

PinPort192™ Reference Manual



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

Version 3.02 – August 2002

© 2000-2002 Exsent Inc. All rights reserved.

PinPort is a trademark of Exsent Inc.

Table of Contents

1 SCOPE.....	4
1.1 Introduction.....	4
2 PINPORT PINS.....	5
2.1 Pin Connector Assignments.....	5
2.2 Pin Argument Bit Assignments.....	5
3 PINPORT PROGRAMMING INTERFACE.....	6
3.1 API Initialization Routines.....	6
3.2 API Configuration Routines.....	8
3.3 API Operational Routines.....	11
3.3.1 Operational Call Example.....	12
3.3.2 Pin Updates and Strobes.....	12
3.4 API Status Routines.....	13
3.5 API Error Codes.....	14
3.6 Utility Routines.....	14
4 BASIC PIN FIRMWARE.....	16
4.1 Application / PinPort Handshake.....	16
4.2 Strobe Algorithm.....	17
4.2.1 Strobe Configuration.....	17
4.2.2 Pin-Strobe Association.....	18
4.3 Pin Enabling and Sampling.....	19
5 INSTALLATION REFERENCE.....	22
5.1 PinPort SCSI Interface.....	22
5.2 Key Files.....	22
6 ELECTRICAL SPECIFICATIONS.....	23
6.1 Clock Frequency.....	23
6.2 Electrical Spec for Pins.....	23
6.3 Electrical Spec for Supply.....	23
6.4 Adapter Board.....	23

Copyright, 2000-2002 Exsent Inc.

PinPort is a trademark of Exsent Inc.

1 SCOPE

This document describes the software, firmware, and hardware features of the PinPort192 software / hardware interface product. This includes the application programming interface (API), basic pin firmware, electrical specifications, license features, and the general operation of the device.

1.1 Introduction

The PinPort family of devices provides a way for a design, verification, or test engineer to communicate with and control a hardware device through a software application, like a logic simulator or a standalone C program. A typical use is system verification. The engineer mounts one or more chips (the hardware sub-system) on an adapter board which plugs into the PinPort device. Physical pins are assigned at this time. To serve as an interface between the simulator and the hardware subsystem, the engineer creates a simple Verilog model, encapsulating the hardware. This allows the engineer to include the hardware in the system level simulation.

A software application running on a workstation can read or write the signal pins on the PinPort adapter board. The PinPort device provides a port on the workstation through which the pin accesses occur. The product name "PinPort" comes from combining the concepts of a "Pin" on an adapter board and a "Port" on a workstation through which it is accessed,

2 PINPORT PINS

2.1 Pin Connector Assignments

The PinPort192 has a single connector. These pins are driven by a 3.3 volt buffer, which is 5.0 volt tolerant and TTL compliant. Each pin can be configured as an input, output, or bi-direct.

2.2 Pin Argument Bit Assignments

The API calls which relate to the configuration and operational routines discussed in the next section are `exsent_cfg_pin_1`, `exsent_cfg_pin_2`, `exsent_write`, `exsent_write_pin_outputs`, `exsent_write_pin_enables`, `exsent_read` and `exsent_write_then_read`. These API calls take one or more vectors, with each bit in the vector corresponding to a PinPort pin. The mapping of bits in the vector to pins on the connector is the same for all of these calls. An example of the `exsent_write_pin_outputs` API call for a PinPort64 device is shown in figure 2.1. The code is shown for both Verilog and C. Note that the least significant bit corresponds to pin “dev1” and the most significant bit corresponds to pin “dev64”. This same least-significant bit to most-significant bit numbering scheme is used for the PinPort192 device.

The C utility routine `exsent_string_to_bits` places the eight least significant bits in the byte at the address given by the first argument, the next eight least significant bits in the byte at the address following this, and so on. This is a little-endian byte organization. If the `exsent_string_to_bits` routine is not used, then the bytes must be explicitly prepared using this convention.

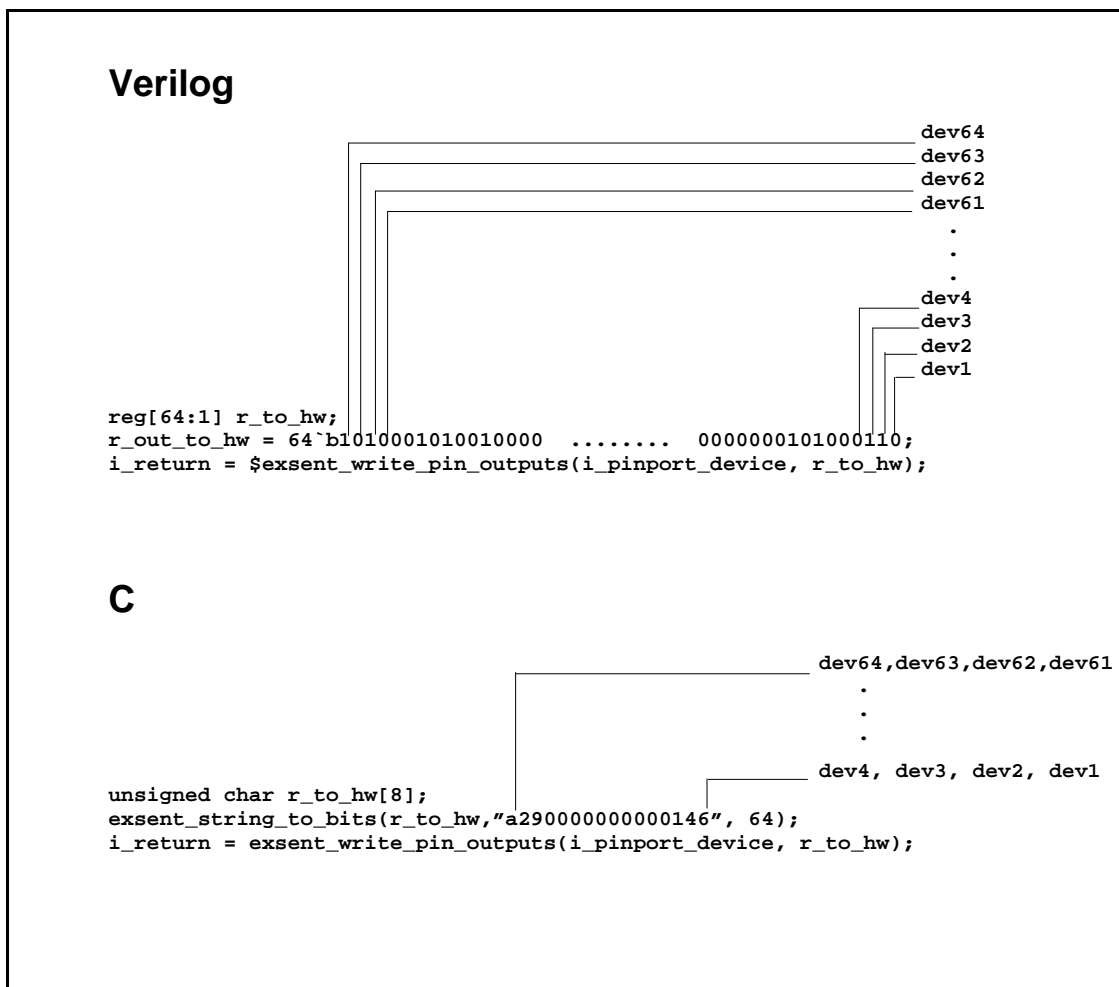


Figure 2.1 Mapping of Vector Bits to Connector Pins

3 PINPORT PROGRAMMING INTERFACE

Access to the PinPort device, and thus its pins, is accomplished through the application programming interface (API). The API is available in both the C programming language as functions and as system tasks in the Verilog hardware description language. The functionality of the API is identical and the arguments to the routines nearly the same, in both C and Verilog. To access the PinPort pins from a host computer application (e.g. a Verilog logic simulator) there are several API routines which can be called. These are comprised of initialization, configuration, operational, and status routines.

