UVVM Essential Mechanisms – Quick Reference

This document explains some of the essential mechanisms necessary for running UVVM, in addition to helpful and important VVC status and configuration records which are accessible directly from the testbench. More details on the VVC Framework and the command mechanism can be found in the VVC Framework Manual.

UVVM™

VHDL 2008 only

1 Libraries

In order to use a VVC the following libraries need to be included:

```
library uvvm_util;
context uvvm_util.uvvm_util_context;

library uvvm_vvc_framework;
use uvvm_vvc_framework.ti_vvc_framework_support_pkg.all;

library bitvis_vip_<name>;
context bitvis_vip_<name>.vvc_context;
```

2 UVVM Initialization

The following mechanisms are required for running UVVM

Mechanism	Description
ti_uvvm_engine	**ti_uvvm_engine** This entity contains a process that will initialize the UVVM environment, and has to be instantiated in the testbench harness, or alternatively in the top-level testbench. Example: `i_ti_uvvm_engine : entity uvvm_vvc_framework.ti_uvvm_engine;`
await_uvvm_initialization()	**await_uvvm_initialization(VOID)** This procedure is a blocking procedure that has to be called from the testbench sequencer, prior to any VVC calls, to ensure that the UVVM engine has been initialized and is ready. This procedure will check the shared_uvvm_state on each delta cycle until the UVVM engine has been initialized. Note that this method is depending on the ti_uvvm_engine mechanism. Note that this method uses the `t_void` parameter, defined in the UVVM Utility Library types package. Example: `await_uvvm_initialization(VOID);`

3 UVVM and VVC Shared Variables

UVVM and VVC shared variables are defined in global_signals_and_shared_variables_pkg and the various vvc_methods_pkg, respectively.

Shared variables

Shared variable	Description

shared_uvvm_status	Shared variable providing access to VVC related information via the `info_on_finishing_await_any_completion` record element, i.e.
	`shared_uvvm_status.info_on_finishing_await_any_completion`
	
	This record element gives access to the name, command index and the time of completion of the VVC that first fulfilled the `await_any_completion()`. The available record fields are:
	
	`vvc_name : string -- default "no await_any_completion() yet"`
	`vvc_cmd_idx : natural -- default 0`
	`vvc_time_of_completion : time -- default 0 ns`
	
	For more information regarding other fields available in the `shared_uvvm_status` see the UVVM Util QuickRef, section 1.4

shared_<vvc_name>_vvc_config	Shared variable providing access to configuration parameters for each VVC instance and channel if applicable. E.g.
	
	`shared_sbi_vvc_config(1).inter_bfm_delay.delay_type := TIME_START2START;`
	`shared_uart_vvc_config(RX,1).bfm_config.bit_time := C_BIT_TIME;`

shared_<vvc_name>_vvc_status	Shared variable providing access to status parameters for each VVC instance and channel if applicable. E.g.
	
	`v_num_pending_cmds := shared_sbi_vvc_status(1).pending_cmd_cnt;`
	`v_current_cmd_idx := shared_uart_vvc_status(TX,2).current_cmd_idx;`
	`v_previous_cmd_idx := shared_uart_vvc_status(TX,2).previous_cmd_idx;`

shared_<vvc_name>_transaction_info	Shared variable providing access to VVC instances transaction information to include in the wave view during simulation.
	Available information is dependent on VVC type and typical information is:
	
	`operation : t_operation; -- default NO_OPERATION`
	`data : std_logic_vector(C_VVC_CMD_DATA_MAX_LENGTH-1 downto 0); -- default 0x0`
	`msg : string(1 to C_VVC_CMD_STRING_MAX_LENGTH); -- default empty`

4 VVC Status, Configuration and Transaction information

The VVC status, configuration and transaction information records are defined in each individual VVC methods package.

Each VVC instance and channel can be configured and useful information can be accessed from the testbench via dedicated shared variables.

From the VVC configuration shared variable, one is given the ability to tailor each VVC to one's needs, in addition to access the BFM configuration record via the `bfm_config` identifier. In addition to BFM configuration possibility, the configuration settings consist of command and result queue settings, BFM access separation delay and a VVC dedicated message ID panel.
Note that some BFMs require user configuration, e.g. the `bit_time` setting in serial interface BFMs.

