The routines **tf_getrealp()** and **tf_igetrealp()** shall return 0.0 if the value being read is a literal string. Therefore, before calling these routines, **tf_typep()** or **tf_itypep()** should be called to check the type of the argument.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

Copyright © 2001 IEEE. All rights reserved.

25.24 tf_getrealtime(), tf_igetrealtime()

tf_getrealtime(), tf_igetrealtime()			
Synopsis:	Get the current simulation time in double-precision format.		
Syntax:	`tf_getrealtime()` `tf_igetrealtime(instance_p)`		
	Type	**Description**	
Returns:	double	Current simulation time	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_gettime() to get the lower 32-bits of simulation time as an integer Use tf_gettime() to get the full 64-bits of simulation time as an integer Use tf_strgettime() to get simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getrealtime()** and **tf_igetrealtime()** shall return the simulation time as a real number in double-precision format.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of a user-defined system task or function.

25.25 tf_gettime(), tf_igettime()

tf_gettime(), tf_igettime()			
Synopsis:	Get the current simulation time as a 32-bit integer.		
Syntax:	`tf_gettime()` `tf_igettime(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Least significant 32 bits of simulation time	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getlongtime() to get the full 64 bits of simulation time Use tf_getrealtime() to get the simulation time as a double-precision real number Use tf_strgettime() to get simulation time as a character string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettime()** and **tf_igettime()** shall return the lower 32 bits of simulation time as an integer.

Time shall be expressed in the timescale unit of the module containing the current instance or a specific instance of a user-defined system task or function.

 Copyright © 2001 IEEE. All rights reserved.

25.26 tf_gettimeprecision(), tf_igettimeprecision()

tf_gettimeprecision(), tf_igettimeprecision()			
Synopsis:	Get the timescale precision of a module or a simulation.		
Syntax:	`tf_gettimeprecision()` `tf_igettimeprecision(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	An integer value that represents a time precision	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function or *null* to represent the simulation
Related routines:	Use tf_gettimeunit() or tf_igettimeunit() to get the timescale time units Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettimeprecision()** and **tf_igettimeprecision()** shall return the timescale precision for the module that contains the current instance or a specific instance of a user-defined system task or function. The time precision is set by the `` `timescale `` Verilog HDL compiler directive in effect when the module was compiled. The routines shall return an integer code representing the time precision, as shown in Table 190.

Table 190—Code returned by tf_gettimeprecision() and tf_igettimeprecision()

Integer code returned	Simulation time precision
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 s
-5	10 s
-6	1 s
-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

When **tf_igettimeprecision()** is called with a `null` instance pointer, the routine shall return the simulation time unit, which is the smallest time precision used by all modules in a design.

Copyright © 2001 IEEE. All rights reserved.

25.27 tf_gettimeunit(), tf_igettimeunit()

tf_gettimeunit(), tf_igettimeunit()			
Synopsis:	Get the timescale unit of a module or a simulation.		
Syntax:	`tf_gettimeunit()` `tf_igettimeunit(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	An integer value that represents a time unit	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function or *null* to represent the simulation
Related routines:	Use tf_gettimeprecision() or tf_igettimeprecision() to get the timescale time precision Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_gettimeunit()** and **tf_igettimeunit()** shall return the timescale time units for the module that contains the current instance or a specific instance of a user-defined system task or function. The time unit for a module is set by the `` `timescale `` Verilog HDL compiler directive in effect when the module was compiled. The routines shall return an integer code representing the time unit, as shown in Table 191.

Table 191—Code returned by tf_gettimeunit() and tf_igettimeunit()

Integer code returned	Simulation time unit
2	100 s
1	10 s
0	1 s
-1	100 ms
-2	10 ms
-3	1 ms
-4	100 s
-5	10 s
-6	1 s
-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

When **tf_igettimeunit()** is called with a `null` instance pointer, the routines shall return the simulation time unit, which is the smallest time precision used by all modules in a design.

 Copyright © 2001 IEEE. All rights reserved.

25.28 tf_getworkarea(), tf_igetworkarea()

tf_getworkarea(), tf_igetworkarea()			
Synopsis:	Get work area pointer.		
Syntax:	`tf_getworkarea()` `tf_igetworkarea(instance_p)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a work area shared by all routines for a specific task/function instance	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setworkarea() or tf_isetworkarea() to put a value into the work area pointer Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_getworkarea()** and **tf_igetworkarea()** shall return the work area pointer value of the current instance or a specific instance of a user-defined system task or function. The value of the work area pointer shall be placed there by a previous call to the routine **tf_setworkarea()** or **tf_isetworkarea()**. These routines can be used as a means for two user applications to share information. For example, a checktf user application might open a file and then place the file pointer into the workarea using **tf_setworkarea()**. Later, the calltf user application can retrieve the file pointer using **tf_getworkarea()**.

25.29 tf_long_to_real()

tf_long_to_real()			
Synopsis:	Convert a 64-bit integer to a real number.		
Syntax:	`tf_long_to_real(low, high, aof_real)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_INT32	low	Least significant (right-most) 32 bits of a 64-bit integer
	PLI_INT32	high	Most significant (left-most) 32 bits of a 64-bit integer
	double *	aof_real	Pointer to a double-precision variable
Related routines:	Use tf_real_to_long() to convert a real number to a 64-bit integer Use tf_longtime_tostr() to convert a 64-bit integer to a character string		

The TF routine **tf_long_to_real()** shall convert a 64-bit integer to a real (double-precision floating-point) number. The variable pointed to by *aof_real* shall contain the converted number upon return from this routine.

Copyright © 2001 IEEE. All rights reserved.

25.30 tf_longtime_tostr()

tf_longtime_tostr()			
Synopsis:	Convert 64-bit integer time value to a character string.		
Syntax:	`tf_longtime_tostr(lowtime, hightime)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string representing the simulation time value	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	lowtime	Least significant (right-most) 32 bits of simulation time
	PLI_INT32	hightime	Most significant (left-most) 32 bits of simulation time
Related routines:	Use tf_getlongtime() to get the current simulation time as a 64-bit integer		

The TF routine **tf_longtime_tostr()** shall convert a 64-bit integer time value to a character string. The time value shall be unsigned.

25.31 tf_message()

tf_message()			
Synopsis:	Report an error or warning message with software product interruption control.		
Syntax:	`tf_message(level, facility, code, message, arg1,...arg5)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	level	A predefined constant indicating the severity level of the error
	quoted string or PLI_BYTE8 *	facility	A quoted character string or pointer to a character string used in the output message
	quoted string or PLI_BYTE8 *	code	A quoted character string or pointer to a character string used in the output message
	quoted string or PLI_BYTE8 *	message	A quoted character string or pointer to a character string that controls the message to be written
(optional)		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the message string
Related routines:	Use tf_text() to store error information prior to calling tf_message Use tf_error() to report error messages Use tf_warning() to report warning messages		

 Copyright © 2001 IEEE. All rights reserved.

The TF routine **tf_message()** shall display warning or error message information using the warning and error message format for a software product. The location information (file name and line number) of the current instance of the user-defined system task or function shall be appended to the message using a format compatible with warning and error messages generated by the software product, and the message shall be written to both the output channel of the software product which invoked the PLI application and the output log file of the product.

