

Using the PinPort PLI

Exsent

EXSENT INC.

750 South Plaza Drive, Suite 317

Saint Paul, MN 55120-1505

Phone : 651-686-7669

Fax : 651-686-7584

E-mail: info@exsent.com

www.exsent.com

Second Edition – October 2001

© 2001 Exsent Inc. All rights reserved.

PinPort is a trademark of Exsent Inc.

Table of Contents

- 1 Overview.....4
- 2 Accessing the PinPort PLI.....5
- 3 PLI Functionality.....6
- 4 Using PinPort PLI.....7
 - Arguments.....7
 - Returned Value.....7
 - Device Addressing.....7
 - Suggested Use.....7
 - Setup.....7
 - Event driven updates.....8
 - Releasing PinPort PLI resources.....8
- 5 Using the Sample Files.....10
- 6 PinPort PLI Function Summary.....11
 - PinPort DLL Initialization and Configuration Functions.....11
 - \$exsent_init(i_options).....11
 - \$exsent_uninit().....11
 - \$exsent_get_sw_version().....11
 - PinPort Device Information and Configuration Functions.....11
 - \$exsent_inquire(i_which_pinport, r_reply).....11
 - \$exsent_get_hw_id(i_which_pinport, r_reply).....11
 - \$exsent_get_hw_version(i_which_pinport).....11
 - \$exsent_get_hw_size(i_which_pinport).....12
 - \$exsent_size_request(i_which_pinport, i_bitsToUse).....12
 - \$exsent_cfg_stb_1(i_which_pinport, i_prescaler, i_frontPorch, i_chimney, i_backPorch, i_polarity, i_strobelogic).....12
 - \$exsent_cfg_stb_2(i_which_pinport, i_prescaler, i_frontPorch, i_chimney, i_backPorch, i_polarity, i_strobelogic).....12
 - \$exsent_cfg_pin_1(i_which_pinport, r_out_to_hw).....12
 - \$exsent_cfg_pin_2(i_which_pinport, r_out_to_hw).....12
 - Pin Data Transfer Functions.....13
 - \$exsent_write_then_read(i_which_pinport, r_in_from_hw, r_out_to_hw, r_out_ena_hw).....13
 - \$exsent_write(i_which_pinport, r_out_to_hw, r_out_ena_hw).....13
 - \$exsent_read(i_which_pinport, r_in_from_hw).....13
 - \$exsent_write_pin_outputs(i_which_pinport, r_out_to_hw).....13
 - \$exsent_write_pin_enables(i_which_pinport, r_out_to_hw).....13
 - PinPort Status Reporting Functions.....13
 - \$exsent_get_tx_errors(i_which_pinport).....13
 - \$exsent_get_rx_errors(i_which_pinport).....14
 - \$exsent_read_cycle_count(i_which_pinport).....14
 - \$exsent_write_cycle_count(i_which_pinport, i_out_to_hw).....14
- 7 Returned Error Codes.....15

1 Overview

The Exsent PinPort devices are a series of versatile multi-pin hardware interfaces that may be connected to your computer by means of SCSI cabling in order to provide direct access to physical hardware pins from your Verilog simulator application.

Your Verilog model may contain one or more modules that interact with Exsent PinPort devices by using Verilog "PLI" language extensions provided by Exsent. This document describes the use of these language extensions.

The Exsent PLI extensions are provided in the form of a dynamically linked program library, usually named *pinport.dll*. The Model Technologies ModelSim simulator can be configured to locate and add these PinPort extensions to your model environment so that you can use them in modules that you write.

2 Accessing the PinPort PLI

This document assumes that you have already followed the instructions that came with your PinPort devices, their cabling and terminators, have correctly installed the needed SCSI interface and its drivers, and have already installed the PinPort PLI software into your computer.

If you have not done these things yet, refer to the appropriate installation instructions and complete these steps before you attempt to proceed with the following.