To make it easier to use the API, particularly with C, a set of utility routines are also provided. These routines are not part of the API proper, since they are not required when interacting with a PinPort device. They exist for convenience sake and are included with the PinPort software in separate files.

The API initialization routines are listed in table 3.1. API Configuration routines are listed in table 3.2. The API operational routines are listed in table 3.3. API status routines are listed in table 3.4. The error codes returned by the API routines are enumerated in table 3.5. Utility routines are listed in table 3.6. Typically, the initialization and configuration routines are done once at the start of simulation, whereas the operational routines are done each time the pin states change, or are read in from a PinPort device.

3.1 API Initialization Routines

Table 3.1 shows the initialization routines. The information is provided for both the Verilog and the C implementations of the API. Note that a "\$" needs to precede a routine name when calling it in Verilog as a system task.

These routines are invoked before interacting with a PinPort device. These are usually invoked only once at the start of the software application. For example, the initialization routines are often called from an initial block within a Verilog module which is part of the system simulation.

Initialization Calls	Values Passed To API		Values Passed From API		Description	
	argument					
	name	Verilog	C	Verilog	C	
exsnt_init	i_options	integer	long			Initialize the PinPort This call provides options for the PinPort API routines. bit 0 : suppress messages printed by API bit 1 : verbose mode bit 2 : trace printing bit 3 : trace messages during API function execution bit 4 - bit 9 : internal use bit 10 - bit 17 : exclude bits; when set the corresponding PinPort is not initialized; bits 10-17 are the exclude bits for PinPorts with SCSI ids 0-7, respectively bit 18 - bit 28 : reserved bit 29 - bit 31 : internal use
	return			integer	int	Returns the number of PinPort devices found on the SCSI bus. A negative value indicates an error.
exsnt_uninit				integer	int	Disables the API and returns any OS resources it is consuming to an unallocated state. 0 indicates success. A negative value indicates an error.
exsnt_get_sw_version				integer	int	Get the software version and revision. Returns a positive integer indicating the current software revision level as a packed series of binary-coded decimal digits. A negative value indicates an error.

<p><code>exsnt_get_hw_version</code></p> <p><code>i_which_pinport</code> integer <code>exsnt DevIdType</code></p> <p>return integer int</p>	<p>Get the hardware version and revision. The PinPort device to access.</p> <p>Returns a positive integer indicating the current hardware revision level as a packed series of binary-coded decimal digits. A negative value indicates an error.</p>
<p><code>exsnt_get_hw_size</code></p> <p><code>i_which_pinport</code> integer <code>exsnt DevIdType</code></p> <p>return integer int</p>	<p>Get the current effective size of the PinPort device. This is the smaller of the size supported by the hardware and the value set by the last call to <code>exsnt_size_request</code>. The PinPort device to access.</p> <p>Returns the current effective size, (bit width - i.e 64, 192, 384, etc.) of the PinPort device. A negative value indicates an error.</p>
<p><code>exsnt_get_fw_function_count</code></p> <p><code>i_which_pinport</code> integer <code>exsnt DevIdType</code></p> <p>return integer int</p>	<p>The PinPort device to access.</p> <p>Returns the number of firmware functions loaded into the PinPort device. A negative value indicates an error.</p>
<p><code>exsnt_get_fw_type</code></p> <p><code>i_which_pinport</code> integer <code>exsnt DevIdType</code></p> <p><code>i_which_fw</code> integer <code>exsnt FwIdType</code></p> <p>return integer int</p>	<p>The PinPort device to access.</p> <p>The firmware function for which information is requested.</p> <p>Returns a positive integer indicating the type of the firmware as a packed series of binary-coded decimal digits. A negative value indicates an error.</p>
<p><code>exsnt_get_fw_version</code></p> <p><code>i_which_pinport</code> integer <code>exsnt DevIdType</code></p> <p><code>i_which_fw</code> integer <code>exsnt FwIdType</code></p> <p>return integer int</p>	<p>The PinPort device to access.</p> <p>The firmware function for which information is requested.</p> <p>Returns a positive integer indicating the current firmware function revision level as a packed series of binary-coded decimal digits. A negative value indicates an error.</p>

Table 3.1 Initialization Routines

The `exsnt_init` API call initializes the PinPort devices and enables communications between the software application and the PinPort devices over the SCSI bus. For each device found, all appropriate identification information is collected and the device interface is prepared for use. The user can also enable certain verbose messaging. Messages can be generated as the result of API operations.

The `exsnt_uninit` API call uninitializes the API by returning OS resources it is consuming to an unallocated state. This releases any hardware locks associated with the PinPort device.

The `exsnt_get_sw_version` API call returns a positive integer representing the current software revision level as a packed series of binary-coded decimal digits. This is for informational purposes and is not required for operation.

The `exsnt_get_hw_id` API call returns the serial number of PinPort device hardware as a 17 character string (16 alpha-numeric digits followed by a null character). This is for informational purposes and is not required for operation.

The `exsnt_get_hw_version` API call returns a positive integer representing the current hardware revision level as a packed series of binary-coded decimal digits. This is for informational purposes and is not required for operation.

The `exsent_get_fw_function_count` API call returns a positive integer indicating the number of firmware functions loaded into the PinPort device. The PinPort device is loaded with one or more firmware functions whenever `exsent_init` is called. This call is for informational purposes and is not required for operation.

The `exsent_get_fw_type` API call returns a positive integer representing the type of the indicated firmware function as a packed series of binary-coded decimal digits. For the basic pin firmware that comes standard with every PinPort this is 3842. This is for informational purposes and is not required for operation.

The `exsent_get_fw_version` API call returns a positive integer representing the current revision level of the indicated firmware function as a packed series of binary-coded decimal digits. This is for informational purposes and is not required for operation.

3.2 API Configuration Routines

Table 3.2 shows the configuration routines for the basic pin firmware of the PinPort device. The configuration routines are usually invoked at the start of verification unless the strobe parameters or the association of pins to strobos needs to change during verification.

