UVVM Utility Library – Quick Reference

UVVM™ bitvis

Checks and awaits

[v_bool :=] **check_value**(value, [exp], alert_level, msg, […])
[v_bool :=] **check_value_in_range**(value, min_value, max_value, alert_level, msg, […])
check_stable(target, stable_req, alert_level, msg, […])
await_change(target, min_time, max_time, alert_level, msg, […])
await_value(target, exp, min_time, max_time, alert_level, msg, […])
await_stable(target, stable_req, stable_req_from, timeout, timeout_from, alert_level, msg, […])

Logging and verbosity control

set_log_file_name(file_name)
log([msg_id], msg, […])
log_text_block(msg_id, text_block, formatting, […])
enable_log_msg(msg_id, […])
disable_log_msg (msg_id, […]),
is_log_msg_enabled (msg_id, [msg_id_panel])
set_log_destination (log_destination, [quietness])

Alert handling

set_alert_file_name(file_name)
alert(alert_level, msg, scope)
[tb_]note(msg, [scope])
[tb_]warning(msg, [scope])
manual_check(msg, [scope])
[tb_]error(msg, [scope])
[tb_]failure(msg, [scope])
set_alert_stop_limit(alert_level, limit)
v_int := **get_alert_stop_limit**(alert_level)
set_alert_attention(alert_level, attention, [msg])
v_attention := **get_alert_attention**(alert_level)
increment_expected_alerts(alert_level, number)
increment_expected_alerts_and_stop_limit(alert_level, [number, [msg, [scope]]])

Reporting

report_global_ctrl(VOID)
report_msg_id_panel(VOID)
report_alert_counters(VOID)
report_alert_counters(order)
v_natural := **shared_uvvm_status.found_unexpected_simulation_warnings_or_worse**
v_natural := **shared_uvvm_status.found_unexpected_simulation_errors_or_worse**
v_natural := **shared_uvvm_status.mismatch_on_expected_simulation_warnings_or_worse**
v_natural := **shared_uvvm_status.mismatch_on_expected_simulation_errors_or_worse**

Randomization

v_slv := **random**(length)
v_sl := **random**(VOID)
{v_int, v_real, v_time} := **random**(min_value, max_value)
random([min_value, [max_val]], v_seed1, v_seed2, v_target)
randomize(seed1, seed2)

String handling

v_string := **to_string**(val, […])
v_string := **justify**(val, justified, width, format_spaces, truncate)
v_string := **fill_string**(val, width)
v_string := **to_upper**(val)
v_character := **ascii_to_char**(ascii_pos, [ascii_allow])
v_int := **char_to_ascii**(character)
v_natural := **pos_of_leftmost**(character, string, [result_if_not_found])
v_natural := **pos_of_rightmost**(character, string, [result_if_not_found])
v_string := **remove_initial_chars**(string, number of chars(natural))
v_string := **get_procedure_name_from_instance_name**(string)
v_string := **get_process_name_from_instance_name**(string)
v_string := **get_entity_name_from_instance_name**(string)
v_string := **replace**(string, target_character, exchange_character)
replace(inout line, target_character, exchange_character)
v_string := **pad_string**(val, char, width, [side])

Signal generators

clock_generator(clock, [clock_count], clock_period, [clock_high_percentage] / [clock_high_time])
clock_generator(clock, clock_ena, [clock_count], clock_period, clock_name, [clock_high_...])
adjustable_clock_generator(clock_signal, clock_ena, clock_period, clock_high_percentage)
adjustable_clock_generator(clock_signal, clock_ena, clock_period, clock_name, clock_high_percentage)
adjustable_clock_generator(clock_signal, clock_ena, clock_count, clock_period, clock_name, clock_high_percentage)
gen_pulse (target, [pulse_value] pulse_duration, [blocking_mode], msg) *or* (target, [pulse_value], clock_signal, num_periods, msg)

Synchronisation

block_flag(flag_name, msg, [already_blocked_severity, [scope]])
unblock_flag(flag_name, msg, trigger, [scope])
await_unblock_flag(flag_name, timeout, msg, [flag_returning, [timeout_severity, [scope]]]
await_barrier(barrier_signal, timeout, msg, [timeout_severity, [scope]]

BFM Common Package

normalize_and_check(value, target, mode, value_name, target_name, msg)
wait_until_given_time_after_rising_edge(clk, wait_time)
wait_until_given_time_before_rising_edge(clk, time_to_edge, clk_period)
wait_num_rising_edge(clk, num_rising_edge)
wait_num_rising_edge_plus_margin(clk, num_rising_edge, margin)
wait_on_bfm_sync_start(clk, bfm_sync, setup_time, config_clock_period, time_of_falling_edge, time_of_rising_edge)
wait_on_bfm_exit(clk, bfm_sync, hold_time, time_of_falling_edge, time_of_rising_edge)
check_clock_period_margin(clock, bfm_sync, time_of_falling_edge, time_of_rising_edge, config_clock_period, config_clock_period_margin, config_clock_margin_severity)

Watchdog

watchdog_timer(watchdog_ctrl, timeout, [alert_level, [msg]])
extend_watchdog(watchdog_ctrl, [time_extend])
reinitialize_watchdog(watchdog_ctrl, timeout)

VHDL 2008 only

1 Method descriptions

Note 1: Arguments common for most methods (green text) are described in chapter 1.12.
Note 2: All methods are defined in uvvm_util.methods_pkg, unless otherwise noted.

Legend: bool=boolean, sl=std_logic, slv=std_logic_vector, u=unsigned, s=signed, int=integer
**IEEE=Method is native for VHDL2008 (Method is listed here for completeness.)*

1.1 Checks and awaits

Name	Parameters and examples	Description
[v_bool :=] check_value()	value(bool), [exp(bool)], alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(sl), exp(sl), [match_strictness], alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(slv), exp(slv), [match_strictness], alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(t_slv_array), exp(t_slv_array), [match_strictness], alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(u), exp(u), alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(t_unsigned_array), exp(t_unsigned_array), [match_strictness], alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(s), exp(s), alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(t_signed_array), exp(t_signed_array), [match_strictness], alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] value(int), exp(int), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(real), exp(real), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(time), exp(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] **Examples** check_value(v_int_a, 42, WARNING, "Checking the integer"); v_check := check_value(v_slv5_a, "11100", MATCH_EXACT, ERROR, "Checking the SLV", "My Scope", HEX, SKIP_LEADING_0, ID_SEQUENCER, shared_msg_id_panel);	Checks if *val* equals *exp*, and alerts with severity *alert_level* if the values do not match. The result of the check is returned as a boolean if the method is called as a function. If *val* is of type *slv*, *unsigned* or *signed*, there are additional optional arguments: - *match_strictness:* Specifies if match needs to be exact or std_match, e.g. 'H' = '1'. (MATCH_EXACT, MATCH_STD) - *radix*: For the vector representation in the log: BIN, HEX, DEC or HEX_BIN_IF_INVALID. (HEX_BIN_IF_INVALID means hexadecimal, unless there are the vector contains any U, X, Z or W, - in which case it is also logged in binary radix.) - *format:* KEEP_LEADING_0 or SKIP_LEADING_0. Controls how the vector is formatted in the log. **Defaults**: *scope*<=C_TB_SCOPE_DEFAULT, *match_strictness*<=MATCH_STD, *radix*<=HEX_BIN_IF_INVALID, *format*<=SKIP_LEADING_0, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel
[v_bool :=] check_value_in_range()	value(u), min_value(u), max_value(u), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(s), min_value(s), max_value(s), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(int), min_value(int), max_value(int), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(time), min_value(time), max_value(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] value(real), min_value(real), max_value(real), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] **Example** check_value_in_range(v_int_a, 10, 100, ERROR, "Checking that integer is in range");	Checks if *min_value* ≤ *val* ≤ *max_value*, and alerts with severity *alert_level* if *val* is outside the range. The result of the check is returned as a boolean if the method is called as a function. **Defaults**: *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel
check_stable()	target(bool), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(sl), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(slv), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(u), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(s), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(int), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(real), stable_req(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] **Example** check_stable(slv8, 9 ns, ERROR, "Checking if SLV is stable");	Checks if the *target* signal has been stable in *stable_req* time. If not, an alert is asserted. **Defaults**: *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel

