

# VVC Implementation Guide

This guide is meant for users that want to make their own VVC.  
Users that only write test cases that are using the VVCs do NOT need to read this guide.

Making test cases using available VVCs is very easy.  
Implementing new VVCs is slightly more complex, but fast, safe and efficient once you understand the VVC mechanisms.



## Understanding and Modifying a VVC

This guide goes through all relevant files needed to make a complete VVC.  
The intention is to allow a VVC implementer to go through file by file and understand and modify as needed.  
All code objects and functionality given in the VVC and UVVM files are considered mandatory unless otherwise noted.

## To implement your own VVC

Prerequisites:

1. Go through 'The\_critically\_missing\_VHDL\_TB\_feature.ppsx' - for a presentation on cycle related corner cases and the need for a far more structured verification approach.
2. Read the 'VVC\_Framework\_manual.pdf' to understand the basic concepts, the communication and handshake between the central sequencer and the VVCs.
3. If your VVC is going to access a DUT interface, you need to have a BFM (Bus Functional Model) for that interface – independent of whether you are going to make a VVC.

Implementing your new VVC:

1. Use the script `vvm_generator` located in the `uvvm_vvc_framework/script/vvc_generator/` to generate a new VVC. Notice that the name length is limited by `C_LOG_SCOPE_WIDTH` (default =20) in `uvvm_util/src/adaptations_pkg`.
2. Then see comments in the code for where to make required changes.
3. If the new VVC uses multiple channels other than TX and RX, modify the `t_channel` type under UVVM-Util `adaptations_pkg`.
4. See this guide for an explanation to all the various sections you need to evaluate or modify – file by file.



## 1 Dependent and independent source

One of the key concepts of the UVVM VVC Framework is the compilation strategy, and how some packages in the UVVM VVC Framework directory are compiled into each of the individual VVC libraries. To avoid confusion about this for future VVC designers, the VVC dependent and VVC independent sources have been marked and split into two source directories. The target dependent source, also known as packages that are compiled into each of the individual VVCs, are placed in the *src\_target\_dependent/* folder. These files are also prefixed with “*td\_\**” for “target dependent”.

The target independent files are compiled into the *uvvm\_vvc\_framework* library. These files are placed in the *src/* folder and prefixed with “*ti\_\**” for “target independent”.

For examples of how the compile order should be handled, please see the example VIPs' QuickRef and Modelsim compile scripts.

## 2 <name>\_vvc.vhd

### 2.1 For single channel VVCs

Code Section	Description
<b>Entity</b>	<p>GC_INSTANCE_IDX: Needed in case there are multiple instances of a given VVC. (E.g. DUT with 2 &lt;VVC-NAME&gt;S). Default is 1, but any natural type is ok.</p> <p>GC_&lt;name&gt;_CONFIG: Recommended. Allows predefined BFM behaviour to be set up for every VVC.</p> <p>GC_CMD_QUEUE_*: Needed to limit the queue size and to generate a warning if more elements in the queue than ever expected.</p> <p>Other generics: Optional and VVC dependent. These generics could for example contain widths of BFM signals.</p> <p>The interface to the DUT and any other needed I/O. The examples show DUT interfaces as single signals, records and records of records. This is optional.</p>
<b>Declarations</b>	<p>C_SCOPE: Used for logs and alerts. C_VVC_NAME is defined in the VVC 'vvc_methods_pkg.vhd'</p> <p>C_VVC_LABELS: A record of constants, e.g. name and channel, used in multiple procedures.</p> <p>Various status signals used as flags between the processes</p> <p>Command termination record. (Fields: <i>set</i>, <i>reset</i>, <i>is_active</i>. Where <i>set</i> and <i>reset</i> signal fields are used to toggle <i>is_active</i>. Used as inter process flags.</p> <p><i>command_queue</i> is the queue of commands to be executed in sequence towards the DUT.</p> <p>The aliases are defined to allow common and simplified names.</p>
<b>Constructor</b>	<p>The constructor is run once only – immediately when starting the simulation. The procedure:</p> <ul style="list-style-type: none"> <li>- Initialises VVC with BFM config and the queue with queue name</li> <li>- Allows constructor log for VVC info (using ID_CONSTRUCTOR), and VVC Queue info (using ID_CONSTRUCTOR_SUB)</li> </ul> <p>The procedure will report alerts with severity TB_FAILURE if one of the following occurs:</p> <ul style="list-style-type: none"> <li>- The instantiated VVC index is higher or equal to the maximum allowed number of VVC instances, given by C_MAX_VVC_INSTANCE_NUM in UVVM-Util 'adaptations_pkg.vhd'</li> <li>- UVVM has not been initialized</li> </ul>