The VVC status shared variable provide access to the command status parameters for each of the VVCs, such as the current and previous command index, and the number of pending commands in the VVCs command queue. This provide a helpful tool, e.g. when synchronizing VVCs in the test sequencer using the `await_completion()` or `await_any_completion()` methods.

When using a wave viewer during simulation, the transaction shared variable provides helpful information regarding current VVC operation and transaction information such as `address` and `data`. Note that the accessible fields depend on the VVC and its implementation. An example of two SBI VVCs performing FIFO write operations, followed by check operations, is shown in Figure 4-1.

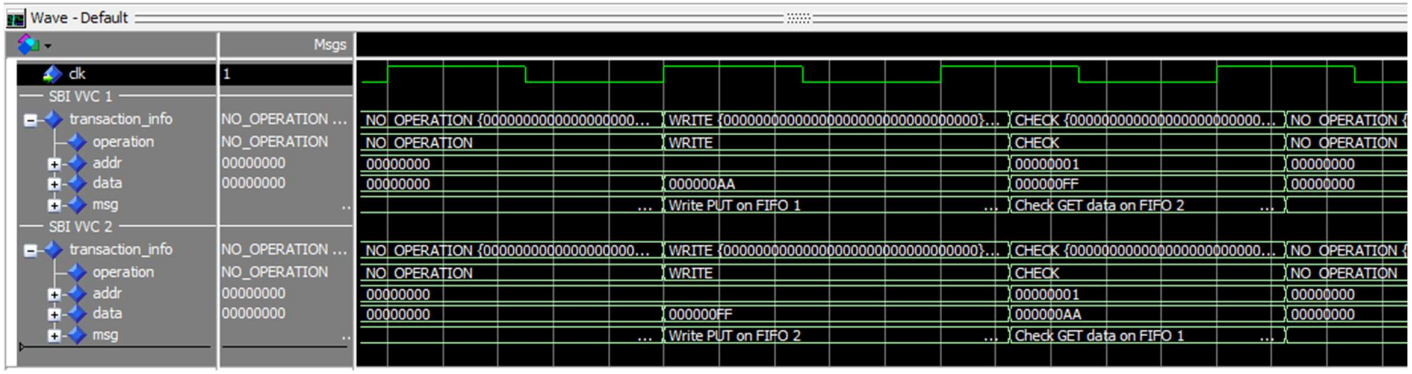

Figure 4-1 VVC Transaction info example

5 Direct Transaction Transfer – From VVCs and/or Monitors

UVVM now supports sharing transaction information in a controlled manner within the complete testbench environment. This allows VVCs and Monitors to provide transaction info to any other part of your testbench – using a predefined structured mechanism. This makes it even easier to make good VHDL testbenches.

Transaction information may be used in many different ways, but the main purpose is to share information inside the testbench of activity or accesses on a given DUT interface. Normally such information is provided from a dedicated interface monitor, but making such a dedicated monitor is sometimes quite time consuming and often not really needed. For that reason. UVVM provides a mechanism for getting the transaction information directly from the VVC.

5.1 Purpose

The purpose of the direct transaction transfer is to allow a model or other part of the testbench to see exactly what accesses have been made on the various interfaces of the DUT, so that the expected DUT behaviour and outputs may be checked. Let us illustrate this with a really simple testbench scenario to verify a UART peripheral with an AXI-lite register/CPU interface on one side and the UART RX and TX ports on the other side. The test sequencer may command the AXI-lite BFM or VVC to write a data byte into the UART TX register, and then it must be checked that the data byte is transmitted out on the DUT TX output some time later.

a) A simple testbench approach could be to have the test sequencer also telling the receiving UART BFM or VVC exactly what to expect. This is a straight forward approach, but requires more action and data control inside the test sequencer. This could of course all be handled in a super-procedure, but for any undetermined behaviour inside the BFM or VVC, like random data generation or error injection, that would not work.
b) A more advanced approach is to have a model overlooking the DUT accesses, generate the expected data and tell the receiving BFM or VVC to check for that data
c) An even more advanced approach would be to use a Scoreboard to check received data (from DUT via VVC) against expected data from a model.

However, for the two latter approaches the model needs information about exactly what happened (the transaction) on the various DUT interfaces, so that it can generate the correct expected data. For the model it doesn't matter if the transaction info comes from a monitor or from a VVC, as long as the information is correct.