Configuration Calls		Values Passed To API		Values Passed From API		Description
argument name	Verilog	C	Verilog	C		
exsent_size_request						Set the current effective width of the PinPort device. The PinPort device to access.
i_which_pinport	integer	exsent DevIdType				
i_bits_to_use	integer	long				This value represents the desired number of bits to access with the PinPort device. It can be equal to or less than the number of pins on the PinPort.
return			integer	int		0 indicates success. A negative value indicates an error.
exsent_cfg_stb_1						Configure strobe 1. The PinPort device to access.
i_which_pinport	integer	exsent DevIdType				
i_prescaler	integer	long				Strobe 1's prescale counter value.
i_front_porch	integer	long				Strobe 1's front porch counter value.
i_chimney	integer	long				Strobe 1's chimney counter value.
i_back_porch	integer	long				Strobe 1's back porch counter value.
i_polarity	integer	long				A value of 1 makes the internal strobe 1 an active low signal prior to gating (output generation).
i_strobe_logic	integer	long				0 : pin output "anded" with strobe 1. 1 : pin output "ored" with strobe 1. 2 : pin output "xored" with strobe 1. 3 : pin input sampled at the end of the chimney section of strobe 1.
return			integer	int		0 indicates success. A negative value indicates an error.
exsent_cfg_stb_2						Configure strobe 2. The PinPort device to access.
i_which_pinport	integer	exsent DevIdType				
i_prescaler	integer	long				Strobe 2's prescale counter value.
i_front_porch	integer	long				Strobe 2's front porch counter value.
i_chimney	integer	long				Strobe 2's chimney counter value.
i_back_porch	integer	long				Strobe 2's back porch counter value.
i_polarity	integer	long				A value of 1 makes the internal strobe 2 an active low signal prior to gating (output generation).
i_strobe_logic	integer	long				0 : pin output "anded" with strobe 2. 1 : pin output "ored" with strobe 2. 2 : pin output "xored" with strobe 2. 3 : pin input sampled at the end of the chimney section of strobe 2.
return			integer	int		0 indicates success. A negative value indicates an error.
exsent_cfg_pin_1						Configure each pin for strobe 1. The PinPort device to access.
i_which_pinport	integer	exsent DevIdType				
	reg[192:1]	exsent ByteType[24]				Bits determine the pin configuration as it relates to strobe 1. A '1' in a bit position means that the corresponding pin - when it is written and enabled - is "anded", "ored", or "xored" with strobe 1. Further, it means that the value read from the pin reflects the value at the end of the chimney section of the strobe. A '0' indicates that no strobe 1 gating should occur when writing to the pin. Also, '0' means the value read is the value at the end of the operational access, unless it is associated with strobe 2.
return			integer	int		0 indicates success. A negative value indicates an error.
exsent_cfg_pin_2						Configure each pin for strobe 2. The PinPort device to access.
i_which_pinport	integer	exsent DevIdType				
	reg[192:1]	exsent ByteType[24]				Bits determine the pin configuration as it relates to strobe 2. A '1' in a bit position means that the corresponding pin - when it is written and enabled - is "anded", "ored", or "xored" with strobe 2. Further, it means that the value read from the pin reflects the value at the end of the chimney section of the strobe. A '0' indicates that no strobe 2 gating should occur when writing to the pin. Also, '0' means the value read is the value at the end of the operational access, unless it is associated with strobe 1.
return			integer	int		0 indicates success. A negative value indicates an error.

Both the `exsent_cfg_stb_1` and `exsent_cfg_stb_2` API calls have the same argument definitions. Since there are two strobes, each strobe configuration applies only to that specific strobe. Both strobes are independent. When the write access to the pin output registers is complete, the strobe period of the operational access is entered for both strobes. If a strobe hasn't been configured, there is no strobe period. If no strobe has been configured, the outputs, enables, and inputs are updated immediately.

A strobe period is comprised of three sections. The first is the front porch, the second is the chimney, and the third is the back porch. Since there are two independent strobes, these sections could be of different lengths. Both strobes must complete all three sections before the operational access to the PinPort device is complete.

The duration of all three strobe sections as well as the function used during output generation is separately configurable for each strobe.

A prescale counter is used to pace three other counters: front porch, chimney, and back porch. All of the counters are four-bit counters. The prescale counter acts as a modulo counter which will count from the value represented by bits 3:0, down to a value of 1, at which time it reloads the prescale count value, (bits 3:0). This counter operates from the main PinPort system clock. Thus if a value of 3 is passed, the other counters will count once every 3 main PinPort clocks. Likewise, a value of 2 will mean the other counters will count every other PinPort main clock, and a value of 1 will mean the counters count every PinPort main clock.

During the front porch section of the strobe period the front porch counter delays the application of both the pin output registers and pin output enable registers to the pins. Once the front porch counter has reached its terminal count of 0, then the chimney section of the strobe period is entered.

The duration of the chimney section is determined by the chimney counter. The chimney counter starts to count when the front porch counter has reached its terminal count. When the chimney counter has reached its terminal count of 0, then the back porch section of the strobe period is entered.

The duration of the back porch section is determined by the back porch counter. The back porch counter starts to count when the chimney counter has reached its terminal count. When the backporch counter has reached its terminal count of 0, the strobe period is ended for the strobe.

Once both strobe periods complete, the operational access is complete. The 2nd through 5th arguments to the `exsent_cfg_stb_1` or `exsent_cfg_stb_2` routines define the duration of each section of a strobe period.

Both strobes act in the same way, but are individually programmed with unique values. Strobes are used to generate clocks to devices, where the setup time is determined by the front porch value. The duration of the clock is determined by the chimney. Using different values for strobes 1 and 2 can provide more sophisticated clock generation and input sampling.

The internal strobe is inverted by passing a 1 to the 6th argument of the `exsent_cfg_stb_1` or `exsent_cfg_stb_2` routine.

A pin output can have 3 operations performed with relationship to the strobes. If the 7th argument passed to these strobe configuration commands is a 0, the internal strobe is "anded" with the register output value. If it is a 1, the internal strobe is "ored" with the register output value. If it is a 2, the internal strobe is "xored" with the register output value. The latching of a pin input is affected by a strobe if the 7th argument to the strobe configuration commands is a 3. When it is a 3, the latching occurs at the end of the chimney section of the strobe, rather than at the end of the backporch.

The `exsent_cfg_pin_1` API call enables strobe 1 to modify the input or output characteristics of a pin when the corresponding bit location is a '1'.

The `exsent_cfg_pin_2` API call enables strobe 2 to modify the input or output characteristics of a pin when the corresponding bit location is a '1'. This requires that the corresponding bit location for the strobe 1 pin enable is a '0', since only one strobe can modify a pin's function.

3.3 API Operational Routines

Table 3.3 shows the operational routines for the basic pin firmware of the PinPort device

The operational routines are used to initiate a write or a read of pins. The operational routines transform the data during the access. The number of bits transferred is restricted by the current effective hardware size as set by `exsent_size_request`.

Operational Calls	argument name	Values Passed To API		Values Passed From API		Description
		Verilog	C	Verilog	C	
exsent_write	i_which_pinport	integer	exsent DevIdType			write the pin outputs and pin output enables. The PinPort device to access.
	r_out_to_hw	reg[192:1]	exsentByte Type[24]			These bits set the output values for pins 1 to 192.
	r_out_ena_hw	reg[192:1]	exsentByte Type[24]			These bits enable the outputs for pins 1 to 192, (1 = enable, 0 = high impedance).
	return			integer	int	0 indicates success. A negative value indicates an error.
exsent_write_pin_outputs	i_which_pinport	integer	exsent DevIdType			write the pin outputs. The PinPort device to access.
	r_out_to_hw	reg[192:1]	exsentByte Type[24]			These bits set the output values for pins 1 to 192.
	return			integer	int	0 indicates success. A negative value indicates an error.
exsent_write_pin_enables	i_which_pinport	integer	exsent DevIdType			write the pin output enables. The PinPort device to access.
	r_out_ena_hw	reg[192:1]	exsentByte Type[24]			These bits enable the outputs for pins 1 to 192, (1 = enable, 0 = high impedance).
	return			integer	int	0 indicates success. A negative value indicates an error.
exsent_read	i_which_pinport	integer	exsent DevIdType			Read the pin inputs. The PinPort device to access.
	r_in_from_hw			reg[192:1]	exsentByte Type[24]	These bits indicate the state of pins 1 to 192.
	return			integer	int	0 indicates success. A negative value indicates an error.
exsent_write_then_read	i_which_pinport	integer	exsent DevIdType			write the pin outputs and pin output enables, then read the pin inputs. The PinPort device to access.
	r_in_from_hw			reg[192:1]	exsentByte Type[24]	These bits indicate the state of pins 1 to 192.
	r_out_to_hw	reg[192:1]	exsentByte Type[24]			These bits set the output values for pins 1 to 192.
	r_out_ena_hw	reg[192:1]	exsentByte Type[24]			These bits enable the outputs for pins 1 to 192, (1 = enable, 0 = high impedance).
	return			integer	int	0 indicates success. A negative value indicates an error.