The *level* field shall indicate the severity level of the error, specified as a predefined constant. There shall be five levels: *ERR_ERROR*, *ERR_SYSTEM*, *ERR_INTERNAL*, *ERR_MESSAGE*, and *ERR_WARNING*. If **tf_message()** is called by the checktf application associated with the user-defined system task or function, the following rules shall apply:

If the checktf application is called when the Verilog HDL source code was being parsed or compiled, and the *level* is *ERR_ERROR*, *ERR_SYSTEM*, or *ERR_INTERNAL*, then parsing or compilation shall be aborted after an error message is reported.
If the checktf application is called when the Verilog HDL source code was being parsed or compiled, and the *level* is *ERR_WARNING* or *ERR_MESSAGE*, then parsing or compilation shall continue after a warning message is reported.
If the checktf application is called when the user-defined task or function was invoked on the interactive command line, the interactive command shall be aborted after a warning message or error message is reported.

The *facility* and *code* fields shall be string arguments that can be used in the Verilog software product message syntax. These strings shall be less than 10 characters in length.

The *message* argument shall be a user-defined control string containing the message to be displayed. The control string shall use the same formatting controls as the C `printf()` function (for example, %d). The message shall use up to a maximum of five variable arguments. There shall be no limit to the length of a variable argument. Formatting characters, such as `\n`, `\t`, `\b`, `\f`, or `\r`, do not need to be included in the message the software product shall automatically format each message.

An example of a **tf_message()** call and the output generated are shown below. Note that the format of the output shall be defined by the software product.

Calling **tf_message()** with the arguments:

```
tf_message(ERR_ERROR, "User", "TFARG",
      "Argument number %d is illegal in task %s", argnum,
taskname);
```

Might produce the output:

```
ERROR!   Argument number 2 is illegal in task      [User-TFARG]
         $usertask
```

The routine **tf_message()** provides more control over the format and severity of error or warning messages than the routines **tf_error()** and **tf_warning()** can provide. In addition, the routine **tf_message()** can be used in conjunction with **tf_text()**, which shall allow an error or warning message to be stored while a PLI application executes additional code before the message is printed and parsing or compilation of Verilog HDL source possibly aborted.

Copyright © 2001 IEEE. All rights reserved.

25.32 tf_mipname(), tf_imipname()

tf_mipname(), tf_imipname()			
Synopsis:	Get the hierarchical module instance path name as a string.		
Syntax:	`tf_mipname()` `tf_imipname(instance_p)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a string containing the hierarchical path name	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_spname() or tf_ispname() to get the scope path name Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_mipname()** shall return the Verilog HDL hierarchical path name to the module instance containing the call to the current instance or a specific instance of a user-defined system task or function.

The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

25.33 tf_movepvc_flag(), tf_imovepvc_flag()

tf_movepvc_flag(), tf_imovepvc_flag()			
Synopsis:	Move system task/function argument value change flags.		
Syntax:	`tf_movepvc_flag(narg)` `tf_imovepvc_flag(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	The value of the pvc flag	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument, or **-1**
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the argument that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy a pvc flag to the saved pvc flag Use tf_testpvc_flag() or tf_itestpvc_flag() to get the value of a saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_movepvc_flag()** and **tf_imovepvc_flag()** shall move the current pvc flag to the saved pvc flag and clear the current flag for the current instance or a specific instance of a user-defined system task or function. The routine shall return the value of the flag that was moved.

 Copyright © 2001 IEEE. All rights reserved.

The argument *narg* shall be the index number of an argument in a specific instance of a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument pvc flags shall be moved and the logical OR of all saved flags returned.

PVC flags shall be used to indicate whether a particular user-defined system task or function argument has changed value. Each argument shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

25.34 tf_multiply_long()

tf_multiply_long()			
Synopsis:	Multiply two 64 bit integers.		
Syntax:	`tf_multiply_long(aof_low1, aof_high1, low2, high2)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_subtract_long() to subtract two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_multiply_long()** shall multiply two 64-bit values. After calling **tf_multiply_long()**, the variables used to pass the first operand shall contain the results of the multiplication. Figure 165 shows the high and low 32 bits of two 64-bit integers and how **tf_multiply_long()** shall multiply them.

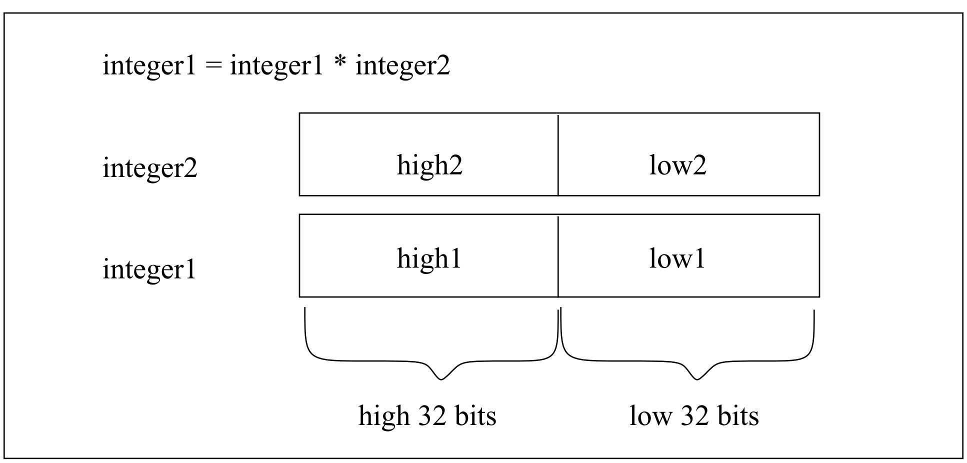


Figure 165—Multiplying with tf_multiply_long()

Copyright © 2001 IEEE. All rights reserved.

25.35 tf_nodeinfo(), tf_inodeinfo()

tf_nodeinfo(), tf_inodeinfo()			
Synopsis:	Get system task/function argument node information.		
Syntax:	`tf_nodeinfo(narg, nodeinfo_p)` `tf_inodeinfo(narg, nodeinfo_p, instance_p)`		
	Type	**Description**	
Returns:	struct t_tfnodeinfo *	The value of the second argument if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	struct t_tfnodeinfo *	nodeinfo_p	Pointer to a variable declared as the `t_tfnodeinfo` structure type
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() for general information on arguments Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_nodeinfo()** and **tf_inodeinfo()** shall obtain information about the specified argument in the current instance or a specific instance of a user-defined system task or function.

The information shall be stored in the C structure `s_tfnodeinfo` as defined in the file `veriuser.h`. The routine shall only be called for arguments that are of the types described in Table 192. Memory space shall first be allocated to hold the information before calling **tf_nodeinfo()** or **tf_inodeinfo()**. For example:

```
   {
   s_tfnodeinfo info;     /* declare a variable of the structure type */
   tf_nodeinfo(n, &info); /* pass tf_nodeinfo a pointer to the variable
*/
   ...
   }
```

The routines shall return the second argument, which is the pointer to the information structure. If *narg* is out of range, or if some other error is found, then **0** shall be returned.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The **tf_nodeinfo()** and **tf_inodeinfo()** routines shall support at least the following Verilog data types as a system task or system function argument:

scalar and vector regs
scalar and vector nets
integer, time and real variables
word select of a one-dimensional reg, integer or time array
null argument

The `s_tfnodeinfo` structure is defined in `veriuser.h` and is listed in Figure 166.