await_change()	target(bool), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(sl), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(slv), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(u), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(s), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(int), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] target(real), min_time, max_time, alert_level, msg, [scope, [msg_id, [msg_id_panel]]] **Example** await_change(bol, 3 ns, 5 ns, ERROR, "Awaiting change on bol signal");	Waits until the *target* signal changes, or times out after *max_time*. An alert is asserted if the signal does not change between *min_time* and *max_time*. Note that if the value changes at exactly *max_time*, the timeout gets precedence. **Defaults**: *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel
await_value()	target(bool), exp(bool), min_time, max_time, alert_level, msg, [scope, (etc.)]] target(sl), exp(sl), [match_strictness], min_time, max_time, alert_level, msg, [scope, (etc.)] target(slv), exp(slv), [match_strictness], min_time, max_time, alert_level, msg, [scope, (etc.)] target(u), exp(u), min_time, max_time, alert_level, msg, [scope, (etc.)] target(s), exp(s), min_time, max_time, alert_level, msg, [scope, (etc.)] target(int), exp(int), min_time, max_time, alert_level, msg, [scope, (etc.)] target(real), exp(real), min_time, max_time, alert_level, msg, [scope, (etc.)] **Examples** await_value(bol, true, 10 ns, 20 ns, ERROR, "Waiting for bol to become true"); await_value(slv8, "10101010", MATCH_STD, 3 ns, 7 ns, WARNING, "Waiting for slv8 value");	Waits until the *target* signal equals the *exp* signal, or times out after *max_time*. An alert is asserted if the signal does not equal the expected value between *min_time* and *max_time*. Note that if the value changes to the expected value at exactly *max_time*, the timeout gets precedence. - *match_strictness:* Specifies if match needs to be exact or std_match , e.g. 'H' = '1'. (MATCH_EXACT, MATCH_STD) **Defaults**: *match_strictness*<=MATCH_EXACT, *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel
await_stable()	target(bool), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(sl), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(slv), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(u), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(s), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(int), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] target(real), stable_req(time), stable_req_from(t_from_point_in_time), timeout (time), timeout_from(t_from_point_in_time), alert_level, msg, [scope, (etc.)] **Example** await_stable(u8, 20 ns, FROM_LAST_EVENT, 100 ns, FROM_NOW, ERROR, "Waiting for u8 to stabilize");	Wait until the target signal has been stable for at least 'stable_req'. Report an error if this does not occurr within the time specified by 'timeout'. Note: 'Stable' refers to that the signal has not had an event (i.e. not changed value). Description of special arguments: `stable_req_from :` `- FROM_NOW Target must be stable 'stable_req' from now.` `- FROM_LAST_EVENT Target must be stable 'stable_req' from the last event of target.` `timeout_from :` `- FROM_NOW The timeout argument is given in time from now.` `- FROM_LAST_EVENT The timeout argument is given in time the last event of target.` **Defaults**: *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_POS_ACK, *msg_id_panel*<=shared_msg_id_panel

1.2 Logging and verbosity control

Name	Parameters and examples	Description
set_log_file_name()	[file_name(string)] **Example** set_log_file_name("new_log_file_name.txt");	Sets the log file name. To ensure that the entire log transcript is written to a single file, this should be called prior to any other procedures (except set_alert_file_name()). If file name is set after a log message has been written to the log file, a warning will be reported. This warning can be disabled by setting C_WARNING_ON_LOG_ALERT_FILE_RUNTIME_RENAME false in the adaptations_pkg. **Defaults**: *file_name*<=C_LOG_FILE_NAME
log()	[msg_id], msg, [scope, [msg_id_panel, [log_destination(t_log_destination), [log_file_name(string), [open_mode(file_open_kind)]]]]] **Examples** log(ID_SEQUENCER, "message to log"); log(ID_BFM, "Msg", "MyScope", local_msg_id_panel, LOG_ONLY, "new_log.txt", write_mode);	Writes message to log. If the *msg_id* is enabled in *msg_id_panel*, log the *msg*. Log destination defines where the message will be written to (CONSOLE_AND_LOG, CONSOLE_ONLY, LOG_ONLY). If log destination is not specified, the default value in shared_default_log_destination found in the adaptations_pkg.vhd will be used. Log file name defines the log file that the text block shall be written to. open_mode indicates how the log file shall be opened (write_mode, append_mode). **Defaults**: *msg_id* <= C_TB_MSG_ID_DEFAULT, *scope*<=C_TB_SCOPE_DEFAULT, *msg_id_panel*<=shared_msg_id_panel, *log_destination*<= shared_default_log_destination, *log_file_name*<=C_LOG_FILE_NAME, *open_mode*<=append_mode
log_text_block()	msg_id, text_block(line), formatting(t_log_format), [msg_header(string), [scope, [msg_id_panel, [log_if_block_empty(t_log_if_block_empty), [log_destination(t_log_destination), [log_file_name(string), [open_mode(file_open_kind)]]]]]]] **Examples** log_text_block(ID_SEQUENCER, v_line, UNFORMATTED); log_text_block(ID_BFM, v_line, FORMATTED, "Header", "MyScope");	Writes text block from VHDL line to log. Formatting either FORMATTED or UNFORMATTED. msg_header is an optional header message for the text_block. log_if_block_empty defines how an empty text block is handled (WRITE_HDR_IF_BLOCK_EMPTY/SKIP_LOG_IF_BLOCK_EMPTY/NOTIFY_IF_BLOCK_EMPTY). Log destination defines where the message will be written to (CONSOLE_AND_LOG, CONSOLE_ONLY, LOG_ONLY). Log file name defines the log file that the text block shall be written to. open_mode indicates how the log file shall be opened (write_mode, append_mode). **Defaults**: *msg_header*<="", *scope*<=C_TB_SCOPE_DEFAULT, *msg_id_panel*<=shared_msg_id_panel, log_if_block_empty<=WRITE_HDR_IF_BLOCK_EMPTY, *log_destination*<= shared_default_log_destination, *log_file_name*<=C_LOG_FILE_NAME, *open_mode*<=append_mode
enable_log_msg ()	msg_id, [quietness(t_quietness), [scope]] msg_id, msg, [quietness(t_quietness), [scope]] msg_id, msg_id_panel, [msg, [scope, [quietness(t_quietness)]]] **Example** enable_log_msg(ID_SEQUENCER);	Enables logging for the given *msg_id*. (See ID-list on front page for special purpose IDs). Logging of enable_log_msg() can be turned off by setting quietness=QUIET. **Defaults**: *msg_id_panel*<=shared_msg_id_panel, *msg*<="", *scope*<=C_TB_SCOPE_DEFAULT, *quietness*<=NON_QUIET **Note:** ID_LOG_MSG_CTRL is always reporting but can be muted by setting quietness=QUIET.
disable_log_msg()	msg_id, [quietness(t_quietness), [scope]] msg_id, msg, [quietness(t_quietness), [scope]] msg_id, msg_id_panel, [msg, [scope, [quietness(t_quietness)]]] **Example** disable_log_msg(ID_LOG_HDR);	Disables logging for the given *msg_id*. (See ID-list on front page for special purpose IDs). Logging of disable_log_msg() can be turned off by setting quietness=QUIET. **Defaults**: *msg_id_panel*<=shared_msg_id_panel, *msg*<="", *scope*<=C_TB_SCOPE_DEFAULT, *quietness*<=NON_QUIET **Note:** ID_LOG_MSG_CTRL is always reporting but can be muted by setting quietness=QUIET.
[v_bool :=] is_log_msg_enabled ()	msg_id, [msg_id_panel] **Example** v_is_enabled := is_log_msg_enabled(ID_SEQUENCER);	Returns Boolean 'true' if given message ID is enabled. Otherwise 'false' **Defaults**: *msg_id_panel*<=shared_msg_id_panel
set_log_destination ()	t_log_destination, [quietness(t_quietness)] **Example** set_log_destination(CONSOLE_ONLY);	Sets the default log destination for all log procedures. The destination specified in this log_destination will be used unless the log_destination argument in the log procedure is specified. A log message is written to log ID ID_LOG_MSG_CTRL if quietness is set to NON_QUIET. **Defaults**: *quietness* <= NON_QUIET