The model could of course look at the interfaces and analyse the transactions itself, but distributing this task to the VVC or monitor makes the testbench far more structured and significantly improves overview, maintenance, extensibility and reuse – at least for anything above medium simple verification challenges.

Another purpose of providing transaction information is for progress viewing and debugging – typically via the wave view or simulation transcripts.

5.2 Transaction definitions

By transactions we normally talk about a complete end to end transfer of data across an interface. This could be anything from a simple write, read or transfer of a single word - to a complete packet in a packet-oriented protocol like Ethernet. The word transaction is however also used for both sub-sets and super-sets of this – depending on the protocol and even on how we want to control our system or testbench. In order to communicate properly and to assure that transactions are properly understood, the following terms are defined:

- **Base transaction (BT)** is the lowest level of a complete transaction as allowed from the central sequencer.
  E.g. AXI-lite read, write or check, UART transmit, receive or expect, Ethernet transmit, receive or expect
- **Sub-transaction (ST)** is the lowest level of an incomplete transaction as allowed from a BFM. The sub-transaction as such is complete seen from a handshake point of view, but the transfer of data is not complete. A split transaction protocol will typically have sub-transactions.
  E.g. Avalon Read Request and Avalon Read Response
- **Leaf transaction (LT)** is not a transaction type in itself, but is the lowest level of complete **or** incomplete transaction defined for a given protocol. I.e. a sub-transaction when this is defined for a given protocol, otherwise a base transaction. This definition is needed in order simplify various explanations.
  (e.g. for Avalon: LT = the sub transactions, and for UART, SBI or Ethernet: LT = the base transactions (as no sub-transaction exist for these protocols)
- **Compound transaction (CT)** is a set of transactions or other methods or statements that as a total is doing a more complex operation.
  E.g. SBI_Poll_Until() or a UART transmit of N consecutive bytes.

5.3 Transaction information

Information about the above transactions is typically provided to a model in the test harness. Depending on whether the transaction info is provided from the VVC or Monitor, different types of information will be available. Common for both is that they always provide info about the operation (read, write, transmit, etc) and often also any other protocol specific info. For a UART this could be data and parity, for an SBI it could be address and data, and for Ethernet: the packet fields.
This minimum is normally what the Monitor can provide from just analysing the interface, and this is also normally enough for a model to generate expected DUT outputs. The VVC on the other hand, can provide more info, which could be useful for instance for progress viewing and debugging.
The transaction information is organised as a transaction record with some predefined fields as shown below. The first table shows the general transaction record, whereas the second table shows a concrete example for the SBI.
Note that for a given interface/protocol, the VVC and the Monitor will use the same interface dedicated transaction record type – with some fields potentially unused.

Table 1 – General Transaction record t_transaction. The greyed-out fields indicate optional or protocol dedicated fields

Field name	Type name	Description
Operation	t_operation	Protocol operation on the given DUT interface. E.g. NO_OPERATION, WRITE, READ, TRANSMIT, POLL_UNTIL, … NO_OPERATION is default and thus used when there is no access. All operations will be separated with a NO_OPERATION for at least 1 delta cycle, e.g. NO_OPERATION – WRITE – NO_OPERATION – READ – NO_OPERATION.
<Optional protocol dedicated field(s)>	<Protocol dedicated>	One or more fields required to complete the transaction info E.g. for UART: Single field 'data'; for SBI: field 1: 'addr', field 2: 'data'; for Ethernet: Most ethernet fields as separate fields here – or a better solution include as a complete sub-record
Transaction_status	t_transaction_status	Transaction status. Handled slightly different from a VVC and a Monitor. VVC: Will show 'IN_PROGRESS' during the transaction and INACTIVE in between (for at least one delta cycle) Monitor: Will show FAILED or SUCCEEDED immediately as soon as this is 100% certain – and keep this info for the display period defined in the Monitor configuration record, or until the next transaction is ready to be displayed. Other than that it will show INACTIVE (even when a transaction has started – before the transaction status is known)
VVC_Meta	t_vvc_meta	Additional transaction information – only known by the VVC. So far 'msg' and 'cmd_idx' (the free running command index) A monitor has no knowledge of this as and will set them to msg = "", cmd_idx = -1
Error_info	t_error_info	Any type of error injection relevant for the given protocol. Typically parity or stop-bit error in an UART or a CRC error in an Ethernet. If no error injection or detection has been implemented, this sub-record may be left out.
NOTES: - For transaction info from a VVC the record reflects the command status, i.e. the status assumed by the VVC when initiating the command, whereas the Monitor will set up the record only after knowing whether the transaction has failed or succeeded. The VVC does not know the BFM status, and this is fine because the BFM will issue an alert for unexpected behavior.		

