

DTEG ITPP Reader Commands

Note: The commands described here are for the latest version (currently 1.3.1). For earlier versions see the release area's [doc/commands.txt](#) file.

Contents:

- Reader data formats and other command controls
 - radix values
 - Signal expecting values allow values specified with the following radices
 - number string values
 - timestr
 - timing controls
 - Vector and TMS repeat handling
 - checking processes
- Vector clocking
 - Vector Clocking
 - Example
 - Left to right timing
 - Pin Group Clocking
 - Example
- Fast pin access
 - fast pin access
 - timing
 - pin size
 - Auto release
 - deposit
 - scan chains
 - Skip Bits/special chains
 - force_fast
 - peek_fast
 - release_fast
 - force_scan
 - check_scan
 - release_scan
 - Debug signals
- Reader (parser) commands
 - label
 - parser_config
 - parser_debug_mode
 - set_active_*
 - time_snapshot
 - macro_map
 - pin_map
 - pin_group
 - Using pin_groups with clocks as a mini BFM
 - vector_clock
 - pin_group_clock
 - vec_pin_group
 - vector
 - wait_for_checks
 - set_cmd_prefix
- TAP commands
 - scani
 - scand
 - to_state
 - tap_hard_reset
 - tap_soft_reset
 - update_to_seldr
 - capture
 - mask
 - capture_fifo
 - TMS vector
 - tap_idle
- STF commands
 - stf_in, stf_out vectors
 - stf_queue_mode
 - stf_drv_cmd
 - stf_shift_count
 - stf_reset_assert
 - stf_reset_deassert
- SIM commands
 - Signal functions:
 - peek_signal
 - deposit_signal
 - force_signal
 - release_signal

- poll_signal
- Timing functions:
 - delay / delay_fall / delay_rise
 - set_delay_inc
 - delay_time
 - timing_cfg
- Checking and clock functions:
 - trigger_check
 - edge_alignment_check
 - pulse_check
 - define_clock
 - check_clock
 - start_clk
 - start_clk_x
 - stop_clk
 - align_sync
- ITPP loops
 - Loops Example
 - start_loop
 - stop_loop
 - start_match_loop
 - stop_match_loop
 - match_tap_vector
 - match_mode_vector
 - match_compare_bit
 - match_compare_value
- STF access without the BFM
 - Vector Clocking for STF
 - Pin Group Clocking for STF
 - STF Test Vectors
- Pin map, Pin groups and Macro map files
 - Pin Map file line Format
 - Pin Group file line Format
 - Sample pin map file
 - Sample pin group file
 - Macro Map file
 - Macro file line Format
 - Sample macro file
- Unsupported ITPP commands
 - start_sync stop_sync
 - expandata
 - start_scan stop_scan
 - start_subroutine stop_subroutine
 - compress: no_compress
 - comment
 - vc2sub
 - pscand
 - misr
- SVF support

Reader data formats and other command controls

radix values

Signal expecting values allow values specified with the following radices

- radix

0b, 'b	for binary
0x, 'h	for hex
0d, 'd, none	for decimal

- Example

```
sim: peek_signal mypin 0x3a
sim: deposit_signal mypin 'b0011_1010
```

- commands accepting this format include peek_signal, deposit_signal, force_signal, release_signal, poll_signal, trigger_check

number string values

signal commands allow verilog logic values and (where appropriate) compare string values for comparing and writing.

- The following values are allowed

```
0,1,x,z are literal values for setting or comparing logic signals.  
'X' means ignore for comparing, but can't be driven.  
'Y' means a 0 or 1 for comparing. (also can't be driven).
```

For compare operations (peek_signal) the input string may be a compare string containing 'H','L','X','Z' as with vector statements. 'H', 'L', and 'Z' are treated as '1','0','z' respectively. 'Y' is also allowed.

binary and hex numbers may contain 'X', 'x', 'z' or 'Y'. in addition to binary or hex digits. Values are applied to all four bits of a hex digit or a single bit for binary. Decimal values may not have special characters.

Underscores are also allowed.

In addition, verilog format is allowed (<size>'<radix><value>) (4'ha).

-Example

```
sim: peek_signal mypins 0b1010;  
sim: peek_signal mypins 0bXX10;  
sim: peek_signal mypins 0b10XX;  
sim: peek_signal mypins 0b10YY;  
sim: peek_signal mypins HLXL;  
sim: force_signal mypins 0b1xz0;  
sim: peek_signal mypins 0b1XX0;  
sim: peek_signal mypins 0b1xz0;  
sim: force_signal mypins 0bx11x;  
sim: peek_signal mypins 0bx11x;  
sim: force_signal mypins 4'h5;  
sim: peek_signal mypins 0b0101;  
sim: peek_signal mypins 4'h5;
```

Note that a lower case x in a compare operation expects a lower case 'x' in the compare, and will fail if it has any other value, while a capital 'X' specifies to ignore the bit(s).

timestr

a timestr is a string indicating a verilog time value. e.g. "10000", "10ns", "500ps", "20ms", "10.25ns"

Time strings are used in many commands and configuration variables. Any command accepting a time value accepts these formats.

timing controls

Signal polling and clocking timing is controlled with the timing configuration variables ITPP_DELAY_TIMEOUT (default 1us), ITPP_DELAY_INCREMENT (default 50ps), and ITPP_DELAY_INCREMENT_EXACT (default 1ps).

The defaults are based on a typical verilog timescale of 1ps/1ps. If your project uses a different timescale, or if you need smaller or larger increments or timeout for better accuracy or performance, these settings can be changed using ovm config variables, or with the timing_cfg command described below.

- commands affected by the time parameters include: vector, delay, align_sync, trigger_check, check_clock, pulse_check, edge_alignment_check
- ITPP_DELAY_INCREMENT_EXACT is used for check_clock, pulse_check and edge_alignment_check always
- The others use ITPP_DELAY_INCREMENT_EXACT if ITPP_DELAY_SYNC_EXACT is set and ITPP_DELAY_INCREMENT (inc) otherwise.

Vector and TMS repeat handling

The definition of a repeat count has changed from a "repeat" count to an exact vector count. Configuration/Plusarg switches ITPP_LEGACY_REPEAT_COUNT_VECTOR, and ITPP_LEGACY_REPEAT_COUNT_TMS were added to control the vector count to return to the repeat count behavior for regular vectors and for TMS vectors respectively.

Additionally though, the JTAG BFM added an additional TCK delay to TMS commands starting in version CHASSIS_JTAGBFM_2016WW30_R2 (2016). To partially correct for this, an additional control bit, ITPP_JTAGBFM_HAS_EXTRA_TMS, was added which removes the extra bit for anything more than 1 TMS. Unfortunately with the JTAG BFM change, it is now impossible to get anything less than 2 TCK delays from the JTAG BFM TMS function.

The SPF tool is accounting for this by reducing the repeat number given in the ITPP vector statement by one. In this case you'll want to leave the control bit ITPP_JTAGBFM_HAS_EXTRA_TMS at zero and thus add the extra cycle in. This control bit is defaulting to 0 beginning with rev 0.9.5. Versions 0.9.0 through 0.9.4 have it set to 1 by default.

- resulting timing with control bit setting:

```

for command vector: TMS(0), 3
we want 3 TCK delays in regular mode
                                BFM<2016  BFM>=2016
EXTRA, LEGACY = 00 : cnt =      3*      4
EXTRA, LEGACY = 01 : cnt =     4**      5
EXTRA, LEGACY = 10 : cnt =      2      3*
EXTRA, LEGACY = 11 : cnt =      3     4**

* this is what you want for tms TCK cycles = repeat value
** this is what you want for tms TCK cycles = repeat value +1

```

NOTE: Due to the confusion over this behavior, new parameters have been added into version 0.9.5: ITPP_REPEAT_ADJ and ITPP_REPEAT_TMS_ADJ. Setting these will override the respective LEGACY and EXTRA_TMS settings and allow the user to directly set the behavior. The adjustment can be any integer value (though only -1 to +1 are expected). The parameter value is added to the vector repeat value to determine how many vectors are sent or how many TMS cycles are requested from the JTAG BFM.

checking processes

At the end of the ITPP file, the ITPP Reader will wait for any active background checking processes to complete. (also see wait_for_checks command). These processes include pulse_check, check_clock, trigger_check, run_edge_alignment_check.

Proper coding would have time delays for the known length of the checks needed, but this is enabled as a fail-safe if someone forgets, or doesn't know how long to wait, but still needs the checks to complete.

The timeout is specified by the ITPP_CHECK_TIMEOUT parameter (default 1us). A timeout of 0 indicates no timeout (wait forever if processes never complete). An error is generated if the timeout (if >0) is exceeded before processes are complete.

The time increment between checks is specified by the ITPP_CHECK_INCREMENT parameter (default 100ns).

Vector clocking

Vector commands that use a BFM have default clocking methods that are controlled by the BFM. These include TMS which uses the JTAG BFM and stf_in/stf_out which uses the STF BFM. There is also the STF TestPort (TP) which uses the TP BFM. For some projects this may also include MCI commands.

For other vectors there are two current methods for clocking Vector statements. For lack of better terms these are called Pin Group Clocking and Vector Clocking. Pin Group Clocking assigns a clock, clock edge and optional delay to any pin group. Vector Clocking assigns a single clock pin to every vector statement as a whole, but edges and delays can be assigned to individual pin groups.

Clocks can be driven before or after the drive/checking. In general for a single set of vectors (e.g. STF) it doesn't matter which is used, but when mixing different types or otherwise expecting specific timing, it can make a difference. Statements below indicate the default clock-before-drive behavior. The difference with clock-after-drive is that instead of waiting for the clock edge first, the delay (if any) is executed, then the drive/sample is executed - on the assumption that the simulation is already at the clock edge from a previous cycle. This means, one must be especially careful on the first cycle to ensure you are at the clock edge, or to have garbage setup/spacing data only.

The before/after default behavior is controlled by the config/plusarg variables ITPP_CLOCK_AFTER_VECTOR, and ITPP_CLOCK_AFTER_PIN_VECTOR. Setting ITPP_CLOCK_AFTER_VECTOR to 1 will put vector clocking in clock-after-drive mode by default. Setting ITPP_CLOCK_AFTER_PIN_VECTOR to 1 sets the pin group default mode to clock-after-drive. The pin group before/after behavior can still be controlled with individual pin group settings. (see pin_group_clock command). And the vector clock default behavior can be changed by the vector_clock command.

Vector Clocking

Vector clocking uses the sim commands "vector_clock" and "pin_group_clock" to set the use parameters. These commands are available in the pin group setup file as well as in the ITPP test case file. See below for detailed syntax.

Vector Clocking assigns a clock and a clock edge (fall or rise) to vector statements to be executed. The vector statement will wait for the next clock edge before executing the vector pins. Individual pin groups can also be assigned a clock edge and a delay for precise timing.

As the pin groups are executed (in order, left to right) the edge (if assigned) is compared with the current edge. If they match, nothing happens. If they do not match, then the simulation is advanced to the next clock edge. After advancing to the correct edge (if any), the edge_delay is checked. If set, the simulation will advance to the time after the edge indicated by edge_delay.

Following any clock edge and delay time advances, the pins are driven/sampled.

Edge delay is the absolute time from the clock edge. So if there are multiple pin groups with edge delays, each will execute at the time specified. They are not cumulative delay values. This means each delay should be larger than the prior one (for the same edge). If not, an error is given. For example, if v_A has a delay of 2ns, v_B has 4ns, and v_C has 5ns, then v_A will execute at 2ns after the edge, v_B will execute 4ns after the edge, and v_C will execute 5ns after the edge.