1.2.1 General string handling features for log()

- All log messages will be given using the user defined layout in adaptations_pkg.vhd
- \n may be used to force line shifts. Line shift will occur after scope column, before message column
- \r may be used to force line shift at start of log message. The result will be a blank line apart from prefix (message ID, timestamp and scope will be omitted on the first line)

1.3 Alert handling

Name	Parameters and examples	Description
set_alert_file_name()	file_name(string)] **Example** set_alert_file_name("new_alert_log_file.txt");	Sets the alert file name. To ensure that the entire log transcript is written to a single file, this should be called prior to any other procedures (except set_alert_file_name()). If file name is set after a log message has been written to the log file, a warning will be reported. This warning can be disabled by setting C_WARNING_ON_LOG_ALERT_FILE_RUNTIME_RENAME false in the adaptations_pkg. **Defaults**: *file_name*<=C_ALERT_FILE_NAME
alert()	alert_level, msg , [scope] **Example** alert(TB_WARNING, "This is a TB warning");	- Asserts an alert with severity given by *alert_level*. - Increment the counters for the given *alert_level.* - If the stop_limit for the given *alert_level* is reached, stop the simulation. **Defaults**: *scope* <=C_TB_SCOPE_DEFAULT
note() error() tb_note() tb_error() warning() failure() tb_warning() tb_failure() manual_check()	msg, [scope] **Examples** note("This is a note"); tb_failure("This is a TB failure", "tb_scope");	Overloads for alert(). Note that: warning(msg, [scope]) = alert(warning, msg, [scope]). **Defaults**: *scope* <=C_TB_SCOPE_DEFAULT
increment_expected_alerts()	alert_level, [number (natural) , [msg, [scope]]] **Example** increment_expected_alerts(WARNING, 2, "Expecting two more warnings");	Increments the expected alert counter for the given alert_level. **Defaults**: *number*<=1,*msg*<="", *scope* <=C_TB_SCOPE_DEFAULT
set_alert_stop_limit()	alert_level, number (natural) **Example** set_alert_stop_limit(ERROR, 2);	Simulator will stop on hitting <number> of specified alert type (0 means never stop).
increment_expected_alerts_and_stop_limit()	alert_level, [number, [msg, [scope]]] **Example** increment_expected_alerts_and_stop_limit(WARNING, 2, "Expecting two more warnings");	Increments the expected alert counter and stop limit for the given alert_level. **Defaults**: *number*<=1, *msg*<="", *scope* <=C_TB_SCOPE_DEFAULT
v_int := get_alert_stop_limit()	alert_level **Example** v_int := get_alert_stop_limit(FAILURE);	Returns current stop limit for given alert type.
set_alert_attention()	alert_level, attention (t_attention), [msg] **Example** set_alert_attention(NOTE, IGNORE, "Ignoring all note-alerts");	Set given alert type to t_attention: IGNORE or REGARD. **Defaults**: *msg* <=""
v_attention := get_alert_attention()	alert_level **Example** v_attention := get_alert_attention(WARNING)	Returns current attention (IGNORE or REGARD) for given alert type.

1.4 Reporting

Name	Parameters	Description
report_global_ctrl()	VOID	Logs the values in the global_ctrl signal, which is described in chapter 1.13
report_msg_id_panel()	VOID	Logs the values in the msg_id_panel, which is described in chapter 1.13
report_alert_counters()	VOID order (t_order) **Example** report_alert_counters(VOID); report_alert_counters(FINAL); report_alert_counters(INTERMEDIATE);	Logs the status of all alert counters, typically at the end of simulation. For each alert_level, the alert counter is compared with the expected counter. If parameter is FINAL, an additional summary concluding success or failure is logged. - type t_order is (FINAL, INTERMEDIATE) VOID parameter gives same result as FINAL.

Shared variable	Signal type	Description
shared_uvvm_status.found_unexpected_simulation_warnings_or_worse	Natural, read only	Status is '0' on success and '1' on failure. The variable is set when actual > expected for WARNING, ERROR or FAILURE alerts.
shared_uvvm_status.found_unexpected_simulation_errors_or_worse	Natural, read only	Status is '0' on success and '1' on failure. The variable is set when actual > expected for ERROR or FAILURE alerts.
shared_uvvm_status.mismatch_on_expected_simulation_warnings_or_worse	Natural, read only	Status is '0' on success and '1' on failure. The variable is set when there is a mismatch between the expected and the actual WARNING, ERROR or FAILURE alerts.
shared_uvvm_status.mismatch_on_expected_simulation_errors_or_worse	Natural, read only	Status is '0' on success and '1' on failure. The variable is set when there is a mismatch between the expected and the actual ERROR or FAILURE alerts.

