IN THIS DOCUMENT

1 Data types

`clock` Clock resource type.

Clocks are declared as global variables and are initialized with the resource identifier of a clock block. When in a running state a clock provides rising and falling edges to ports configured using that clock.

2 Port Configuration Functions

```
void configure_in_port_handshake(void port p,
                                 in port readyin,
                                 out port readyout,
                                 clock clk)
```

Configures a buffered port to be a clocked input port in handshake mode.

If the ready-in or ready-out ports are not 1-bit ports, an exception is raised. The ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. The port samples its pins on its sampling edge when both the ready-in and ready-out ports are asserted.

By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

`p` The buffered port to configure.

readyin	A 1-bit port to use for the ready-in signal.
readyout	A 1-bit port to use for the ready-out signal.
clk	The clock used to configure the port.

```
void configure_out_port_handshake(void port p,
                                  in port readyin,
                                  out port readyout,
                                  clock clk,
                                  unsigned initial)
```

Configures a buffered port to be a clocked output port in handshake mode.

If the ready-in or ready-out ports are not 1-bit ports an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. The ready-in port is read on the sampling edge of the buffered port. Outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port was high. On the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

p	The buffered port to configure.
readyin	A 1-bit port to use for the ready-in signal.
readyout	A 1-bit port to use for the ready-out signal.
clk	The clock used to configure the port.
initial	The initial value to output on the port.

```
void configure_in_port_strobed_master(void port p,
                                      out port readyout,
                                      const clock clk)
```

Configures a buffered port to be a clocked input port in strobed master mode.

If the ready-out port is not a 1-bit port, an exception is raised. The ready-out port is asserted on the falling edge of the clock when the port's buffer is not full. The port samples its pins on its sampling edge after the ready-out port is asserted.

By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

p	The buffered port to configure.

readyout	A 1-bit port to use for the ready-out signal.
clk	The clock used to configure the port.

```
void configure_out_port_strobed_master(void port p,
                                       out port readyout,
                                       const clock clk,
                                       unsigned initial)
```

Configures a buffered port to be a clocked output port in strobed master mode.

If the ready-out port is not a 1-bit port, an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. Outputs are driven on the next falling edge of the clock. On the falling edge of the clock the ready-out port is driven high if data is output on that edge, otherwise it is driven low.

This function has the following parameters:

p	The buffered port to configure.
readyout	A 1-bit port to use for the ready-out signal.
clk	The clock used to configure the port.
initial	The initial value to output on the port.

```
void configure_in_port_strobed_slave(void port p,
                                     in port readyin,
                                     clock clk)
```

Configures a buffered port to be a clocked input port in strobed slave mode.

If the ready-in port is not a 1-bit port, an exception is raised. The port samples its pins on its sampling edge when the ready-in signal is high. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

p	The buffered port to configure.
readyin	A 1-bit port to use for the ready-in signal.
clk	The clock used to configure the port.

```
void configure_out_port_strobed_slave(void port p,
                                      in port readyin,
                                      clock clk,
                                      unsigned initial)
```

Configures a buffered port to be a clocked output port in strobed slave mode.

If the ready-in port is not a 1-bit port, an exception is raised. The port drives the initial value on its pins until an output statement changes the value driven. The ready-in port is read on the buffered port's sampling edge. Outputs are driven on the next falling edge of the clock where the previous value read from the ready-in port is high. By default the port's sampling edge is the rising edge of clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

`p` The buffered port to configure.

`readyin` A 1-bit port to use for the ready-in signal.

`clk` The clock used to configure the port.

`initial` The initial value to output on the port.

`void configure_in_port(void port p, const clock clk)`

Configures a port to be a clocked input port with no ready signals.

This is the default mode of a port. The port samples its pins on its sampling edge. If the port is unbuffered, its direction can be changed by performing an output. This change occurs on the next falling edge of the clock. Afterwards, the port behaves as an output port with no ready signals.

By default the port's sampling edge is the rising edge of the clock. This can be changed by the function set_port_sample_delay().

This function has the following parameters:

`p` The port to configure, which may be buffered or unbuffered.

`clk` The clock used to configure the port.

`void configure_in_port_no_ready(void port p, const clock clk)`

Alias for configure_in_port().

`void configure_out_port(void port p, const clock clk, unsigned initial)`

Configures a port to be a clocked output port with no ready signals.

The port drives the initial value on its pins until an input or output statement changes the value driven. Outputs are driven on the next falling edge of the clock and every port-width bits of data are held for one clock cycle. If the port is unbuffered, the direction of the port can be changed by performing an input. This change occurs on the falling edge of the clock after any pending outputs have been held for one clock period. Afterwards, the port behaves as an input port with no ready signals.

This function has the following parameters:

`p` The port to configure, which may be buffered or unbuffered.

clk The clock used to configure the port.

initial The initial value to output on the port.

```
void configure_out_port_no_ready(void port p,
                                 const clock clk,
                                 unsigned initial)
```

Alias for configure_out_port().

```
void configure_port_clock_output(void port p, const clock clk)
```

Configures a 1-bit port to output a clock signal.