Example

Using STF without a BFM, mixed with other vectors.

```

sim: pin_map: stf_clk my_top_path.xxSTF_CLK;
sim: pin_map: BCLK my_top_path.xxBCLK;

sim: vector_clock: clk=stf_clk fall enable=1;
pin_group_clock: stf_in edge=fall edge_delay=200ps;
pin_group_clock: stf_out edge=rise edge_delay=400ps;

vector: stf_in(0101010101010) stf_out(XXXXXXXXHLHLHLHL);
vector: stf_in(0101010101010) stf_out(XXXXXXXXHLHLHLHL);
vector: stf_in(0101010101010) stf_out(XXXXXXXXHLHLHLHL);
# stf_in 200ps after fall of stf_clk
# stf_out 400ps after rise of stf_clk

sim: vector_clock: clk=BCLK rise enable=1;
vector: my_bck_pins(10100010001010) my_bck_chk(XXHHHXXLLXXLHLHLHL);
vector: my_bck_pins(10100010001010) my_bck_chk(XXHHHXXLLXXLHLHLHL);
vector: my_bck_pins(10100010001010) my_bck_chk(XXHHHXXLLXXLHLHLHL);
# my_bck_pins drive at rise of BCLK
# my_bck_chk check at rise of BCLK

```

results:

```

< fall stf_clk >
< 200ps >      < stf_in >
< rise stf_clk >
< 400ps >      < stf_out >
< fall stf_clk >
< 200ps >      < stf_in >
< rise stf_clk >
< 400ps >      < stf_out >
< fall stf_clk >
< 200ps >      < stf_in >
< rise stf_clk >
< 400ps >      < stf_out >
### Note change of clock here
< rise BCLK >
< my_bck_pins >  < my_bck_chk >
< rise BCLK >
< my_bck_pins >  < my_bck_chk >
< rise BCLK >
< my_bck_pins >  < my_bck_chk >

```

The Vector clock is temporarily disabled if a vector statement has BFM controlled pin groups or pin group clocks (the latter can be disabled - see configuration parameter ITPP_PIN_CLOCK_WITH_VECTOR_CLOCK). The vector statement is scanned for BFM controlled commands and pin groups with clocks before waiting for the vector clock. If found, the BFM timing or pin group timing is used instead of the vector clock.

In the above example the STF BFM is assumed to be disabled, so you are controlling the STF Clock and the pin groups are assigned properly for the STF Pins. However, the Example still works if the STF BFM is being used. In this case the Vector clocking will not activate for the stf pins, and instead the commands will be sent to the STF BFM for timing. The same is true for TMS vectors. TMS will be sent to the JTAG BFM for execution. Testport and MCI (if enabled) also do this. Pin groups with clocks assigned will behave similarly (unless disabled).

Left to right timing

Keep in mind that, as mentioned above, vector statements execute the pin groups in order from left to right, so if two vectors have different edges or timing, you need to place the later timing to the right of earlier timing elements. Also, if you have different edges on pin groups, make sure the pin groups with edges that match the vector clock edge execute first. Otherwise you may get undesired clocking effects.

For example, if you have a falling edge vector clock, then execute pin group A with rising edge followed by pin group B with falling edge, the vector statement will start on the falling edge, wait for the next rising edge, execute group A, wait for the falling edge, execute group B, then the next vector will wait for the NEXT falling edge.

Example: `***(bad)***`

```

sim: vector_clock: BCLK fall enable=1;
pin_group: A my_pins_A;
pin_group: B my_pins_B;
sim: pin_group_clock: A edge=rise;
sim: pin_group_clock: B edge=fall;
vector: A(101010) B(HLHLHL);

```

results:

```

< fall BCLK >
< rise BCLK > < pin group A >
< fall BCLK > < pin group B >
< rise BCLK >
< fall BCLK >
< next vector start >

```

Example: *****(good)*****

```

sim: vector_clock: BCLK rise;
pin_group: A my_pins_A;
pin_group: B my_pins_B;
sim: pin_group_clock: A edge=rise;
sim: pin_group_clock: B edge=fall;
vector: A(101010) B(HLHLHL);

```

results:

```

< rise BCLK >
                < pin group A >
< fall BCLK > < pin group B >
< rise BCLK >
< next vector start >

```

Pin Group Clocking

Pin Group Clocking assigns a clock and edge to a given pin group. When the pin group is executed from a vector statement, ITPP Reader will wait for the clock edge (e.g. rise of TCK, or fall of BCLK) before driving/sampling pins. Pin group clocking also uses the edge delay as in vector clocking for precise timing control.

The main difference with Vector Clocking, and the main reason Vector Clocking was added, is the pin group clock is associated only with a single pin group. So a vector with no associated clock would have no timing associated with it. You could then have vector statements driving and sampling pins, but not advancing time. This is not how it is handled in TVPV, so there would be incompatibilities introduced.

Proper use of Pin Group Clocking has at least one of the pin groups in a vector statement assigned a clock for timing. In the example below, my_grp_in is assigned my_clk. The rising edge is assumed by default. my_grp_out has no timing associated with it. During processing my_grp_out is sampled immediately, while my_grp_in waits for the next rise of my_clk before driving.

Example

```

pin_group: my_grp_out pin345
pin_group: my_grp_in pin1 pin210
# Add a clock
sim: pin_group_clock: my_grp_in clock=my_clk
# vector commands issue data
vector: my_grp_out(HHL) my_grp_in(1010);
vector: my_grp_out(LLH) my_grp_in(0011);

```

results:

```

< my_grp_out >
< rise my_clk > < my_grp_in >
< my_grp_out >
< rise my_clk > < my_grp_in >

```

Note that in this case, since the clock is on the second pin group in the vector, my_grp_in and my_grp_out from the NEXT vector are happening at the same time. In this way, you are sampling the output from after the previous cycle's rising edge with my_grp_out. To change the timing, adjust the clocking and/or the position within the vector statement of the various pin groups, according to the desired behavior.

```

pin_group: my_stf_in clock_fall=stf_clk my_stf_input_pins;
pin_group: my_stf_out clock_rise=stf_clk my_stf_input_pins;

vector: my_stf_in(0101010101) my_stf_out(HHLLHHLLHHHLL);
vector: my_stf_in(0101010101) my_stf_out(HHLLHHLLHHHLL);
vector: my_stf_in(0101010101) my_stf_out(HHLLHHLLHHHLL);

```

results:

```
< fall stf_clk > < my_stf_in >
< rise stf_clk > < my_stf_out >
< fall stf_clk > < my_stf_in >
< rise stf_clk > < my_stf_out >
< fall stf_clk > < my_stf_in >
< rise stf_clk > < my_stf_out >
```

Fast pin access

An optional pin interface has been added to enable fast pin access using direct vcs access rather than relying on SAOLA or UVM generic access by string. Using this should speed up test cases that require heavy use of certain signals - and particularly clock and other polling commands. Additionally, for GLS special sets of such signals are available for scan testing.

The Fast signals are provided in a list before compile time and a script is run (described below) to generate the needed RTL code for the pins.

For further information see "Adding a Fast Pin interface" in the install guide.

fast pin access

By default the normal pin access functions, including pins in pin groups, vectors and polling functions (e.g. delay) will use the fast pins if available. So no special functions are required. Access is the same as other pins. This can be disabled if needed by setting config variables ITPP_FAST_PIN_OVD or ITPP_POLLING_FAST_PIN_OVD.

```
ITPP_FAST_PIN_OVD - enable normal write commands to use fast pins
                    if available (def=1)
ITPP_POLLING_FAST_PIN_OVD - enable polling commands to use fast pins
                           if available (def=1)
```

timing

Fast pins use an array based method to interface with the rtl model. Using this, the signals are available in the model on the following sim tick. This is the same as usual, however there can be a difference within the reader itself. Using string based methods, itpp commands will see the affects of a write immediately. But using the fast pins, the input values won't be seen until the next rtl sim tick. Normally this won't be an issue since this is how the model typically behaves. But this can cause some behavioral timing differences depending on the coding methods used and whether ITPP is driving the signals or they are coming from rtl sources.

Fast pins are forced with vcs force commands (thus bypassing string based methods). To mimic the affect of a deposit command signals can be automatically released. If enabled the signals will be released on the following sim tick. (see Auto release and deposit below).

pin size

Fast pins can be up to 32 bits wide, similar to other pins defined in the pin map. However, each different size or bit pattern to be used in the itpp file must be specified in the fast pin list pre-compile. Only those listed will be identified as fast pins and use the fast pin methods. Others will use the standard string based methods. Scan pins are all single bit signals.

Auto release

There is a mechanism which will release the signals automatically. This is off by default, so the pins need to be manually released if needed.

(see "deposit" below for deposit_signal exception to this behavior.)

If enabled, auto release will release a signal on the following simulation tick. To set/clear auto release use these config settings.

```
ITPP_FAST_PIN_AUTO_RELEASE - fast pins are automatically released after
                             force (def=0)
```

deposit

The mechanism for fast pins uses force commands to drive the signals. Using the deposit_signal with fast pins, or if deposit is the default access method, will enable the auto release for each such signal individually in order to mimic deposit behavior as closely as possible. The signal will be forced for one simulation tick, then released.

scan chains

The scan commands below (force_scan, check_scan, release_scan) operate on special sets of fast pins organized by scan chain. One set is for write operations ("scan write" pins) and another is for read operations ("scan read" pins). A third is "extra" chains which are independent of write and read scan chains and their special circumstances (particularly skip bits). The write and read chains are intended to represent the input and output nodes respectively of the same set of scan chains, so they are expected to have the same number of bits in each chain and the same number of chains. But other than chains being sticky across scan commands by default, they are operated independently, so the user can set it up in any manner they see fit, if they find it more useful. Keep in mind though that the chains are set up at compile time, so no signal changes are allowed during a simulation.

Skip Bits/special chains

Special chains are created for each of the skip bits (skip_bits and pre_skip_bits) and for the start bit from each chain (start_chain). These can be renamed as needed and accessed with the chain's name from the force_scan commands's chainid option only. The skip bit chains are named "skipbit_chain_<num>" and "pre_skipbit_chain_<num>" by default. The number starts at zero from the outside. i.e. skipbit_chain_0 has each of the msb bits of each scan chain, and pre_skipbit_chain_0 has each of the lsb bits of each scan chain. Note that pre_skipbit_chain_0, if it exists, has the same bits as the start_chain.

The write scan chains can have skip bits and 'pre' skip bits. These are a fixed number of bits at the end or beginning (for 'pre' skip bits) set in the scan chain setup file with the commands below.

```
scan_write_skip_bits <number>          (default: 2)
scan_write_pre_skip_bits <number>      (default: 0)
scan_write_skip_bit_chain_name <name>   (default: skipbit_chain)
scan_write_pre_skip_bit_chain_name <name> (default: pre_skipbit_chain)
scan_write_start_chain_name <name>     (default: start_chain)
```

Skip bits are meant to be undriven while the rest of the chain is forced. They can be driven if specified in the data of the force_scan command, but are driven separately from the rest of the chain. This is slower, and should be avoided unless necessary. The number of skip bits is a compile time setting and cannot be changed.

force_fast

- syntax

```
sim: force_fast: <path> <value>;
```

-parameters path - pin or rtl path to the fast pin signal value - value to force.

The force_fast command will force a given fast pin (and only a fast pin) to the provided value using the fast pin mechanism, regardless of the configuration bit status. A pin name defined in the pin_map that also is a fast pin can also be specified. Value is the same as for the force_signal command. The force takes place one sim tick after the command is executed.

peek_fast

- syntax

```
sim: peek_fast: <pin> <value>;
```

-parameters path - pin or rtl path to the fast pin signal value - value to force.

peek_fast compares the pin value with the given value and generates an error if there is no match. The value is the same as for a peek_signal command.

release_fast

- syntax

```
sim: release_fast: <pin>;
```

-parameters path - pin or rtl path to the fast pin signal

release_fast causes a fast pin to be released. The release takes place one sim tick after the command is executed.

force_scan

Writes to the specified bits in a scan chain.

- syntax

```
sim: force_scan [chain=<#>] [chainid=<idstring>] [start=<#>] [reverse] [last_chain] <bitvector> ;
```

- parameters

chain	Scan chain number to process. Chooses which scan chain to force. Sticky from last scan command (any command type) (last chain accessed)
chainid	String identifying the desired chain force. Can also specify extra chains including the start_chain, or skip_bit chains. Sticky from last scan command (any command type) (last chain accessed)
start	Chooses which bit to start scanning at. Default is bit 0.
reverse	Specifies the start pin is the lsb of the bitvector default is msb is the start pin.
last_chain	Start with chain and start bit from previous scan command.
bitvector or	string of bit values (can contain underscores for spacing) determining the data to drive/compare. Valid bits include 1,0,x,X,z

Only one of chain or chainid is needed. If none is specified, the chain used is the last one used in a prior scan command (any type).