1.5 Randomization

Name	Parameters and examples	Description
v_slv := random()	length(int) **Example** v_slv := random(v_slv'length);	Returns a random std_logic_vector of size *length*. The function uses and updates a global seed.
v_sl := random()	VOID **Example** v_sl := random(VOID);	Returns a random std_logic. The function uses and updates a global seed
{v_int,v_real,v_time} := random()	min_value(int), max_value(int) min_value(real), max_value(real) min_value(time), max_value(time) **Example** v_int := random(1, 10);	Returns a random *integer*, *real* or *time* between min_value and max_value. The function uses and updates a global seed
random()	v_seed1(positive *variable*), v_seed2(positive *variable*), v_target(slv *variable*) **Example** random(v_seed1, v_seed2, v_slv)	Sets v_target to a random value. The procedure uses and updates v_seed1 and v_seed2.
random()	min_value(int), max_value(int), v_seed1(positive *var*), v_seed2(positive *var*), v_target(int *var*) min_value(real), max_value(real), v_seed1(positive *var*), v_seed2(positive *var*), v_target(real *var*) min_value(time), max_value(time), v_seed1(positive *var*), v_seed2(positive *var*), v_target(time *var*) **Example** random(0.01, 0.03, v_seed1, v_seed2, v_real);	Sets v_target to a random value between min_value and max_value. The procedure uses and updates v_seed1 and v_seed2.
randomize()	seed1(positive), seed2(positive) , [msg, [scope]] **Example** randomize(12, 14, "Setting global seeds");	Sets the global seeds to *seed1* and *seed2*.

1.6 String handling

(Methods are defined in uvvm_util.string_methods)

Name	Parameters and examples	Description
v_string := to_string() *IEEE	value({ANY_SCALAR_TYPE}) value(slv) value(time), unit(time) value(real), digits(natural) value(real), format(string) -- C-style formatting	IEEE defined to_string functions. Return a *string* with the value of the argument 'value'.
v_string := to_string()	val(bool), width(natural), justified(side), format_spaces(t_format_spaces), [truncate(t_truncate_string)] val(int), width(natural), justified(side), format_spaces(t_format_spaces), [truncate(t_truncate_string), [radix(t_radix), [prefix(t_radix_prefix), [format(t_format_zeros)]]]] val(int), radix(t_radix), prefix(t_radix_prefix), [format(t_format_zeros)] val(slv), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(t_slv_array), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(u), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(t_unsigned_array), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(s), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(t_signed_array), radix(t_radix), [format(t_format_zeros), [prefix(t_radix_prefix)]] val(string) -- Removes non printable ascii characters **Examples** v_string := to_string(v_u8, DEC); v_string := to_string(v_slv8, HEX, AS_IS, INCL_RADIX);	Additions to the IEEE defined to_string functions. Return a *string* with the value of the argument 'val'. - type t_radix is (BIN, HEX, DEC, HEX_BIN_IF_INVALID) - type t_format_spaces is (KEEP_LEADING_SPACE, SKIP_LEADING_SPACE) - type t_truncate_string is (DISALLOW_TRUNCATE, ALLOW_TRUNCATE) - type t_format_zeros is (AS_IS, SKIP_LEADING_0) - type t_radix_prefix is (EXCL_RADIX, INCL_RADIX) - type t_format_zeros is (KEEP_LEADING_0, SKIP_LEADING_0) **Defaults**: *justified*<=RIGHT, *truncate*<=DISALLOW_TRUNCATE, *prefix*<=EXCL_RADIX
v_string := to_upper()	val(string) **Example** v_string := to_upper("lowercase string");	Returns a *string* containing an upper case version of the argument 'val'
v_string := justify() *IEEE	value(string), [justified(side)], [field(width)]	IEEE implementation of justify. Returns a *string* where 'value' is justified to the side given by 'justified' (right, left). **Defaults**: *justified*<=right, field<=0
v_string := justify()	val(string), justified(side), width(natural), format_spaces(t_format_spaces), truncate(t_truncate_string) **Example** v_string := justify("string", RIGHT, C_STRING_LENGTH, ALLOW_TRUNCATE, KEEP_LEADING_SPACE);	Addition to the IEEE implementation of justify(). Returns a *string* where 'val' is justified to the side given by 'justified' (right, left, center). In addition to right and left, center is also an option. The string can be truncated with the 'truncate' parameter (ALLOW_TRUNCATE, DISALLOW_TRUNCATE) or leading spaces can be removed with 'format_spaces' (KEEP_LEADING_SPACE, SKIP_LEADING_SPACE).
v_string := fill_string()	val(character), width(natural) **Example** v_string := fill_string('X', 10);	Returns a *string* filled with the character 'val'.
v_character := ascii_to_char()	ascii_pos(int), [ascii_allow (t_ascii_allow)] **Example** v_char := ascii_to_char(65); -- ASCII 'A'	Return the ASCII to character located at the argument 'ascii_pos' - type t_ascii_allow is (ALLOW_ALL, ALLOW_PRINTABLE_ONLY) **Defaults**: *ascii_allow*<=ALLOW_ALL

v_int := char_to_ascii()	char (character) **Example** v_int := char_to_ascii('A'); -- Returns 65	Return the ASCII value (integer) of the argument 'char'
v_natural := pos_of_leftmost()	target(character), vector(string), [result_if_not_found (natural)] **Example** v_natural := pos_of_leftmost('x', v_string);	Returns position of left most 'character' in 'string', alternatively return-value if not found **Defaults:** *result_if_not_found*<=1
v_natural := pos_of_rightmost()	target(character), vector(string), [result_if_not_found (natural)] **Example** v_natural := pos_of_rightmost('A', v_string);	Returns position of right most 'character' in 'string', alternatively return-value if not found **Defaults:** *result_if_not_found*<=1
v_string := remove_initial_chars()	source(string), num(natural) **Example** v_string :=remove_initial_chars("abcde",1); -- Returns "bcde"	Return string less the num (number of chars) first characters
v_string := get_procedure_name_from_instance_name()	val(string) **Example** v_string := get_procedure_name_from_instance_name(c_int'instance_name);	Returns procedure, process or entity name from the given instance name as *string*. The instance name must be <object>'instance_name, where object is a signal, variable or constant defined in the procedure, process and entity/process respectively e.g. get_entity_name_from_instance_name(my_process_variable'instance-name)
v_string := get_process_name_from_instance_name()	val(string) **Example** v_string := get_process_name_from_instance_name(c_int'instance_name);	
v_string := get_entity_name_from_instance_name()	val(string) **Example** v_string := get_entity_name_from_instance_name(c_int'instance_name);	
v_string := replace()	val(string), target_char(character), exchange_char(character) **Example** v_string := replace("string_x", 'x', 'y'); -- Returns "string_y"	String function returns a *string* where the target character has been replaced by the exchange character.
replace()	variable text_line(inout line), target_char(character), exchange_char(character) **Example** replace(str, 'a', 'b');	Similar to function version of replace(). Line procedure replaces the input with a line where the target character has been replaced by the exchange character.
v_string := pad_string()	val(string), char(character), width(natural), [side(side)] **Example** v_string := pad_string("abcde", '-', 10, LEFT);	Returns a string of width 'width' with the string 'val' on the side of the string given in 'side' (LEFT, RIGHT). The remaining width is padded with 'char'. **Defaults:** *side* <= *LEFT*