If the port is not a 1-bit port, an exception is raised. Performing inputs or outputs on the port once it has been configured in this mode results in undefined behaviour.

This function has the following parameters:

p The 1-bit port to configure.

clk The clock to output.

```
void set_port_no_sample_delay(void port p)
```

Sets a port to no sample delay mode.

This causes the port to sample input data on the rising edge of its clock. This is the default state of the port.

This function has the following parameters:

p The port to configure.

```
void set_port_sample_delay(void port p)
```

Sets a port to sample delay mode.

This causes the port to sample input data on the falling edge of its clock.

This function has the following parameters:

p The port to configure.

```
void set_port_clock(void port p, const clock clk)
```

Attaches a clock to a port.

This corresponds to using the SETCLK instruction on a port. The edges of the clock are used to sample and output data. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`clk` The clock to attach.

`void set_port_ready_src(void port p, void port ready)`

Sets a 1-bit port as the ready-out for another port.

This corresponds with using the SETRDY instruction on a port. If the ready-out port is not a 1-bit port then an exception is raised. The ready-out port is used to indicate that the port is ready to transfer data. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

`p` The port to configure.

`ready` The 1-bit port to use for the ready-out signal.

`void set_port_use_on(void port p)`

Turns on a port.

The port state is initialised to the default state for a port of its type. If the port is already turned on its state is reset to its default state.

This function has the following parameters:

`p` The port to turn on.

`void set_port_use_off(void port p)`

Turns off a port.

No action is performed if the port is already turned off. Any attempt to use the port while off will result in an exception being raised.

This function has the following parameters:

`p` The port to turn off.

`void set_port_mode_data(void port p)`

Configures a port to be a data port.

This is the default state of a port. Output operations on the port are use to control its output signal.

This function has the following parameters:

`p` The port to configure.

`void set_port_mode_clock(void port p)`

Configures a 1-bit port to be a clock output port.

The port will output the clock connected to it. If the port is not a 1-bit port, an exception is raised. The function set_port_mode_data() can be used to set the port back to its default state.

This function has the following parameters:

p The port to configure.

`void set_port_mode_ready(void port p)`

Configures a 1-bit port to be a ready signal output port.

The port will output the ready-out of a port connected with set_port_ready_src(). If the port is not a 1-bit port, an exception is raised. The function set_port_mode_data() can be used to set the port back to its default state. Usually the use of the configure_*_port_* functions is prefered since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_drive(void port p)`

Configures a port in drive mode.

Values output to the port are driven on the pins. This is the default drive state of a port. Calling set_port_drive() has the side effect disabling the port's pull up or pull down resistor.

This function has the following parameters:

p The port to configure.

`void set_port_drive_low(void port p)`

Configures a port in drive low mode.

For 1-bit ports when 0 is output its pin is driven low and when 1 is output no value is driven. If the port is not a 1-bit port, the result of an output to the port is undefined. On XS1-G devices calling set_port_drive_low() has the side effect of enabling the port's internal pull-up resistor. On XS1-L devices calling set_port_drive_low() has the side effect of enabling the port's internal pull-down resistor.

This function has the following parameters:

p The port to configure.

`void set_port_pull_up(void port p)`

Enables a port's internal pull-up resistor.

When nothing is driving a pin the pull-up resistor ensures that the value sampled by the port is 1. The pull-up is not strong enough to guarantee a defined external value. On XS1-G devices calling set_port_pull_up() has the side effect of configuring the port in drive low mode. On XS1-L devices no pull-up resistors are available and an exception will be raised if set_port_pull_up() is called.

This function has the following parameters:

p The port to configure.

`void set_port_pull_down(void port p)`

Enables a port's internal pull-down resistor.

When nothing is driving a pin the pull-down resistor ensures that the value sampled by the port is 0. The pull-down is not strong enough to guarantee a defined external value. On XS1-G devices no pull-down resistors are available and an exception will be raised if set_port_pull_down() is called. On XS1-L devices calling set_port_pull_down() has the side effect of configuring the port in drive low mode.

This function has the following parameters:

p The port to configure.

`void set_port_pull_none(void port p)`

Disables the port's pull-up or pull-down resistor.

This has the side effect of configuring the port in drive mode.

This function has the following parameters:

p The port to configure.

`void set_port_master(void port p)`

Sets a port to master mode.

This corresponds to using the SETC instruction on the port with the value XS1_SETC_MS_MASTER. Usually the use of the functions configure_in_port_strobed_master() and configure_out_port_strobed_master() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_slave(void port p)`

Sets a port to slave mode.

This corresponds to using the SETC instruction on the port with the value XS1_SETC_MS_SLAVE. Usually the use of the functions configure_in_port_strobed_slave() and configure_out_port_strobed_slave() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_no_ready(void port p)`

Configures a port to not use ready signals.

This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_NOREADY. Usually the use of the functions configure_in_port() and configure_out_port() is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_strobed(void port p)`

Sets a port to strobed mode.

This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_STROBED. Usually the use of the configure_*_port_strobed_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_handshake(void port p)`

Sets a port to handshake mode.