## Command Interpreter

Waits for commands from the central test sequencer distributed to this VVC, then puts the command on the queue for execution or immediately performs the required action – depending on command type. Afterwards, it acknowledges the command and waits for the next command from the sequencer.

### Step 0

Initialize\_interpreter():

- Initialises parameters to default passive/initial values (e.g. terminate\_current\_cmd.set := '0')

### Step 1

await\_cmd\_from\_sequencer():

- Waits for a command from the central sequencer. Continues on matching VVC, instance index, name and channel
- Log at start using ID\_CMD\_INTERPRETER\_WAIT and at the end using ID\_CMD\_INTERPRETER.
- Will only accept exact matches of instance index and name, and either the correct address or "ALL\_CHANNELS"

### Step 2a (Only if command type is QUEUED)

put\_command\_on\_queue():

- Puts the received command on the VVC queue (for later retrieval by the Command Executor)

### Step 2b (Only if command type is IMMEDIATE)

Execute the requested command/operation.

- For the VVC methods these procedures will correspond to the UVVM methods, but prepended with "interpreter\_", e.g. "interpreter\_await\_completion". These UVVM methods are documented in the Common\_VVC\_methods.pdf document. Other commands are documented in their respective QuickRefs.
- format\_command\_idx(): (ti\_vvc\_framework\_support\_pkg) Converts the command index to string, enclosed by C\_CMD\_IDX\_PREFIX and C\_CMD\_IDX\_SUFFIX (found in UVVM-Util adaptations\_pkg).

### Step 3

acknowledge\_cmd():

- Acknowledges the command from the sequencer by driving global\_vvc\_ack signal (to '1') for 1 delta cycle, then setting it back to 'Z'. This lets the central sequencer know that it can continue execution.

## Command Executor

Fetches commands from the command queue – if any. Then executes the command and fetches or waits for the next command in the command queue

### Step 0

initialize\_executor():

- Initialises parameters to default passive/initial values (e.g. terminate\_current\_cmd.reset := '0')

### Step 1

fetch\_command\_and\_prepare\_executor():

- Fetches a command from the queue (waits until available if needed).
- Sets relevant flag parameters.
- Log command using ID\_CMD\_EXECUTOR (or Log using ID\_CMD\_EXECUTOR\_WAIT if queue is empty)

Set transaction information for wave-view.

- transaction\_info\_for\_waveview (from vvc\_methods\_pkg.vhd) is a shared variable intended for use in a wave-view – to yield a better overview of transaction info. Setting this information is optional. The pad\_string and to\_string procedures are documented in the UVVM-Util QuickRef.

Insert inter BFM delay if requested.

- insert\_inter\_bfm\_delay\_if\_requested(): Inserts either start-to-start or finish-to-start delay between BFM accesses if this is set in the inter\_bfm\_delay parameter in 'vvc\_config'. Logs information using ID\_CMD\_EXECUTOR.
- If the command currently being processed by the executor is a BFM access, a timestamp will be stored in v\_timestamp\_start\_of\_current\_bfm\_access.

### Step 2

---

Executes a command depending on the requested command/operation.

- *terminate\_current\_cmd* is only checked inside operations that require multiple BFM accesses – like for instance a POLL\_UNTIL command.
- *store\_result()* is executed for any BFM, where it makes sense for you to store the result of a BFM access. In our example for SBI we think it only makes sense for 'READ'.
- Logging as defined by your BFM.
- Transaction info can be stored in the *transaction\_info\_for\_waveview* struct for each command type, but this is optional.

Update the BFM access timestamps if this was a BFM access.

- *v\_timestamp\_of\_last\_bfm\_access* is set to *now*
- *v\_timestamp\_start\_of\_last\_bfm\_access* is set to *v\_timestamp\_start\_of\_current\_bfm\_access*

The *terminate\_current\_cmd* flag is reset if it has been active.