Table 3.3 Operational Routines

The `exsent_write` API call drives the output pins by passing an integer identifying the PinPort device to access and two 192 bit values: one for the pin output register and another for the pin enable register. The states of those pins that are enabled are updated with their value from the output register.

The `exsent_write_pin_outputs` API call drives the output pins by passing an integer identifying the PinPort device to access and one 192 bit value for the pin output register. When the outputs are written, the states of the pins that are enabled are updated.

The `exsent_write_pin_enables` API call updates the pin enable holding register by passing an integer identifying the PinPort device to access and one 192 bit value for the pin enable holding register. Note that just the holding register is updated, not the actual pin enable register. The pin enable register is updated with the value in the holding register when the pin output values are written with an `exsent_write_pin_outputs` call,

The `exsent_write_then_read` API call is a composite of both the write and read API calls. Thus, in addition to the integer passed to identify the PinPort device to access, three 192 bit values are used, the first provides the pin output values, the second provides the pin output enable values, and the third is given the pin states read after the strobe time.

3.3.1 Operational Call Example

An example of the `exsent_write_then_read` call in Verilog

```
integer i_exsent_device;
reg[192:1] r_input_pin_value;
reg[192:1] r_output_pin_value;
reg[192:1] r_output_pin_enable;

always @(posedge clk)
    $exsent_write_then_read(i_exsent_device,
        r_input_pin_value,
        r_output_pin_value,
        r_output_pin_enable);
```

where `i_exsent_device` identifies the PinPort device (allowed values are 0 to 7), `r_output_pin_value` is a vector containing the desired output value for each pin, `r_output_pin_enable` is a vector containing the desired output enable for each pin, and `r_input_pin_value` is a vector that gets set by the routine to the detected state for each pin. This line is part of a Verilog behavioral model where the routine is called on every positive edge of the `clk` signal.

The physical pins on the PinPort have the pin output register and the pin output enable register applied to the pins at the start of the strobe activation. There is no staggered application of the pin states; they are all updated at the same time.

3.3.2 Pin Updates and Strokes

The pin outputs, enables, and inputs operate as follows. Each pin has an output and enable register. These are not directly written by the routines listed in table 3.3. These routines write pin output and enable *holding* registers instead. These holding registers are then written to the pin output and enable registers during the strobe process, if no transmission errors are detected, (see *API Status Routines* section). Writing to the output holding register with either the `exsent_write`, `exsent_write_pin_outputs`, or `exsent_write_then_read` calls initiates the strobe period.

3.4 API Status Routines

Table 3.4 shows the status routines. These are useful in determining statistics about the usage of the PinPort device. They can also alert the user to potential communications problems.

Status Calls	argument		Values Passed To API		Values Passed From API		Description
	name	Verilog	C	Verilog	C		
exsent_get_tx_errors	i_which_pinport	integer	exsent DevIDType				Get transmit error count. The PinPort device to access.
	return			integer	int		Returns the number of communication errors detected while transmitting data from the API to the PinPort, since the last call to the exsent_init routine. A negative value indicates an error.
exsent_get_rx_errors	i_which_pinport	integer	exsent DevIDType				Get receive error count. The PinPort device to access.
	return			integer	int		Returns the number of communication errors detected while receiving data from the PinPort, since the last call to the exsent_init routine. A negative value indicates an error.
exsent_write_cycle_count	i_which_pinport	integer	exsent DevIDType				Write the transaction counter. The PinPort device to access.
	i_out_to_hw	integer	long				The value to load into the transaction counter.
	return			integer	int		0 indicates success. A negative value indicates an error.
exsent_read_cycle_count	i_which_pinport	integer	exsent DevIDType				Read the transaction counter. The PinPort device to access.
	return			integer	int		Returns an integer giving the current value of the transaction counter. A negative value indicates an error.

Table 3.4 Status Routines

The `exsent_get_tx_errors` API call returns the number of errors detected in data transmitted by the API to the PinPort, since the last `exsent_init` call.

The `exsent_get_rx_errors` API call returns the number errors detected in data received by the API from the PinPort, since the last `exsent_init` call.

A software counter is incremented each time the API initiates a transaction with a PinPort. A transaction occurs when output registers are updated during a strobe period. The transaction counter increments each time an `exsent_write`, `exsent_write_pin_outputs`, or `exsent_write_then_read` API call is executed. This is a way to track the number of potential pin changes, or the number of times during the simulation that the pin outputs are written.

The `exsent_write_cycle_count` API call initializes the transaction counter with the supplied integer value.

The `exsent_read_cycle_count` API call returns the value of the transaction counter.

3.5 API Error Codes

It is recommended that the user be familiar with the basics of API usage prior to using a PinPort device. It is very important to always check the return value for the proper operation of the PinPort in the verification application. The PinPort API calls will not always alert the user to errors. Thus, by always checking the return value for errors, the correct operation of the verification is assured. Table 3.5 provides a description of each of the currently documented error codes that can be returned by API calls. Any other negative integer value should be treated as an “undefined” error, but still considered serious.

Note that the `exsent_error_trap` utility routine can be used to process values returned by API calls.

CODE	DESCRIPTION
-5000	Error retrieving value from the Verilog Programming Language Interface (PLI)
-4999	The API had not been bound for Verilog PLI. Call <code>exsent_init()</code> first before using other API functions.
-4000	Invalid number of bits argument passed to <code>exsent_inquire()</code> SPI function.
-1996	Overflow error (rollover) in cycle counter.
-1995	Invalid size argument passed to <code>exsent_size_request()</code> API function.
-1994	Authorization error. The data in the file <code>\$PINPORT_DIR/keys/<serial number>/key.dat</code> is not valid. This file is supplied by Exsent and should not be modified. Reinstall the keys.
-1993	PinPorts with duplicate serial number have been detected.
-1991	Invalid PinPort identifier argument passed to an API function. It must be greater or equal to zero and less than the number of installed PinPorts.
-1990	Invalid firmware identifier argument passed to an API function. It must be greater or equal to zero and less than the number of installed firmware functions.
-1989	Invalid prescaler argument passed to a strobe configuration API function. It must be between 0 and 15.
-1988	Invalid front porch argument passed to a strobe configuration API function. It must be between 0 and 15.
-1987	Invalid chimney argument passed to a strobe configuration API function. It must be between 0 and 15.
-1986	Invalid back porch argument passed to a strobe configuration API function. It must be between 0 and 15.
-1985	Invalid polarity argument passed to a strobe configuration API function. It must be either 0 or 1.
-1984	Invalid strobe logic argument passed to a strobe configuration API function. It must be between 0 and 3.
-1600	The directory <code>\$PINPORT_DIR/keys</code> does not exist or is not readable. Check that the <code>PINPORT_DIR</code> environment variable is set to the installation location of the PinPort software.
-1599	The directory <code>\$PINPORT_DIR/keys/<serial number></code> does not exist or is not readable. Verify that the keys you received from Exsent were placed in the correct location and that they match the serial number of your PinPort device.
-1598	The file <code>\$PINPORT_DIR/keys/<serial number>/key.dat</code> does not exist or is not readable. This file is supplied by Exsent and should not be modified. Reinstall the keys.
-1500	Problem extracting information from a load file in the keys directory.
-1499	A load file in the keys directory is either missing or can't be opened for reading.
-1498	The data read from a load file in the keys directory has been corrupted.
-1497	Unable to read from a load file in the keys directory.
-1496	The initialization of the FPGAs that hold downloaded firmware has failed.
-1495	The loading of the FPGAs that hold downloaded firmware has failed.
-1494	A status error has occurred while downloading firmware into an FPGA.
-1493	A command error has occurred while downloading firmware into an FPGA.
-1492	A data write error has occurred while downloading firmware into an FPGA.
-1491	A CRC error has occurred while downloading firmware into an FPGA.
-1000	No SCSI winASPI detected. Check the WINASPI software installation.
-999	No SCSI host adapter found. Check the installation of the SCSI adapter hardware.
-996	SCSI transport error, SCSI host adapter, SCSI cable, or PinPort communication failure.
-995	An error occurred reading data received from a PinPort.
-994	An error occurred reading the serial number from the PinPort device.
-993	A CRC error was detected in the data received from a PinPort.
-992	An error has occurred reading the PinPort EEROM.