This corresponds to using the SETC instruction on the port with the value XS1_SETC_RDY_HANDSHAKE. Usually the use of the configure_*_port_handshake functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

p The port to configure.

`void set_port_no_inv(void port p)`

Configures a port to not invert data that is sampled and driven on its pins.

This is the default state of a port.

This function has the following parameters:

p The port to configure.

`void set_port_inv(void port p)`

Configures a 1-bit port to invert data which is sampled and driven on its pin.

If the port is not a 1-bit port, an exception is raised. If the port is used as the source for a clock then setting this mode has the effect of the swapping the rising and falling edges of the clock.

This function has the following parameters:

p The 1-bit port to configure.

`void set_port_shift_count(void port p, unsigned n)`

Sets the shift count for a port.

This corresponds with the SETPSC instruction. The new shift count must be less than the transfer width of the port, greater than zero and a multiple of the port width otherwise an exception is raised. For a port used for input this function will cause the next input to be ready when the specified amount of data has been shifted in. The next input will return transfer-width bits of data with the captured data in the most significant bits. For a port used for output this will cause the next output to shift out this number of bits. Usually the use of the functions partin() and partout() is preferred over `setpsc()` as they perform both the required configuration and the input or output together.

This function has the following parameters:

`p` The buffered port to configure.

`n` The new shift count.

`void set_pad_delay(void port p, unsigned n)`

Sets the delay on the pins connected to the port.

The input signals sampled on the port's pins are delayed by this number of processor-clock cycles before they they are seen on the port. The default delay on the pins is 0. The delay must be set to values in the range 0 to 5 inclusive. If there are multiple enabled ports connected to the same pin then the delay on that pin is set by the highest priority port.

This function has the following parameters:

`p` The port to configure.

`n` The number of processor-clock cycles by which to delay the input signal.

3 Clock Configuration Functions

`void configure_clock_src(clock clk, void port p)`

Configures a clock to use a 1-bit port as its source.

This allows I/O operations on ports to be synchronised to an external clock signal. If the port is not a 1-bit port, an exception is raised.

This function has the following parameters:

`clk` The clock to configure.

`p` The 1-bit port to use as the clock source.

`void configure_clock_ref(clock clk, unsigned char divide)`

Configures a clock to use the reference clock as it source.

If the divide is set to zero the reference clock frequency is used, otherwise the reference clock frequency divided by 2 * is used. By default the reference clock is configured to run at 100 MHz.

This function has the following parameters:

`clk` The clock to configure.

`divide` The clock divide.

`void configure_clock_rate(clock clk, unsigned a, unsigned b)`

Configures a clock to run at a rate of (MHz.

If the specified rate is not supported by the hardware, an exception is raised. The hardware supports rates of MHz and rates of the form (MHz where is the reference clock frequency and is a number in the range 1 to 255 inclusive.

This function has the following parameters:

`clk` The clock to configure.

`a` The dividend of the desired rate.

`b` The divisor of the desired rate.

`void configure_clock_rate_at_least(clock clk, unsigned a, unsigned b)`

Configures a clock to run the slowest rate supported by the hardware that is equal to or exceeds (MHz.

An exception is raised if no rate satisfies this criterion.

This function has the following parameters:

`clk` The clock to configure.

`a` The dividend of the desired rate.

`b` The divisor of the desired rate.

`void configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)`

Configures a clock to run at the fastest non-zero rate supported by the hardware that is less than or equal to (MHz.

An exception is raised if no rate satisfies this criterion.

This function has the following parameters:

`clk` The clock to configure.

`a` The dividend of the desired rate.

`b` The divisor of the desired rate.

`void set_clock_src(clock clk, void port p)`

Sets the source for a clock to a 1-bit port.

This corresponds with using the SETCLK instruction on a clock. If the port is not a 1-bit port, an exception is raised. In addition if the clock was previously configured with a non-zero divide then an exception is raised. Usually the use of configure_clock_src() which does not suffer from this problem is recommended.

This function has the following parameters:

`clk` The clock to configure.

`p` The 1-bit port to use as the clock source.

`void set_clock_ref(clock clk)`

Sets the source for a clock to the reference clock.

This corresponds with the using SETCLK instruction on a clock. The clock divide is left unchanged.

This function has the following parameters:

`clk` The clock to configure.

`void set_clock_div(clock clk, unsigned char div)`

Sets the divide for a clock.

This corresponds with the SETD instruction. The clock source must be set to the reference clock, otherwise an exception is raised. If the divide is set to zero the source frequency is left unchanged, otherwise the source frequency is divided by 2 * .

This function has the following parameters:

`clk` The clock to configure.

`div` The divide to use.

`void set_clock_rise_delay(clock clk, unsigned n)`

Sets the delay for the rising edge of a clock.

Each rising edge of the clock by processor-clock cycles before it is seen by any port connected to the clock. The default rising edge delay is 0 and the delay must be set to values in the range 0 to 512 inclusive. If the clock edge is delayed by more than the clock period then no rising clock edges are seen by the ports connected to the clock.

This function has the following parameters:

`clk` The clock to configure.

`n` The number of processor-clock cycles by which to delay the rising edge of the clock.