- example

```
sim: force_scan chain=0 start=0 111000_zx1000;
sim: force_scan chainid=my_scan_chain start=0 111000_zx1000;
sim: force_scan chainid=start_chain start=3 000_zx1000;
sim: delay_time 10ps;
sim: release_scan chain=0;
delay: clk10(0);
sim: check_scan chain=0 start=0 100100_zx1000;
sim: check_scan last_chain reverse 0001xz_001001;
```

Note that the entire chain is forced together regardless of which bits are set in the data, with the exception of Skip bits. This is done for performance, as a single vector force takes about as much time as a single bit force. Hence, we want to include as many bits as possible into the force operation. Typically, if there are skip bits, it is expected that the start bit will be set to the number of skip bits, and the data vector will only have bits up to the last non-skip bit on the pre_skip_bit side (chain start side). This means no valid data is set corresponding to the skip bits. Only if there is data on the skip bits will they be forced on an individual bit basis. This is slows simulation by requiring extra forces to occur. If all the bits from each chain corresponding to a particular skip bit need to be forced, the extra chain "skipbit_chain_<num>" or "pre_skipbit_chain_<num>" can be used. (The name can be customized).

check_scan

- syntax

```
sim: check_scan [chain=<#>] [chainid=<idstring>] [start=<#>] [reverse] [wrap] [next_chain]
[last_chain] <bitvector> ;
```

- parameters see force_scan valid bits include 1,0,X,x,z Capital 'X' is don't care. Lower case 'x' is comparing for a logic 'x' value.

release_scan

- syntax

```
sim: release_scan [chain=<#>] [chainid=<idstring>] [last_chain] ;
```

- parameters see force_scan

The entire chain is released on a release_scan command.

Debug signals

If using the ITPP Reader Pin Interface, Debug signals are made available for tracking ITPP flow using waveforms.

- signals include

itpp_last_label	string with the last label value executed.
itpp_command	string with the last ITPP Command started.

itpp_loop_info	string with current loop data
itpp_line_start	ITPP line number of the last ITPP command started.
itpp_line_end	ITPP line number of the last ITPP command completed.

Reader (parser) commands

label

Simulation treats this as a comment. Print the string in the log file. Tester uses this to tag input/output with the label

- syntax

```
label: <string>
```

parser_config

set configuration variables. Normally config settings are made via PlusArg or OVM/UVM config vars. This option is in place to allow setting of certain variables within the ITPP file, which can be more convenient at times.

- syntax

```
sim: parser_config: <cfgid> <value>
```

parser_debug_mode

- Controls display features of the ITPP reader.
- Can set/clear debug mode which prints some debug info for which OVM_DEBUG doesn't work, and enables deeper debug for some functions.
- Also show or hide remark (rem:) statements in the log files. Default is to display remarks, some projects need to remove them due to disk space concerns on very long ITPP simulations.
- Can also set the OVM verbosity for the ITPP Reader.
- Can set certain configuration bits that normally are only set with config and/or plusarg values.
- syntax

```
sim: parser_debug_mode: [ hide_remarks | show_remarks | <debug value> | verbosity <val> | config <var> <val> ]
```

- parameters

hide_remarks	do NOT display remark statements in the log file
show_remarks	display all remark statements in the log file
debug_value	Sets the debug mode. 1= on, 0=off. Displays some more debug info that OVM_DEBUG doesn't capture.
verbosity	set the ITPP Reader OVM verbosity level
config	set a config variable

set_active_*

NOT FUNCTIONAL CURRENTLY	FUTURE IMPLEMENTATION
---------------------------	------------------------

- syntax:

```
sim: set_active_tap <value>
sim: set_active_stf <value>
sim: set_active_tp <value>
sim: set_active_sim <value>
```

Future Implementation. These functions are intended to activate one or more of multiple instances of a BFM. For example, multiple TAP networks in a model. They currently control which of several sequencers are active for a pipe. All sequencers are sent to each appropriate command pipe. This “should” work, but has not been tested, is not currently approved by the ITPP Working Group, and is thus not available on the Tester or Emulation.

time_snapshot

- syntax

```
sim: time_snapshot [tolerance <tol>] [delta <time>]
```

- parameters

tolerance	amount of leeway allowd in time comparison
delta	expected time difference from last snapshot

This will save the current time for future comparison. If <delta> is specified, a comparison is done with the previous time snapshot. If the difference is not equal to the value specified within the tolerance specified (if any) an error will be given.

This is intended for internal testing, but could be useful so is being made generally available.

macro_map

- syntax

```
sim: macro_map [<name> <value>]
```

- parameters:

name	the name of the pin
value	the macro value to replace name

Macro_map defines macros to use as shortcuts when defining RTL signal names to be used in later commands. These are added to the macro map created with the (optional) macro map file upon startup.

Macros are referenced using '<macro name>' in a signal reference command.

If no parameters are given, the contents of the macro map are displayed to the log file.

- Example

```
sim: macro_map top_pins cabist_subsystem_TbTop
sim: pin_map my_pin_2 `top_pins.my_val_pin_2
sim: peek_signal my_pin_2 0
sim: peek_signal `top_pins.my_val_pin_4 1
```

pin_map

- syntax

```
sim: pin_map [<name> <path>]
```

- parameters:

name	the name of the pin
path	the RTL path of the signal

pin_map defines RTL pin names to be used in later commands. Most commands that require a signal, require the signal to be defined in the pin map. Those that don't require it can still use it. Pins can be multiple bit. The size is determined by the path name given, so a partial vector can be used.

If no parameters are given, the contents of the pin map are displayed to the log file.

pin_map also allows the alternate TMS command to be set by giving a pin name of "TMS".

- Example

```
sim: pin_map my_input_vec my_rtl_path.my_signal[8:5];
sim: pin_map TMS xxtms;
```

pin group

- syntax

```
pin_group: <group name> [<clock_desc> [=] <clock pin>] <pin> [,] [more pins ...]
```

Pin groups are used in vector commands and some other sim commands to drive or compare groups of rtl signals. The pins making up a pin group must be defined in the pin map. If enabled (default) the RTL path may be specified instead of a pin name and the pin map entry will be made automatically. This, however, removes the naming of the pin out of user control.

A user can associate a clock with a pin group with the “clock [pin](#)” parameter. (See Pin Group Clocking above). Doing so causes the pin group to wait for the clock edge before proceeding to read or write signals. “clock_hl” or “clock_fall” specifies to wait for the falling edge, while “clock” or “clock_lh” specifies the rising edge.

- parameters:

<group name>	the name of the group
<clock_desc>	clock clock_lh clock_hl clock_fall to associate a clock with this group. clock_fall or clock_hl specifies the falling edge.
<clock pin>	the pin name or rtl path of the clock
<pin name>	name of the pin or RTL path to use

If no parameters are given, the contents of the pin group map are displayed to the log file.

- options:

ITPP_CLOCK_AFTER_PIN_VECTOR=1	config variable or plusarg to specify waiting for clock after driving pins.
-------------------------------	---

Using pin_groups with clocks as a mini BFM

```
# Define the pins:
sim: pin_map bpin5 bpin5[4:0] <- bpin5, size 5, lo = 0, hi = 4
sim: pin_map pin1 path[4] <- pin1, size 1, lo=4, hi=4
sim: pin_map pin2 path <- pin2, size 1, lo=0, hi=0
sim: pin_map pin210 path[2:0] <- pin210 size 3, lo = 0, hi = 2
sim: pin_map pin345 path[5:3] <- pin345 size 3, lo = 3, hi = 5
sim: pin_map my_clk path
# Define the groups
pin_group: my_pins pin1 pin2 pins345[2:0] pins
pin_group: my_grp_out pin345
pin_group: my_grp_in pin1 pin210
# Add a clock
sim: pin_group_clock: my_grp_in clock=my_clk
# vector commands issue data to BFM
vector: my_grp_out(HHL) my_grp_in(1010);
vector: my_grp_out(LLH) my_grp_in(0011);
```

This would send the my_grp_out followed by my_grp_in to the cmd_pipe. Since the clock is associated with the my_grp_in, it doesn't toggle until it sees it. so you get Check, Clock, Drive - in that order. You could also do it the other way and have the clock associated with the out group in which case you'd get Clock first, then Check, then Drive. To emulate the STF and TP, associate the clock with the input and issue in first -

```
vector: my_stf_in(0101010101) my_stf_out(HHLLHHLLHHLL);
```

A full example is shown below in the section titled “STF Access without the BFM” Similar methods can be applied for MCI and many other interfaces.

NOTE# a couple of bugs prior to version 0.4.1

- clock_hl (falling edge) operates as rising edge, so there's not a working falling edge option. To work around it, you may use an inverted clock.
- The clock is waited for AFTER driving the pins, not before, as stated.
- The ITPP_CLOCK_AFTER_PIN_VECTOR option was added to allow either method.
- For driving STF without a BFM, this works out nicely as specifying the positive stf_clk on the input vector, ends up giving correct timing for driving and comparing signals.

vector_clock

- syntax:

```
sim: vector_clock: [<pin>] [rise/fall] [before/after] [enable=0/1]
```

- parameters:

<pin>	pin or path to set as the vector clock, enable turned on
enable	enable(1)/disable(0) vector clocking
[rise]	vectors will wait for the rising edge
[fall]	vectors wait for the falling edge
[after]	clock occurs after drive/sample vector
[before]	clock occurs before drive/sample vector (default)

Parameter values are sticky unless a pin is specified. If a pin is specified, other parameters are set to default (enable=1, rising edge) unless specified. Upon completion, the current status is printed to the log. if no options are given, nothing changes - only the current status is printed to the log. See Vector Clocking description above for more details.

Time does not advance for vectors if Vector Clocking is disabled (enable=0 or never set), unless a pin group in the vector statement has an associated clock, either through a BFM (e.g. TMS or stf_in/stf_out), or through pin group clocking. This is not typical for TVPV. Therefore, it is expected that Vector clocking will be enabled for cases that are expected to go to Silicon testing with non-BFM driven pin groups.

pin_group_clock

- syntax

```
sim: pin_group_clock: <group> [edge=rise/fall] [edge_delay=<timestr>]
                        [before/after] [<clock_desc>=<clock pin>]
```

pin_group_clock sets the timing details for a pin group using either vector clocking or pin group clocking. Setting a clock pin using one of the <clock_desc> settings puts vectors with this pin group into pin group clocking mode. In pin group clocking mode vector clocks (acting on the line as a whole) are ignored and timing control switches to using the pin group clock(s). (see Pin Group Clocking above)

- parameters:

group	name of the pin group
edge	which clock edge should this pin group execute after (rise or fall) for vector or pin group clocking.
edge_delay	time string indicating how long after the clock edge to execute (drive/sample) the pin group.
before	put this pin group in clock-before-drive mode (default)
after	put this pin group in clock-after-drive mode
<clock_desc>	clock clock_lh clock_hl clock_fall identifies a pin group clock for this group clock clock_lh indicates transactions after rising edge clock hl clock_fall indicates transactions after falling edge
<clock pin>	the pin name of the pin group clock.

- for vector clocking:

group	must be a previously defined group.
edge	identifies the rise/fall edge before driving/sampling the group.
edge_delay	specifies a delay after the edge before driving/sampling.

See Vector Clocking description above.

- for pin group clocking:

group	must be a previously defined group.
clock_pin	the clock pin must be a previously defined pin.

clock_desc	the clock type specifier sets the edge (rising or falling) before driving/sampling signals. The edge set by the 'clock_desc' is overridden by the 'edge' setting.
edge	identifies the rise/fall edge before driving/sampling the group. overrides the clock_desc setting and
edge_delay	specifies a delay after the edge before driving/sampling.
before	This pin group will wait for the clock before driving/sampling data (default).
after	This pin group will drive/sample data before waiting for the clock edge.