Note: See section 1.2.1 for general string handling features for the log() procedure

1.7 Signal generators

Name	Parameters and examples	Description
clock_generator()	clock_signal(sl), [clock_count (natural)], clock_period(time), [clock_high_percentage(natural)] clock_signal(sl), [clock_count (natural)], clock_period(time), [clock_high_time(time)] clock_signal(sl), clock_ena(boolean), [clock_count(natural)], clock_period(time), clock_name(string), [clock_high_percentage(natural range 1 to 99)] clock_signal(sl), clock_ena(boolean), [clock_count(natural)], clock_period(time), clock_name(string), [clock_high_time(time)] **Examples** clock_generator(clk50M, 20 ns); clock_generator(clk100M, clk100M_ena, 10 ns, "100 MHz with 60% duty cycle", 60); clock_generator(clk100M, clk100M_ena, clk100M_cnt, 10 ns, "100 MHz with 60% duty cycle", 6 ns);	Generates a clock signal. Usage: Include the clock_generator as a concurrent procedure from your test bench. By using the variant with the *clock_ena* input, the clock can be started and stopped during simulation. Each start/stop is logged (if the msg_id ID_CLOCK_GEN is enabled). Duty cycle can be set either by percentage or time. An optional output signal *clock_count* can be used to keep track of the number of clock cycles that have passed. Always starts on 0. **Defaults:** *clock_high_percentage*<=50
adjustable_clock_generator()	clock_signal(sl), clock_ena(boolean), clock_period(time), clock_high_percentage(natural) clock_signal(sl), clock_ena(boolean), clock_period(time), clock_name(string), clock_high_percentage(natural) clock_signal(sl), clock_ena(boolean), clock_count(natural), clock_period(time), clock_name(string), clock_high_percentage(natural) **Examples** adjustable_clock_generator(clk50M, clk50M_ena, 20 ns, 50); adjustable_clock_generator(clk50M, clk50M_ena, 20 ns, "100MHz clock with 50% duty cycle", 50); adjustable_clock_generator(clk50M, clk50M_ena, clk50M_cnt, 20 ns, "100MHz clock with 60% duty cycle", 60);	Generates a clock with adjustable duty cycle. Usage: Include the adjustable_clock_generator as a concurrent procedure from your test bench. Duty cycle can be adjusted by changing the clock_high_percentage. Note that clock_high_percentage has to be set in the range of 1 to 99, and that an TB_ERROR will be raised if scale limits are exceeded. Input parameter clock_period and clock_name are constants. An optional output signal *clock_count* can be used to keep track of the number of clock cycles that have passed. Always starts on 0.
gen_pulse()	target(sl), [pulse_value(sl)], pulse_duration(time), [blocking_mode(t_blocking_mode)], msg, [scope, [msg_id, [msg_id_panel]]] target(sl), [pulse_value(sl)], clock_signal(sl), num_periods(int), msg, [scope, [msg_id, [msg_id_panel]]] target(boolean), [pulse_value(boolean)], pulse_duration(time), [blocking_mode(t_blocking_mode)], msg, [scope, [msg_id, [msg_id_panel]]] target(boolean), [pulse_value(boolean)], clock_signal(sl), num_periods(int), msg, [scope, [msg_id, [msg_id_panel]]] target(slv), [pulse_value(slv)], pulse_duration(time), [blocking_mode(t_blocking_mode)], msg, [scope, [msg_id, [msg_id_panel]]] target(slv), [pulse_value(slv)], clock_signal(sl), num_periods(int), msg, [scope, [msg_id, [msg_id_panel]]] **Examples** gen_pulse(sl_1, 50 ns, BLOCKING, "Pulsing for 50 ns"); gen_pulse(sl_1, '1', 50 ns, BLOCKING, "Pulsing for 50 ns"); gen_pulse(slv8, 50 ns, "Pulsing SLV for 50 ns", ALLOW_PULSE_CONTINUATION); gen_pulse(slv8, x"AB", clk100M, 2, "Pulsing SLV for 2 clock periods");	Generates a pulse on the target signal for a certain amount of time or a number of clock cycles. - If blocking_mode = BLOCKING: Procedure blocks the caller (f.ex the test sequencer) until the pulse is done. (default) - If blocking_mode = NON_BLOCKING : Procedure starts the pulse and schedules the end of the pulse, so that the caller can continue immediately. Note that the clock_signal version will synchronize the pulse to clock signal and begin the pulse on falling edge and end the pulse on a succeeding falling edge. **Defaults**: *pulse_value*<=('1'|true|(others=>'1')), *scope*<=C_TB_SCOPE_DEFAULT, *msg_id*<=ID_GEN_PULSE, *msg_id_panel*<=shared_msg_id_panel

slettet: ,

1.8 Synchronisation

Name	Parameters and examples	Description
block_flag()	flag_name(string), msg, [already_blocked_severity(t_alert_level), [scope]] **Example** block_flag("my_flag","blocking my flag") block_flag(C_MY_FLAG_1,"blocking " & C_MY_FLAG_1, WARNING, "My Scope")	Blocks a flag to allow synchronisation between processes. Adds a new blocked flag if it does not already exist. Maximum number of flags can be modified in adaptation_pkg. Sets an alert with *already_blocked_severity* if the flag already is blocked. **Defaults:** *already_blocked_severity*<=WARNING, *scope*<=C_TB_SCOPE_DEFAULT **Note:** use a constant for flag_name to avoid typing errors
unblock_flag	flag_name(string), msg, trigger(sl), [scope] **Example** unblock_flag("my_flag","unblocking my flag", global_trigger) unblock_flag(C_MY_FLAG_1,"unblocking" & C_MY_FLAG_1, global_trigger, "My Scope")	Unblocks a flag to allow a process that is waiting on that flag to continue. Adds a new unblocked flag if it does not already exist. Parameter *trigger* is included to pulse the global signal *global_trigger* used to allow *await_unblock_flag()* to detect unblocking. **Mandatory:** *trigger*<=global_trigger **Defaults:** *scope*<=C_TB_SCOPE_DEFAULT **Note:** use a constant for flag_name to avoid typing errors.
await_unblock_flag	flag_name(string), timeout(time), msg, [flag_returning(t_flag_returning), [timeout_severity(t_alert_level), [scope]]] **Examples** await_unblock_flag("my_flag", 0 ns, "waiting for my_flag to be unblocked) await_unblock_flag("my_flag", 10 us, "waiting for my_flag to be unblocked", RETURN_TO_BLOCK, WARNING) await_unblock_flag(C_MY_FLAG_1, 10 us, "waiting for "C_MY_FLAG_1 & " to be unblocked", RETURN_TO_BLOCK, WARNING, "My Scope")	Waits for a flag to be unblocked. Continues immediately if the flag already is unblocked. Adds a new blocked flag if it does not already exist. If so *await_unblock_flag()* will wait for the flag to be unblocked. Sets an alert with *timeout_severity* if the flag is not unblocked within *timeout*. A timeout of 0 ns means wait forever. The flag can be re-blocked when leaving the process by setting *flag_returning*=RETURN_TO_BLOCK. **Defaults:** *flag_returning*<=KEEP_UNBLOCKED, *timeout_severity*<=ERROR, *scope*<=C_TB_SCOPE_DEFAULT **Note:** use a constant for flag_name to avoid typing errors
await_barrier	barrier_signal(sl), timeout(time), msg, [timeout_severity(t_alert_level), [scope]] **Example** await_barrier(global_barrier, 100 us, "waiting for global barrier", ERROR, "My Scope")	For the barrier_signal you can either use the predefined global_barrier or you can define your own barrier_signal of type sl. The function can be used to synchronise between several sequencers. When the function is called, it waits for all sequencer using the same barrier_signal to reach their call of await_barrier.