Update the *last\_cmd\_idx\_executed* variable with the current command index, *v\_cmd.cmd\_idx*.

---

## Command Terminator

The command terminator concurrent procedure sets the *is\_active* flag based on the *set* and *reset* flags.

---

## 2.2 Additional for multi-channel VVCs

Please note that we strongly recommend implementing the VVCs such that each leaf VVC handles one independent DUT communication thread (here: 'Channel'). No more; no less. This allows a single command queue and a single executor handling DUT communication. (Additional processes to handle other characteristics is fine. E.g., a parallel bit-rate check thread.)

Note that SBI\_VVC must handle both read and write accesses, but never simultaneously and always in the given order.

Multi-channel VVCs may be implemented in many different ways – depending on your preferences and priorities. Some examples:

### 1. As unique VVC implementation

Unique VVCs may be used in order to omit the channel input, e.g. UART RX VVC and UART TX VVC. UART TX VVC would only contain TX specific BFM procedures, while UART RX VVC would only contain RX specific BFM procedures. With this approach the test bench sequencer calls would look like e.g. (assuming both VVCs in this pair are set to instance index 1):

- a. *uart\_transmit(UART\_TX\_VVCT,1,...)*
- b. *uart\_receive(UART\_RX\_VVCT,1,...)*

### 2. As shared VVC implementation with usage restricted by user, and multiple VVC instances

A combined VVC with different VVC instances for different channels e.g. RX and TX. The TX instance could e.g. be instance 1, and the RX instance could be e.g. instance 2. Using this UART VVC with this implementation would look like:

- a. *uart\_transmit(UART\_VVCT,1,...)*
- b. *uart\_receive(UART\_VVCT,2,...)*

### 3. As shared VVC implementation with GC\_CHANNEL generic input

A combined VVC with the same combined VVC implementation, but separate instances for different channels e.g. RX and TX (both functionalities inside the same leaf VVC). The downside of this implementation is that it would be possible to call TX BFM procedures when calling the RX VVC channel. Using this UART VVC would look like:

- a. *uart\_transmit(UART\_VVCT,TX,1,...)*
- b. *uart\_receive(UART\_VVCT,RX,1,...)*

### 4. As unique VVC implementation with GC\_CHANNEL generic input

This approach uses unique VVC implementations for each channel, e.g. in `uart_rx_vvc.vhd` and `uart_tx_vvc.vhd`, but they both share the VVC target parameter, `UART_VVCT`. They both use the `GC_CHANNEL` generic input to specify their channel, i.e. TX or RX. This is similar to the method described in 3., but with restrictions that ensure that e.g. the UART TX VVC can't use the UART RX BFM procedures. The included `bitvis_vip_uart` example is implemented with this method. Using this UART VVC would look like:

- a. `uart_transmit(UART_VVCT,TX,1,...)`
- b. `uart_receive(UART_VVCT,RX,1,...)`

When using multiple leaf VVCs it is recommended to use a wrapper architecture to encapsulate the channels. This way, it is possible to instantiate a single VVC rather than each VVC channel individually. For more information about the wrapper architecture, see the `uart_vvc.vhd` example in the `bitvis_vip_uart/src/` directory.

### 3 vvc\_cmd\_pkg.vhd

Section	Comment
<b>t_operation</b>	Contains all UVVM common operations, e.g. <code>AWAIT_COMPLETION</code> and <code>ENABLE_LOG_MSG</code> , in addition to the VVC specific operations such as e.g. <code>WRITE</code> and <code>READ</code> . The VVC specific will have to be evaluated and potentially replaced when implementing a new VVC. The <code>t_operation</code> type is used when relaying commands from the sequencer to the VVC. The <code>t_operation</code> type also has its own <code>to_string()</code> function in this package.
<b>t_vvc_cmd_record</b>	Record type used for relaying a command from the test bench sequencer to the VVC. The record contains fields needed in the common UVVM procedures (listed under the "Common UVVM fields" comment), and VVC specific fields needed to relay data to the VVC executor. The VVC specific data fields should contain any data fields that the BFM procedures might need, e.g. data, address, timeouts etc. There is also a default for this type called <code>C_VVC_CMD_DEFAULT</code> in this package.
<b>Constants</b>	The <code>vvc_cmd_pkg.vhd</code> should contain constants that needs to be set for the entire VVC. In most VVCs this will include the <code>C_VVC_CMD_STRING_MAX_LENGTH</code> which determines the maximum size of msg variables in the VVC. It is also a good idea to declare constants for maximum VVC data/address bus sizes here. It will be possible to construct VVCs with bus sizes up to and including the sizes declared here.
<b>Shared Variables</b>	The <code>shared_vvc_cmd</code> shared variable (type <code>t_vvc_cmd_record</code> ) is used for relaying commands between sequencer methods and the VVC. It is default set to <code>C_VVC_CMD_DEFAULT</code> , which is also declared in this file.