`void set_clock_fall_delay(clock clk, unsigned n)`

Sets the delay for the falling edge of a clock.

Each falling edge of the clock is delayed by processor-clock cycles before it is seen by any port connected to the clock. The default falling edge delay is 0. The delay can be set to values in the range 0 to 512 inclusive. If the clock edge is delayed by more than the clock period then no falling clock edges are seen by the ports connected to the clock.

This function has the following parameters:

`clk` The clock to configure.

`n` The number of processor-clock cycles by which to delay the falling edge of the clock.

`void set_clock_ready_src(clock clk, void port ready)`

Sets a clock to use a 1-bit port for the ready-in signal.

This corresponds with using the SETRDY instruction on a clock. If the port is not a 1-bit port then an exception is raised. The ready-in port controls when data is sampled from the pins. Usually the use of the configure_*_port_* functions is preferred since they ensure that all the port configuration changes required for the desired mode are performed in the correct order.

This function has the following parameters:

`clk` The clock to configure.

`ready` The 1-bit port to use for the ready-in signal.

`void set_clock_on(clock clk)`

Turns on a clock.

The clock state is initialised to the default state for a clock. If the clock is already turned on then its state is reset to its default state.

This function has the following parameters:

`clk` The clock to turn on.

`void set_clock_off(clock clk)`

Turns off a clock.

No action is performed if the clock is already turned off. Any attempt to use the clock while it is turned off will result in an exception being raised.

This function has the following parameters:

`clk` The clock to turn off.

4 Port Manipulation Functions

`void start_port(void port p)`

Activates a port.

The buffer used by the port is cleared.

This function has the following parameters:

`p` The port to activate.

`void stop_port(void port p)`

Deactivates a port.

The port is reset to being a no ready port.

This function has the following parameters:

`p` The port to deactivate.

`unsigned peek(void port p)`

Instructs the port to sample the current value on its pins.

The port provides the sampled port-width bits of data to the processor immediately, regardless of its transfer width, clock, ready signals and buffering. The input has no effect on subsequent I/O performed on the port.

This function has the following parameters:

`p` The port to peek at.

This function returns:

The value sampled on the pins.

`void clearbuf(void port p)`

Clears the buffer used by a port.

Any data sampled by the port which has not been input by the processor is discarded. Any data output by the processor which has not been driven by the port is discarded. If the port is in the process of serialising output, it is interrupted immediately. If a pending output would have caused a change in direction of the port then that change of direction does not take place. If the port is driving a value on its pins when clearbuf() is called then it continues to drive the value until an output statement changes the value driven.

This function has the following parameters:

`p` The port whose buffer is to be cleared.

`void sync(void port p)`

Waits until a port has completed any pending outputs.

Waits output all until a port has completed any pending outputs and the last port-width bits of data has been held on the pins for one clock period.

This function has the following parameters:

`p` The port to wait on.

`unsigned endin(void port p)`

Ends the current input on a buffered port.

The number of bits sampled by the port but not yet input by the processor is returned. This count includes both data in the transfer register and data in the shift register used for deserialisation. Subsequent inputs on the port return transfer-width bits of data until there is less than one transfer-width bits of data remaining. Any remaining data can be read with one further input, which returns transfer-width bits of data with the remaining buffered data in the most significant bits of this value.

This function has the following parameters:

`p` The port to end the current input on.

This function returns:

The number of bits of data remaining.

`unsigned partin(void port p, unsigned n)`

Performs an input of the specified width on a buffered port.

The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The value returned is undefined if the number of bits in the port's shift register is greater than or equal to the specified width.

This function has the following parameters:

`p` The buffered port to input on.

`n` The number of bits to input.

This function returns:

The inputted value.

`void partout(void port p, unsigned n, unsigned val)`

Performs an output of the specified width on a buffered port.

The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The least significant bits of are output.

This function has the following parameters:

p The buffered port to output on.

n The number of bits to output.

val The value to output.

`unsigned partout_timed(void port p, unsigned n, unsigned val, unsigned t)`

Performs a output of the specified width on a buffered port when the port counter equals the specified time.

The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The least significant bits of are output.

This function has the following parameters:

p The buffered port to output on.

n The number of bits to output.

val The value to output.

t The port counter value to output at.

`{unsigned , unsigned } partin_timestamped(void port p, unsigned n)`

Performs an input of the specified width on a buffered port and timestamps the input.

The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The value returned is undefined if the number of bits in the port's shift register is greater than or equal to the specified width.

This function has the following parameters:

p The buffered port to input on.

n The number of bits to input.

This function returns:

The inputted value and the timestamp.

`unsigned partout_timestamped(void port p, unsigned n, unsigned val)`

Performs an output of the specified width on a buffered port and timestamps the output.

The width must be less than the transfer width of the port, greater than zero and a multiple of the port width, otherwise an exception is raised. The least significant bits of are output.

This function has the following parameters:

`p`	The buffered port to output on.
`n`	The number of bits to output.
`val`	The value to output.

This function returns:

The timestamp of the output.

5 Clock Manipulation Functions

`void start_clock(clock clk)`

Puts a clock into a running state.