Table 2 – SBI specific Transaction record t_transaction. The greyed-out fields indicate protocol dedicated fields

Field name	Type name	Description
Operation	t_operation	Either of WRITE, READ or CHECK, but could also be POLL_UNTIL or a more complex compound transaction
Address	unsigned	
Data	std_logic_vector	
Transaction_status	t_transaction_status	
VVC_Meta	t_vvc_meta	
Note: No error_info field as no error injection or detection has been implemented in neither VVC nor Monitor – at this stage.		

Other interfaces will of course have different protocol dedicated fields, or even a complete protocol dedicated sub-record (e.g. for Ethernet packet fields)

5.4 Transaction info transfer

In order to reduce the number of signals from a VVC or Monitor, all possible simultaneous transactions (and their transaction records) are collected into a single transaction group record. For an SBI interface this will consist of a BT record and potentially a CT record, whereas for an Avalon it will in addition also consist of two ST records because for instance a read request may be active at the same time as a read response. (And the sub-transactions are part of a base transaction and may also be part of a CT).

Table 3 below shows the maximum transaction group record for an SBI, whereas Table 4 below shows the maximum transaction group record for an Avalon. The greyed-out CT is optional for both, and thus depends on whether CTs have been defined in the VVC. Multiple parallel STs may be written to the transaction group record simultaneously – as these are handled by different "threads" (concurrent statements like a process).

A Monitor cannot know about CTs, and thus a monitor will never fill inn that sub-record. A Monitor for a split transaction protocol (i.e. with multiple STs) may or may not provide BT info. If it does, this should normally be implemented in a higher level "wrapper"

Note again:
- A VVC will update its DTT leaf transaction details at the start of the transaction when the BFM is called. and turned off when BFM is finished.
- A monitor will set its DTT information after the transaction is finished (or transaction status is known) and keep it on for a pre-defined time – or until the next transaction is finished if earlier.

It is recommended that the model (or any other user of the DTT signals) triggers on transaction_status changing to 'INACTIVE' and then sample <signal>'last_value

Table 3 – Maximum transaction group record t_transaction_group – for an SBI interface

Field name	Type name	Description
bt	t_transaction	Base transaction
ct	t_transaction	Compound transaction

Table 4 - Maximum transaction group record t_transaction_group – for an Avalon MM interface

Field name	Type name	Description
st_request	t_transaction	Sub-transaction
st_response	t_transaction	Sub-transaction
Bt	t_transaction	Base transaction
ct	t_transaction	Compound transaction

5.5 Transaction info transfer signals

The DTT (Direct Transaction Transfer) is provided out of the VVC and Monitor using global signals. These signals and all DTT related VHDL types are defined in transaction_pkg, located in the VIP src folder.
- Monitor DTT signal : global_<protocol-name>_monitor_transaction, e.g. global_uart_monitor_transaction
- VVC DTT signal: global_<protocol-name>_transaction, e.g. global_uart_transaction. The VVC is also responsible for filling out the vvc_meta record field.

6 VVC local sequencers

UVVM testbenches may have one or more central sequencers – also known as test sequencers or test drivers. A single test sequencer is recommended in order to reduce complexity – as synchronization between multiple parallel test sequencer could be really complex.
UVVM does however also provide support for so called local sequencers. These sequencers will typically run inside the VVCs executor process. The executor will typically run a single transaction via a BFM procedure towards the DUT interface, like an sbi_write() or uart_expect() procedure. For more advanced VVCs it would however make sense to send even higher level commands to a VVC, like requesting it to transmit N random bytes, or keep on receiving and checking data until a certain functional coverage is reached, or even setting up a peripheral by writing to multiple configuration registers. In these cases, a single command to the VVC will trigger a complete sequence of accesses towards the DUT. The code inside the VVC executors handling these sequences are called local sequencers as they are local to the VVC and thus also improves re-use. These sequences of transactions are also defined as Compound Transactions in chapter 5.2.

Some examples of local sequencers are the randomisation and functional coverage sequences in the UART VVC, and poll_until in the SBI VVC.