 Copyright © 2001 IEEE. All rights reserved.

```
typedef struct t_tfnodeinfo
{
  PLI_INT16 node_type;
  PLI_INT16 padding;
  union
    {
      struct t_vecval *vecval_p;
      struct t_strengthval *strengthval_p;
      PLI_BYTE8 *memoryval_p;
      double    *real_val_p;
    } node_value;
  PLI_BYTE8 *node_symbol;
  PLI_INT32 node_ngroups;
  PLI_INT32 node_vec_size;
  PLI_INT32 node_sign;
  PLI_INT32 node_ms_index;
  PLI_INT32 node_ls_index;
  PLI_INT32 node_mem_size;
  PLI_INT32 node_lhs_element;
  PLI_INT32 node_rhs_element;
  PLI_INT32 *node_handle;
} s_tfnodeinfo, *p_tfnodeinfo;
```

Figure 166—The s_tfnodeinfo structure definition

The following paragraphs define the fields of the `s_tfnodeinfo` structure.

The *node_type* field of the `s_tfnodeinfo` structure shall indicate the Verilog HDL data type of the argument, and is one of the predefined constants as given in Table 192 and defined in `veriuser.h`.

Table 192—Predefined constants for node_type

Predefined constant	Description
tf_null_node	Not a writable argument
tf_reg_node	Argument references a reg variable
tf_integer_node	Argument references an integer variable
tf_real_node	Argument references a real variable
tf_time_node	Argument references a time variable
tf_netvector_node	Argument references a vector net
tf_netscalar_node	Argument references a scalar net
tf_memory_node	Argument references a memory

The *node_value* field of the `s_tfnodeinfo` structure shall be a union of pointers to value structures defining the current value on the node referenced by the argument. The union member accessed shall depend on the *node_type*. The union members are given in Table 193.

Copyright © 2001 IEEE. All rights reserved.

Table 193—How the node_value union is used

When the node_type is	The union member used is
tf_reg_node, tf_integer_node, tf_time_node, or **tf_netvector_node**	`vecval_p`
tf_real_node	`real_val_p`
tf_netscalar_node	`strengthval_p`
tf_memory_node	`memoryval_p`

If the *node_type* is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then *node_value* shall be a pointer to an array of `s_vecval` structures that gives the resultant value of the node. The `s_vecval` structure for representing vector values is defined in `veriuser.h` and is listed in Figure 167.

```
typedef struct t_vecval
{
  PLI_INT32 avalbits;
  PLI_INT32 bvalbits;
} s_vecval, *p_vecval;
```

Figure 167—The s_vecval structure definition

If the number of bits in the vector (defined by the *node_vec_size* field of the `s_tfnodeinfo` structure) is less than or equal to 32, then there shall only be one `s_vecval` group in the `node_value.vecval_p` array. For 33 bits to 64 bits, two groups shall be in the array, and so on. The number of groups shall also be given by the value of *node_ngroups*. The fields for *avalbits* and *bvalbits* of the `s_vecval` structure shall hold the bit patterns making up the value of the argument. The lsb in the value shall be represented by the lsb s in the *avalbits* and *bvalbits* components, and so on. The bit coding shall be as given in Table 194.

Table 194—avalbits/bvalbits encoding

aval / bval	Logic value
00	0
10	1
01	High impedance
11	Unknown

If the *node_type* field of the `s_tfnodeinfo` structure is **tf_netscalar_node**, then the *node_value.strengthval_p* field of the `s_tfnodeinfo` structure shall point to an `s_strengthval` structure of the form given in Figure 168.

```
typedef struct t_strengthval
{
  PLI_INT32 strength0;
  PLI_INT32 strength1;
} s_strengthval, *p_strengthval;
```

Figure 168—The s_strengthval structure definition

 Copyright © 2001 IEEE. All rights reserved.

In the s_strengthval structure, *strength0* shall give the 0-strength bit pattern for the value, and *strength1* shall give the 1-strength bit pattern. Refer to 7.10 for details about these bit patterns.

If the *node_type* field of the s_tfnodeinfo structure is **tf_memory_node**, then *node_value.memoryval_p* shall point to a memval structure giving the total contents of the memory. The structure is organized as shown in Figure 169.

```
struct
{
    PLI_BYTE8 avalbits[node_ngroups];
    PLI_BYTE8 bvalbits[node_ngroups];
} memval[node_mem_size];
```

Figure 169—The memval structure definition

Note that a pointer to the memval structure data structure cannot be represented in C, so the *node_value.memoryval_p* field of the s_tfnodeinfo structure is declared as a pointer to a PLI_BYTE8 type. The memory element with the lowest number address in the Verilog array declaration shall be located in the first group of bytes, which is the byte group represented by memval[0].

The *node_symbol* field of the s_tfnodeinfo structure shall be a string pointer to the identifier of the argument.

If the *node_type* field of the s_tfnodeinfo structure is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then the *node_ngroups* field of the s_tfnodeinfo structure shall indicate the number of groups for the argument *nodevalue* and shall determine the array size of the *node_value.vecval_p* value structure. If the *node_type* is **tf_real_node**, then *node_ngroups* shall be 0.

If the *node_type* field of the s_tfnodeinfo structure is **tf_reg_node**, **tf_integer_node**, **tf_time_node**, or **tf_netvector_node**, then the *node_vec_size* field of the s_tfnodeinfo structure shall indicate the total number of bits in the array of the *node_value.vecval_p* structure. If *node_type* is **tf_real_node**, then *node_vec_size* shall be 0.

The *node_sign* field of the s_tfnodeinfo structure shall indicate the sign type of the node as follows: 0 for unsigned, nonzero for signed.

If the *node_type* is **tf_memory_node**, then *node_mem_size* shall indicate the number of elements in the *node_value.memoryval_p* structure.

If the *node_type* field of the s_tfnodeinfo structure is **tf_reg_node** or **tf_netvector_node**, then the *node_value.node_ms_element* and *node_value.node_ls_element* fields shall contain the msb and lsb of the given vector.

If the *node_type* field of the s_tfnodeinfo structure is **tf_reg_node** or **tf_netvector_node**, and the argument is a part-select, then the *node_value.node_rhs_index* and *node_value.node_lhs_index* fields shall contain the msb and lsb of the given part-select.

The field *node_handle* is not used.

Copyright © 2001 IEEE. All rights reserved.

25.36 tf_nump(), tf_inump()

tf_nump(), tf_inump()			
Synopsis:	Get number of task or function arguments.		
Syntax:	`tf_nump()` `tf_inump(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of arguments	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_nump()** and **tf_inump()** shall return the number of task/function arguments specified in the current instance or a specific instance of a user-defined task or function statement in the Verilog source description. The number returned shall be greater than or equal to zero.

Note: null arguments are counted. Therefore, $foo() returns a count of 1 and $foo(,) returns a count of 2. The routine **tf_typep()** returns a type of **tf_nullparam** for a null argument.

25.37 tf_propagatep(), tf_ipropagatep()

tf_propagatep(), tf_ipropagatep()			
Synopsis:	Propagate a system task/function argument value.		
Syntax:	`tf_propagatep(narg)` `tf_ipropagatep(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_exprinfo() or tf_iexprinfo() to get an argument expression value Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_propagatep()** and **tf_ipropagatep()** shall write a value to an argument node of the current instance or a specific instance of a user-defined system task or function, and then propagate the value to any loads that read the value of the node.

 Copyright © 2001 IEEE. All rights reserved.

In order to write values back into a Verilog software product data structure using **tf_propagatep()** and **tf_ipropagatep()**, the value shall first be placed into the value structure pointed to by the component `expr_value_p` as allocated by calling **tf_exprinfo()** or **tf_iexprinfo()**. The structure for **tf_exprinfo()** and **tf_iexprinfo()** shall be used for all argument types except memories.