See Pin Group Clocking and Vector Clocking descriptions above.

For pin group clocking, This command allows clock association without changing the pin_group command line, which doesn't have a clock option for silicon testing.

The clock pin associates a clock pin with a pin group, enabling pin group clocking. Vector statements will wait for the clock before driving or checking pins in a pin group with a clock association.

Clock descriptors are the same as for a pin_group command. if "clock" or "clock_lh" is specified, the pin group will wait for the rising edge of the clock. if "clock_fall" or "clock_hl" is specified, pin group will wait for the falling edge of the clock. The edge setting, if specified, overrides these clock descriptor settings.

In clock-after-drive mode, the edge_delay happens first, then the drive/sample, and finally wait for the clock edge. A before/after setting overrides the default set by the config/plusarg variable ITPP_CLOCK_AFTER_PIN_VECTOR (which defaults to "before").

vec_pin_group

- syntax

```
sim: vec_pin_group: <group name> [clock <clock pin>] <pin name or path> [more pins ...]
```

Same as pin_group, but will allow multiple bit pins even if disabled for regular pin_group commands.

vector

- syntax

```
vector: <name>(value) [<name>(value) ...][, <repeat value>];
vector: <name>(value) [<name>(value) ...]
[<name>(value) ...]
[<name>(value) ...][, <repeat value>];
```

drive or compare pins, pin groups or pre-defined vector groups to the model. Pre defined vector names include TMS (tap), stf_in, stf_out (STF network), tp_in, tp_out (STF TestPort). The TMS vector is handled by the TAP pipe, the STF vectors are handled by the STF pipe, and TP vectors are handled by the TP pipe. Other vector commands can be defined by custom implementations.

As of version 0.9.1 vectors can use pins in the pin map, or rtl paths directly. There can be no timing associated with these other than vector clocking.

As of version 0.9.1, the vector command can be split over multiple lines. A vector line not ending in a semicolon is assumed to continue to the next line. The next line(s) are processed as a vector continuation until a semicolon is reached. If a vector command is missing the semicolon at the end, you will likely get a strange error message about the command on the next line not being a valid rtl path or missing a vector value. Any repeat value specified on the last line applies to the entire vector command.

For clocking and timing considerations see Vector Clocking and Pin Group Clocking above. Also see command definitions for "pin_group", "vector_clock", and "pin_group_clock".

Vector statements are processed left to right, and any timing considerations need to account for this.

A repeat value applies to the entire line as if the whole line were listed that many times in the file. Repeat counts prior to version 0.6.0 were processed as number+1 executions of the vector line. Meaning it meant "how many times to repeat the line". Post 0.6.0, it now is the exact number of executions of the vector line. Meaning "how many times to execute the line". Zero is invalid and produces an error.

There are switches to use the legacy mode if needed. (see "ITPP Reader OVM Config Variables", and "ITPP Reader PLUSARGS" sections).

Also see "vector and tms repeat handling" for notes on TMS timing using the JTAG BFM.

wait_for_checks

- syntax

```
sim: wait_for_checks: [timeout=<time>] [inc=<time>;
```

- Example:

```
sim: wait_for_checks: inc=10ns timeout=1us;
```

This comand waits for background checking processes to complete. These include pulse_check, check_clock, trigger_check, run_edge_alignment_check.

The timeout is specified by the ITPP_CHECK_TIMEOUT parameter (default 1us) or can be overridden by specifying the “timeout” parameter. A timeout of 0 indicates no timeout (wait forever if processes never complete). An error is generated if the timeout (if >0) is exceeded before processes are complete.

The time increment between checks is specified by the ITPP_CHECK_INCREMENT parameter (default 100ns) or can be overridden by the “inc” setting.

set_cmd_prefix

- syntax

```
sim: set_cmd_prefix: <TAP|STF|TP> <prefix> <id>
```

- Example:

```
sim: set_cmd_prefix: TAP tap1 1;  
tap1_scand: 00100100, HHLXXXX;
```

set_cmd_prefix is available as a sim command or as a pin groups file entry (without the “sim:”). set_cmd_prefix enables the user to set a prefix name for commands issued to a BFM - in particular the TAP and STF commands. The ITPP sequencer pre-defines names to access different BFM sequencers if enabled.

For TAP the first two prefixes are “pri” and “sec” and will correspond to the first an second BFM sequencer attached during setup (using the set_tap_sequencer function). After that names default to “tap<id>” starting at id=2 (pri=0, sec=1). For STF, the default names are all “stf<id>”. Command names allowed are the base command, and <prefix>_<command>. The base command accesses the currently active BFM, if set_active* is used (non-standard), or the first (primary) if not. Note that this means there can be more than on way to access the primary BFM (e.g. scani and pri_scani).

Though changing the command prefixed during a test is allowed, it is not recommended. The prefixes and the BFM’s associated with them should be set at the beginning of simulation - either when setting up the sequencers initially, or through the pin groups file.

TAP commands

scani

- syntax

```
scani: <bin command>[, <compare data>] ;
```

- Example

```
scani: 00110001 ;  
scani: 00110001, XXXXXXXH ;
```

Shifts larger than MAX_DR_REG_LENGTH are split into multiple shifts going through pause-ir between shifts. Multiple shift behavior is controlled with the config or plusarg variables ITPP_ALLOW_SPLIT_SCAN_CHUNKS. Set it to 0 to disable multiple shifts.

Optional comparison operations can only do a single shift	no pause, and exit to run-test-idle instead of Exit1-IR. This is due to a limitation in the JTAG BFM. The BFM sequence that captures TDO goes to idle state, and the sequence that goes to Exit1-IR does not capture TDO.
---	---

If a compare operation is specified, a previously set MASK vector will mask the TDO output for this command. The mask is cleared after one instruction. Also data is captured for the capture_fifo command if a capture vector is set. Neither mask nor capture is available if a compare operation is not specified. Note that the comparison operation on a scani op is not standard and does not translate to the tester env - only in simulation.

scand

- syntax

```
scand: <drive data>, <compare data>
```

- Example

```
scand: 10001001, HLLHLL ;
```

Tap shift data operation. Shifts longer than MAX_DR_REG_LENGTH are split into multiple shifts going through pause-dr between shifts. Multiple shift behavior is controlled with the config or plusarg variables ITPP_ALLOW_SPLIT_SCAN_CHUNKS. Set it to 0 to disable multiple shifts. A previously set MASK vector will mask the TDO output for this command. The mask is cleared after one instruction. Data is captured for the capture_fifo command if a capture vector is set

to_state

- syntax

```
to_state: <state name> ;
```

- Example

```
to_state: Run-Test/Idle ;
to_state: Pause-DR ;
```

```
"Test-Logic-Reset":      target = TAP_TLRS;
"Run-Test/Idle":         target = TAP_IDLE;
"Select-DR-Scan":        target = TAP_SELECT_DR;
"Capture-DR":            target = TAP_CAPTURE_DR;
"Shift-DR":              target = TAP_SHIFT_DR;
"Exit1-DR":              target = TAP_EXIT1_DR;
"Pause-DR":              target = TAP_PAUSE_DR;
"Exit2-DR":              target = TAP_EXIT2_DR;
"Update-DR":             target = TAP_UPDATE_DR;
"Select-IR-Scan":        target = TAP_SELECT_IR;
"Capture-IR":            target = TAP_CAPTURE_IR;
"Shift-IR":              target = TAP_SHIFT_IR;
"Exit1-IR":              target = TAP_EXIT1_IR;
"Pause-IR":              target = TAP_PAUSE_IR;
"Exit2-IR":              target = TAP_EXIT2_IR;
"Update-IR":             target = TAP_UPDATE_IR;
```

tap_hard_reset

issue hard reset (pull TLR pin)

tap_soft_reset

issue soft reset (TMS=1 for 5 cycles)

update_to_seldr

A True setting of This command will give a warning if BFM calls are to be made to a function that automatically goes to IDLE. (Currently this occurs only on scan commands which check TDO values.) In silicon testing this command indicates to send the TAP directly from Update-DR or Update-IR to the Select-DR-Scan state instead of routing through IDLE. In simulation, the JTAGBFM handles routing and already goes the short route. In order to go to IDLE, one must include a "to_state: Run-Test/Idle;" command, even if the setting is False.

- syntax

```
update_to_seldr: True|False;
```

capture

set a list of bit ranges to capture TDO bits from scand ops

- syntax

```
capture: <bit/bit range>[, ...]
```

- Example

```
capture: 15, 14, 13, 12, 0-7, 4-0, 7-4
```

mask

set a mask vector for scand ops the mask will ignore bits in the return data. similar to comparing to X, instead of H or L

- syntax

```
mask: <bit/bit range>[, ...]
```

- Example

```
mask: 15, 14, 13, 12, 0-7, 4-0, 7-4
```

capture_fifo

compare the last scand data captured (setup by capture command) comparisons are done in reverse order - i.e. first compare is for the last field listed. available compare commands are: equal, not, less, greater, between

- Example

```
capture: 15, 14, 13, 12, 0-7, 4-0, 7-4
scand: 00000000000000000000000000000000, LLLLLLHHLLLLLLLLHHHLHLHLHH_HHLL_HHLH ;
to_state: Run-Test/Idle ;
sim: capture_fifo equal 0b1100 <- compares bits 7:4 (HHLL) with 1100
sim: capture_fifo less 0b01111 <- compares bits 4:0 (LHHLH) with 01111
sim: capture_fifo equal 0b11001101 <- compares bits 7:0 (HHLL_HHLH)
sim: capture_fifo equal 0 <- compares bit 12
sim: capture_fifo equal 1 <- compares bit 13
sim: capture_fifo equal 0 <- compares bit 14
sim: capture_fifo equal 1 <- compares bit 15
```

TMS vector

set the TMS vector

- syntax

```
vector: TMS(<val>) [, <repeat>];
```

Often used to cause a TCK delay when in IDLE state Can also use the delay command if the TCK is free-running, or the tap_idle command below.

See section "vector and TMS repeat handling" for specific timing concerns. <val> is a single bit value (0 or 1) and is sent to the JtagBfm as one string repeated using the repeat value. For the newer BFM (2018 and beyond) this has the affect of delaying repeat+1 cycles in the IDLE state with <val>=0.

An alternate vector command instead of "TMS" is allowed and can be set using pin_map (or in the pin map file) or as the config/plusarg variable ITPP_ALT_TMS_CMD.

- alternate TMS example

```
sim: pin_map TMS xxtms;
```

tap_idle

IDLE for a number of cycles.

- ```
sim: tap_idle: <val>;
```

## STF commands

execute STF vectors

- ```
vector: stf_in(<drive vec>) stf_out(<compare vec>) [, <repeat>]
```

- ```
vector< stf_in(000000000000000000000000000000001000001) stf_out(LLLLLLHLLLLLLHLHLHLLLLLHLHLLLLXX
XXXXXXX);
```

- syntax

```
sim: stf_queue_mode [<ON>] [<OFF>] [<DEFAULT>] [<timeout=cycles>]
```

- ```
sim: stf_queue_mode ON timeout=15
```

- syntax

```
sim: stf_drv_cmd <cmd> [<data>]
```

- syntax

```
sim: stf_shift_count [<value>] [end] [cfg_iv=0|1] [cfg_op=0|1] [spofi_enable=1|0]
```

A shift count for STF vectors is displayed in the log for GLS to determine failure info. This count increments for each valid non-zero STF instruction sent to the STF BFM by the ITPP Reader. By default the count resets to zero after each "label" command. (first STF Vector is shift number 1). This function modifies the counting behavior. Setting a value sets the current value of the shift count, and disables clearing of the shift count by labels. Specifying "end" restores clearing by labels. `cfg_iv` and `cfg_op` tell when to increment shift count `iv=1` means input valid must be asserted, else any allowed `op=1` means opcode must be non-zero, else any allowed when enabled, increment the count and add it to the message, so GLS can parse it upon detecting an error. Default is both are zero, counting every cycle. `spofi_enable` enables or disables `spofi` messages for failures in the STF BFM.