6.1 Local sequencer requirements

The following requirements should be followed when making local sequencers (basically any VVC command resulting in more than one transaction):

1. If Direct Transaction Transfer is supported, then both the leaf transaction and the compound transaction info should be updated.
2. The sequence should be handled directly inside the VVC executor – and not inside the BFM
   (Otherwise updating the leaf transactions for Direct Transaction transfer could be difficult
3. It should be possible to terminate the sequence immediately after each leaf transaction – on request from the central sequencer issuing a terminate_current_command() or terminate_all_commands().

7 Protocol aware Error Injection

Error injection into the DUT could be very useful in a testbench in order to test how the DUT handles interface errors when these errors are a) to be detected and corrected, b) detected only, and c) not detected but may or may not affect the behaviour. Protocol aware error injection is defined here as intelligent error injection, given knowledge about the interface and protocol, e.g. to inject a parity error in a protocol rather than just inverting or delaying a signal without pre-defined detailed support to do this at the right place. The latter is supported by a dedicated "brute force" error injection VIP 'bitvis_vip_error_injection' in UVVM.
UVVM has a pre-defined methodology for handling protocol aware error injection in a structured way.
Note that only some VVCs and BFMs currently support error injection. The principles shown for these VVCs and BFMs may be applied directly also for user defined VIP.

7.1 UVVM error injection principles

Error injection may be applied randomly, with no limitations. For UVVM however, we recommend the following approach:

1. No randomisation of behaviour inside BFMs when this could affect the DUT behaviour or output. (and a monitor would be required to check the actual DUT stimuli.)
   Hence BFM procedures should only be called with parameters explicitly defining the interface behaviour (from the BFM side). Thus no parity error randomisation inside.
   The only exception is for behaviour that should not affect the DUT. Thus the position of a data bit error could be randomised inside the BFM.
2. It is recommended that more advanced VVC include randomisation – in order to distribute this away from the test sequencer and increase the re-use value of a VVC.
   Thus a VVC may be told to apply say 10% parity errors for a UART_VVC transmission into the DUT. In that case the VVC will randomly – with a 10% probability - inject a parity error into the DUT. As the VVC uses a BFM to handle the actual interface/protocol, this means that in 10% of the BFM transmit calls the VVC will request a parity error to be injected.

7.2 Error injection in BFMs

In order to simplify the specification of which errors to inject, the complete error injection specification is given as a sub-record inside the BFM configuration.
E.g. inside the UART BFM configuration the following sub-record is defined – with fields specifying the error injection details (Details given in the UART VIP doc)

- error_injection (fixed name, but type will differ)
  - parity_bit_error (boolean)
  - stop_bit_error (boolean)

In order to initiate error injection, the BFM config record must be modified and included in the BFM procedure call

7.3 Error injection in VVCs

In order to simplify the specification of which errors to inject, the complete error injection specification is given as a sub-record inside the VVC configuration (Note: not the BFM config)
E.g. inside the UART VVC configuration the following sub-record is defined – with fields specifying the error injection details (Details given in the UART VIP doc)

- error_injection
  - parity_bit_error_prob (real between 0.0 and 1.0)
  - stop_bit_error_prob (real between 0.0 and 1.0)

In order to initiate error injection, the VVC config record must be assigned the wanted values via the VVC configuration shared variable. (see ch 4)
Note that the Error injection sub-record inside the VVC configuration will override that of the BFM configuration.
Any compound or more advanced transactions may of course also request error injection directly or indirectly via the VVC command itself.

7.4 Naming and type usage

The error injection sub-record will be VVC and BFM dedicated, and thus any names and types may be used, and even sub-records under 'error_injection' is required. The VVC and BFM error injection records may differ or be the same. The only requirement is that readability is prioritised. Values should be checked against legal ranges or values.

8 Randomisation

UVVM provides functions and procedures for simple generation of random numbers (real, integer, time) and vectors. This is described in detail in the documentation for the UVVM Utility Library. Other libraries for generation of random data may also be used seamlessly with UVVM.
Note that only some VVCs currently include randomisation (e.g. UART TX VVC and SBI VVC.) The principles shown for these VVCs may be applied directly also for user defined VIP.