25.38 tf_putlongp(), tf_iputlongp()

<table>
<tr><th colspan="4">tf_putlongp(), tf_iputlongp()</th></tr>
<tr><td>Synopsis:</td><td colspan="3">Write a 64-bit integer value to a system task/function argument or function return.</td></tr>
<tr><td>Syntax:</td><td colspan="3"><code>tf_putlongp(narg, lowvalue, highvalue)</code>
<code>tf_iputlongp(narg, lowvalue, highvalue, instance_p)</code></td></tr>
<tr><td></td><td>Type</td><td colspan="2">Description</td></tr>
<tr><td>Returns:</td><td>PLI_INT32</td><td colspan="2">0 if successful; 1 if an error occurred</td></tr>
<tr><td></td><td>Type</td><td>Name</td><td>Description</td></tr>
<tr><td>Arguments:</td><td>PLI_INT32</td><td>narg</td><td>Index number of the user-defined system task or function argument or 0 to return a function value</td></tr>
<tr><td></td><td>PLI_INT32</td><td>lowvalue</td><td>Least significant (right-most) 32 bits of value</td></tr>
<tr><td></td><td>PLI_INT32</td><td>highvalue</td><td>Most significant (left-most) 32 bits of value</td></tr>
<tr><td></td><td>PLI_BYTE8 *</td><td>instance_p</td><td>Pointer to a specific instance of a user-defined system task or function</td></tr>
<tr><td>Related routines:</td><td colspan="3">Use tf_putp() or tf_iputp() to put an argument value as a 32-bit integer
Use tf_putrealp() or tf_iputrealp() to put an argument value as a double
Use tf_strdelputp() to put a value as a formatted string with delay
Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function</td></tr>
</table>

The TF routines **tf_putlongp()** and **tf_iputlongp()** shall write a 64-bit integer value to the argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putlongp()** and **tf_iputlongp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the values to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to 24.3 for more details on proper data type selection with put routines.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

Copyright © 2001 IEEE. All rights reserved.

25.39 tf_putp(), tf_iputp()

tf_putp(), tf_iputp()			
Synopsis:	Put an integer value to a system task/function argument or function return.		
Syntax:	`tf_putp(narg, value)` `tf_iputp(narg, value, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument or 0 to return a function value
	PLI_INT32	value	An integer value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_putlongp() or tf_iputlongp() to put an argument value as a 64-bit integer Use tf_putrealp() or tf_iputrealp() to put an argument value as a double Use tf_strdelputp() to put a value as a formatted string with delay Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_putp()** and **tf_iputp()** shall write an integer value to the argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putp()** or **tf_iputp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the value to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to Section 24.3 for more details on proper data type selection with put routines.

NOTE Calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

 Copyright © 2001 IEEE. All rights reserved.

25.40 tf_putrealp(), tf_iputrealp()

tf_putrealp(), tf_iputrealp()			
Synopsis:	Write a real value to a system task/function argument or function return.		
Syntax:	`tf_putrealp(narg, value)` `tf_iputrealp(narg, value, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument or 0 to return a function value
	double	value	A double-precision value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_putp() or tf_iputp() to put an argument value as a 32-bit integer Use tf_putlongp() or tf_iputlongp() to put an argument value as a 64-bit integer Use tf_strdelputp() to put a value as a formatted string with delay Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_putrealp()** and **tf_iputrealp()** shall write a double-precision real value to the argument specified by *narg* of the current instance or a specific instance of a user-defined system task or function. If *narg* is 0, **tf_putrealp()** and **tf_iputrealp()** shall write the value as the return of a user-defined system function. If *narg* is out of range or the argument cannot be written to, then the routines shall do nothing. Should the calltf routine for a user defined system function fail to put a value during its execution, the default value of 0.0 shall be applied.

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

The data type of the value to be written should be consistent with the type of put routine and the type of the argument to which the value shall be written. Refer to 24.3 for more details on proper data type selection with put routines.

NOTE calling put routines to TF argument 0 (return of a function) shall only return a value in a calltf application, when the call to the function is active. The action of the put routine shall be ignored when the function is not active.

Copyright © 2001 IEEE. All rights reserved.

25.41 tf_read_restart()

tf_read_restart()			
Synopsis:	Get a block of data from a previously written save file.		
Syntax:	`tf_read_restart(blockptr, blocklen)`		
	Type	**Description**	
Returns:	PLI_INT32	Nonzero if successful; zero if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	blockptr	Pointer to block of saved data
	PLI_INT32	blocklen	Length of block
Related routines:	Use tf_write_save() to save a block of data		

The TF routine **tf_read_restart()** shall read back a block of memory that was saved with **tf_write_save()**. This routine shall only be called from the misctf application when the misctf routine is invoked with **reason_restart**.

The argument *blockptr* shall be a pointer to an allocated block of memory to which the saved data shall be restored.

The argument *blocklen* shall be the length in bytes of the allocated block of memory. Exactly as many bytes have to be restored as were written with **tf_write_save()**.

If any user task instance pointers have been saved (for use with tf_i* calls), **tf_getinstance()** has to be used to get new instance pointer values after the restart. If pointers to user data were saved, the application of the user has to implement a scheme to reconnect them properly.

25.42 tf_real_to_long()

tf_real_to_long()			
Synopsis:	Convert a real number to a 64-bit integer.		
Syntax:	`tf_real_to_long(realvalue, aof_low, aof_high)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	double	realvalue	Value to be converted
	PLI_INT32 *	aof_low	Pointer to an integer variable for storing the least significant (right-most) 32 bits of the converted value
	PLI_INT32 *	aof_high	Pointer to an integer variable for storing the most significant (left-most) 32 bits of the converted value
Related routines:	Use tf_long_to_real() to convert a 64-bit integer to a real number		

 Copyright © 2001 IEEE. All rights reserved.

The TF routine **tf_real_to_long()** shall convert a double-precision floating-point number to a 64-bit integer. The converted value shall be returned in the variables pointed to by *aof_low* and *aof_high*.

25.43 tf_rosynchronize(), tf_irosynchronize()

tf_rosynchronize(), tf_irosynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	`tf_rosynchronize()` `tf_irosynchronize(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function Use tf_synchronize() to synchronize to end of simulation time step Use tf_getnextlongtime() to get next time at which a simulation event is scheduled		

The TF routines **tf_rosynchronize()** and **tf_irosynchronize()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called with a reason of **reason_rosynch** at the end of the current simulation time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** have different functionality. The routine **tf_synchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_synch**, and the misctf application shall be allowed to schedule additional simulation events using routines such as **tf_strdelputp()**.

The routine **tf_rosynchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_rosynch**, and the PLI shall not be allowed to schedule any new events. This guarantees that all simulation events for the current time are completed. Calls to routines such as **tf_strdelputp()** and **tf_setdelay()** are illegal during processing of the misctf application with reason **reason_rosynch**.

The routine **tf_getnextlongtime()** shall only return the next simulation time for which an event is scheduled when used in conjunction with the routines **tf_rosynchronize()** and **tf_irosynchronize()**.

Copyright © 2001 IEEE. All rights reserved.