- parameters:

<value>	shift count value and disable clearing by label
end	restore clearing shift count by labels
cfg_iv	if 1, shift count increments only if iv=1 (def=0)
cfg_op	if 1, shift count increments only if opcode is non-zero (def=0)
spofi_enable	1/0, enable/disable spofi messages from stfbfm (def=0)

`cfg_iv` and `cfg_op` also set identifiers displayed in STF debug/error messages.

- Message values are:

"NOP"	iv=0 op=0
"IV"	iv=1 op=0
"OP"	iv=0 op=1
"IVOP"	iv=1 op=1

stf_reset_assert

Call the STF_BFM sequence `STF_BFM_AssertResetSeq` to assert the STF Reset pin. Allows setting of the sequence's timing parameters. Default is zero to allow ITPP file to control the timing by default.

- syntax:

```
sim: stf_reset_assert [pre=<cycles>] [post=<cycles>] [trans=<cycles>]
```

- parameters:

pre	cycles delay before reset assertion
trans	cycles delay after reset assertion
post	additional cycles delay after reset assertion

stf_reset_deassert

Call the STF_BFM sequence `STF_BFM_DeassertResetSeq` to deassert the STF Reset pin Allows setting of the sequence's timing parameters. Default is zero to allow ITPP file to control the timing by default.

- syntax:

```
sim: stf_reset_deassert [pre=<cycles>] [post=<cycles>] [trans=<cycles>]
```

- parameters:

pre	cycles delay before reset deassertion
trans	cycles delay after reset deassertion
post	additional cycles delay after reset deassertion

SIM commands

Various commands for interacting with simulation without interfacing with a BFM.

All SIM commands are started with "sim:" or "rem:" followed by the action command (detailed below). "rem:" and "sim:" are functionally equivalent, except "sim:" command lines will generate errors for unknown commands, but "rem:" treats unknown commands as comments. "sim:" is preferred to rem: in order to leave the rem: comments for silicon test cases, since these comments appear in the silicon test log. sim: commands will not appear in the silicon test log.

Signal functions:

peek_signal

- syntax

```
sim: peek_signal <pin/group> <value / '!x'>
```

Read the given signal and compare with the given value fail if the value does not match. If no value is given, The read is performed and the value is printed in the log file. A pin can be a pre-defined pin in the pin map, an rtl path, or a pin group. The macro map can be used to complete the rtl path. set PlusArg +ITPP_XCHECK_WARN to issue a warning instead of an error when checking for X

- Example

```
sim: macro_map top_pins cabist_subsystem_TbTop
sim: pin_map my_pin_2 `top_pins.my_val_pin_2
sim: peek_signal my_pin_2 0
sim: peek_signal `top_pins.my_val_pin_4 !x
sim: peek_signal `top_pins.my_val_pins[4:0] 0xa
sim: peek_signal mygroup 0b111100XX1010
```

- valid radix values include (for all signal ops)

```
binary - 0b 'b
hex - 0x 'h
decimal - 0d 'd or none
```

Don't care bits can be specified by using 'X' in a binary or hex string. (4 bits for a hex digit 'X') See "number string values" for more details.

deposit_signal

- syntax

```
sim: deposit_signal <pin/group> <value>
```

set the given pin to the given value A pin can be a pre-defined pin in the pin map, an rtl path, or a pin group. The macro map can be used to complete the rtl path.

force_signal

- syntax

```
sim: force_signal <pin/group> <value>
```

set the given pin to the given value using a force command. A pin can be a pre-defined pin in the pin map, an rtl path, or a pin group. The macro map can be used to complete the rtl path.

release_signal

- syntax

```
sim: release_signal <pin/group> <value>
```

undo a force operation so deposits or rtl activity can drive the pin A pin can be a pre-defined pin in the pin map, an rtl path, or a pin group. The macro map can be used to complete the rtl path. The value seems to have no effect - the pin is released no matter what value is specified

poll_signal

- syntax

```
sim: poll_signal <pinname> <pinval> [<timeout>] [increment]
```

Wait for the given signal to attain the given value. Checks every <increment> (default 1ns) time period. Fails if the value is not set after timeout (default 1ms). A pin can be a pre-defined pin in the pin map, or an rtl path, or it can be a pin group. The macro map can be used to complete an rtl path.

- Parameters

pinname	pin name, rtl path or pin group to check
pinval	value to look for on the signal
timeout	time limit for search. fails if exceeded. (default=1ms)
increment	(optional) (default 1ns) time step for checking.

If using pin group polling, the pinval can be a compare string with don't care's (includes x,z,X,Y,Z). A pin/path can only have x,z in addition to digits and radix. See "number string values" for more details.

Note that pin/path polling is more time efficient than pin group polling due to the string manipulation involved in pin group data collecting. So, if don't care's and pin groups are not needed, pin/path is preferred.

Timing functions:

delay / delay_fall / delay_rise

- syntax

```
delay: <pin>(cycles);
sim: delay_fall: <pin>(cycles);
sim: delay_rise: <pin>(cycles);
```

The clock pin can be a pin in the pin map, a pin group (if more than one bit, only the last bit is used), or an RTL path.

Sync up with the clock, then Wait for specified number of clock cycles. If time is already at the clock boundary, then the affect is to wait cycles+1 clocks. So to advance one clock, use delay: clk(0); "delay_fall" waits for a falling edge of the clock (high, then low). delay_rise waits for a rising edge of the clock (low, then high). delay waits for a rising edge by default, but waits for the falling edge of a pin which is defined as a clock (see define_clock) with sync parameter set to "fall". The default edge can be changed with the config/plusarg parameter ITPP_DELAY_SYNC_DEFAULT.

set_delay_inc

- syntax

```
sim: set_delay_inc: [<timestr>];
```

Override the default time increment for delay commands. If no value or zero is specified, restore default behavior, otherwise use the value specified for all future delay and delay_fall commands.

This is intended to be used to improve simulation performance by reducing the number of polls between clocks in a "delay" command. It is easy to use for an individual clock and then reset to default settings as needed.

- Example

```
sim: set_delay_inc: 100ns;
delay: clk(1);
sim: set_delay_inc: 0;
```

```
sim: set_delay_inc 1ns;
delay: stf_clk(5);
sim: time_snapshot delta 60ns tolerance 1ns;
sim: set_delay_inc 0.5ns;
delay: stf_clk(5);
sim: time_snapshot delta 60ns tolerance 0.5ns;
*** back to default
sim: set_delay_inc;
delay: stf_clk(0);
sim: time_snapshot;
sim: set_delay_inc 1ns;
delay: stf_clk(5);
# should start at the edge, so even 1ns delays should match up exactly
sim: time_snapshot delta 60ns tolerance 1ps;
```

delay_time

- syntax

```
sim: delay_time: <timestr>
```

a timestr is a string indicating a verilog time e.g. "10000", "10ns", "500ps", "20ms"

Delay for a given time

timing_cfg

- syntax

```
sim: timing_cfg: [timeout[=]<timestr>] [inc[=]<timestr>] [short_inc[=]<timestr>] [check_timeout[=]<time>] [check_inc[=]<time>]
```

Set and/or display the delay command parameters which are used for all polling events, such as delay commands, checking and generating clocks and vector clocking. Also displays the current settings to the log file. If no parameters are given, it will just display the current settings.

This command changes (or displays) the values set by the configuration variables ITPP_DELAY_TIMEOUT, ITPP_DELAY_INCREMENT, and ITPP_DELAY_INCREMENT_EXACT, ITPP_CHECK_TIMEOUT, ITPP_CHECK_INCREMENT respectively.

- Example

```
sim: timing_cfg: timeout=1us inc=50ps
```

- Parameters

timeo ut	the timeout value for polling signals
inc	updates the parameter ITPP_DELAY_INCREMENT which is the time increment between signal polling checks
short_ inc	updates the parameter ITPP_DELAY_SYNC_EXACT which is the time increment for minimum delay polling. this includes pulse_check and check_clock commands, and all clock/polling commands except sim: poll_signal when ITPP_DELAY_SYNC_EXACT mode is set.
check_ time out	timeout for background process completion checks
check_ _inc	time increment for background process completion checks

- see timing controls above

Checking and clock functions:

trigger_check

- syntax (single line)

```

sim: trigger_check trig=<pin/path> clock=<pin/path> [edge=<rise|fall>]
    [trig_val=<1|H|0|L>]
    pin_group/pin/path [pin_group/pin/path ...]
    [check_val=<value>]
    (cycles=<val>[:<max>]|cycles_sync=<val>[:<max>])
    [inc_time=<timestr>] [timeout=<#cycles>]

```

A non-blocking command. This command runs in the background allowing other ITPP commands to be executed, including other similar commands.

Check that a group of signals asserts within a given cycle range after a trigger event. Events are timed based on the given clock signal provided. The optional "inc_time" parameter specifies how often to sample the clock for determining the timing events. "edge" specifies on which clock edge events are to occur.

The trigger is a single bit signal and the timing of events start when the trigger reaches the specified trigger value (default 1). A timeout error occurs if the number of cycles specified in "timeout" occurs before the trigger is asserted (default 10000).

The response signals can be pin groups, pins, or rtl paths and multiple pins/groups can be specified. All pins are grouped into a single multi-bit value and compared against the "check_val" specified. If no value is specified, all one's is the default.

"cycles" or "cycles_sync" specifies the timing window to expect responses. A single value like "cycles=3" is a single cycle window and is the same as "cycles=3:3". "cycles=x:y" gives an error if a response occurs before "x" cycles and if any of the pins does not respond by "y" cycles. "cycles_sync" specifies that, though the responses can come at any time within the window, all the responses must occur at the same time.

- summary A trigger asserts. A number of cycles later, response signals assert. check that the responses all happen X cycles after the trigger, and none occur before X cycles
- options:

cycles=x	check all responses occur after exactly x cycles
cycles=x:y	check all responses are active at y cycles, none before x
cycles_sync=x:y	check all responses are simultaneous, and occur between x and y cycles

- parameters

trig=	trigger pin/path
clock=	clock pin/path
edge=<rise fall>	sample on rising or falling edge
<pin name>	check pins/pin_group
check_val=	check value (one bit per pin bit) (default 1)
trig_val=	trigger value (0/L or 1/H - default 1)
cycles=<x x:y>	valid response times between x and y cycles
cycles_sync=<x x:y>	valid response times between x and y cycles responses must be synchronous
timeout=	number of cycles to wait for trigger before error default 10000
inc_time=	time increment (default= short inc setting)

- Example:

```

sim: trigger_check: trig=pin1 clock=stf_clk edge=fall cycles=10 mybpins pin3 cabist_subsystem_TbTop.v
al_pin_4 check_val=0b1111111

```

checks that each of the signals in (mybpins + pin3 + val_pin_4) goes from 0 to 1 (check_val is all ones) after 10 STF clock cycles. Sample on fall of stf_clk. Cycle count starts when pin1 goes high.

- Example:

```

sim: trigger_check: trig=pin1 trig_val=L clock=stf_clk cycles=5:8 mybpins check_val=0b111100

```

checks that the upper three bits of mybpins goes from 0 to 1 and the lower two bits goes from 1 to 0 within 5 to 8 cycles of stf_clk after pin1 goes low.

edge_alignment_check

- syntax

```
sim: edge_alignment_check: pin_group=<pin_group> [ window=<timestr> ]
    [ edge=<rise/fall> ] [ inc_time=<timestr> ] [ timeout=<timestr> ]
```

A non-blocking command. This command runs **in** the background allowing other ITPP commands to be executed, including other similar commands.