Table 3.5 API Call Return Error Codes

3.6 Utility Routines

Table 3.6 shows the utility routines. These are useful in interacting with the API. It is possible to interact with a PinPort device without using these routines. Note that because the Verilog version of the routines are not system tasks, they are **not** preceded with a “\$” like the routines in the Verilog version of the API.

The `exsent_error_trap` utility call evaluates the return value from API routines. It is designed to be called after each API call, taking as input the return value and the identifier of the API call. When an error occurs, it prints out a message and updates an error count. Optionally, it will print out the return value and/or terminate the verification. This routine is provided for both Verilog and C.

Utility Calls	Values Passed To Routine		Values Passed From Routine		Description	
	argument name	Verilog	C	Verilog		C
	exsent_error_trap					
	i_return_value	integer	int			
	i_routine_id	integer	int			
	r_debug_level	reg[15:0]	unsigned short			
	i_err	integer	int*	integer	int*	
	return			-	void	
exsent_bits_to_string					Set a string buffer with a hexadecimal literal given by an array of bytes. An ASCII string containing the hexadecimal literal. The array of bytes.	
	r_string			-	char*	
	r_bits	-	exsentByte Type*			
	i_count	-	int			
	return			-	void	
exsent_string_to_bits					Set an array of bytes with the value given by the hexadecimal string literal. The array of bytes.	
	r_bits			-	exsentByte Type*	
	r_string	-	char*			
	i_count	-	int			
	return			-	void	
exsent_complement_bits					Complement the bits in the array of bytes. The array of bytes.	
	r_bits	-	exsentByte Type*	-	exsentByte Type*	
	i_count	-	int			
	return			-	void	
exsent_leftshift_bits					Shift the bits in the array of bytes left with the most-significant bit wrapping around. The array of bytes.	
	r_bits	-	exsentByte Type*	-	exsentByte Type*	
	i_count	-	int			
	return			-	void	

Table 3.6 Utility Routines

The **exsent_bits_to_string** utility routine provides for C, when used with the standard C library function `printf`, a subset of the capability that `$display` provides in Verilog for printing the values of registers. This routine accepts an array of bytes, with the most-significant bits in the first byte. The string of hexadecimal digits that is returned is NULL-terminated for convenient handling by `printf`. Currently, the routine expects the number of bits to be a multiple of 8. The routine is provided only in C as similar capability is built into Verilog.

The **exsent_string_to_bits** utility routine provides for C a subset of the capability that the Verilog procedural assignment does for assigning integer literals to registers. This routine accepts a string of hexadecimal digits. The array of bytes that it produces has the most-significant bits in the first byte. Currently, the routine expects string of hexadecimal digits to be NULL-terminated and the number of bits to be a multiple of 8. The routine is provided only in C as the capability is provided directly in Verilog.

The **exsent_complement_bits** utility routine operates on the bits in the array of bytes in C as the bitwise operator `~` does on the bits of an arbitrarily sized register in Verilog. It inverts each bit. Currently, the routine expects the number of bits to be a multiple of 8. The routine is provided only in C as the capability is provided directly in Verilog.

The **exsent_leftshift_bits** utility routine operates on the bits in the array of bytes in C similarly to how the bitwise operator `<<` does on the bits of an arbitrarily sized register in Verilog. It shifts each bit left one position. It differs from Verilog in that the most-significant bit wraps around to the least-significant position. Currently, the routine expects the number of bits to be a multiple of 8. The routine is provided only in C as similar capability is built into Verilog.

4 BASIC PIN FIRMWARE

The section provides more information on the operation of the basic pin firmware that is included with each PinPort device. Through the basic pin firmware the PinPort device's pins can be configured in a variety of ways.

4.1 Application / PinPort Handshake

The application (logic simulator or C program) interacts with the PinPort device via a handshake in which the passage of verification time stops while the PinPort device is executing. The operational routines of the PinPort API are blocking. Another way to look at this is that the execution of the PinPort device and the application are not concurrent.

For a Verilog simulation this means that the PinPort device execution happens in zero simulation time. This is illustrated in figure 4.1. The top of the figure shows a clock in the Verilog simulation starting at time zero. In this example, PinPort device execution occurs on a rising edge of the clock. The user determines the event which will trigger PinPort device execution. The trace for the clock signal is shown disconnected as no time is elapsing in the simulation while the PinPort device is executing.

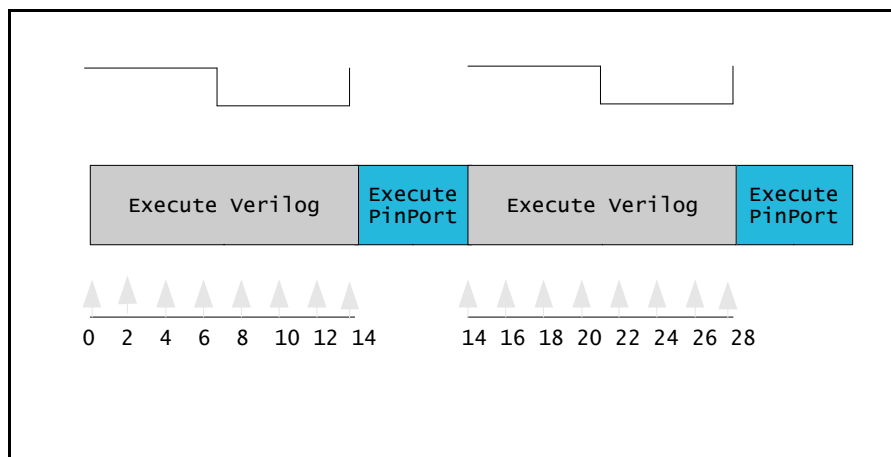


Figure 4.1 Verilog and PinPort Handshake

Input pin states are delivered to and output pin states are received from the device-under-test on the PinPort adapter card. The PinPort device applies the input states to the pins, executes detailed timing and strobe functions based on the configured strobe algorithm, then reads the state of the pins, returning the output pin values to the Verilog simulator. When execution is returned to the simulator, from the simulation point of view, the last event was the rising edge of the clock.

The inputs to and the outputs from the PinPort device can each have delays applied to them in the simulation to ensure the fidelity of the overall verification. Delays are applied to inputs before making a call to an algorithm-triggering PinPort API routine. The routines which trigger the algorithm are `exsent_write`, `exsent_write_pin_outputs`, and `exsent_write_then_read`. Output delays are added after the triggering routine returns.