25.44 tf_scale_longdelay()

tf_scale_longdelay()			
Synopsis:	Convert a 64-bit integer delay to the timescale of the module instance.		
Syntax:	`tf_scale_longdelay(instance_p, delay_lo, delay_hi, aof_delay_lo, aof_delay_hi)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	PLI_INT32	delay_lo	Least significant (right-most) 32 bits of the delay to be converted
	PLI_INT32	delay_hi	Most significant (left-most) 32 bits of the delay to be converted
	PLI_INT32 *	aof_delay_lo	Pointer to a variable to store the least significant (right-most) 32 bits of the conversion result
	PLI_INT32 *	aof_delay_hi	Pointer to a variable to store the most significant (left-most) 32 bits of the conversion result
Related routines:	Use tf_scale_realdelay() to scale real number delays Use tf_unscale_longdelay() to convert a delay to the time unit of a module Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_scale_longdelay()** shall convert a 64-bit integer delay into the timescale of the module containing the instance of the user-defined system task or function pointed to by *instance_p*. The arguments *aof_delay_lo* and *aof_delay_hi* shall contain the address of the converted delay returned by the routine.

25.45 tf_scale_realdelay()

tf_scale_realdelay()			
Synopsis:	Convert a double-precision floating-point delay to the timescale of the module instance.		
Syntax:	`tf_scale_realdelay(instance_p, realdelay, aof_realdelay)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	double	realdelay	Value of the delay to be converted
	double *	aof_realdelay	Pointer to a variable to store the conversion result
Related routines:	Use tf_scale_longdelay() to scale 64-bit integer delays Use tf_unscale_realdelay() to convert a delay to the time unit of a module Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

 Copyright © 2001 IEEE. All rights reserved.

The TF routine **tf_scale_realdelay()** shall convert a double-precision floating-point delay into the timescale of the module containing the instance of the user-defined system task or function pointed to by *instance_p*. The argument *aof_realdelay* shall contain the address of the converted delay returned by the routine.

25.46 tf_setdelay(), tf_isetdelay()

tf_setdelay(), tf_isetdelay()			
Synopsis:	Activate the misctf application at a particular simulation time.		
Syntax:	`tf_setdelay(delay)` `tf_isetdelay(delay, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	delay	32-bit integer delay time
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setlongdelay() or tf_isetlongdelay() for 64-bit integer reactivation delays Use tf_setrealdelay() or tf_isetrealdelay() for real number reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setdelay()** and **tf_isetdelay()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The misctf application shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setdelay()** and **tf_isetdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled. Multiple calls to **tf_setdelay()** and **tf_isetdelay()** for the same time step are permitted and shall result in multiple calls to the misctf application for that time step.

The *delay* argument shall be a 32-bit integer and shall be greater than or equal to 0. The delay shall assume the timescale units specified for the module containing the specific system task call.

Copyright © 2001 IEEE. All rights reserved.

25.47 tf_setlongdelay(), tf_isetlongdelay()

tf_setlongdelay(), tf_isetlongdelay()			
Synopsis:	Activate the misctf application at a particular simulation time.		
Syntax:	`tf_setlongdelay(lowdelay, highdelay)` `tf_isetlongdelay(lowdelay, highdelay, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	lowdelay	Least significant (right-most) 32 bits of the delay time to reactivation
	PLI_INT32	highdelay	Most significant (left-most) 32 bits of the delay time to reactivation
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() for 32-bit integer reactivation delays Use tf_setrealdelay() or tf_isetrealdelay() for real number reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setlongdelay()** and **tf_isetlongdelay()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The misctf routine shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setlongdelay()** and **tf_isetlongdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled. Multiple calls to **tf_setlongdelay()** and **tf_isetlongdelay()** for the same time step are permitted and shall result in multiple calls to the misctf application for that time step.

The *delay* argument shall be a 64-bit integer and shall be greater than or equal to 0. The delay shall assume the timescale units specified for the module containing the specific system task call.

25.48 tf_setrealdelay(), tf_isetrealdelay()

tf_setrealdelay(), tf_isetrealdelay()			
Synopsis:	Activate the *misctf* application at a particular simulation time.		
Syntax:	`tf_setrealdelay(realdelay)` `tf_isetrealdelay(realdelay, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error occurred	
	Type	**Name**	**Description**
Arguments:	double	realdelay	Double-precision delay time to reactivation
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_setdelay() or tf_isetdelay() for 32-bit integer reactivation delays Use tf_setlongdelay() or tf_isetlongdelay() for 64-bit integer reactivation delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

 Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_setrealdelay()** and **tf_isetrealdelay()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called at a future *reactivation time*. The reactivation time shall be the current simulation time plus the specified delay. The misctf application shall be called at the reactivation time with a reason of **reason_reactivate**. The **tf_setrealdelay()** and **tf_isetrealdelay()** routines can be called several times with different delays, and several reactivations shall be scheduled. Multiple calls to **tf_setrealdelay()** and **tf_isetrealdelay()** for the same time step are permitted and shall result in multiple calls to the misctf application for that time step.

The *delay* argument shall be a double-precision value and shall be greater than or equal to 0.0. The delay shall assume the timescale units specified for the module containing the specific system task call.

25.49 tf_setworkarea(), tf_isetworkarea()

tf_setworkarea(), tf_isetworkarea()			
Synopsis:	Store user data pointer in work area.		
Syntax:	`tf_setworkarea(workarea)` `tf_isetworkarea(workarea, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	workarea	Pointer to user data
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getworkarea() or tf_igetworkarea() to retrieve the user data pointer Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_setworkarea()** and **tf_isetworkarea()** shall store a pointer to user data in the work area of the current instance or a specific instance of a user-defined system task or function. The pointer that is stored can be retrieved by calling **tf_getworkarea()** or **tf_igetworkarea()**.

The work area can be used for

Saving information during one call to a PLI routine, which can be retrieved upon a subsequent invocation of the routine

Passing information from one type of PLI application to another, such as from a checktf application to a calltf application

Note that the workarea pointer is a *PLI_BYTE8 ** type. If the memory allocated for the user data is of some other type, it should be cast to *PLI_BYTE8 **.

Copyright © 2001 IEEE. All rights reserved.

25.50 tf_sizep(), tf_isizep()

tf_sizep(), tf_isizep()			
Synopsis:	Get the bit length of a system task/function argument.		
Syntax:	`tf_sizep(narg)` `tf_isizep(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	The number of bits of the system task/function argument	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_sizep()** and **tf_isizep()** shall return the value size in bits of the specified argument in the current instance or a specific instance of a user-defined system task or function.

If the specified argument is a literal string, **tf_sizep()** and **tf_isizep()** shall return the string length.

If the specified argument is real or if an error is detected, **tf_sizep()** and **tf_isizep()** shall return 0.

The argument *narg* shall be the index number of an ARGUMENT in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.51 tf_spname(), tf_ispname()

tf_spname(), tf_ispname()			
Synopsis:	Get scope hierarchical path name as a string.		
Syntax:	`tf_spname()` `tf_ispname(instance_p)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string with the hierarchical path name	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

 Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_spname()** and **tf_ispname()** shall return a pointer to the Verilog HDL hierarchical path name to the scope containing the call of a specific instance of a user-defined system task or function.