A clock generates edges only after it has been put into this state. The port counters of all ports attached to the clock are reset to 0.

This function has the following parameters:

`clk`	The clock to put into a running state.

`void stop_clock(clock clk)`

Waits until a clock is low and then puts the clock into a stopped state.

In a stopped state a clock does not generate edges.

This function has the following parameters:

`clk`	The clock to put into a stopped state.

6 Logical Core/Tile Control Functions

`void set_core_fast_mode_on(void)`

Sets the current logical core to run in fast mode.

The scheduler always reserves a slot for a logical core in fast mode regardless of whether core is waiting for an input or a select to complete. This reduces the worst case latency from a change in state happening to a paused input or select completing as a result of that change. However, putting a core in fast mode means that other logical cores are unable to use the extra slot which would otherwise be available while the core is waiting. In addition setting logical cores to run in fast mode may also increase the power consumption.

`void set_core_fast_mode_off(void)`

Sets the current logical core to run in normal execution mode.

If a core has previously been put into fast mode using set_core_fast_mode_on() this function resets the execution mode it to its default state.

`unsigned getps(unsigned reg)`

Gets the value of a processor state register.

This corresponds with the GETPS instruction. An exception is raised if the argument is not a legal processor state register.

This function has the following parameters:

`reg` The processor state register to read.

This function returns:

The value of the processor state register.

`void setps(unsigned reg, unsigned value)`

Sets the value of a processor state register.

Corresponds with the SETPS instruction. An exception is raised if the argument is not a legal processor state register.

This function has the following parameters:

`reg` The processor state register to write.

`value` The value to set the processor state register to.

`int read_pswitch_reg(unsigned tileid, unsigned reg, unsigned &data)`

Reads the value of a processor switch register.

The read is of the processor switch which is local to the specified tile id. On success 1 is returned and the value of the register is assigned to . If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the read packet then 0 is returned.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value read from the register.

This function returns:

Whether the read was successful.

`int write_pswitch_reg(unsigned tileid, unsigned reg, unsigned data)`

Writes a value to a processor switch register.

The write is of the processor switch which is local to the specified tile id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the write packet then 0 is returned.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the write was successful.

`int write_pswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)`

Writes a value to a processor switch register without acknowledgement.

The write is of the processor switch which is local to the specified tile id. Unlike write_pswitch_reg() this function does not wait until the write has been performed. If the register number or tile identifier is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the parameters are valid.

`int read_sswitch_reg(unsigned tileid, unsigned reg, unsigned &data)`

Reads the value of a system switch register.

The read is of the system switch which is local to the specified tile id. On success 1 is returned and the value of the register is assigned to . If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the read packet then 0 is returned.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value read from the register.

This function returns:

Whether the read was successful.

`int write_sswitch_reg(unsigned tileid, unsigned reg, unsigned data)`

Writes a value to a system switch register.

The write is of the system switch which is local to the specified tile id. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number or tile identifier is too large to fit in the write packet then 0 is returned.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the write was successful.

`int write_sswitch_reg_no_ack(unsigned tileid, unsigned reg, unsigned data)`

Writes a value to a system switch register without acknowledgement.

The write is of the system switch which is local to the specified tile id. Unlike write_sswitch_reg() this function does not wait until the write has been performed. If the register number or tile identifier is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.

This function has the following parameters:

`tileid` The tile identifier.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the parameters are valid.

`int read_node_config_reg(tileref tile, unsigned reg, unsigned &data)`

Reads the value of a node configuration register.

The read is of the node containing the specified tile. On success 1 is returned and the value of the register is assigned to . If an error acknowledgement is received or if the register number is too large to fit in the read packet then 0 is returned.

This function has the following parameters:

`tile` The tile.

`reg` The number of the register.

`data` The value read from the register.

This function returns:

Whether the read was successful.

`int write_node_config_reg(tileref tile, unsigned reg, unsigned data)`

Writes a value to a node configuration register.

The write is of the node containing the specified tile. If a successful acknowledgement is received then 1 is returned. If an error acknowledgement is received or if the register number is too large to fit in the write packet then 0 is returned.

This function has the following parameters:

`tile` The tile.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the write was successful.

`int write_node_config_reg_no_ack(tileref tile, unsigned reg, unsigned data)`

Writes a value to a node configuration register without acknowledgement.

The write is of the node containing the specified tile. Unlike write_node_config_reg() this function does not wait until the write has been performed. If the register number is too large to fit in the write packet 0 is returned, otherwise 1 is returned. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.

This function has the following parameters:

`tile` The tile.

`reg` The number of the register.

`data` The value to write to the register.

This function returns:

Whether the parameters are valid.

```
int read_periph_8(tileref tile,
                  unsigned peripheral,
                  unsigned base_address,
                  unsigned size,
                  unsigned char data[])
```

Reads bytes from the specified peripheral starting at the specified base address.

The peripheral must be a peripheral with a 8-bit interface. On success 1 is returned and is filled with the values that were read. Returns 0 on failure.

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address` The base address.

`size` The number of 8-bit values to read.