1.9 BFM Common package

(Methods are defined in uvvm_util.bfm_common_pkg)

Name	Parameters and examples	Description
{slv, u, s, t_slv_array, t_signed_array, t_unsigned_array} := normalize_and_check()	value(slv), target(slv), mode (t_normalization_mode), value_name, target_name, msg value(t_slv_array), target(t_slv_array), mode (t_normalization_mode), value_name, target_name, msg value(u), target (u), mode (t_normalization_mode), value_name, target-_name, msg value(t_unsigned_array), target(t_unsigned_array), mode (t_normalization_mode), value_name, target_name, msg value(s), target (s), mode (t_normalization_mode), value_name, target_name, msg value(t_signed_array), target(t_signed_array), mode (t_normalization_mode), value_name, target_name, msg **Example** v_slv8 := normalize_and_check(v_slv5, v_slv8, ALLOW_NARROWER, "8bit slv", "5bit slv", "Normalizing and checking slv");	Normalize 'value' to the width given by 'target'. If value'length > target'length, remove leading zeros (or sign bits) from value. If value'length < target'length, add padding (leading zeros, or sign bits) to value. Mode (t_normalization_mode) is used for sanity checks, and can be one of : ALLOW_WIDER : Allow only value'length > target'length ALLOW_NARROWER : Allow only value'length < target'length ALLOW_WIDER_NARROWER : Allow both of the above ALLOW_EXACT_ONLY : Allow only value'length = target'length
wait_until_given_time_after_rising_edge()	clk(sl), wait_time(time) **Example** wait_until_given_time_after_rising_edge(clk50M, 5 ns);	Wait until wait_time after rising_edge(clk) If the time passed since the previous rising_edge is less than wait_time, don't wait until the next rising_edge, just wait_time after the previous rising_edge.
Wait_until_given_time_before_rising_edge()	clk(sl), time_to_edge(time), clk_period(time) **Example** wait_until_given_time_after_rising_edge(clk50M, 2 ns, 10 ns);	Wait until time_to_edge before rising_edge(clk) If the time until rising_edge is less than time_to_edge, wait until the next rising_edge and afterwards until time_to_edge before rising_edge
wait_num_rising_edge()	clk(sl), num_rising_edge(natural) **Example** wait_num_rising_edge(clk10M, 5);	Waits for 'num_rising_edge' rising edges of the clk signal
wait_num_rising_edge_plus_margin()	clk(sl), num_rising_edge(natural), margin(time) **Example** wait_num_rising_edge_plus_margin(clk50M, 3, 4 ns);	Waits for 'num_rising_edge' rising edges of the clk signal, and then waits for 'margin'.
wait_on_bfm_sync_start()	clk(sl), bfm_sync(t_bfm_sync), setup_time(time), config_clock_period(time), time_of_falling_edge(time), time_of_rising_edge(time) **Example** wait_on_bfm_sync_start(clk, config.bfm_sync, 2.5 ns, 10 ns, v_time_of_falling_edge, v_time_of_rising_edge);	Synchronizes the start of a BFM procedure depending on bfm_sync: -SYNC_ON_CLOCK_ONLY: waits until the falling_edge of the clk signal. -SYNC_WITH_SETUP_AND_HOLD: waits until the setup time before the clock's rising_edge. It returns the times of falling and rising edges. When not found returns -1 ns.
wait_on_bfm_exit()	clk(sl), bfm_sync(t_bfm_sync), hold_time(time), time_of_falling_edge(time), time_of_rising_edge(time) **Example** wait_on_bfm_exit(clk, config.bfm_sync, 2.5 ns, v_time_of_falling_edge, v_time_of_rising_edge);	Synchronizes the exit of a BFM procedure depending on bfm_sync: -SYNC_ON_CLOCK_ONLY: waits until one quarter of the clock period (measured with the falling and rising edges) after the clock's rising_edge. -SYNC_WITH_SETUP_AND_HOLD: waits until the hold time after the clock's rising_edge. The times of falling and rising edges must be consecutive to be able to calculate the correct clock period.

check_clock_period_margin()	clock(sl), bfm_sync(t_bfm_sync), time_of_falling_edge(time), time_of_rising_edge(time), config_clock_period(time), config_clock_period_margin(time), config_clock_margin_severity(t_alert_level) **Example** check_clock_period_margin(clk, config.bfm_sync, v_time_of_falling_edge, v_time_of_rising_edge, 10 ns, 2 ns, TB_ERROR);	Checks that the clock signal behaves according to configured specifications. Only when bfm_sync = SYNC_WITH_SETUP_AND_HOLD. The procedure must be called after the clock's rising_edge.

1.10 Watchdog

Name	Parameters and examples	Description
watchdog_timer()	watchdog_ctrl(t_watchdog_ctrl), timeout (time), [alert_level, [msg]] **Example** watchdog_timer(watchdog_ctrl, 500 us, ERROR, "Watchdog timer");	This procedure has to be instantiated as a concurrent procedure in the testbench or test harness. Initializes the watchdog timer as a concurrent procedure that will run until the timeout expires. A signal of the type t_watchdog_ctrl must be defined in the testbench, this is needed to call the other procedures on the watchdog.
extend_watchdog()	watchdog_ctrl(t_watchdog_ctrl), [time_extend (time)] **Example** extend_watchdog(watchdog_ctrl, 100 us)	Extends the timeout of the watchdog timer by the specified time. If no time is given, the original timeout will be used as the time extension.
reinitialize_watchdog()	watchdog_ctrl(t_watchdog_ctrl), timeout (time) **Example** reinitialize_watchdog(watchdog_ctrl, 1 ms)	Reinitializes the watchdog timer with a new timeout.
terminate_watchdog()	watchdog_ctrl(t_watchdog_ctrl) **Example** terminate_watchdog(watchdog_ctrl)	Terminates the concurrent procedure where the watchdog timer is running. Once this is done the watchdog can't be started again. This should normally be called at the end of the simulation.