8.1 UVVM VIP randomisation principles

Randomisation may of course be applied with no limitations in a UVVM based testbench. For UVVM VIP however, we recommend the same general approach as for error injection randomisations:

1. No randomisation of data inside BFMs as this would affect the DUT behaviour or output. (and a monitor would be required to check the actual DUT stimuli.)
   Hence BFM procedures should only be called with explicit data.
2. It is recommended that more advanced VVCs include functionality for randomisation of data – in order to distribute this away from the test sequencer and increase the re-use value of a VVC. Thus, a VVC may be told to apply random data, in which case the VVC will randomly generate data according to a given profile (e.g. uniform) and provide that data to the interface via the BFM call. The profile and constraints will depend on the needs and the VVC implementation

8.2 Data randomisation in BFMs

There is no data randomisation inside a normal BFM, for the reason given above.

8.3 Data randomisation in VVCs

A VVC may be commanded to generate constrained random data, where data in this sense could also be addresses, lengths, etc. Typically such commands would allow flexibility for the number of accesses and other important aspects – like scoreboards, common buffers, files, etc.
A few randomisation profiles have been predefined both as typical use cases and as examples for future extensions, when needed.
The profile names are defined in the type t_randomisation, which is declared in the adaptations package to allow users to add more profiles.

NA	Not applicable (To be used in a record where the field is present, but no randomisation wanted)
RANDOM	Uniform distribution
RANDOM_FAVOUR_EDGES	Significantly more edge cases, where “edge” differs between various interfaces. E.g. UART: Cover patterns like 01111111, 00000000, 11111111, 11111110, 01010101, 10101010,
<user-defined>	

8.4 VVC Command Syntax

See chapter 12 for parameter sequence and options.

9 Functional Coverage

External libraries for handling functional coverage be used seamlessly with UVVM. The only exception here is that log and alert messages will go to a different file set.
Note that only some VVCs currently include functional coverage (e.g. UART RX VVC.) The principles shown for these VVCs may be applied directly also for user defined VIP.

9.1 UVVM VIP Functional Coverage principles

Functional coverage may of course be applied with no limitations in a UVVM based testbench. For UVVM VVCs however, we recommend to structure this as follows:

1. For advanced testbenches using functional coverage, it is recommended to include as much of this functionality in the VVCs as possible – in order to distribute this away from the test sequencer and increase the re-use value of a VVC.
2. A VVC may be told to apply functional coverage either on the master/transmitter side or the slave/receiver side – or both, all depending on how the functional coverage is applied and utilised inside the testbench. In any case the VVC will receive a command to start or modify a predefined functional coverage task.
   (An example on how functional coverage is applied and used to terminate a test section when sufficient functional coverage is reached in the receiver, is shown in the UART VVC.)

9.2 Functional Coverage in VVCs

A VVC may be commanded to just gather functional coverage and potentially flag when required coverage is achieved, or to keep on doing something (like receiving or transmitting data) until the requested coverage is reached.
A few functional coverage profiles have been predefined both as typical use cases and as examples for future extensions, when needed.
The profile names are defined in the type t_coverage, which is declared in the adaptations package to allow users to add more profiles.

NA	Not applicable (To be used in a record where the field is present, but no functional coverage wanted)
COVERAGE_FULL	Full coverage meaning all possible permutations
COVERAGE_EDGES	A predefined set of important edge cases – for instance like the RANDOM_FAVOUR_EDGES in the previous chapter.
<user-defined>	

9.3 VVC Command Syntax

See chapter 12 for parameter sequence and options.

10 Testbench Data routing

Direct transaction transfer is providing a mechanism for passively routing source data (data entered into the DUT) out of the VVCs to other parts of the testbench. This data routing is passive in the sense that the transaction data are just provided as a global signal – for anyone to read. This is covered in chapter 0.
There is however also a need for routing data actively inside the testbench, where routing means fetching from or sending to predefined sources and destinations.

10.1 To/from Buffer

UVVM has a global buffer that is divided into multiple smaller buffers that may be indexed and accessed from anywhere in the testbench.
This functionality is described in detail in 'uvm_vvc_framework/doc/UVVM_FIFO_Collection_QuickRef.pdf'.
VVC commands requesting sourcing data from or sending data to these buffers use parameter TO_BUFFER or FROM_BUFFER, followed by the buffer index.