`data` The values read from the peripheral.

This function returns:

Whether the read was successful.

```
int write_periph_8(tileref tile,
                   unsigned peripheral,
                   unsigned base_address,
                   unsigned size,
                   const unsigned char data[])
```

Writes bytes to the specified peripheral starting at the specified base address.

The peripheral must be a peripheral with a 8-bit interface. On success 1 is returned. Returns 0 on failure.

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address` The base address.

`size` The number of 8-bit values to write.

`data` The values to write to the peripheral.

This function returns:

Whether the write was successful.

```
int write_periph_8_no_ack(tileref tile,
                          unsigned peripheral,
                          unsigned base_address,
```

```
                unsigned size,
                const unsigned char data[])
```

Writes bytes to the specified peripheral starting at the specified base address without acknowledgement.

The peripheral must be a peripheral with a 8-bit interface. Unlike write_periph_8() this function does not wait until the write has been performed. Because no acknowledgement is requested the return value does not reflect whether the write succeeded.

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address`
The base address.

`size` The number of 8-bit values to write.

`data` The values to write to the peripheral.

This function returns:

Whether the parameters are valid.

```
int read_periph_32(tileref tile,
                   unsigned peripheral,
                   unsigned base_address,
                   unsigned size,
                   unsigned data[])
```

Reads 32-bit words from the specified peripheral starting at the specified base address.

On success 1 is returned and is filled with the values that were read. Returns 0 on failure. When reading a peripheral with an 8-bit interface the most significant byte of each word returned is the byte at the lowest address (big endian byte ordering).

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address`
The base address.

`size` The number of 32-bit words to read.

`data` The values read from the peripheral.

This function returns:

Whether the read was successful.

```
int write_periph_32(tileref tile,
                    unsigned peripheral,
                    unsigned base_address,
                    unsigned size,
                    const unsigned data[])
```

Writes 32-bit words to the specified peripheral starting at the specified base address.

On success 1 is returned. Returns 0 on failure. When writing to a peripheral with an 8-bit interface the most significant byte of each word passed to the function is written to the byte at the lowest address (big endian byte ordering).

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address` The base address.

`size` The number of 32-bit words to write.

`data` The values to write to the peripheral.

This function returns:

Whether the write was successful.

```
int write_periph_32_no_ack(tileref tile,
                           unsigned peripheral,
                           unsigned base_address,
                           unsigned size,
                           const unsigned data[])
```

Writes 32-bit words to the specified peripheral starting at the specified base address without acknowledgement.

Unlike write_periph_32() this function does not wait until the write has been performed. Because no acknowledgement is requested the return value does not reflect whether the write succeeded. When writing to a peripheral with an 8-bit interface the most significant byte of each word passed to the function is written to the byte at the lowest address (big endian byte ordering).

This function has the following parameters:

`tile` The tile.

`peripheral` The peripheral number.

`base_address` The base address.

`size` The number of 32-bit words to write.

`data` The values to write to the peripheral.

This function returns:

Whether the parameters are valid.

`unsigned get_local_tile_id(void)`

Returns the identifier of the tile on which the caller is running.

The identifier uniquely identifies a tile on the network.

This function returns:

The tile identifier.

`unsigned get_logical_core_id(void)`

Returns the identifier of the logical core on which the caller is running.

The identifier uniquely identifies a logical core on the current tile.

This function returns:

The logical core identifier.

7 Channel Functions

`void start_streaming_master(chanend c)`

Start streaming communication on the channel.

A call to this function must be matched with a call to start_streaming_slave() on the other end of the channel. A path between the two channel ends is opened which can be used to perform unsynchronized communication using the streaming input and output functions. This path is held open until it is closed using the function stop_streaming_master(). Note that if the number of channels held open between two points on the network is equal to the number of possible paths these two points then no other channel communication can take place between these two points, which may cause the program deadlock.

This function has the following parameters:

`c` The channel end to start streaming on

`void stop_streaming_master(chanend c)`

Stop streaming communication on the channel.

A call to this function must be matched with a call to a stop_streaming_slave() on the other end of the channel. The the path previously opened using start_streaming_master() is closed down, making it available for other channel communications.

This function has the following parameters:

`c` The channel end to stop streaming on

`void start_streaming_slave(chanend c)`

Start streaming communication on the channel.

A call to this function must be matched with a call to start_streaming_master() on the other end of the channel. A path between the two channel ends is opened which can be used to perform unsynchronized communication using the streaming input and output functions. This path is held open until it is closed using the function stop_streaming_slave(). Note that if the number of channels held open between two points on the network is equal to the number of possible paths these two points then no other channel communication can take place between these two points, which may cause the program deadlock. The function start_streaming_slave() may be called in a case of a select, in which case it becomes ready when start_streaming_master() is called on the other end of the channel.

This function has the following parameters:

`c` The channel end to start streaming on

`void stop_streaming_slave(chanend c)`

Stop streaming communication on the channel.

A call to this function must be matched with a call to stop_streaming_master() on the other end of the channel. The the path previously opened using start_streaming_slave() is closed down, making it available for other channel communications.