Before you can use the PinPort PLI extensions you need to make sure that the *modelsim.ini* file used to control the options for your ModelSim simulator sessions correctly describes the location of the PinPort PLI library. Here is one way to do that:

- Locate your *modelsim.ini* file and open it using a text editor.
- Find the portion of that file containing commented lines which look something like this:

```
; List of dynamically loaded objects for Verilog PLI applications  
; Veriuser=veriuser.s1
```

- After those lines add the following line:

```
veriuser=C:/Program Files/exsent/pinport/pinport.dll
```

This assumes that the location of the Exsent folder is in *C:\Program Files\exsent\pinport*. If you have installed your Exsent files somewhere else, change this line to reflect the actual path.

You must also ensure that you have correctly installed the matching security authorization key files for your devices and that they are currently valid. Remember that one of these key files needs to be present for each of your PinPort devices and that it must match the serial number of the devices you are trying to use.

3 PLI Functionality

Once all the hardware and the dynamic link library are correctly installed and operating, ModelSim will automatically make the Exsent PinPort PLI extensions available for use by your model. These extensions are a set of 19 functions that may be used by your model to communicate with and control PinPort devices. They allow you to configure and control the PLI library itself and each of the attached devices. Functions are provided that allow your model to:

- initialize the PLI module to prepare it for use and to release any system resources it uses when they are no longer needed;
- configure each of your PinPort devices and report on their capabilities;
- specify automatically timed strobe generation for the output pins of each of those devices;
- transfer pin vectors between your model and each of your PinPort devices to update their pin values or read them back;
- and verify the integrity of data transfers between your computer and PinPort devices.

General advice about using these functions can be found in the next section of this document. A detailed description of each function and its input and output arguments is also available and can be found in the section titled "PinPort PLI Summary"

4 Using PinPort PLI

Arguments

PinPort PLI functions take standard Verilog arguments where needed in the form of Verilog Integer or Register values as appropriate. They always return Integer values as the value of the function.

In cases such as that of **\$exsent_read**, where the result of the function cannot be completely expressed as a single positive integer, the result is returned in a Verilog register object that must be supplied by the caller as one of the arguments to the function. The supplied register must be large enough to hold the expected result or an error will occur and the result contained in this register will be undefined.

This makes it extremely important to check that every function called returns a successful result code (as described below), since failure to detect an error at the point at which it occurs can cause confusing behavior that can be difficult to troubleshoot.

Returned Value

In most cases, the integer value returned by the PLI call indicates whether the operation succeeded or failed, and if it failed, an error code is returned to assist troubleshooting. Refer to the PinPort PLI Summary for a list of these codes and their meanings.

The returned value is always negative in the case of an operational error. This allows zero or a positive value to be returned as a simple indication of success or a positive integer value in cases where that is sufficient to convey any numeric data that the function needs to return.

Remember that the result code merely indicates that the mechanics of the operation succeeded or failed, but there are times when you need to worry about other kinds of errors.

For example. The **\$exsent_init** function returns the number of PinPort devices attached to the system if it succeeds, or an error code if it fails. If no devices are detected, 0 will be returned - indicating that no devices were found. Since this is a positive value, it is an indication that the function operated properly. The information it returned, however is a sign that your model cannot proceed, since no available PinPort devices were found!

Device Addressing

Operations that affect PinPort devices need to specify which device they are intended to address. They do this by passing a zero-based index value corresponding to the intended device.

That is to say: the first device is specified by the value "0", the second by the value "1", and so forth. The devices are assigned indexes at initialization, in the order in which they are found by searching through the SCSI device addresses starting from the lowest numbered device to the highest.

Suggested Use

Setup

During the initialization of your module, you would first typically call **\$exsent_init**. The single integer argument to this function is an "option control" value that can be used to turn on or off a number of useful settings affecting the behavior of the dll.

At this time you may wish to also confirm that the expected driver and devices have been found in the expected configuration and that all the proper capabilities for use with your

model. The following can be useful for these purposes: **\$exsent_get_sw_version**, **\$exsent_get_hw_id**, **\$exsent_get_hw_version**, **\$exsent_get_hw_size**,

If all goes well, you may now proceed to configure each of the PinPort devices to agree with the way that your external hardware is connected to each of them and to match its timing needs. Use **\$exsent_cfg_stb_1**, **\$exsent_cfg_stb_2**, **\$exsent_cfg_pin_1** and **\$exsent_cfg_pin_2** to configure pulsed output timing. Use **\$exsent_write_pin_enables** to configure pins as output or input as desired.