10.2 To scoreboard

Scoreboards may be used anywhere inside the testbench, but for UVVM the following is recommended:

1. Use Scoreboards only on the destination side of the testbench, i.e. where data is received or fetched out of the DUT.
   I.e. for a UART on the DUT UART TX side (= UART_VVC RX side)
2. Every VVC may be connected to one single Scoreboard
3. The Scoreboard instance number should be the same as the VVC instance number
4. When using VVCs make sure the VVC passes the received data to its scoreboard. Do not check the data in the VVC.
   I.e. for a UART_VVC use the receive-command (and not the expect-command) to forward received data to the scoreboard

VVC commands requesting sending data to the scoreboard use parameter TO_SB.

10.3 Data routing options

The profile names are defined in the type t_data_routing, which is defined in UVVM Util types_pkg.

NA	Not applicable (To be used in a record where the field is present, but no data routing wanted)
TO_SB	Data is passed on to the scoreboard for the given VVC
FROM_BUFFER	Data is sourced from the UVVM global buffer
TO_BUFFER	Data is also sent to the UVVM global buffer
TO_FILE	TBD – Not yet implemented (Do not use – as this may change)
FROM_FILE	TBD – Not yet implemented (Do not use – as this may change)
<user-defined>	

10.4 VVC Command Syntax

See chapter 12 for parameter sequence and options.

11 Controlling property checkers

A major VVC advantage is that lots of additional very useful functionality may be added inside the VVC entity, meaning that all the verification support for a given interface can be encapsulated inside a single VHDL entity. A major advantage of UVVM is that adding additional functionality and controlling it from the test sequencers is really simple. One very useful additional functionality is property checkers, and some typical examples of this could be to check the minimum allowed bit period, the minimum inter-packet gap, back-to-back restrictions, etc., or in general to check a given requirement continuously – especially when this is easier to do outside the BFM – for instance in a dedicated checker process.

A dedicated checker process would typically just wait for a trigger condition on the interface (like a UART data bit changing its level), then wait again for a next trigger (the next data bit), and then check that the time between the two changes is not less than the minimum allowed bit period. This check could then be repeated forever. It is however recommended that the check could be turned on and off for more flexibility.

11.1 Property check configuration

In UVVM turning checkers on and off is controlled by the VVC configuration (chapter 4), and often additional control of the checker behaviour is also required. Thus, it is recommended to include the checker control for each individual check in a dedicated sub-record. An example on this (for the UART VVC) is shown below. See UART_VVC RX for implementation.

```
.bit_rate_checker                       -- Sub-record containing all control of the property checker behaviour
    .enable         boolean             -- Enables or disables the complete bit rate checker.
    .min_period     time
    .alert_level    t_alert_level
```

For this example the bit rate checker inside the UART_VVC RX will trigger on changes on the DUT TX and execute the check if enable is TRUE.

11.2 Setting up the configuration

The bit rate checker configuration may be changed directly from the sequencer via the shared variable VVC configuration.

12 VVC parameters and sequence for Randomisation, Functional Coverage, Sources and Destinations

In order to assure a common syntax and understanding for the various VVC commands controlling these features, the sequence and type of parameters have been defined as follows:

Parameter sequence	Preceding command part	[Number of repetitions]	Randomness/Coverage type	Data routing type	[Data routing index]
Example a	uart_transmit(UART_VVCT,1,TX,	4	RANDOM_FAVOUR_EDGES	TO_BUFFER	5
Example b	uart_receive(UART_VVCT,1,RX,		COVERAGE_FULL	TO_SB	

Example a means: make 4 transactions with random data (using predefined profile RANDOM_FAVOUR_EDGES) and send the data also to BUFFER 5
(e.g. uart_transmit(UART_VVCT,1,TX, 4, RANDOM_FAVOUR_EDGES, TO_BUFFER, C_UART_BUFFER, “my message”);
Example b means: keep on receiving data until given coverage is reached and send the received data also to the local Scoreboard

Exactly what variants will be available for each VVC is up to the VVC designer, but this gives the sequence and the options.

13 Multiple Central Sequencers

A structured test environment is important and we recommend the use of a test harness to instantiate VVCs, DUT, clock generator and so forth. The testbench may consist of one or more test sequencers which are used to control the complete testbench architecture with any number of VVCs, although for a better testbench overview we recommend to have a single central test sequencer only.