Check that once one of the signals **in** the pin **group** has a rising (or falling) edge, all others also change within an amount of time specified **by** window. The **final** value should be all ones (rise) or all zeros (fall)

```
- parameters
  pin_group= - pin group containing signals to check
  window=    - timestring specifying window size within which
               all signals should transition (default 0)
  edge=<rise/fall> - check rising edge or falling edge (default rise)
  timeout=    - maximum time to wait for edge (default 1us)
  inc_time=   - time increment (default= short inc setting)

- Example:
(code)
sim: edge_alignment_check: pin_group=mybpins window=100ps edge=rise timeout=1ns
checks that each of the signals in mybpins rise together within 100ps of each other.
```

pulse_check

- syntax

```
sim: pulse_check [name] pin=<pin/sig> [time=<check time>] [timeout=<val>]
    [enable_pin=<pin/sig>] [enable_val=<val>]
    [halt_pin=<pin/sig>] [halt_val=<val>]
    [inc_time=<inc>] [val=<1/H/0/L>]
    [[pulses=<val>] | [pulses_min=<val>] [pulses_max=<val>]]
    [[width=<time val>] | [width_min=<time val>] [width_max=<time val>]]
    [partial_count] [partial_check]
```

A non-blocking command. This command runs in the background allowing other ITPP commands to be executed, including other similar commands.

Counts the number of pulses in a given time period and prints to the log file. Optionally checks the number and/or width of pulses against the pulses* and width* parameters. An error is given if the number of pulses or pulse width is out of the expected range.

A pulse found at the very beginning of the time window or a pulse that does not complete by the end of the time window is a partial pulse. Since we can't know the full width of these pulses, and they may overlap the time window, which may or may not be allowed for a particular instance, these are treated specially.

By default partial pulses are not counted or checked. Specifying "partial_count" will count the partial pulses, and check the maximum width (if width checking is enabled), but will not check the minimum width. Specifying "partial_check" will count and check widths (min and max) of partial pulses, as if they were full pulses beginning or ending at the time window edge. Specifying both "partial_count" and "partial_check" is the same as just specifying "partial_check".

The check interval can be controlled with an explicit duration, or by enable/halt signals. An explicit duration is specified using the "time" parameter. An Enable signal and value can be specified with "enable_pin" and "enable_val". The default enable value is 1.

A halt signal is specified by "halt_pin" and "halt_val". The default halt value is 0. The default values allow both enable and halt pins to be the same signal and specifies a time window by the width of the assertion on the signal. At least one of "time" or "halt_pin" must be specified.

A timeout can be specified to stop checking and error out if signals are unresponsive (mainly enable and halt signals).

The name is used to identify this particular pulse check internally and in the log file. If specified, the checker name must be the first parameter. If not specified, the pin/path name is used for the checker name.

- Example

```
# check for a running clock
sim: pulse_check: pchk4 pin=clk1 time=520ns pulses_min=1

# check for a stopped clock
sim: pulse_check: pchk5 pin=clk2 time=520ns pulses_max=0

# check for between 9 and 11 active-low pulses
# with widths between 20ns and 40ns each
sim: pulse_check: pchk3 pin=clk2 time=520ns val=0 pulses_min=9 pulses_max=11 width_min=20ns
width_max=40ns
```

- parameters

<name>	an optional identifier used in the log file
pin=	pin name or path
time=<timestr>	time to check/timeout
enable_pin	pin or path to the enable signal
enable_val	enable signal assert value (default 1)
halt_pin	path to the halt signal
halt_val	halt signal assert value (default 0)
inc_time	time increment per check (default short inc time)
timeout	timeout for unresponsive signals
val=	value to check (1 H= high pulse, 0 L = low pulse)
pulses=x	specify exact number of pulses to expect
pulses_max	maximum number of pulses to expect
pulses_min	minimum number of pulses to expect
width=<timestr>	expected width of each pulse (exact)
width_max=<timestr>	maximum expected width of each pulse
width_min=<timestr>	minimum expected width of each pulse
partial_count	count partial pulses
partial_check	check minimum width of partial pulses

define_clock

- syntax

```
sim: define_clock: <clock name> [path=<pin/path>] [period=<time>] [high=<time>]
[low=<time>] [jitter=<time>] [jitter_rate=<chance of jitter>]
[randx=<0/1>] [x_low=<time>] [x_high=<time>] [x_min=<time>]
[sync=<rise/fall>] [init=<clkinit>]
```

Defines the given pin to be a clock that the ITPP Reader will drive or check with the given parameters. If the clock pin is not defined in the pin map, the clock name and path will be used to define an entry. An error will be given if the pin exists, but the path doesn't match. The path can be another pin, in which case the path is copied from that pin definition.

- parameters

clock name	pin name of the clock
path	pin or path, if not already defined in the pin map
period	clock period
high	clock high time
low	clock low time
jitter	maximum a clock edge is allowed to move from the base time
jitter_rate	real number 0 to 1 indicating the chance of a jitter event happening (default 0)
x_low	width of X pulse from low to high. (default 0)
x_high	width of X pulse from high to low. (default 0)
x_min	minimum width of a random x pulse. (default 0)
randx	if 1, randomize the width of x pulses. (default 0) x_low and x_high represent the maximum width. See start_clk_x command.

clear	clear a prior definition (for reuse). if a new definition is used without clearing, prior parameters (e.g. high time) will hold their values from the previous definition. Path is not cleared, only clock stats.
sync	rise fall specifies whether the delay command syncs to the rising or falling edge of the specified pin.
init=< clkinit>	indicates the initialization state for the clock. valid states are high, low, and opp. "high/low" indicates to drive the first edge to 1 or 0. "opp" indicates to drive the first edge opposite the current value. if the value is 'x' the initial state will drive low to start.

time parameters are strings indicating a verilog time. no spaces allowed. e.g. "10000", "10ns", "500ps", "20ms"

Options for clock timing

- at least one of low, high, or period must be set.
- if all are set, period must equal high + low or an error occurs.
- incomplete settings are filled as follows:

given 2 of the three	period = low + high; low = period - high; high = period - low;
or given period	high = period/2; low = period - high;
or given just low or high	high = low; period = high + low;

- jitter

The jitter options specify to add random jitter to any clock edge with the given rate and max jitter.

```
base:  -----|-----|-----|-----|-----|
clk:   -----|-----|-----|-----|-----|
jitter:  >--<      >--<      >--<      >--<      >--<
```

- Example

```
sim: pin_map r_clk mypath/r_clk;
sim: pin_map clk_sync mypath2/trigger_clk;
sim: define_clock: r_clk period=100ns jitter=10ps jitter_rate=0.10 sync=rise;
sim: poll_signal clk_sync 1 1ps;
start_clk: r_clk;
delay r_clk(10);
stop_clk: r_clk;
```

- X clock generation

The start_clk_x command generates x's on the transitions.

```
x-clock with jitter:
base:  -----|-----|-----|-----|-----|
clk:   -----xx|-----xxx|---x|-----xx|-----xx|---
```

see also: start_clk, start_clk_x, check_clock, stop_clk

check_clock

- syntax

```
sim: check_clock: <clock name> [path=<pin/path>] [period=<time>]
[high=<high time>] [low=<low time>] [jitter=<max jitter time>]
[cycles=<cycles to check>] [stop] [timeout=<val>]
[enable_pin=<pin/sig>] [enable_val=<val>]
[halt_pin=<pin/sig>] [halt_val=<val>] [inc_time=<inc>]
```

A non-blocking command. This command runs in the background allowing other ITPP commands to be executed, including other similar commands.

Defines a pin with the given clock name to be a clock to check. Starts a process that will check the clock's rate with the given parameters. If the clock pin is not defined in the pin map, the clock name and path will be used to define an entry. An error will be given if the pin exists, but the path doesn't match. The path can be another pin, in which case the path is copied from that pin definition. You can use the same clock name as for define_clock /start_clk, but it is not recommended, as the control parameters can conflict. Rather you should rename the checking version and have separate entries for running and checking clocks.

Setting period, high, and/or low indicates which of these clock parameters should be checked.

The check interval can be controlled with an explicit duration, or by enable/halt signals. An explicit duration is specified using the “cycles” parameter. Cycles specifies the number of clock cycles to check, then stop checking.

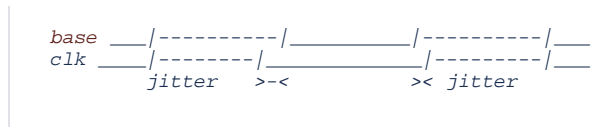
An Enable signal and value can be specified with “enable_pin” and “enable_val”. The default enable value is 1.

A halt signal is specified by “halt_pin” and “halt_val”. The default halt value is 0. The defaults values allow both enable and halt pins to be the same signal and specifies a time window by the width of the assertion on the signal.

Checking begins on the rising edge of the clock signal. If a clock edge is not detected within two expected clock periods, an error is given. wait_for_start disables this error, and checking will then wait until the rising clock edge or the timeout period elapses.

Checking continues until then cycle limit (if specified) is reached, a ‘stop’ command is received, the halt signal is asserted, or timeout is reached. Timeout is enabled with any value greater than zero, and is primarily for unresponsive enable and/or halt signals. Checking is done in a forked process so processing ittp commands may continue.

jitter is max allowed jitter of any clock edge from the base value.



Checking begins at the rise of the clock. Each cycle's high and low time are checked (if set) and the period, each check allowing for the jitter set on each edge. The number of cycles is checked upon completion of the end condition. This expected number of cycles is computed based on the start and end times and jitter or frequency tolerance are accounted for.

The cycle computation is as follows:

```
exp_cyc_min = $floor( (end_time - start_time - maxjit) / period );
exp_cyc_max = $ceil( (end_time - start_time + maxjit) / period );
exp_cyc_min = exp_cyc_min - max_cycle_err;
exp_cyc_max = exp_cyc_max + max_cycle_err;
```

The actual number of cycles should fall within the range or an error is generated. These cycle Checks are also performed every 100 cycles during the run for longer running clock checks.

- Example

```
sim: pin_map r_clk mypath/r_clk;
sim: check_clock: r_clk period=100ns jitter=10ps jitter_rate=0.10 cycles=100;
delay r_clk 100;
sim: check_clock: r_clk period=100ns jitter=10ps jitter_rate=0.10;
delay r_clk 100;
sim: check_clock: r_clk stop;
```

time parameters are strings indicating a verilog time. no spaces allowed. e.g. “10000”, “10ns”, “500ps”, “20ms”

- parameters

<clockname>	pin name of the clock
path=<rtl_path>	pin/path, if not already defined in the pin map
period=<timestr>	clock period
high=<timestr>	clock high time
low=<timestr>	clock low time
jitter=<timestr>	the allowed jitter for a clock edge
cycles	number of cycles to check
stop	stop checking this clock

clear	clear a prior definition (for reuse). if a new definition is used without clearing, prior parameters (e.g. high time) will hold their values from the previous definition. Path is not cleared, only clock stats.
enable_pin	pin or path to the enable signal
enable_val	enable signal assert value (default 1)
halt_pin	path to the halt signal
halt_val	halt signal assert value (default 0)
inc_time	time increment per check (default short inc time)
timeout	timeout for unresponsive signals
wait_for_start	start checking only when clock activates.(def=no). This disables the two period error waiting for the first rising edge of the clock. A timeout error will still occur if timeout is exceeded before the first rising edge.
max_cycle_err	maximum cycle count can vary from computed value based on expected period and elapsed time (def=1)
freq	expected frequency. Enables a post-run check of the frequency based on the total elapsed time and the number of cycles counted.
freq_tol	allowed frequency tolerance for computed value based on number of cycles and elapsed time
enable_cycle_check	set to 1 (default) this enables a final cycle count check based on total elapsed time and the specified period. if 0, period checks are only made on individual clock cycles, and cycle number checks are disabled. if "freq" is specified this is disabled unless explicitly set.

start_clk

```
start_clk: <clkname>
```

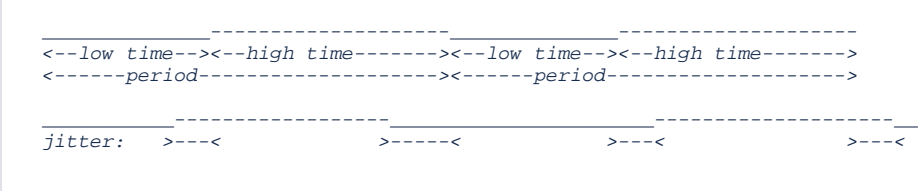
- Arguments

clkname	the clock name to start
---------	-------------------------

A non-blocking command. This command runs in the background allowing other ITPP commands to be executed, including other similar commands.