4.2 Strobe Algorithm

When the pin outputs are written by making an `exsent_write`, `exsent_write_pin_outputs`, or `exsent_write_then_read` call, the algorithm which generates strobes (or clocks) is activated. A strobe is comprised of three distinct sections of time referred to as the *front porch*, *chimney*, and *back porch* as shown in figure 4.2. Each is programmable to an integral number of clocks. For the example in the figure, these are 6, 4, and 8, respectively.

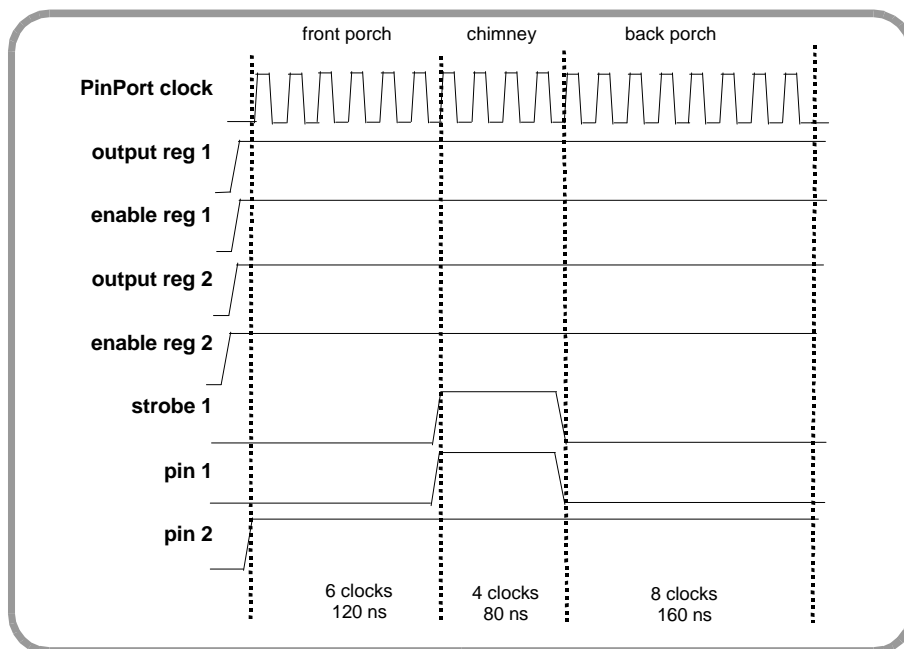


Figure 4.2 Output Strobe Example

The algorithm generates two distinct strobes: *strobe 1* and *strobe 2*. A strobe is configured to either: effect when/how a value written to a pin appears as an output, or determine when an input to a pin is latched for reading. Hence, a strobe can effect outputs or inputs, but not both.

Each pin of a PinPort device is associated with one or the other strobe (but not both) or no strobe at all. When a pin is associated with a strobe that has been configured to effect outputs, a logic function combines the pin with the strobe as part of the algorithm. The programmable logic function for a strobe is either *and*, *or*, or *xor*. When a pin is associated with a strobe that has been configured to effect inputs, the input value is latched at the end of the chimney section of the strobe period, rather than at the end of the backporch.

4.2.1 Strobe Configuration

Each strobe is configured independently. Strobe generation for strobe 1 and strobe 2 is configured by calling the `exsent_cfg_stb_1` and `exsent_cfg_stb_2` API routines, respectively. These routines have identical seven-argument, argument lists. The first argument identifies the PinPort device. The second argument is a prescale counter. It is used to scale the clock internal to the PinPort device. The third, fourth, and fifth arguments are each multiplied by the prescale counter (second argument) to determine the number of clocks comprising the front porch, chimney, and back porch, respectively. The sixth argument determines whether the generated strobe is negated before it is input to the logic function that combines it with the value of a pin being output. The seventh argument specifies whether the strobe effects values output (and if so the logic function to perform) or when input values are latched.

For example,

```
exsent_cfg_stb_1(1,2,3,2,4,0,0)
```

configures strobe 1 for PinPort device number 1. The second through fifth arguments generate the strobe depicted in figure 4.2 as the trace “strobe 1”. The sixth argument indicates the strobe is active high. The seventh argument indicates that it is combined with a pin through an *and* function to effect the output value.. Note that “strobe 1” is active during the chimney portion of the strobe period.

As another example,

```
exsent_cfg_stb_2(1,2,5,2,2,0,3)
```

configures strobe 2 for PinPort device number 1. The second through fifth arguments generate the strobe depicted in figure 4.3 as the trace “strobe2”. The sixth argument indicates the strobe is active high. The seventh argument indicates that it affects when input values are latched.

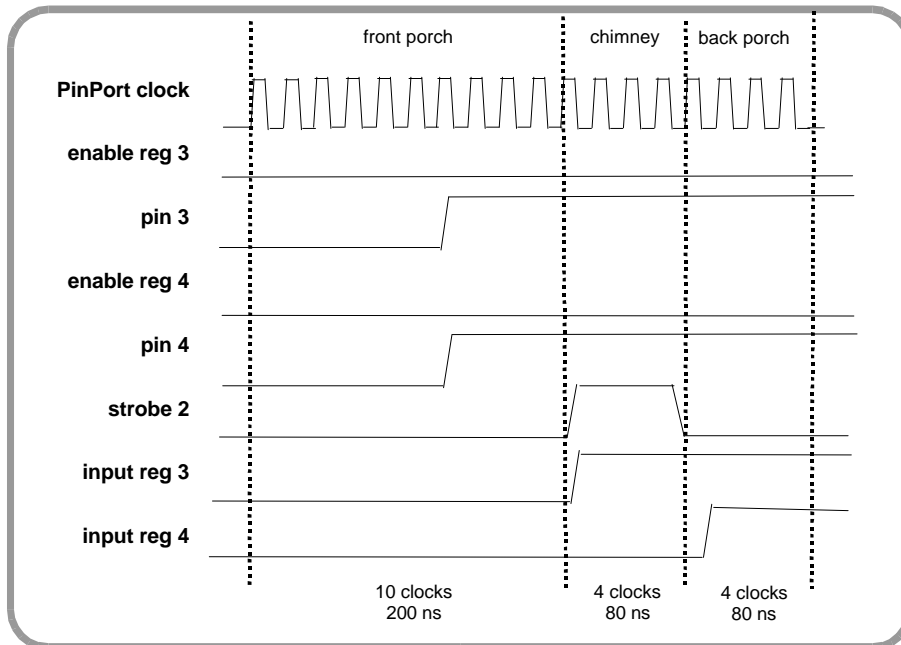


Figure 4.3 Input Strobe Example

4.2.2 Pin-Strobe Association

Pins are associated with strobe1 or strobe 2 by calling the `exsent_cfg_pin_1` and `exsent_cfg_pin_2` API routines, respectively. These routines have identical two-argument argument lists. The first argument identifies the PinPort device. The second argument is a vector consisting of a bit for each pin of the PinPort device. If a bit position corresponding to a pin in this argument is a 0, then the strobe has no effect on the value output by the pin driver or latched by the pin buffer. If it is a 1, then the strobe will modify how the pin's output driver or input buffer operates. It is an error if the same bit position is set to a 1 in the second argument of calls to both `exsent_cfg_pin_1` and `exsent_cfg_pin_2` for the same PinPort device.