Note! The VVCs support an activity watchdog. See UVVM Essential Mechanisms PDF in UVVM VVC Framework for more details.

formaterte: Skrift: 10 pkt

slettet: From UVVM version 3 t

formaterte: Skrift: 10 pkt

formaterte: Skrift: 10 pkt, Engelsk (USA)

1.11 Message IDs

A sub set of message IDs is listed in this table. All the message IDs are defined in uvvm_util.adaptations_pkg.

Message ID	Description
ID_LOG_HDR	For all test sequencer log headers. Special format with preceding empty line and underlined message (also applies to ID_LOG_HDR_LARGE and ID_LOG_HDR_XL).
ID_SEQUENCER	For all other test sequencer messages
ID_SEQUENCER_SUB	For general purpose procedures defined inside TB and called from test sequencer
ID_POS_ACK	A general positive acknowledge for check routines (incl. awaits)
ID_BFM	BFM operation (e.g. message that a write operation is completed) (BFM: Bus Functional Model, basically a procedure to handle a physical interface)
ID_BFM_WAIT	Typically BFM is waiting for response (e.g. waiting for ready, or predefined number of wait states)
ID_BFM_POLL	Used inside a BFM when polling until reading a given value, i.e., to show all reads until expected value found.
ID_PACKET_INITIATE	A packet has been initiated (Either about to start or just started)
ID_PACKET_COMPLETE	Packet completion
ID_PACKET_HDR	Packet header information
ID_PACKET_DATA	Packet data information
ID_LOG_MSG_CTRL	Dedicated ID for enable/disable_log_msg
ID_CLOCK_GEN	Used for logging when clock generators are enabled or disabled
ID_GEN_PULSE	Used for logging when a gen_pulse procedure starts pulsing a signal
ID_NEVER	Used for avoiding log entry. Cannot be enabled.
ALL_MESSAGES	Not an ID. Applies to all IDs (apart from ID_NEVER)

Message IDs are used for verbosity control in many of the procedures and functions in UVVM-Util, and are toggled by using the procedures *enable_log_msg()* and *disable_log_msg()* that are described in this document.

Example: A check is performed each clock cycle;

check_value(my_boolean_condition, error, "Verifying condition", C_SCOPE, ID_POS_ACK, my_msg_id_panel);

The message ID "ID_POS_ACK" is enabled by default, and will report a positive acknowledge if the check passes. Since the check is performed each clock cycle, the positive acknowledge will be printed each clock cycle. There are two possibilities if you wish to turn off the positive acknowledge message:

- Disable "ID_POS_ACK" in *my_msg_id_panel* (or use another msg_id_panel) by calling *disable_log_msg(ID_POS_ACK, my_msg_id_panel)*. This will disable positive acknowledge messages for any procedure call that uses this msg_id_panel.
- Call *check_value()* with "ID_NEVER" instead of "ID_POS_ACK". This will disable the positive acknowledge for this particular call of *check_value()*, but all other calls to *check_value()* will report a positive acknowledge.

1.12 Common arguments in checks and awaits

Most check and await methods have two groups of arguments:

- arguments specific to this function/procedure
- **common_args**: arguments common for all functions/procedures:
  - alert_level, msg, [scope], [msg_id], [msg_id_panel]

For example: check_value(*val, exp,* ERROR, "Check that the val signal equals the exp signal", C_SCOPE);

The common arguments are described in the following table.

Argument	Type	Example	Description
alert_level	t_alert_level;	ERROR	Set the severity for the alert that may be asserted by the method.
msg	string;	"Check that bus is stable"	A custom message to be appended in the log/alert.
scope	string;	"TB Sequencer"	A string describing the scope from which the log/alert originates.
msg_id	t_msg_id	ID_BFM	Optional message ID, defined in the adaptations package. Default value for check routines = ID_POS_ACK;
msg_id_panel	t_msg_id_panel	local_msg_id_panel	Optional msg_id_panel, controlling verbosity within a specified scope. Defaults to a common ID panel defined in the adaptations package.

1.13 Using Hierarchical Alert Reporting

Enable hierarchical alerts via the constant C_ENABLE_HIERARCHICAL_ALERTS in the adaptations package.
The procedures used for hierarchical alert reporting are described in the following table.

Name	Parameters and examples	Description
add_to_alert_hierarchy()	scope(string), [parent_scope(string), [stop_limit(t_alert_counters)]] **Example** add_to_alert_hierarchy("tier_2", "tier_1");	Add a scope in the alert hierarchy tree. **Defaults:** *parent_scope* <= C_BASE_HIERARCHY_LEVEL, *stop_limit* <= (others => 0)
increment_expected_alerts()	scope(string), alert_level, [amount(natural)] **Example** increment_expected_alerts("tier_2", ERROR, 2);	Increment the expected alert counter for a scope. **Defaults:** *amount* <= 1
set_expected_alerts()	scope(string), alert_level, expected_alerts(natural) **Example** set_expected_alerts("tier_2", WARNING, 5);	Set the expected alert counter for a scope.
increment_stop_limit()	scope(string), alert_level, [amount(natural)] **Example** increment_stop_limit("tier_1", ERROR);	Increment the stop limit for a scope. **Defaults:** *amount* <= 1
set_stop_limit()	scope(string), alert_level, stop_limit (natural) **Example** set_stop_limit("tier_1", ERROR, 5);	Set the stop limit for a scope.

- By default, there is only one level in the hierarchy tree, and one scope with name given by C_BASE_HIERARCHY_LEVEL in the adaptations package. This scope has a stop limit of 0 by default.
- To add a scope to the hierarchy, call *add_to_alert_hierarchy()*.
- Any scope that is not registered in the hierarchy will be automatically registered if an alert is triggered in that scope. The parent scope will then be C_BASE_HIERARCHY_LEVEL. Changing the parent is possible by calling *add_to_alert_hierarchy()* with another scope as parent. This is only allowed if the parent is C_BASE_HIERARCHY_LEVEL and may cause an odd-looking summary (total summary will be correct).

slettet: default

slettet: ,

Intended use:
In UVVM mostly use the scope to describe components, e.g. VVCs. It can also be smaller structures, but it has to have its own sequencer.
A good way to set up the hierarchy is to let every scope register themselves with the default parent scope, and then in addition make every parent register each of its children. This is because the child scope doesn't have to have the same parent scope in all testbenches/testharnesses, i.e. the child doesn't know its parent.

- In the child, call *add_to_alert_hierarchy(<child scope>)*. This will add the scope of the child to the hierarchy with the default (base) parent.
- In the parent, first call *add_to_alert_hierarchy(<parent scope>)*. Then call immediately *add_to_alert_hierarchy(<child scope>, <parent scope>)* for each of the scopes that shall be children of this parent scope. This will re-register the children to the correct parent.