A scope shall be

A top-level module
A module instance
A named begin-end block
A named fork-join block
A Verilog HDL task
A Verilog HDL function

The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

25.52 tf_strdelputp(), tf_istrdelputp()

tf_strdelputp(), tf_istrdelputp()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a 32-bit integer delay.		
Syntax:	`tf_strdelputp(narg, bitlength, format, value_p,delay, delaytype)` `tf_istrdelputp(narg, bitlength, format, value_p,delay, delaytype, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	value_p	Quoted character string or pointer to a character string with the value to be written
	PLI_INT32	delay	Integer value representing the time delay before the value should be written to the argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_strlongdelputp() or tf_istrlongdelputp() for 64-bit integer delays Use tf_strrealdelputp() or tf_istrrealdelputp() for real number delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strdelputp()** and **tf_istrdelputp()** shall write a string value to the specified argument of the current instance or a specific instance of a user-defined system task or function. The actual change to the argument shall be scheduled as an event on the argument in the Verilog model at a future simulation time. An argument value of 0 (system function return) shall be illegal.

Copyright © 2001 IEEE. All rights reserved.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters given in Table 195.

Table 195—Format characters

Format character	Description
`'b'` or `'B'`	Value is in binary
`'o'` or `'O'`	Value is in octal
`'d'` or `'D'`	Value is in decimal
`'h'` or `'H'`	Value is in hexadecimal

The *delay* argument shall represent the amount of time before the value shall be applied to the argument, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same reg or variable. The *delaytype* shall be one of integer values shown in Table 196.

Table 196—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument in the Verilog model are removed before scheduling a new event the last event to be scheduled is not necessarily the last one to occur

 Copyright © 2001 IEEE. All rights reserved.

25.53 tf_strgetp(), tf_istrgetp()

tf_strgetp(), tf_istrgetp()			
Synopsis:	Get formatted system task/function argument values.		
Syntax:	`tf_strgetp(narg, format)` `tf_istrgetp(narg, format, instance_p)`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string with the argument value	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	format	Character in single quotes controlling the return value format
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getp() or tf_igetp() to get an argument value as a 32-bit integer Use tf_getlongp() or tf_igetlongp() to get an argument value as a 64-bit integer Use tf_getrealp() or tf_igetrealp() to get an argument value as a double Use tf_getcstringp() or tf_igetcstringp() to get an argument value as a string Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strgetp()** and **tf_istrgetp()** shall return a pointer to a string that contains the value of the argument expression of the current instance or a specific instance of a user-defined system task or function.

The string format is specified by *format*, and shall be one of the following characters shown in Table 197.

Table 197—Format characters

Format character	Description
`'b'` or `'B'`	Value is in binary
`'o'` or `'O'`	Value is in octal
`'d'` or `'D'`	Value is in decimal
`'h'` or `'H'`	Value is in hexadecimal

The string value returned shall have the same form as output from the formatted built-in system task **$display()** in terms of value lengths and value characters used. The length shall be of arbitrary size (not limited to 32 bits as with the **tf_getp()** routine), and unknown and high-impedance values shall be obtained.

The referenced argument can be a string, in which case a pointer to the string shall be returned (the *format* shall be ignored in this case). The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

A `null` pointer shall be returned for errors.

Copyright © 2001 IEEE. All rights reserved.

25.54 tf_strgettime()

tf_strgettime()			
Synopsis:	Get the current simulation time as a string.		
Syntax:	`tf_strgettime()`		
	Type	**Description**	
Returns:	PLI_BYTE8 *	Pointer to a character string with the simulation time	
	Type	**Name**	**Description**
Arguments:			No arguments
Related routines:	Use tf_gettime() to get simulation time as a 32-bit integer value Use tf_getlongtime() to get simulation time as a 64-bit integer value Use tf_getrealtime() to get simulation time as a real value		

The TF routine **tf_strgettime()** shall return a pointer to a string, which shall be the ASCII representation of the current simulation time. The string obtained shall be stored in a temporary buffer. If the string is needed across multiple calls to the PLI application, the string should be preserved.

Time shall be expressed in simulation time units, which is the smallest time precision used by all modules in a design.

25.55 tf_strlongdelputp(), tf_istrlongdelputp()

tf_strlongdelputp(), tf_istrlongdelputp()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a 64-bit integer delay.		
Syntax:	`tf_strlongdelputp(narg, bitlength, format, value_p, lowdelay, highdelay, delaytype)` `tf_istrlongdelputp(narg, bitlength, format, value_p, lowdelay, highdelay, delaytype, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	value_p	Quoted character string or pointer to a character string with the value to be written
	PLI_INT32	lowdelay	Least significant (right-most) 32 bits of delay before the value is be written to the argument
	PLI_INT32	highdelay	Most significant (left-most) 32 bits of delay before the value is be written to the argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_strdelputp() or tf_istrdelputp() for 32-bit integer delays Use tf_strrealdelputp() or tf_istrrealdelputp() for real number delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_strlongdelputp()** and **tf_istrlongdelputp()** shall write a string value to the specified argument of the current instance or a specific instance of a user-defined system task or function. The actual change to the argument shall be scheduled as an event on the argument in the Verilog model at a future simulation time. An argument value of 0 (system function return) shall be illegal.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters shown in Table 198.

Table 198—Format characters

Format character	Description
'b' or 'B'	Value is in binary
'o' or 'O'	Value is in octal
'd' or 'D'	Value is in decimal
'h' or 'H'	Value is in hexadecimal

The *delay* argument shall represent the amount of time before the value shall be applied to the argument, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same reg or variable. The *delaytype* shall be one of integer values shown in Table 199.

Table 199—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument in the Verilog model are removed before scheduling a new event the last event to be scheduled is not necessarily the last one to occur

Copyright © 2001 IEEE. All rights reserved.

25.56 tf_strrealdelputp(), tf_istrrealdelputp()

tf_strrealdelputp(), tf_istrrealdelputp()			
Synopsis:	Write a value to a system task/function argument from string value specification, using a real number delay.		
Syntax:	`tf_strrealdelputp(narg, bitlength, format, value_p, realdelay, delaytype)` `tf_istrrealdelputp(narg, bitlength, format, value_p, realdelay, delaytype, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	1 if successful; 0 if an error is detected	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_INT32	bitlength	Number of bits the value represents
	PLI_INT32	format	A character in single quotes representing the radix (base) of the value
	quoted string or PLI_BYTE8 *	value_p	Quoted character string or pointer to a character string with the value to be written
	double	realdelay	Double-precision value representing the time delay before the value shall be written to the argument
	PLI_INT32	delaytype	Integer code representing the delay mode for applying the value
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_strdelputp() or tf_istrdelputp() for 32-bit integer delays Use tf_strlongdelputp() or tf_istrlongdelputp() for 64-bit integer delays Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_strrealdelputp()** and **tf_istrrealdelputp()** shall write a string value to the specified argument of the current instance or a specific instance of a user-defined system task or function. The actual change to the argument shall be scheduled as an event on the argument in the Verilog model at a future simulation time. An argument value of 0 (system function return) shall be illegal.

The *bitlength* argument shall define the value size in bits.

The *format* shall define the format of the value specified by *value_p* and shall be one of the characters given in Table 200.

Table 200—Format characters

Format character	Description
`'b'` or `'B'`	Value is in binary
`'o'` or `'O'`	Value is in octal
`'d'` or `'D'`	Value is in decimal
`'h'` or `'H'`	Value is in hexadecimal

 Copyright © 2001 IEEE. All rights reserved.

The *delay* argument shall represent the amount of time before the value shall be applied to the argument, and it shall be greater than or equal to 0. The delay shall assume the timescale units of the module containing the instance of the user-defined system task or function.

The *delaytype* argument shall determine how the value shall be scheduled in relation to other simulation events on the same reg or variable. The *delaytype* shall be one of integer values shown in Table 201.