When an output pin is not associated with a strobe, it is updated directly with the corresponding data bit in the output register during the front porch period. When it is associated with a strobe configured to effect outputs, it is combined with the strobe through the logic function during the chimney period.

For output pins associated with a strobe, the back porch period provides any required settling time for signals prior to the next pin access. The front porch is configured so that pins not associated with a strobe have sufficient setup time with respect to an output pin associated with a strobe. Similarly, the back porch is configured so that pins not associated with a strobe have sufficient hold time with respect to an output pin associated with a strobe.

Continuing with the example of figure 4.2, pin 1 is associated with strobe 1 and pin 2 is **not** associated with a strobe. The “output reg 1” and “output reg 2” traces show that both pin 1 and pin 2 have been set to a logic 1 through API calls. Further, the “enable reg 1” and “enable reg 2” traces show that both pin 1 and pin 2 have been enabled for output through API calls. The value driven for pin 1 is shown as the trace “pin 1” and results from the ANDing of “output reg 1” with “strobe 1”. The value driven for pin 2 is shown as the trace “pin 2” and directly reflects “output reg 2” during the front porch, chimney, and back porch segments.

Continuing with the example of figure 4.3, pin 3 is associated with strobe 2 and pin 4 is **not** associated with a strobe. The “enable reg 3” and “enable reg 4” traces show that neither pin 3 nor pin 4 has been enabled for output through API calls – they are both inputs. The “pin 3” and “pin 4” traces show that the device-under-test on the PinPort adapter card has changed the value of these pins from a logic 0 to a logic 1 during the front

4.3 Pin Enabling and Sampling

The PinPort device has a flexible and programmable pin architecture. The block diagram of figure 4.4 shows the function and control of each PinPort device pin.

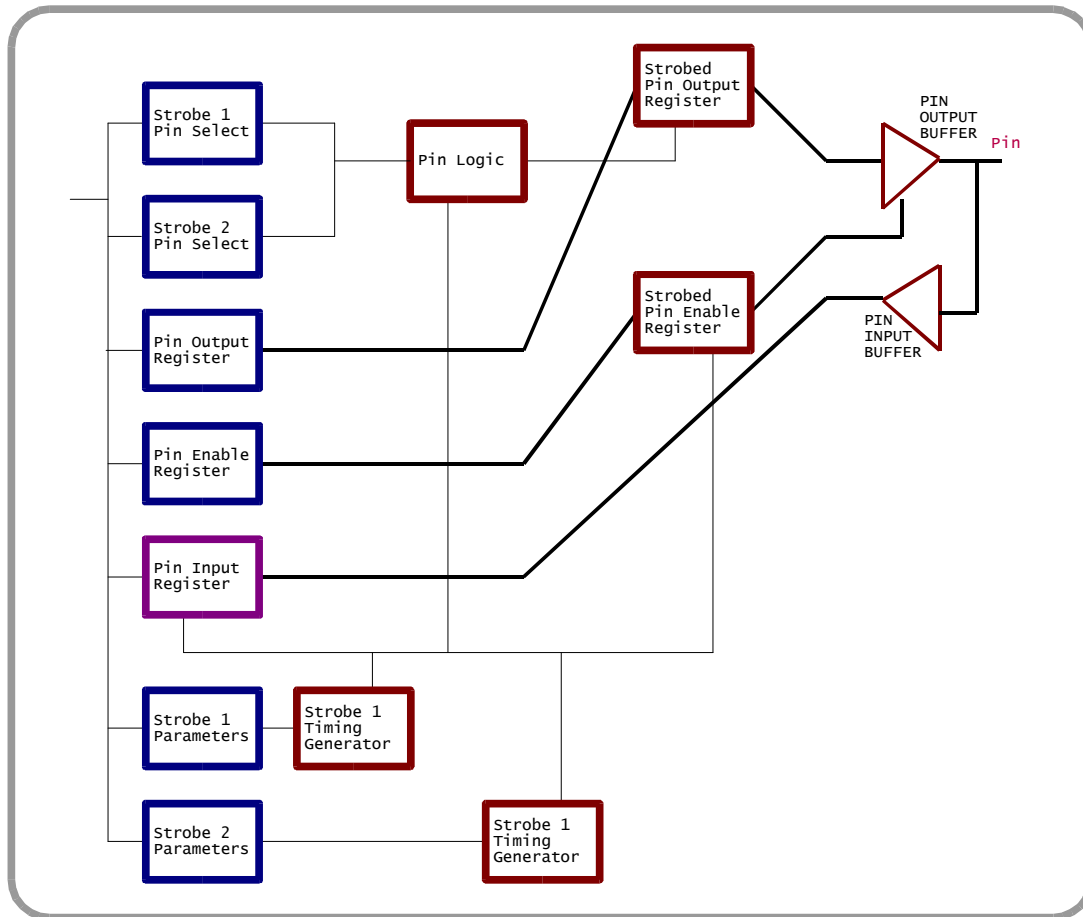


Figure 4.4 PinPort Pin Architecture

During a strobe period each pin is configured as an input or an output based on its enable bit. If the enable bit is a 1, the pin is an output. If it is a 0, the pin is an input. A bidirectional pin is simply a pin whose enable bit is 1 for some strobe periods and a 0 for others. The API routines `exsent_write`, `exsent_write_pin_enables`, and `exsent_write_then_read` each set the enable bits for all pins.

For any given strobe period a bidirectional pin is either an output pin or an input pin. The strobe algorithm controls the level, function, and timing of each output pin and when an input pin is latched as described in the previous section. At the completion of the strobe period, the input register holds the state of the pin. This is true regardless of whether the pin is acting as an output or an input for the strobe period. Both of the API routines `exsent_read` and `exsent_write_then_read` return the value of the input register for all pins.

Both figures 4.5 and 4.6 show a bidirectional pin over three strobe periods. In each figure, the pin is an output for the first two periods and an input for the last period. The pin in figure 4.5 is **not** associated with a strobe. The pin in figure 4.6 is associated with a strobe.

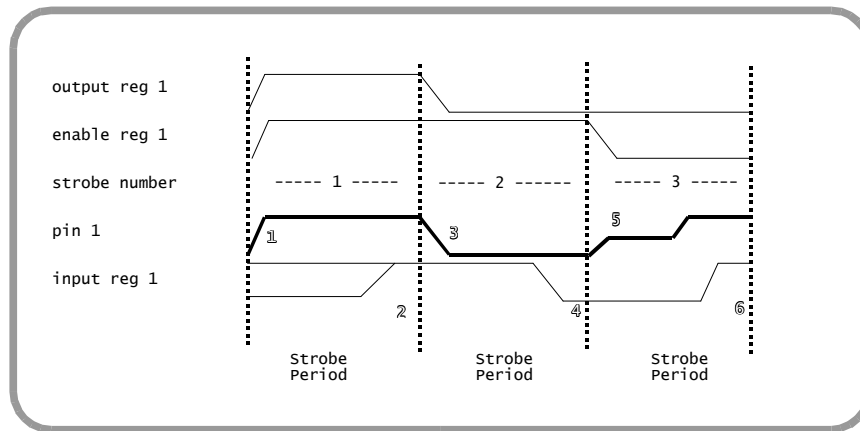


Figure 4.5 Bidirectional Pin Not Associated with a Strobe

At point 1 the output is 1, and the enable is 1, so in the absence of any strobe enables for this pin, the pin value becomes 1. At point 2 the strobe period for cycle 1 is complete, so the value of the pin is stored in the input register. Likewise for cycle 2, the enable value is 1, the output is now 0, so the pin will change to 0 at point 3. At point 4 the input register will store this value for cycle 2. In cycle 3, the enable is now 0, so even though the output value is transferred to the pin output register, the pin goes to a non-driven or hi Z state. If the device-under-test on the PinPort adapter card drives the pin, then at point 6 the input register will store this value. Recall that in the verification application (simulator or C program) no time elapses during the strobe period. This activity described here happens in the PinPort physical environment.