Start generating a given clock. The clock must be defined with the define_clock command in order to tell what the clock rate should be. If the name is not a valid defined clock, a warning is given. We don't fail in order to allow start_clk lines in an ITPP file meant for the tester, but driven by the simulation model elsewhere.

By default, the clock starts assuming the pin is low and just starting the low phase. This behavior can be changed by use of the 'init' parameter in 'define_clock'. valid settings are high, low, and opp. See 'define_clock' for more details.



start_clk_x

- syntax

```
sim: start_clk_x: <clkname>
```

- Arguments

clkname	the clock name to start
---------	-------------------------

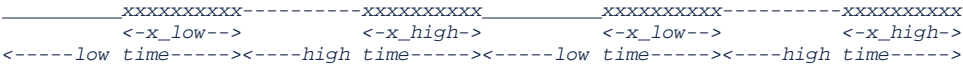
A non-blocking command. This command runs in the background allowing other ITPP commands to be executed, including other similar commands.

Start generating a given clock, generating x's between transitions. The clock must be defined with the define_clock command in order to tell what the clock rate should be. If the name is not a valid defined clock, a warning is given. The width of the X pulses are controlled by the randx, x_low and x_high parameters of the define_clock command. Jitter is allowed also. see start_clk, define_clk

By default, the clock starts assuming the pin is low and just starting the low phase. This behavior can be changed by use of the 'init' parameter in 'define_clock'. valid settings are high, low, and opp. See 'define_clock' for more details.

- Example

```
sim: define_clock: clk2 period=50ns high=20ns jitter=0 jitter_rate=0
sim: define_clock: clk_x path=cabist_subsystem_TbTop.val_pin_4 period=50ns high=20ns x_high=5ns x_low=
5ns randx=1 jitter=5ns jitter_rate=1.00
start_clk: clk2;
sim: start_clk_x: clk_x;
delay: clk2(10);
stop_clk: clk2 0
stop_clk: clk_x 0
```



stop_clk

- syntax

```
stop_clk: <clkname> [<stpval>] [WAIT|NOWAIT]
```

- Arguments

clkname	the clock name to stop
stpval	(optional) value to stop at
WAIT NOWAIT	(optional) wait for completion condition

Stop generating a clock. If a stop value is provided, stop the clock when it reaches that value (0 or 1). This only works for clocks the ITPP Reader is generating. It does not stop anything the model or other validation code is doing. If the name is not a valid defined clock, a warning is given. We don't fail in order to allow stop_clk lines in an ITPP file meant for the tester, but driven by the simulation model elsewhere.

If no value is specified, the clock will stop generating at the next phase transition.

if NOWAIT is specified, the command will exit immediately and continue to the next ITPP command. if WAIT is specified or no wait option is given, then the command waits until the run clock process completes and generates an error if it fails to complete within the specified clock period. if NOWAIT is specified, no waiting or checking will occur and the command exits immediately. This will allow multiple clocks to be stopped together, or other commands to execute before the final clock phase.

align_sync

- syntax

```
sim: align_sync: <pin/path>;
```

- Arguments

pin	pin or rtl pah to align to
-----	----------------------------

Wait for the rising edge of the provided signal. On the tester, align_sync will move the pointer to start the next command to the alignment pulse of the sync signal. Any following commands will start at the alignment point. The primary use is to align a vector action at a specific point in time. Waiting for the rising edge is the nearest thing we can do in the simulator.

ITPP loops

Loops in the ITPP code are allowed with the start_loop/stop_loop and start_match_loop/stop_match_loop command pairs. Loops execute until the specified count (loop iterations) , clock cycle count (on a given clock), or match condition is met (the latter two for match loops only).

Loops may be nested so long as each has a unique name.

Match loops execute until a specified match condition is found, or the timeout condition(s) is met. These are implemented with `match_tap_vector` and `match_mode_vector` commands, plus simulation only `match_compare_bit` and `match_compare_value` commands. These match commands modify the next tap or vector instruction to detect the match condition. When a match is found, the loop terminates. An error occurs if the number of loops executed or the number of clock cycles matches or exceeds the timeout specified before a match is found.

Match compare instructions can execute any time within the loop, but there can only be one active match operation at a time. So, if nested match loops are desired, the outer loop's compare must happen after the inner loop is completed.

Loops Example

```
vector: mybpins(00000);
start_loop: loopA 2
vector: mybpins(00001);
delay: stf_clk(1);

start_match_loop: loopB count=3
vector: mybpins(00011);
delay: stf_clk(1);

start_match_loop: loopC clock=stf_clk cycles=40 count=4
scani: 00110100 ;
to_state: Run-Test/Idle ;
match_tap_vector: 8; # match bit -----v
scand: 00000000000000000000000000000000, LLLLLLHLLLLLLHLLLLLLHLLLLLL ;
## Only the match bit is checked and the result is returned for loop end check
## no pass/fail occurs
to_state: Run-Test/Idle ;
delay: stf_clk(1);
stop_match_loop: loopC

start_match_loop: loopCB clock=stf_clk cycles=40 count=4
scani: 00110100 ;
to_state: Run-Test/Idle ;
sim: match_compare_bit: 8 1; # match bit -----v
scand: 00000000000000000000000000000000, LLLLLLHLLLLLLHLLLLLLHLLLLLL ;
## The non-match bits will compare pass/fail each time
## the match bit will return the result for loop end check
to_state: Run-Test/Idle ;
delay: stf_clk(1);
stop_match_loop: loopCB

start_match_loop: loopD count=6
delay: stf_clk(4);
vector: counter_g(LLLL_HXXX);
match_mode_vector: counter_g(XXXX_XHHL);
# match_mode_vector checks counter_g and compares the lower three bits.
# the result is used to determine if loopD completes or continues
stop_match_loop: loopD

start_match_loop: loopDB count=6
delay: stf_clk(4);
match_compare_value: XXXX_XHHL;
vector: counter_g(LLLL_HXXX);
# match_compare_value checks counter_g and compares the lower three bits.
# the result is used to determine if loopD completes or continues
# the other bits (7:3) are compared pass/fail each time through the loop
stop_match_loop: loopDB

scani: 00110101 ;
to_state: Run-Test/Idle ;
sim: match_compare_value: XXXXXXXXXXXXXXXHLLLLLLHLLLLLL ;
scand: 00000000000000000000000000000000, LLLLLLHLLLLLLHLLLLLLHLLLLLL ;
## Here, the upper bits of the scand compare corresponding to the bits marked X
## in match value, will compare pass/fail each time through the loop.
## The lower bits of the scand compare corresponding to the bits marked H,L
## in match value will not be checked for pass/fail. They are marked X in the
## pass/fail compare.
## The lower bits of the match value, marked H,L in match value, will be used
## to compare for loop end. If they match, the loop terminates.
## If not, then the loop continues unless the timeout is reached. (In which
## case an error is given)

stop_match_loop: loopB

vector: mybpins(00001);
delay: stf_clk(1);
stop_loop: loopA
vector: mybpins(00000);
delay: stf_clk(1);
```

Note that the compare bits in the scand instructions above are not used. Those bits are compared only from the match value (or bit) specified. Also note that there are multiple match compare operations above, but never simultaneously active. The inner loop completes before the outer loop's compare starts. If the outer loop's compare happened before the inner loop, or within the inner loop, then the result will be lost and an error would occur.

start_loop

- syntax

```
start_loop: <name> <count>;
```

- Arguments

name	name given to the loop
count	number of times to execute the loop

start_loop/stop_loop indicate to loop through the intervening ITPP instructions <count> number of times. Loops can be nested but must have unique names within the nesting. Loop names can be re-used after completion. Count must be an integer > 0. The loop re-executes instructions starting at the line following the start_loop command. A start_loop or start_match_loop command within an already existing loop of the same name (i.e. before a matching stop_loop) will generate an error.

stop_loop

- syntax

```
stop_loop: <name>
```

- Arguments

name	name of the loop
------	------------------

stop_loop checks the number of times the loop of the same name has executed. If the number matches the count in the matching start_loop command, the loop is completed and execution falls through to the next ITPP line. If the loop has not completed <count> loops, then execution continues at the line following the start_loop command with the matching name.

You can nest loops. A stop_loop without a matching start_loop will generate an error.

An error is issued if no time has elapsed between the start_match_loop and the stop_match_loop commands. This ensures no hung simulations if there are no timed commands within the loop.

start_match_loop

- syntax

```
start_match_loop: <name> [internal] ( clock=<clock> cycles=<cycles> | <count> ) [mismatch_mode=<true/false>;
```

- Arguments

name	name of the loop
internal	specifies match signal is an internal signal.
clock	pin/pin group/rtl path of the clock
cycles	cycles of clock until timeout
count	number of loops until timeout
mismatch_mode	if true, set loop to mismatch mode

Similar to start_loop/stop_loop, a start_match_loop/stop_match_loop iterates through a loop of code a number of times. The match loop, however, looks for a compare result from either a TAP scand command or a vector command. The compare command for the match result is selected using the match_tap_vector, match_mode_vector, match_compare_bit or match_compare_value commands. The loop continues to execute until a matching result is found or a timeout occurs.

If mismatch mode is set, the loop continues as long as the compare matches and exits when the compare no longer matches.

A timeout occurs if the number of iterations meets the count specified, or the number of cycles of the specified clock elapses before a matching result is found. In either timeout case an error is given.

An error is issued if no time has elapsed between the start_match_loop and the stop_match_loop commands. This ensures no hung simulations if there are no timed commands within the loop.

Either the loop count limit or a clock and cycles parameters are required. Both can also be specified.

Nested loops are allowed. Match loops can be nested within standard loops and vice-versa, so long as all the names are unique. An error is given if a start_loop or start_match_loop command is given if there is an active loop with the same name.

Match loops can be nested in other match loops, provided the match condition (see match_compare_bit and match_compare_value) is set after completion of all inner loops, and not before since only one match condition is valid at a time. An error is given if more than one match compare is active.

stop_match_loop

- syntax

```
<match_compare command: bit or value>;  
<compare operation: either scand or vector>;  
...  
stop_match_loop: <name>;
```

- Arguments

name	name of the loop
------	------------------

stop_match_loop checks the result of the compare operation indicated with the match_tap_vector, match_mode_vector, match_compare_bit or match_compare_value commands. If the compare matched the expected result, the loop terminates and flow continues to the next ITPP line. If the match failed, execution continues with the line following the matching start_match_loop command.

Or if the loop is in mismatch mode (see start_match_loop) the loop terminates on a mismatch and continues on a match.

If no match is found and the number of loop iterations meets or exceeds the loop count limit or the number of cycles on the specified clock meets or exceeds the cycle count limit set in the start_match_loop command, then an error is given.

You can nest loops. A stop_match_loop without a matching start_match_loop will generate an error. An error is also given if a stop_match_loop executes without a valid match condition being specified (matching or not matching).

An error is issued if no time has elapsed between the start_match_loop and the stop_match_loop commands. This ensures no hung simulations if there are no timed commands within the loop.

match_tap_vector

- syntax

```
match_tap_vector: <bit number>;  
scand: <scan in> <scan out>;  
...  
stop_match_loop <name>;
```

match_tap_vector specifies that the next scand instruction should not be a standard compare, but should instead be used as a match condition for a match loop. The standard pass/fail compare is not done, so no failure can occur. Instead, the TDO bit specified as the match bit will be compared and the result used in the next stop_match_loop command to determine whether to continue the loop or fall through to the next ITPP line. An error occurs if the TDO match bit is not an H or L.

Only one match compare (whether match_tap_vector, match_mode_vector, match_compare_bit or match_compare_value) is valid at a time. An error is given if a match compare is executed with another match compare active (i.e. before the next stop_match_loop command).

match_mode_vector

- syntax

```
[sim:] match_mode_vector: <signal>(<value>);  
...  
stop_match_loop <name>;
```

- Arguments

signal	pin name, pin group name or rtl path
value	H,L,X compare value bit string

The `match_mode_vector` command reads the indicated signal and compares it to the given value. No failures result from the compare. Instead, the result is used in the next `stop_match_loop` command to determine whether to continue the loop or fall through to the next ITPP line.