This function has the following parameters:

`c` The channel end to stop streaming on

`void outuchar(chanend c, unsigned char val)`

Streams out a value as an unsigned char on a channel end.

The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value to output.

`void outuint(chanend c, unsigned val)`

Streams out a value as an unsigned int on a channel end.

The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value to output.

`unsigned char inuchar(chanend c)`

Streams in a unsigned char from a channel end.

If the next token in the channel is a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data in on.

This function returns:

The value received.

`unsigned inuint(chanend c)`

Streams in a unsigned int from a channel end.

If the next word of data channel in the channel contains a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data in on.

This function returns:

The value received.

`void inuchar_byref(chanend c, unsigned char &val)`

Streams in a unsigned char from a channel end.

The inputted value is written to . If the next token in channel is a control token then an exception is raised. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data in on.

`val` The variable to set to the received value.

`void inuint_byref(chanend c, unsigned &val)`

Streams in a unsigned int from a channel end.

The inputted value is written to . This function may be called in a case of a select, in which case it becomes ready as soon as there data available on the channel. The protocol used is incompatible with the protocol used by the input (:>) and output (<:) operators.

This function has the following parameters:

`c` The channel end to stream data in on.

`val` The variable to set to the received value.

`void outct(chanend c, unsigned char val)`

Streams out a control token on a channel end.

Attempting to output a hardware control token causes an exception to be raised.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value of the control token to output.

`void chkct(chanend c, unsigned char val)`

Checks for a control token of a given value.

If the next byte in the channel is a control token which matches the expected value then it is input and discarded, otherwise an exception is raised.

This function has the following parameters:

`c` The channel end.

`val` The expected control token value.

`unsigned char inct(chanend c)`

Streams in a control token on a channel end.

If the next byte in the channel is not a control token then an exception is raised, otherwise the value of the control token is returned.

This function has the following parameters:

`c` The channel end to stream data in on.

This function returns:

The received control token.

`void inct_byref(chanend c, unsigned char &val)`

Streams in a control token on a channel end.

The inputted value is written to . If the next byte in the channel is not a control token then an exception is raised.

This function has the following parameters:

`c` The channel end to stream data in on.

`val` The variable to set to the received value.

`int testct(chanend c)`

Tests whether the next byte on a channel end is a control token.

The token is not discarded from the channel and is still available for input.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

1 if the next byte is a control token, 0 otherwise.

`int testwct(chanend c)`

Tests whether the next word on a channel end contains a control token.

If the word does contain a control token the position in the word is returned. No data is discarded from the channel.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

`void soutct(streaming chanend c, unsigned char val)`

Outputs a control token on a streaming channel end.

Attempting to output a hardware control token causes an exception to be raised. Attempting to output a or control token is invalid.

This function has the following parameters:

`c` The channel end to stream data out on.

`val` The value of the control token to output.

`void schkct(streaming chanend c, unsigned char val)`

Checks for a control token of a given value on a streaming channel end.

If the next byte in the channel is a control token which matches the expected value then it is input and discarded, otherwise an exception is raised.

This function has the following parameters:

`c` The streaming channel end.

`val` The expected control token value.

`unsigned char sinct(streaming chanend c)`

Inputs a control token on a streaming channel end.

If the next byte in the channel is not a control token then an exception is raised, otherwise the value of the control token is returned.

This function has the following parameters:

`c` The streaming channel end to stream data in on.

This function returns:

The received control token.

`void sinct_byref(streaming chanend c, unsigned char &val)`

Inputs a control token on a streaming channel end.

The inputted value is written to . If the next byte in the channel is not a control token then an exception is raised.

This function has the following parameters:

`c` The streaming channel end to stream data in on.

`val` The variable to set to the received value.

`int testct(streaming chanend c)`

Tests whether the next byte on a streaming channel end is a control token.

The token is not discarded from the channel and is still available for input.

This function has the following parameters:

`c` The channel end to perform the test on.

This function returns:

1 if the next byte is a control token, 0 otherwise.

`int testwct(streaming chanend c)`

Tests whether the next word on a streaming channel end contains a control token.

If the word does contain a control token the position in the word is returned. No data is discarded from the channel.

This function has the following parameters:

`c` The streaming channel end to perform the test on.

This function returns:

The position of the first control token in the word (1-4) or 0 if the word contains no control tokens.

`transaction out_char_array(chanend c, const char src[], unsigned size)`

Output a block of data over a channel.

A total of bytes of data are output on the channel end. The call to out_char_array() must be matched with a call to in_char_array() on the other end of the channel. The number of bytes output must match the number of bytes input.

This function has the following parameters:

`c` The channel end to output on.

`src` The array of values to send.

`size` The number of bytes to output.

`transaction in_char_array(chanend c, char src[], unsigned size)`

Input a block of data from a channel.

A total of bytes of data are input on the channel end and stored in an array. The call to in_char_array() must be matched with a call to out_char_array() on the other end of the channel. The number of bytes input must match the number of bytes output.

This function has the following parameters:

`c` The channel end to input on.

`src` The array to store the values input from on the channel.

`size` The number of bytes to input.