The waveform in figure 4.6 shows the same pin values, but with an output-controlling strobe enabled. In this case, the strobe is active high, and the logic function for combining the strobe with the pin value is an *and*. Note that at point 7 a one is latched into the input register even though the output is not enabled. This implies that the device-under-test on the PinPort adapter card is driving a logic 1.

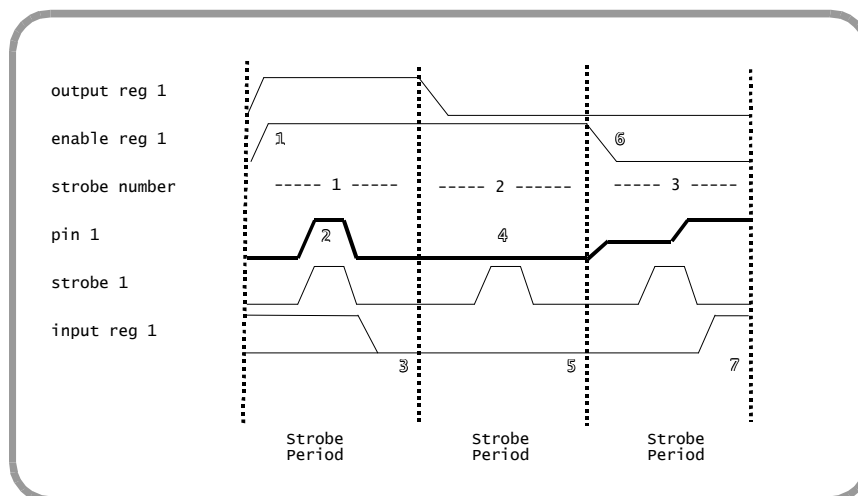


Figure 4.6 Bidirectional Pin Associated with a Output-Controlling Strobe

Figure 4.7 shows traces and strobe periods similar to Figure 4.6. However, now an input-controlling strobe has been added to determine when the value gets latched into the input register. Note that points 3, 5, and 7 now coincide with the chimney portions of the strobe periods.

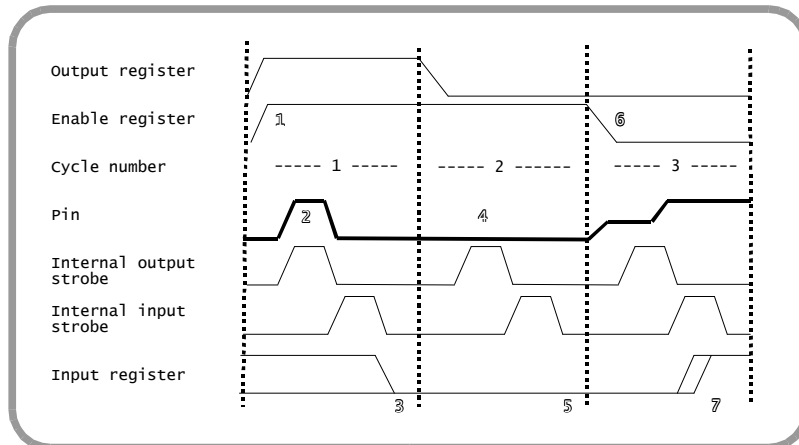


Figure 4.7 Pin Associated with an Input-Controlling Strobe

5 INSTALLATION REFERENCE

5.1 *PinPort SCSI Interface*

The PinPort192 supports a standard SCSI interface. Two 50 pin single-ended SCSI connectors are provided. One is an upstream connection to the computer, and the other is a downstream connection to another SCSI device, or used for the SCSI terminator. Multiple SCSI devices can be put on the SCSI bus, and this includes multiple PinPort devices. They can be accessed individually. All accesses to the PinPort device are currently on LUN0. No other LUN is supported at this time.

Each SCSI device has a unique SCSI ID number (0-7). The SCSI host adapter (in the computer used for simulation) will usually get the address 7. Thus there are 7 addresses left for external devices. If only one PinPort device is used, there is usually no need to change the SCSI setting.

If more than one SCSI device is connected to the SCSI bus, then they are considered (physically) daisy chained. One cable goes from the computer to the first SCSI device, one cable goes from the first to the second SCSI device, and so on. The last SCSI device in the daisy chain has the responsibility of terminating the SCSI bus. A connector for this termination, or for daisy-chaining is provided on the PinPort192.

5.2 *Key Files*

Key files are required to enable the PinPort API software to function with the PinPort device. These key files should be in a sub-directory or sub-folder of the "keys" directory with the directory or folder name being the same as the serial number of the PinPort device. The "keys" directory or folder exists at the same location as the PinPort API software. The serial number is located on the bottom of the device.

6 ELECTRICAL SPECIFICATIONS

6.1 Clock Frequency

The PinPort192 has a clock which operates at 50MHz. This clock determines strobe timing for the pins, along with the strobe configuration API calls.

6.2 Electrical Spec for Pins

The PinPort192 provides 192 user definable pins for accessing physical devices. These pins typically have a 3.3 volt output swing, but are 5.0 volt tolerant, and compliant with the TTL spec.

In addition to the 192 assigned signal pins, there are six 5.0 volt power pins, four 3.3 volt power pins, four 2.5 volt power pins, and 36 ground pins available.

Table 6.1 shows the power and ground pins, and their connector pin assignments.

Pins	Voltage	Max Power
A101, A114, A115, A116, A117, B101,	5.0v 4.5v min 5.5v max	167-500ma note 1
A120, A121, B120, B121	3.3v 2.97v min 3.63v max	167-500ma note 1
A102, A119 B102, B119	2.5v 2.25v min 2.75v max	167-500ma note 1
A1, A7, A17, A27, A37, A45, A50, A51, A57, A67, A77, A87, A95, A100, A105, A107, A109, A111, B1, B7, B17, B27, B37, B45, B50, B51, B57, B67, B77, B87, B95, B100, B105, B107, B109, B111	0v, ground	N/A

Table 6.1 Power and Ground Pins

NOTE 1: The total for all 3 power supplies is 500ma. The power can be used in any combination as long as the total power for all 3 does not exceed the maximum.

The min and max parameters for the signal pins are shown in table 6.2. All output drives are rated at 2.0 ma max for source or sink. Special care should be taken in the layout of any board which connects to a PinPort device so that signal integrity is maintained when switching multiple signals.

DESCRIPTION	MIN	MAX	UNITS
Input, high level	1.815	5.5	v
Input, low level	0	0.891	v
Output, high level	2.4	-	v
Output, low level	-	0.400	v

Table 6.2 Pin Voltage Levels

If extensive circuitry is connected to the PinPort, then external power should be considered.

Proper attention to grounding should assure signal integrity.

Pins A1 - A121 are on one side of the 242 pin device connector, and pins B1 - B121 are on the opposite side.

6.3 Electrical Spec for Supply

The PinPort192 requires an external, 12 volt DC, 1.5 amp power supply.

The power supply provided with PinPort is the only power supply which should be used.

6.4 Adapter Board

The PinPort192 has an edge connector socket at the top of the PinPort device. Inserting an edge connector adapter into this socket will allow connections to be made from the adapter to the verification environment.