Table 201—delaytype codes

delaytype code	Definition	Description
0	Inertial delay	All scheduled events on the output argument in the Verilog model are removed before scheduling a new event
1	Modified transport delay	All events that are scheduled for times later than the new event on the output argument in the Verilog model are removed before scheduling a new event
2	Pure transport delay	No scheduled events on the output argument in the Verilog model are removed before scheduling a new event the last event to be scheduled is not necessarily the last one to occur

25.57 tf_subtract_long()

tf_subtract_long()			
Synopsis:	Subtract two 64-bit integers.		
Syntax:	`tf_subtract_long(aof_low1, aof_high1, low2, high2)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	PLI_INT32 *	aof_low1	Pointer to least significant 32 bits of first operand
	PLI_INT32 *	aof_high1	Pointer to most significant 32 bits of first operand
	PLI_INT32	low2	Least significant 32 bits of second operand
	PLI_INT32	high2	Most significant 32 bits of second operand
Related routines:	Use tf_add_long() to add two 64-bit integers Use tf_multiply_long() to multiply two 64-bit integers Use tf_divide_long() to divide two 64-bit integers Use tf_compare_long() to compare two 64-bit integers		

The TF routine **tf_subtract_long()** shall subtract two 64-bit values. After calling **tf_subtract_long()**, the variables used to pass the first operand shall contain the results of the subtraction. The operands shall be assumed to be in two s complement form. Figure 170 shows the high and low 32 bits of two 64-bit integers and how **tf_subtract_long()** shall subtract them.

Copyright © 2001 IEEE. All rights reserved.

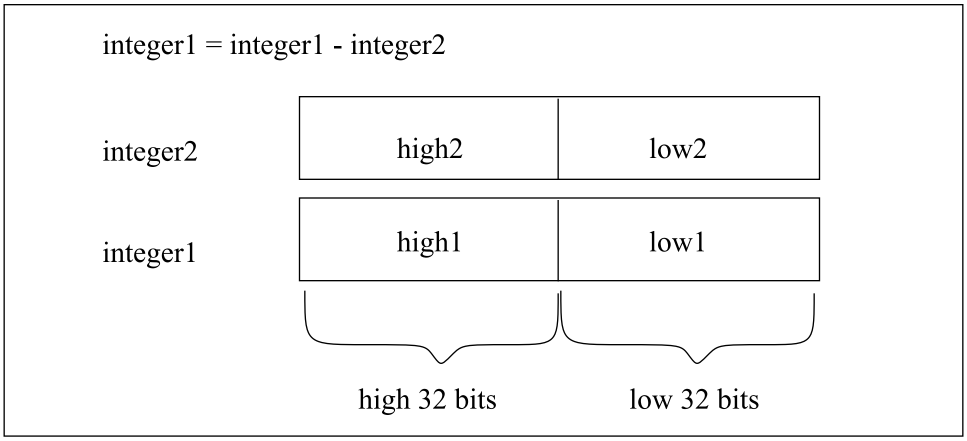


Figure 170—Subtracting with tf_subtract_long()

The example program fragment shown in Figure 171 uses **tf_subtract_long()** to calculate the relative time from the current time to the next event time (this example assumes that the code is executed during a misctf application call with reason of **reason_rosynch**).

The text message generated by this example is split between the two **io_printf()** calls. If done in a single **io_printf()**, the second call to **tf_longtime_tostr()** would overwrite the string from the first call, since the string is placed in a temporary buffer.

```
PLI_INT32 currlow, currhigh;
PLI_INT32 relalow, relahigh;

currlow = tf_getlongtime(&currhigh);
io_printf("At time %s: ", tf_longtime_tostr(currlow, currhigh));
if(tf_getnextlongtime(&relalow, &relahigh) == 0)
{
   tf_subtract_long(&relalow, &relahigh, currlow, currhigh);
   io_printf ("relative time to next event is %s",
   tf_longtime_tostr(relalow, relahigh));
}
else
   printf("there are no future events");
```

Figure 171—Using tf_subtract_long()

 Copyright © 2001 IEEE. All rights reserved.

25.58 tf_synchronize(), tf_isynchronize()

tf_synchronize(), tf_isynchronize()			
Synopsis:	Synchronize to end of simulation time step.		
Syntax:	`tf_synchronize()` `tf_isynchronize(instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	0 if successful; 1 if an error occurred	
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_rosynchronize() for read-only synchronization Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_synchronize()** and **tf_isynchronize()** shall schedule a callback to the misctf application associated with the current instance or a specific instance of a user-defined system task or function. The misctf application shall be called with a reason of **reason_synch** at the end of the current simulation time step.

The routines **tf_synchronize()** and **tf_rosynchronize()** have different functionality. The routine **tf_synchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_synch**, and the misctf application shall be allowed to schedule additional simulation events using routines such as **tf_strdelputp()**.

The routine **tf_rosynchronize()** shall call the associated misctf application at the end of the current simulation time step with **reason_rosynch**, and the PLI shall not be allowed to schedule any new events. This guarantees that all simulation events for the current time are completed. Calls to routines such as **tf_strdelputp()** and **tf_setdelay()** are illegal during processing of the misctf application with reason **reason_rosynch**.

The routine **tf_getnextlongtime()** shall only return the next simulation time for which an event is scheduled when used in conjunction with the routines **tf_rosynchronize()** and **tf_irosynchronize()**.

25.59 tf_testpvc_flag(), tf_itestpvc_flag()

tf_testpvc_flag(), tf_itestpvc_flag()			
Synopsis:	Test system task/function argument value change flags.		
Syntax:	`tf_testpvc_flag(narg)` `tf_itestpvc_flag(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	The value of the saved pvc flag	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument, or **-1**
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_asynchon() or tf_iasynchon() to enable pvc flags Use tf_getpchange() or tf_igetpchange() to get the index number of the argument that changed Use tf_copypvc_flag() or tf_icopypvc_flag() to copy a pvc flag to the saved pvc flag Use tf_movepvc_flag() or tf_imovepvc_flag() to move a pvc flag to the saved pvc flag Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

Copyright © 2001 IEEE. All rights reserved.

The TF routines **tf_testpvc_flag()** and **tf_itestpvc_flag()** shall return value of the saved pvc flag. The argument *narg* shall be the index number of an argument in a specific instance of a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1. If *narg* is **-1**, then all argument pvc flags shall be tested and the logical OR of all saved flags returned.

PVC flags shall be used to indicate whether a particular user-defined system task or function argument has changed value. Each argument shall have two pvc flags: a current pvc flag, which shall be set by a software product when the change occurs, and a saved pvc flag, which shall be controlled by the user.

NOTE PVC flags shall not be set by the software product until **tf_asynchon()** or **tf_iasynchon()** has been called.

25.60 tf_text()

tf_text()			
Synopsis:	Store error message information.		
Syntax:	`tf_text(message, arg1,...arg5)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	message	A quoted character string or pointer to a character string with a message to be stored
(optional)		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the message string
Related routines:	Use tf_message() to display the stored error message		

The TF routine **tf_text()** shall store text messages about an error in a buffer, which will be printed when the routine **tf_message()** is called. The routine shall provide a method for a PLI application to store information about one or more errors before it calls the **tf_message()** TF routine. This allows an application to process all of a routine, such as syntax checking, before calling **tf_message()**, which can be set to abort processing after printing messages. An application shall be able to call **tf_text()** any number of times before it calls **tf_message()**.