8 Predicate Functions

`void pinseq(unsigned val)`

Wait until the value on the port's pins equals the specified value.

This function must be called as the expression of an input on a port. It causes the input to become ready when the value on the port's pins is equal to the least significant port-width bits of .

This function has the following parameters:

`val` The value to compare against.

`void pinsneq(unsigned val)`

Wait until the value on the port's pins does not equal the specified value.

This function must be called as the expression of an input on a port. It causes the input to become ready when the value on the port's pins is not equal to the least significant port-width bits of .

This function has the following parameters:

`val` The value to compare against.

`void pinseq_at(unsigned val, unsigned time)`

Wait until the value on the port's pins equals the specified value and the port counter equals the specified time.

This function must be called as the expression of an input on a unbuffered port. It causes the input to become ready when the value on the port's pins is equal to the least significant port-width bits of and the port counter equals .

This function has the following parameters:

`val` The value to compare against.

`time` The time at which to make the comparison.

`void pinsneq_at(unsigned val, unsigned time)`

Wait until the value on the port's pins does not equal the specified value and the port counter equals the specified time.

This function must be called as the expression of an input on a unbuffered port. It causes the input to become ready when the value on the port's pins is not equal to the least significant port-width bits of and the port counter equals .

This function has the following parameters:

`val` The value to compare against.

`time` The time at which to make the comparison.

`void timerafter(unsigned val)`

Wait until the time of the timer equals the specified value.

This function must be called as the expression of an input on a timer. It causes the input to become ready when timer's counter is interpreted as coming after the specified value timer is after the given value. A time A is considered to be after a time B if the expression is true.

This function has the following parameters:

`val` The time to compare against.

9 XS1-S Functions

These functions to control the analogue-to-digital converter (ADC) on XS1-S devices.

```
void enable_xs1_su_adc_input(unsigned number, chanend c)
```

Enables the ADC input specified by .

Samples are sent to chanend .

This function has the following parameters:

`number` The ADC input number.

`c` The channel connected to the XS1-SU ADC.

```
void enable_xs1_su_adc_input_streaming(unsigned number,
                                       streaming chanend c)
```

Enables the ADC input specified by .

Samples are sent to chanend .

This function has the following parameters:

`number` The ADC input number.

`c` The channel connected to the XS1-SU ADC.

```
void disable_xs1_su_adc_input(unsigned number, chanend c)
```

Disables the ADC input specified by .

This function has the following parameters:

`number` The ADC input number.

`c` The channel connected to the XS1-SU ADC.

```
void disable_xs1_su_adc_input_streaming(unsigned number,
                                        streaming chanend c)
```

Disables the ADC input specified by .

This function has the following parameters:

`number` The ADC input number.

`c` The channel connected to the XS1-SU ADC.

10 Miscellaneous Functions

`void crc32(unsigned &checksum, unsigned data, unsigned poly)`

Incorporate a word into a Cyclic Redundancy Checksum.

The calculation performed is

This function has the following parameters:

`checksum` The initial value of the checksum, which is updated with the new checksum.

`data` The data to compute the CRC over.

`poly` The polynomial to use when computing the CRC.

`unsigned crc8shr(unsigned &checksum, unsigned data, unsigned poly)`

Incorporate 8-bits of a word into a Cyclic Redundancy Checksum.

The CRC is computed over the 8 least significant bits of the data and the data shifted right by 8 is returned. The calculation performed is

This function has the following parameters:

`checksum` The initial value of the checksum which is updated with the new checksum.

`data` The data.

`poly` The polynomial to use when computing the CRC.

This function returns:

The data shifted right by 8.

`{unsigned, unsigned} lmul(unsigned a, unsigned b, unsigned c, unsigned d)`

Multiplies two words to produce a double-word and adds two single words.

The high word and the low word of the result are returned. The multiplication is unsigned and cannot overflow. The calculation performed is

This function returns:

The high and low halves of the calculation respectively.

`{unsigned, unsigned} mac(unsigned a, unsigned b, unsigned c, unsigned d)`

Multiplies two unsigned words to produce a double-word and adds a double word.

The high word and the low word of the result are returned. The calculation performed is:

This function returns:

The high and low halves of the calculation respectively.

`{signed, unsigned} macs(signed a, signed b, signed c, unsigned d)`

Multiplies two signed words and adds the double word result to a double word.

The high word and the low word of the result are returned. The calculation performed is:

This function returns:

The high and low halves of the calculation respectively.

`signed sext(unsigned a, unsigned b)`

Sign extends an input.

The first argument is the value to sign extend. The second argument contains the bit position. All bits at a position higher or equal are set to the value of the bit one position lower. In effect, the lower b bits are interpreted as a signed integer. If b is less than 1 or greater than 32 then result is identical to argument a.

This function returns:

The sign extended value.

`unsigned zext(unsigned a, unsigned b)`

Zero extends an input.

The first argument is the value to zero extend. The second argument contains the bit position. All bits at a position higher or equal are set to the zero. In effect, the lower b bits are interpreted as an unsigned integer. If b is less than 1 or greater than 32 then result is identical to argument a.

This function returns:

The zero extended value.