You may also wish to use the data transfer commands described below during your initialization to establish predictable initial hardware values.

Event driven updates

Typically all of the preceding operations are performed within the initialization code provided by your module. Everything that follows is typically performed within the event driven portion of your module.

Whenever your model needs to update the PinPort output pin values at the hardware connected to the PinPort device, it can do so by using the PinPort data transfer commands **\$exsent_write** (which modifies both the outputs and their corresponding enables) or by using **\$exsent_write_pin_outputs** to modify the outputs only.

Pins used as PinPort outputs may have their enable state changed at any time, either by using **\$exsent_write** (which updates both the outputs and their enables) or by using **\$exsent_write_pin_enables** to update only the enables.

Whenever your model needs to update its internal registers so that they agree with the value of PinPort input pins it can do so by using **\$exsent_read**.

Since most of the overhead associated with communication with SCSI devices is consumed on a "per-transaction" basis, minimizing the overall number of PLI operations can result in considerably improved performance. You can efficiently update your hardware under test by using the **\$exsent_write_then_read** function which allows updating output pins and their associated enables and also returning the input pin values in a single PLI operation. Judicious use of this function can considerably improve model performance.

Always check return codes from PLI functions and act upon them since a failed operation can cause confusing behavior later, especially if your hardware is state driven.

Because problems with your SCSI hardware can't always be reported as communication errors, you may wish to confirm that you are not experiencing transmission errors by using the explicit functions **\$exsent_get_tx_errors** and **\$exsent_get_rx_errors**, which record communication errors.

No such errors should ever occur. If these counters indicate any value except zero, your SCSI card, cabling, termination or drivers are not operating properly. These counters are normally not needed, but have been provided so that you can insert code into your model to confirm the communication if you suspect that it is not reliable for some reason.

Remember that in any case, communications failures result in properly reported errors returned by the operation that depends upon it - although the error code returned may indicate that the intended purpose of the operation has failed, rather than that the underlying communication caused the failure.

Releasing PinPort PLI resources

If your module has any reason to release the PinPort hardware resources (the PinPort boxes and their operating system interfaces) when they are no longer needed, so that other programs can use them without conflict it may do so by calling the **\$exsent_uninit** function.

That is the only time this function would normally need to be used. Once this operation has been performed, your model may no longer make use of other PLI functions unless **\$exsent_init** is once more called.

5 Using the Sample Files

To get you started, we have provided a few simple samples in files in the subdirectories called *test*, *examples*, and *tutorials*. These have been installed in the directory where your *pinport.dll* function is located. The sample files are obviously not useful without adding more detail to them, but they have been kept simple in order to make them easy to understand.

Serious models will require complete error handling and more realistic input and output handling. Of course, practical modules would also need to be designed so that their input and output bit assignments agree with the actual wiring of the hardware you intend to connect to your PinPort devices.

1. *tutorial/exersize.v* is a complete file describing a typical interface to a physical device wired to a prototyping board installed in a PinPort 64. It provides a fairly simple, but complete real-world use of the PinPort and is an excellent template to use for building your own models.
2. *examples/utilities/util_01.v* is a simple set of utility operations that provides some useful routines which you may find helpful in structuring your own models.
3. Although not intended as an example, the file *test_01.v* included in the subdirectory named *test* is an extremely complete use of the Exsent PinPort PLI interface to communicate with the PinPort for purposes of confidence testing your PinPort. Although you are unlikely to ever need to model a PinPort device with no external hardware plugged into it's IO pins, it is still a gold mine of useful code fragments demonstrating basic techniques.

6 PinPort PLI Function Summary

PinPort DLL Initialization and Configuration Functions

\$exsent_init(i_options)

Initializes and resets the PLI code and searches the SCSI bus for installed PinPort devices. For each device found, all appropriate identification information is collected and the device interface is prepared for use. In the current PLI revision, the device hardware registers are NOT modified by this command.