14 Monitors

Monitors could be great to check the interface accesses to a DUT – to report the transaction to the testbench – with all relevant info like operation (write, read, transmit, …), data, address, etc. This information may be critical in order to understand the operation of the DUT and its expected outputs. A monitor is not a protocol checker, but may of course check various properties of an interface/protocol. A typical Monitor will however only provide the relevant basic information and leave more advance interpretation to other parts of a testbench. For simple protocols like the UART, UVVM also includes basic error checking in the Monitor – as this happens at a very low level. For more advanced protocols it would make sense to just pass on the low level info to a higher level checker. The reason for making a dedicated monitor rather than leaving that to the testbench model is to achieve a better testbench structure and more efficient reuse.

It should however be mentioned that implementing Direct Transaction Transfer (ch. 5) inside a VVC significantly reduces the need for a dedicated monitor, as the VVC will then be able to pass the complete transaction information on to for instance a model inside the Testbench.

14.1 Transfer of Monitor information to the testbench

The mechanism for passing the monitor deduced transaction out of the monitor is almost exactly the same as for passing transaction info out of a VVC – as described in chapter 5. The only difference is that the monitor can only provide parts of what the VVC can provide – as follows compared to the tables given in chapter 5.

- No monitor can provide info about compound transactions
- For a split transaction protocol like Avalon – only the sub-transactions could be provided (which could be analysed at the higher level to provide Base transactions)
- A monitor cannot provide meta data like command index or command message.

As the monitor does not know what to expect at the beginning of a transaction the following field limitations apply:

- Operation: Can only be known some time after the start of the transaction. Will be set when the result of the transaction is known (otherwise NO_OPERATION) **** IN_PROGRESS????
- Transaction_status: Will be set to FAILED or SUCCEEDED as soon as the result is 100% given. Prior to that – during the transaction : IN_PROGRESS. FAILED/SUCCEEDED will remain for the transaction_display_time given inside the monitor configuration record, or until the next transaction FAILED or SUCCEEDED.

An example of a complete monitor is shown in the UART VIP directory.

14.2 Transaction info transfer signals

The Transaction info provided out of a Monitor uses a global signal. This signal and all related VHDL types are defined in transaction_pkg.
- Monitor DTT signal : global_<protocol-name>_monitor_transaction, e.g. global_uart_monitor_transaction(instance number, channel)

15 Compile scripts

In the script folder in the root directory the compile_all.do compiles all UVVM components. This script may be called with one to three input arguments. The first input argument is the directory of the script folder at the root directory from the working directory. The second input argument is the target directory of the compiled libraries, by default every library is compiled in a sim folder in the corresponding components directory. The third input argument is the directory to a custom component list in .txt format. The script will only compile the components listed in that file. By default, the script uses the file component_list.txt located in uvvm/script. This file can be modified so that only some components are compiled.

Example: do uvvm/script/compile_all.do uvvm/script

There are also compile scripts for all UVVM components located in the script folder of each UVVM component. These scripts can be called with two input arguments. The first input argument is the directory of the component folder from the working directory. The second input argument is the target directory of the compiled library, default is the sim folder in the respective component.

Example: do uvvm/uvvm_util/script/compile_src.do uvvm/uvvm_util

16 Scope of verbosity control

Message IDs are used for verbosity control in many of the procedures and functions in UVVM, as well as log messages and checks in VVCs, BFMs and Scoreboards.
Note that VVCs and Scoreboards comes with dedicated message ID panels and are not affected by the global message ID panel, but accessed by addressing targeting VVC or Scoreboard and, if applicable, instance number or with a broadcast.

E.g.

```
-- Global message ID panel. Does not apply to VVCs or Scoreboards, as they have their own local message ID panel
disable_log_msg(ALL_MESSAGES);
enable_log_msg(ID_SEQUENCER);

-- VVC message ID panel
disable_log_msg(VVC_BROADCAST, ALL_MESSAGES); -- broadcast to all VVCs and instances
enable_log_msg(I2C_VVCT, C_VVC_INSTANCE_1, ID_BFM_WAIT); -- I2C VVC instance 1
enable_log_msg(I2C_VVTC, C_VVC_INSTANCE_2, ID_BFM_WAIT); -- I2C VVC instance 2

-- Scoreboard message ID panel
shared variable sb_under_test : record_sb_pkg.t_generic_sb;
…
sb_under_test.disable_log_msg(ALL_INSTANCES, ID_CTRL); -- broadcast to all SB instances
sb_under_test.enable_log_msg(C_SB_INSTANCE_1, ID_DATA); -- SB instance 1
```

A subset of message IDs is listed in UVVM Utility Library QR, section 1.10.