When the application calls **tf_message()**, the information stored by **tf_text()** shall be displayed before the information in the call to **tf_message()**. Each call to **tf_message()** shall clear the buffer where **tf_text()** stores its information.

The *message* argument is a user-defined control string containing the message to be displayed. The control string uses the same formatting controls as the C `printf()` function (for example, %d). The message shall use up to a maximum of five variable arguments. There shall be no limit to the length of a variable argument. Formatting characters, such as `\n`, `\t`, `\b`, `\f`, or `\r`, do not need to be included in the message the software product shall automatically format each message.

An example of using **tf_text()** and **tf_message()** calls and the output generated follow. Note that the format of the output shall be defined by the software product.

 Copyright © 2001 IEEE. All rights reserved.

Calling **tf_text()** and **tf_message()** with the arguments:

```
tf_text ("Argument number %d", argnum);
...
tf_message(ERR_ERROR, "User", "TFARG",
        " is illegal in task %s", taskname);
```

Might produce the output:

```
ERROR!  Argument number 2 is illegal in task     [ User-TFARG]
        $usertask
```

25.61 tf_typep(), tf_itypep()

tf_typep(). tf_itypep()			
Synopsis:	Get a system task/function argument type.		
Syntax:	`tf_typep(narg)` `tf_itypep(narg, instance_p)`		
	Type	**Description**	
Returns:	PLI_INT32	A predefined integer constant representing the Verilog HDL data type for the argument	
	Type	**Name**	**Description**
Arguments:	PLI_INT32	narg	Index number of the user-defined system task or function argument
	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
Related routines:	Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routines **tf_typep()** and **tf_itypep()** shall return an integer constant indicating the type of an argument for the current instance or a specific instance of a user-defined system task or function. The integer constants shall be as shown in Table 202.

Table 202—Predefined tf_typep() constants

Predefined constant	Description
tf_nullparam	The argument is a *null* expression (where no text has been given as the argument), or *narg* is out of range
tf_string	The argument is a literal string
tf_readonly	The argument is a expression with a value that can be read but not written
tf_readwrite	The argument is a expression with a value that can be read and written
tf_readonlyreal	The argument is a real number expression with a value that can be read but not written
tf_readwritereal	The argument is a real number expression with a value that can be read and written

Copyright © 2001 IEEE. All rights reserved.

A *read only* expression shall be any expression that would be illegal as a left-hand-side construct in a Verilog HDL procedural assignment (e.g., an expression using *net* data types or *event* data types)

A *read/write* expression shall be any expression that would be legal as a left-hand-side construct in a Verilog HDL procedural assignments (e.g., an expression using *reg*, *integer*, *time*, or *real* data types)

The argument *narg* shall be the index number of an argument in a user-defined system task or function. Task/function argument index numbering shall proceed from left to right, with the left-most argument being number 1.

25.62 tf_unscale_longdelay()

tf_unscale_longdelay()			
Synopsis:	Convert a delay from internal simulation time units to the timescale of a particular module.		
Syntax:	`tf_unscale_longdelay(instance_p, delay_lo, delay_hi, aof_delay_lo, aof_delay_hi)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	PLI_INT32	delay_lo	Least significant (right-most) 32 bits of the delay to be converted
	PLI_INT32	delay_hi	Most significant (left-most) 32 bits of the delay to be converted
	PLI_INT32 *	aof_delay_lo	Pointer to a variable to store the least significant (right-most) 32 bits of the conversion result
	PLI_INT32 *	aof_delay_hi	Pointer to a variable to store the most significant (left-most) 32 bits of the conversion result
Related routines:	Use tf_unscale_realdelay() to unscale real number delays Use tf_scale_longdelay() to convert a delay to the timescale of the module instance Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_unscale_longdelay()** shall convert a 64-bit integer delay expressed in internal simulation time into the time units of the module containing the user-defined system task or function referenced by the *instance_p* pointer. The argument *aof_delay_lo* and *aof_delay_hi* shall contain the address of the converted delay returned by the routine.

 Copyright © 2001 IEEE. All rights reserved.

25.63 tf_unscale_realdelay()

tf_unscale_realdelay()			
Synopsis:	Convert a delay expressed in internal simulation time units to the timescale of a particular module.		
Syntax:	`tf_unscale_realdelay(instance_p, realdelay, aof_realdelay)`		
	Type	**Description**	
Returns:	void		
	Type	**Name**	**Description**
Arguments:	PLI_BYTE8 *	instance_p	Pointer to a specific instance of a user-defined system task or function
	double	delay	Value of the delay to be converted
	double *	aof_realdelay	Pointer to a variable to store the conversion result
Related routines:	Use tf_unscale_longdelay() to unscale 64-bit integer delays Use tf_scale_realdelay() to convert a delay to the timescale of the module instance Use tf_getinstance() to get a pointer to an instance of a user-defined system task or function		

The TF routine **tf_unscale_realdelay()** shall convert a double-precision delay expressed in internal simulation time into the time units of the module containing the user-defined system task or function referenced by the *instance_p* pointer. The argument *aof_realdelay* shall contain the address of the converted delay returned by the routine.

25.64 tf_warning()

tf_warning()			
Synopsis:	Report a warning message.		
Syntax:	`tf_warning(format, arg1,...arg5)`		
	Type	**Description**	
Returns:	PLI_INT32	Always returns 0	
	Type	**Name**	**Description**
Arguments:	quoted string or PLI_BYTE8 *	format	A quoted character string or pointer to a character string that controls the message to be written
(optional)		arg1...arg5	One to five optional arguments of the format control string; the type of each argument should be consistent with how it is used in the format string
Related routines:	Use tf_message() to write warning messages with additional format control Use tf_error() to write an error message Use io_printf() or io_mcdprintf() to write a formatted message		

The TF routine **tf_warning()** shall provide a warning reporting mechanism compatible with warning messages generated by the software product.

Copyright © 2001 IEEE. All rights reserved.

The *format* control string shall use the same formatting controls as the C *printf()* function (for example, %d).

The maximum number of arguments that shall be used in the format control string is 5.

The location information (file name and line number) of the current instance of the user-defined system task or function shall be appended to the message using a format compatible with error messages generated by the software product.

The message shall be written to both the output channel of the software product which invoked the PLI application and the output log file of the product.

The **tf_warning()** routine shall not abort parsing or compilation of Verilog HDL source code.

25.65 tf_write_save()

tf_write_save()			
Synopsis:	Append a block of data to a save file.		
Syntax:	`tf_write_save(blockptr, blocklen)`		
Returns:	**Type**	**Description**	
	PLI_INT32	Nonzero value if successful, zero if an error is encountered	
Arguments:	**Type**	**Name**	**Description**
	PLI_BYTE8 *	blockptr	Pointer to the first byte of the block of data to be saved
	PLI_INT32	blocklen	Number of bytes are to be saved
Related routines:	Use tf_read_restart() to retrieve the data saved		

The TF routine **tf_write_save()** shall write user-defined data to the end of a save file being written by the **$save** built-in system task. This routine shall be called from the misctf application when misctf is invoked with **reason_save**.

The argument *blockptr* shall be a pointer to an allocated block of memory containing the data to be saved.

The argument *blocklen* shall be the length in bytes of the allocated block of memory. Note that exactly as many bytes shall be restored using **tf_read_restart()** as were written with **tf_write_save()**.

 Copyright © 2001 IEEE. All rights reserved.