On success, returns a positive integer equal to the number of PinPort Devices found on the appropriate SCSI bus. On failure, returns a negative value which is an error code.

Possible decimal option argument values which can be added together to enable their respective functions are:

- (+1) Don't print ANYTHING from the PLI code
- (+2) Print more verbose trace information if tracing is enabled
- (+4) Print trace messages during PLI function execution.

Or, you may prefer to treat these as hexadecimal values and to think of them as option control "flags".

\$exsent_uninit()

Disables PLI and returns any OS resources it is consuming to an unallocated state. (Releasing any hardware locks). On success, returns zero. On failure, returns a negative value which is an error code.

\$exsent_get_sw_version()

On success, returns a positive integer indicating the current software revision level as a packed series of hexadecimal digits. On failure, returns a negative value which is an error code.

PinPort Device Information and Configuration Functions

\$exsent_inquire(i_which_pinport, r_reply)

On success, returns 0 and updates r_reply to contain a copy of the IEEE SCSI inquiry command returned by the PinPort Device whose index is i_which_pinport. r_reply needs to be at least 800 bits long. On failure, returns a negative value which is an error code.

\$exsent_get_hw_id(i_which_pinport, r_reply)

On success, returns 0 and updates r_reply to contain a copy of the ASCII hardware ID string returned by the PinPort Device whose index is i_which_pinport. r_reply needs to be at least 136 bits long. On failure, returns a negative value which is an error code.

\$exsent_get_hw_version(i_which_pinport)

On success, returns a positive integer indicating the current hardware revision level of the PinPort Device whose index is i_which_pinport as a packed series of hexadecimal digits. On failure, returns a negative value which is an error code.

\$exsent_get_hw_size(i_which_pinport)

On success, returns a positive integer indicating the current effective size in bits of the PinPort Device whose index is `i_which_pinport`. On failure, returns a negative value which is an error code. Note that the size will be the smaller of the natural size of the hardware device, or the value set by any previous use of the `$exsent_size_request` function.

\$exsent_size_request(i_which_pinport, i_bitsToUse)

On success 0 and sets the current effective width of the PinPort Device whose index is `i_which_pinport` to the number of bits passed in the integer `i_bitsToUse`. This cannot be larger than the actual width of the hardware device . On failure, returns a negative value which is an error code.

This function can be useful when writing models that are independent of whether they have been created for use with PinPort 64, 192, or wider PinPort models.

\$exsent_cfg_stb_1(i_which_pinport, i_prescaler, i_frontPorch, i_chimney, i_backPorch, i_polarity, i_strobelogic)

On success, returns 0 and sets strobe generator 1 to the count values specified by the integer arguments. On failure, returns a negative value which is an error code.

The prescaling and timing values are passed as integers and can range from values of one to 16 counts of the basic timing

The strobe logic argument is passed, encoded as follows:

0 : output bit "anded" with strobe 1

1 : output bit "ored" with strobe 1

2 : output bit "xored" with strobe 1

\$exsent_cfg_stb_2(i_which_pinport, i_prescaler, i_frontPorch, i_chimney, i_backPorch, i_polarity, i_strobelogic)

On success, returns 0 and sets strobe generator 2 to the count values specified by the integer arguments. On failure, returns a negative value which is an error code.

The prescaling and timing values are passed as integers and can range from values of one to 16 counts of the basic timing

The strobe logic argument is passed, encoded as follows:

0 : output bit "anded" with strobe 1

1 : output bit "ored" with strobe 1

2 : output bit "xored" with strobe 1

\$exsent_cfg_pin_1(i_which_pinport, r_out_to_hw)

On success, returns 0 and assigns the output pins of the PinPort Device whose index is `i_which_pinport` corresponding to the bits in `r_out_to_hw` to obey the strobe timing currently specified as `strobe1`. On failure, returns a negative value which is an error code.

\$exsent_cfg_pin_2(i_which_pinport, r_out_to_hw)

On success, returns 0 and assigns the output pins of the PinPort Device whose index is

`i_which_pinport` corresponding to the bits in `r_out_to_hw` to obey the strobe timing currently specified as `strobe1`. On failure, returns a negative value which is an error code.