Example output:

```
================================================================================================================================================================
*** FINAL SUMMARY OF ALL ALERTS  ***     Format: REGARDED/EXPECTED/IGNORED
================================================================================================================================================================
                                      NOTE      TB_NOTE     WARNING   TB_WARNING  MANUAL_CHECK   ERROR     TB_ERROR     FAILURE   TB_FAILURE
TB seq                          :     5/5/5       5/5/5       5/5/5       5/5/5       5/5/5       5/5/5       5/5/5       5/5/5       5/5/5
`- scope_1                      :     4/4/4       4/4/4       4/4/4       4/4/4       4/4/4       4/4/4       4/4/4       4/4/4       4/4/4
   |- scope_2                    :     1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1
   `- scope_3                    :     2/2/2       2/2/2       2/2/2       2/2/2       2/2/2       2/2/2       2/2/2       2/2/2       2/2/2
      `- scope_4                 :     1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1       1/1/1
================================================================================================================================================================
>> Simulation SUCCESS: No mismatch between counted and expected serious alerts
================================================================================================================================================================
```

1.14 Adaptation package

The `adaptations_pkg.vhd` is intended for local modifications to library behaviour and log layout.
This way only one file needs to merge when a new version of the library is released.
This package may of course also be used to set up a company or project specific behaviour and layout.
The layout constants and global signals are described in the following tables.

Constant	Description
C_ALERT_FILE_NAME	Name of the alert file.
C_LOG_FILE_NAME	Name of the log file.
C_SHOW_UVVM_UTILITY_LIBRARY_INFO	General information about the UVVM Utility Library will be shown when this is enabled.
C_SHOW_UVVM_UTILITY_LIBRARY_RELEASE_INFO	Release information will be shown when this is enabled.
C_UVVM_TIMEOUT	General timeout for UVVM wait statements.
C_LOG_PREFIX	The prefix to all log messages. "UVVM: " by default.
C_LOG_PREFIX_WIDTH	Number of characters to be used for the log prefix.
C_LOG_MSG_ID_WIDTH	Number of characters to be used for the message ID.
C_LOG_TIME_WIDTH	Number of characters to be used for the log time. Three characters are used for time unit, e.g., ' ns'.
C_LOG_TIME_BASE	The unit in which time is shown in the log. Either ns or ps.
C_LOG_TIME_DECIMALS	Number of decimals to show for the time.
C_LOG_SCOPE_WIDTH	Number of characters to be used to show log scope.
C_LOG_LINE_WIDTH	Number of characters allowed in each line in the log.
C_LOG_INFO_WIDTH	Number of characters of information allowed in each line in the log. By default, this is set to C_LOG_LINE_WIDTH – C_LOG_PREFIX_WIDTH.
C_LOG_HDR_FOR_WAVEVIEW_WIDTH	Number of characters for a string in the waveview indicating last log header.
C_WARNING_ON_LOG_ALERT_FILE_RUNTIME_RENAME	Whether or not to report a warning if the log or alert files are renamed after they have been written.
C_USE_BACKSLASH_N_AS_LF	If true '\n' will be interpreted as line feed.
C_USE_BACKSLASH_R_AS_LF	If true '\r' placed as the first character in the string will be interpreted as a LF where the timestamp, Id etc. will be omitted.
C_SINGLE_LINE_ALERT	If true prints alerts on a single line. Default false.
C_SINGLE_LINE_LOG	If true prints logs messages on a single line. Default false.
C_TB_SCOPE_DEFAULT	The default scope in the test sequencer.
C_LOG_TIME_TRUNC_WARNING	Yields a single TB_WARNING if time stamp truncated. Otherwise none.
C_DEFAULT_MSG_ID_PANEL	Sets the default message IDs that shall be shown in the log.
C_MSG_ID_INDENT	Sets the indentation for each message ID.
C_DEFAULT_ALERT_ATTENTION	Sets the default alert attention.
C_DEFAULT_STOP_LIMIT	Sets the default alert stop limit.
C_ENABLE_HIERARCHICAL_ALERTS	Whether or not to enable hierarchical alert summary. Default false.
C_BASE_HIERARCHY_LEVEL	The name of the base/top level node that all other nodes in the tree will originate from.
C_DEPRECATE_SETTING	Sets how the user is to be notified if a procedure has been deprecated and will be removed in later versions.
C_VVC_RESULT_DEFAULT_ARRAY_DEPTH	Default for how many results (e.g. reads) a VVC can store before overwriting old results
C_VVC_MSG_ID_PANEL_DEFAULT	Default message ID panel to use in VVCs
C_SHOW_LOG_ID	Whether or not to show the Log ID field
C_SHOW_LOG_SCOPE	Whether or not to show the Log Scope field

slettet: d

slettet: .

slettet: ,

Global signal	Signal type	Description
global_show_msg_for_uvvm_cmd	boolean	If true messages for Bitvis UVVM commands will be shown if applicable.

Shared variable	Signal type	Description
shared_default_log_destination	t_log_destination	The default destination for the log messages (Default: CONSOLE_AND_LOG)

Additional Documentation

There are two other main documents for the UVVM Utility Library (available from our Downloads page)
- 'Making a simple, structured and efficient VHDL testbench – Step-by-step'
- 'Bitvis Utility Library – Concepts and Usage'

There is also a webinar available on 'Making a simple, structured and efficient VHDL testbench – Step-by-step' (via Aldec). Link on our downloads page.

2 Compilation

UVVM Utility Library may only be compiled with VHDL 2008.

Compile order for UVVM Utility Library:

Compile to library	File
uvvm_util	uvvm_util/src/types_pkg.vhd
uvvm_util	uvvm_util/src/adaptations_pkg.vhd
uvvm_util	uvvm_util/src/string_methods_pkg.vhd
uvvm_util	uvvm_util/src/protected_types_pkg.vhd
uvvm_util	uvvm_util/global_signals_and_shared_variables_pkg.vhd
uvvm_util	uvvm_util/src/hierarchy_linked_list_pkg.vhd
uvvm_util	uvvm_util/src/alert_hierarchy_pkg.vhd
uvvm_util	uvvm_util/src/license_pkg.vhd
uvvm_util	uvvm_util/src/methods_pkg.vhd
uvvm_util	uvvm_util/src/bfm_common_pkg.vhd
uvvm_util	uvvm_util/src/uvvm_util_context.vhd

Modelsim and Riviera-PRO users can compile the library by sourcing the following files:
`script/compile_src.do`

Note that the compile script compiles the Utility Library with the following Modelsim directives for the vcom command:

Directive	Description
-suppress 1346,1236	Suppress warnings about the use of protected types. These can be ignored.

The uvvm_util project is opened by opening `sim/uvvm_util.mpf` in Modelsim.

3 Simulator compatibility and setup

UVVM Utility Library has been compiled and tested with Modelsim, Riviera-PRO and Active HDL.
See README.md for a list of supported simulators.

Required setup:

- Textio buffering should be removed or reduced. (Modelsim.ini: Set UnbufferedOutput to 1)
- Simulator transcript (and log file viewer) should be set to a fixed width font type for proper alignment (e.g. Courier New 8)
- Simulator must be set up to break the simulation on failure (or lower severity)