## 4 vvc\_methods\_pkg.vhd

Section	Comment
Constants and aliases	The vvc_methods_pkg contain constants for the VVC name, e.g. "<NAME>_VVC". There are also aliases created to make the code more readable.
<NAME>_VVCT	The <NAME>_VVCT signal (e.g. SBI_VVCT) is the VVC target record signal. The target type <i>t_vvc_target_record</i> is a record that contains the parameters needed to trigger a VVC, and to identify the correct target of a VVC command.
t_vvc_config	<p>This type contains the needed configuration for setting up the VVC and BFM. In Bitvis VVCs the BFM configuration is encapsulated in a bfm_config record, of type <i>t_bfm_name_bfm_config</i>. This record is placed in this file and compiled into each VVC since the VVC/BFM configuration will differ for each VVC. Record contents:</p> <ul style="list-style-type: none"> <li>- <i>inter_bfm_delay</i>: A record containing the potential inter-bfm delay specifications, e.g. if BFM accesses shall be separated with a given time</li> <li>- <i>cmd_queue_count_*</i>: Command queue specifications</li> <li>- <i>msg_id_panel</i>: The ID panel that the VVC shall use</li> <li>- <i>bfm_config</i>: A record containing all settings for the BFM, e.g. clock periods, message IDs etc.</li> </ul> <p>A constant <i>C_&lt;name&gt;_VVC_CONFIG_DEFAULT</i> is defined for this type to use as default value. A shared variable array of <i>t_vvc_config</i> <i>shared_&lt;name&gt;_vvc_config</i> is declared and all elements are set to the default value.</p>
t_vvc_status	<p>The optional status record is created in order for the test bench sequencer to have access to the status of the VVC. The status record can contain anything that is relevant for the outside, and it is recommended to have at least these three fields:</p> <ul style="list-style-type: none"> <li>- <i>current_cmd_idx</i>: The current command index being processed in the executor</li> <li>- <i>previous_cmd_idx</i>: The previous command index being processed in the executor</li> <li>- <i>pending_cmd_idx</i>: The number of pending commands to be processed by the executor</li> </ul> <p>A constant <i>C_VVC_STATUS_DEFAULT</i> is defined for this type to use as default value. A shared variable array of <i>t_vvc_status</i> <i>shared_&lt;name&gt;_vvc_status</i> is declared and all elements are set to the default value.</p>
t_transaction_info_for_waveview	<p>The <i>t_transaction_info_for_waveview</i> type is an optional status record to be used in the wave-view. This record should be modified to suit the BFM fields, but it can also contain the VVC field <i>t_operation</i>, which can be updated with the VVC operation currently being processed by the executor. The <i>transaction_info_for_waveview</i> is meant as a way of improving the readability of wave-views.</p> <p>A constant <i>C_TRANSACTION_INFO_FOR_WAVEVIEW_DEFAULT</i> is defined for this type to use as default value. A shared variable of <i>t_transaction_info_for_waveview</i> <i>t_&lt;name&gt;_transaction_info_for_waveview</i> is declared and all elements are set to the default value.</p>
VVC Dedicated Methods	<p>The vvc_methods_pkg.vhd file also contains the VVC procedures that are called from the test bench sequencer. These procedures should reflect the procedures in the BFM, e.g. <i>&lt;name&gt;_write</i> or <i>&lt;name&gt;_receive</i>. The parameters of these procedures are mostly up to the user, but it is recommended that the BFM arguments that are rarely altered be placed into the <i>bfm_config</i> parameter, while the parameters that changes often are used as input arguments.</p> <p>Since these VVC methods are reused for all instances of this VVC, it is necessary with some extra parameters in order to specify which VVC instance to forward the call to. This is done with the first two (or three) parameters:</p>