The signal can be a pin in the pin map, a pin group or an RTL path. The compare value can contain H,L,X as with TAP and vector compare operations. Simulation time and timing of the signal sampling are exactly as a normal vector operation (see pin group clocking). Pins, direct rtl paths, and un-clocked pin groups, which do not typically advance simulation time, will advance one clock cycle of the match loop's designated clock after reading the value.

This command has both a standard and sim variant. For simulation an internal signal may be specified to match in order to speed up delay loops. This condition will be specified by an "internal" option on the `start_match_loop` command, and a "sim:" designation on the `match_mode_vector` command. The Tester will ignore the sim command and only count the delay loop. The simulation will treat both versions the same.

Only one match compare (whether `match_tap_vector`, `match_mode_vector`, `match_compare_bit` or `match_compare_value`) is valid at a time. An error is given if a match compare is executed with another match compare active (i.e. before the next `stop_match_loop` command).

match_compare_bit

- syntax

```
sim: match_compare_bit: <bit number> <value>;
<compare operation: either scand or vector>;
...
stop_match_loop <name>;
```

- Arguments

bit number	the bit number of the compare bit
value	the value the bit should have to terminate the loop

`match_compare_bit` is a non-standard simulation only command for enhancing match loop behavior in simulation. `match_compare_bit` specifies that the next compare instruction - either TAP scand or a vector compare operation - should not be a standard compare, but should instead be used as a match condition for a match loop. Two compares will be done. The standard pass/fail compare and the match bit compare. The match bit number specified will be marked as don't care (i.e. "X") in the standard compare operation, so that bit will not cause any failure in the simulation. The remaining bits will still behave as normal and if mismatches arise, will cause an error as usual. The specified match bit will be compared against the value specified and the result used in the next `stop_match_loop` command to determine whether to continue the loop or fall through to the next ITPP line.

Only one match compare (whether `match_compare_bit` or `match_compare_value`) is valid at a time. An error is given if a match compare is executed with another match compare active (i.e. before the next `stop_match_loop` command).

match_compare_value

- syntax

```
sim: match_compare_value: <compare value>;
<compare operation: either scand or vector>;
...
stop_match_loop <name>;
```

`match_compare_value` is a non-standard simulation only command for enhancing match loop behavior in simulation. As with `match_compare_bit`, the `match_compare_value` command modifies the next compare instruction - either a TAP scand or a vector compare. But a full width compare value is allowed. The width of the compare value should match the width of the expected data in the compare or an error is given. The compare value can contain H,L,X as with TAP and vector compare operations. Any compare bits (i.e. not 'X') will be marked for don't care in the standard compare (as with `match_compare_bit`) so no failures can result from the match compare checked bits. Where the match compare value has 'X', the regular compare bits are valid and may cause failure on a mismatch. The checked match bits will be compared against the values specified and the result used in the next `stop_match_loop` command to determine whether to continue the loop or fall through to the next ITPP line.

Only one match compare (whether `match_compare_bit` or `match_compare_value`) is valid at a time. An error is given if a match compare is executed with another match compare active (i.e. before the next `stop_match_loop` command).

STF access without the BFM

Two examples of driving the STF network without using the STF BFM are shown below. One using the Vector Clocking method and one using the Pin Group Clocking method.

Vector Clocking for STF

This is an example of how to drive and check the STF interface when you don't have the BFM installed. It uses the Vector Clocking method described above.

```
#=====
# STF ITPP Test to check the flow
#=====
# file: mphy_stf_itpp_test.itpp

# This test is an example to demonstrate the methods of driving STF
# without using the BFM

#####
# Define the STF input and output pins
# 32 bit size limit, so break it up into two vectors
sim: pin_map stf_input_vec1 top.i_dfx_top.pstf_ring_pkt[41:32] ;
sim: pin_map stf_input_vec2 top.i_dfx_top.pstf_ring_pkt[31:0] ;
sim: pin_map stf_output_vec1 top.i_dfx_top.nstf_ring_pkt[41:32] ;
sim: pin_map stf_output_vec2 top.i_dfx_top.nstf_ring_pkt[31:0] ;

# STF reset pin
sim: pin_map stf_resetb top.i_dfx_top.pstf_ring_reset_b ;

# define the stf_clk pin
sim: pin_map stf_clk top.i_dfx_top.pstf_ring_clk ;

# Define the STF Clock
# this defines the 'stf_clk' pin as a clock with the given rate.
# It can then be used with start_clk and stop_clk commands to generate the clock.
sim: define_clock: stf_clk period=50ns high=25ns

#####
# STF_BFM behavior:
# called on falling edge of stf_clk
# drive data shortly after falling edge (501ps)
# sample data shortly after rising edge (3ns)
# return at falling edge
#####

#####
# Define the pin groups for the STF vector statements

pin_group: stf_in stf_input_vec1 stf_input_vec2 ;
pin_group: stf_out stf_output_vec1 stf_output_vec2 ;

#####
# To emulate the STF BFM behavior:
# Set the vector clock to be the STF Clock, falling edge
# set stf_in to use falling edge, plus a small delay (501ps)
# set stf_out to use rising edge, plus a small delay (3ns)

sim: vector_clock: stf_clk fall enable=1;
pin_group_clock: stf_in edge=fall edge_delay=501ps;
pin_group_clock: stf_out edge=rise edge_delay=3ns;

# Start generating the STF Clock
start_clk: stf_clk;

# End of Setup. begin the test vectors
```

Pin Group Clocking for STF

This is an example of how to drive and check the STF interface even if you don't have the BFM installed. It was created for the Gen4phy team and uses the Pin Groups and clocks as described above.

```

#=====
# STF ITPP Test to check the flow
#=====
# file: mphy_stf_ittp_test.itpp

# This test is an example to demonstrate the methods of driving STF
# without using the BFM

# Define the STF input and output pins
# 32 bit size limit, so break it up into two vectors
sim: pin_map stf_input_vec1 top.i_dfx_top.pstf_ring_pkt[41:32] ;
sim: pin_map stf_input_vec2 top.i_dfx_top.pstf_ring_pkt[31:0] ;
sim: pin_map stf_output_vec1 top.i_dfx_top.nstf_ring_pkt[41:32] ;
sim: pin_map stf_output_vec2 top.i_dfx_top.nstf_ring_pkt[31:0] ;

# STF reset pin
sim: pin_map stf_resetb top.i_dfx_top.pstf_ring_reset_b ;

# define the stf_clk pin
sim: pin_map stf_clk top.i_dfx_top.pstf_ring_clk ;

# Define the STF Clock
# this defines the 'stf_clk' pin as a clock with the given rate.
# It can then be used with start_clk and stop_clk commands to generate the clock.
#*****
# sim: define_clock: <clock name> [path=<rtl path>] [period=<time>] [high=<high time>] [low=<low
time>] [jitter=<max jitter time>] [jitter_rate=<% chance>]
#*****
sim: define_clock: stf_clk period=50ns high=20ns

# Define the pin groups for the STF vector statements
# include the clock in the pin group for input
# that way vector statements wait for the rise of stf_clk before
# driving the input pins or checking the output pins
pin_group: stf_out stf_output_vec1 stf_output_vec2 ;
pin_group: stf_in clock=stf_clk stf_input_vec1 stf_input_vec2 ;

# Start generating the STF Clock
start_clk: stf_clk;

# End of Setup. begin the test vectors

```

STF Test Vectors

For both Vector Clocking and Pin Group Clocking, Vector statements for STF packets will look identical to previous ones for the STF_BFM. The difference is the SIM pipe will handle the data rather than the STF pipe and STF BFM, and the clocking is handled by the ITPP Reader instead of the STF BFM. This format is compatible with the STF BFM, so you could add the BFM in anytime without affecting cases. Or, conversely, you can use tests meant for the STF BFM. But you do need to drive the reset if the BFM model isn't handling it for you. This would be no different from TVPV cases, but could be a difference for simulation, depending on the DFT testing environment used.


```
# Pin Map
#
`include ${MODEL_ROOT}/verif/my/stuff/my_extra_pin_maps.txt
stf_input_vec1 top.i_dfx_top.pstf_ring_pkt[41:32]
stf_input_vec2 top.i_dfx_top.pstf_ring_pkt[31:0]
stf_output_vec1 top.i_dfx_top.nstf_ring_pkt[41:32]
stf_output_vec2 top.i_dfx_top.nstf_ring_pkt[31:0]
stf_resetb top.i_dfx_top.pstf_ring_reset_b
stf_clk top.i_dfx_top.pstf_ring_clk
```

Sample pin group file

```
# Pin Groups
#
stf_out stf_output_vec1 stf_output_vec2
stf_in clock=stf_clk stf_input_vec1 stf_input_vec2
`include ${MODEL_ROOT}/verif/my/stuff/my_extra_pingroups.txt
```

The Pin Groups file can also contain the following additional setup and debug commands. Command syntax for each is the same as if in the ITPP file, except without the "sim:" command prefix and colons are required to distinguish commands from pin groups. pin_group_clock: timing_cfg: vector_clock: set_cmd_prefix: parser_debug_mode:

Macro Map file

Macros are used to substitute a long rtl path string for a short macro name. Macros in a path name begin with a backtick ("") and end at the next period ("."). so in the name "abcd1234.my_module.sig2", "my_module" would be replaced by the macro definition.

Use the PlusArg +ITPP_MACRO_FILE=<macro map file name> to tell the file containing the macro definitions.

Macro file line Format

```
`define <macro name> <substitute pattern>
```

Sample macro file

```
`define top_pin_if cabist_subsystem_TbTop.i_cabist_subsystem_PinIf
`define top_pins cabist_subsystem_TbTop
`include ${MODEL_ROOT}/verif/my/stuff/my_extra_macros.txt
```

Macros are referenced using '<macro name>' in a signal reference command.

Pin map files and Pin map definitions can use the macros as well. (but not pin groups)

- Example pin map file using macros

```
#
#Pin map file for simcmd_test.itpp
#
# includes macros, needs macro_map_file.txt
#
pin1 `top_pins.val_pin_1
pin2 `top_pins.val_pin_2
pin3 `top_pins.val_pin_3
pin4 `top_pins.val_pin_4
bpin5[4] `top_pins.val_bpin5[4]
bpin5[3] `top_pins.val_bpin5[3]
bpin5[2] `top_pins.val_bpin5[2]
bpin5[1] `top_pins.val_bpin5[1]
bpin5[0] `top_pins.val_bpin5[0]
bpin5 `top_pins.val_bpin5
tap_clk `top_pins.jtag_clk
stf_clk `top_pins.tb_cabist_clk_STF
cab_clk `top_pins.tb_cabist_clk
```

- Example

```
sim: macro_map top_pins cabist_subsystem_TbTop
sim: pin_map my_pin_2 `top_pins.my_val_pin_2
pin_group my_group_a my_pin2 `top_pins.my_val_pin_3 `top_pins.my_val_pin_4
sim: peek_signal my_pin_2 0
sim: peek_signal `top_pins.my_val_pin_4 1
vector my_group_a(101);
delay TCK(0);
vector my_group_a(HLH);
```

Unsupported ITPP commands

Unsupported commands will be parsed and not fail simulation, but will produce no results, other than a message stating that the command is not supported.

start_sync stop_sync

sync pulses are supposed to be alignment pulses generated by the tester, aligned to a clock, and used for aligning commands to a certain time. No generic way has been found to do this in simulation that matches the usage on tester. The Simulator ITPP Reader is limited by the use of arbitrary signals to polling for edges. So you can never get a precise edge to sync to. You're always off by at least a sim tick, and usually more (e.g. 1ps). There are some ideas out there, but finding a solution that matches tester usage/behavior has not been and will not be easy. align_sync has a partial implementation. See align_sync command.

expandata

start_scan stop_scan

start_subroutine stop_subroutine

compress: no_compress

comment

vc2sub

pscand

misr

SVF support

SVF commands can be executed from the ITPP Reader. Initial usage is the same as DTEG SVF Reader version 0.0.7.

For commands info and available configuration, see the SVF Reader documentation.