Pin Data Transfer Functions

\$exsent_write_then_read(`i_which_pinport`, `r_in_from_hw`, `r_out_to_hw`, `r_out_ena_hw`)

Transfers the pin values found in `r_out_to_hw` and `r_out_ena_hw` respectively to the corresponding output and output enable bits of the PinPort Device whose index is `i_which_pinport` and then transfers the pin values found on the pins of that device to the register `r_in_from_hw`. On success, returns a positive integer or zero. On failure, returns a negative value which is an error code.

The number of bits transferred can be restricted by the current hardware size setting.

\$exsent_write(`i_which_pinport`, `r_out_to_hw`, `r_out_ena_hw`)

Transfers the pin values found in `r_out_to_hw` and `r_out_ena_hw` respectively to the corresponding output and output enable bits of the PinPort Device whose index is `i_which_pinport`. On success, returns a positive integer or zero. On failure, returns a negative value which is an error code.

The number of bits transferred can be restricted by the current hardware size setting.

\$exsent_read(`i_which_pinport`, `r_in_from_hw`)

Transfers the pin values found on the pins of the PinPort Device whose index is `i_which_pinport` to the register `r_in_from_hw`. On success, returns a positive integer or zero. On failure, returns a negative value which is an error code.

The number of bits transferred can be restricted by the current hardware size setting.

\$exsent_write_pin_outputs(`i_which_pinport`, `r_out_to_hw`)

Transfers the pin values found in `r_out_to_hw` to the corresponding output bits of the PinPort Device whose index is `i_which_pinport`. On success, returns a positive integer or zero. On failure, returns a negative value which is an error code.

The number of bits transferred can be restricted by the current hardware size setting.

\$exsent_write_pin_enables(`i_which_pinport`, `r_out_to_hw`)

Transfers the pin values found in `r_out_to_hw` to the corresponding output enable bits of the PinPort Device whose index is `i_which_pinport`. On success, returns a positive integer or zero. On failure, returns a negative value which is an error code.

The number of bits transferred can be restricted by the current hardware size setting.

PinPort Status Reporting Functions

\$exsent_get_tx_errors(`i_which_pinport`)

On success, returns a positive integer representing the number of data communication errors detected since `$exsent_init` was last called on transmissions to the the PinPort device whose index is `i_which_pinport`. On failure, returns a negative value which is an error code.

\$exsent_get_rx_errors(i_which_pinport)

On success, returns a positive integer representing the number of data communication errors detected since **\$exsent_init** was last called on transmissions from the the PinPort device whose index is *i_which_pinport*. On failure, returns a negative value which is an error code.

\$exsent_read_cycle_count(i_which_pinport)

On success, returns a positive integer representing the value of the cycle counter for the PinPort device whose index is *i_which_pinport*. On failure, returns a negative value which is an error code.

\$exsent_write_cycle_count(i_which_pinport, i_out_to_hw)

Sets the operation cycle counter for the PinPort device whose index is *i_which_pinport* to the value *i_out_to_hw*. Returns zero if succeeds or a negative integer error code if it fails.

7 Returned Error Codes

These are the currently documented error codes. Any other negative integer values should be treated as "undefined" errors, but still considered as serious errors.

-3000	Register error
-2999	CRC error
-2998	Serial number error
-2997	Eeprom access error
-2000	Hardware / PLI version incompatibility
-1999	Verilog Simulator / PLI incompatibility
-1998	Hardware / bit width request incompatibility
-1997	Incorrect values or number of values presented to PLI
-1996	Overflow error (rollover) in cycle counter
-1995	Argument size error, (out of range)
-1994	Authorization error
-1993	Duplicate serial number error
-1992	Missing key file error
-1000	No SCSI WinASPI detected
-999	No SCSI Host adapter found
-998	Incorrect values or number of values presented to SCSI
-997	Incorrect PinPort device ID
-996	SCSI transport error, host adapter, cable, or PinPort communication failure
-995	Host adapter failure
-994	SCSI PinPort reset error
-993	PinPort device unavailable - in use by another software process