- 
- *signal VVCT : inout t\_vvc\_target\_record;*
  - *constant vvc\_instance\_idx : in integer;*
  - *constant channel : in t\_channel; -- Only if the VVC is multi-channel.*

The method bodies are quite similar for all VVC commands:

1. First, the shared\_vvc\_cmd record is set to its default value, resetting the data from any potential previous command.
2. The general VVC fields (e.g. name and instance index) are set using the UVVM method `set_general_target_and_command_fields()`
3. The VVC specific fields are set in the shared\_vvc\_cmd shared variable. This means e.g. address and data fields.
4. The command is sent to all VVCs using the UVVM method `send_command_to_vvc(VVCT)`

All VVC instances and channels of this type receive the command, but only the VVC with the correct instance index, channel and name will accept it and acknowledge it.

---

## 5 vvc\_context.vhd

This file contains all the necessary packages that are used when simulating a VVC. When adding a VVC to a testbench, the user only needs to add the following lines in the header:

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

The generated file from the `vvc_generator` script has the following as default:

- **transaction\_pkg.all**
- **vvc\_methods\_pkg.all**
- **td\_vvc\_framework\_common\_methods\_pkg.all**
- **<name>\_bfm\_pkg.t\_<name>\_if** → BFM interface type (remove if not used)
- **<name>\_bfm\_pkg.t\_<name>\_bfm\_config** → BFM configuration type
- **<name>\_bfm\_pkg.C\_<name>\_CONFIG\_DEFAULT** → BFM default configuration constant

Additional types or constants can be added if needed.

## 6 BFM prerequisites

There are no firm restrictions of how to implement the BFM in order for the VVC to function, but if the VVC generated with the `vvc_generator` script is to work out of the box, it is necessary to have some components in the BFM:

- The BFM needs to be called `<name>_bfm_pkg.vhd`. If this is not the case, the package use clauses in each of the VVC files needs to be altered.
- The BFM needs to contain a `bfm_config` record type with an associated default constant. The generated VVC file assumes that this bfm config type is called `t_<name>_bfm_config` and the constant is called `C_<NAME>_BFM_CONFIG_DEFAULT`. In order to support the delay operation in the VVC executor the BFM config type will also need to have a parameter `clock_period`. If this is not needed, the "INSERT\_DELAY" case in the generated VVC can be removed.

A BFM skeleton that contains the necessary structure is created by the `vvc_generator` script, and can be used as a base for a BFM that includes the necessary structure for the VVC to work out of the box.

## 7 UVVM Framework Packages

### 7.1 td\_target\_support\_pkg.vhd

The UVVM VVC dedicated support package contains VVC support that is common for all VVCs, but needs to be compiled specifically into each of the VVC libraries.

Section	Comment
<b>Target record</b>	<p>The target record type, <i>t_vvc_target_record</i>, is used to target a VVC command to a specific VVC implementation. This is needed since many of the UVVM common commands are shared between all VVCs, e.g. <i>await_completion()</i> which is compiled into each VVC library.</p> <p>For a sequencer with two VVCs, A and B, there must be a way of determining if <i>await_completion</i> is to be executed in VVC A or VVC B. To resolve this, each VVC has a signal in their <i>vvc_methods_pkg</i> that is compiled into their own library. For VVC A and B this signal will be called <i>A_VVCT</i> and <i>B_VVCT</i>. When <i>await_completion(A_VVCT,...)</i> is called from the sequencer, the compiler will understand that this <i>await_completion</i> is called with target type <i>library_a.t_vvc_target_record</i>, which only complies with the <i>await_completion</i> procedure in the VVC A library.</p> <p><i>td_target_support_pkg</i> also contains a default value for the <i>t_vvc_target_record</i> type, and a function <i>set_vvc_target_defaults</i> for setting the VVC target based on the VVC name.</p>
<b>String methods</b>	<p>The package contains two string methods:</p> <ul style="list-style-type: none"> <li>- <i>to_string()</i>: This function converts a <i>t_vvc_target_record</i>, <i>vvc_instance</i> and <i>vvc_channel</i> into a string</li> <li>- <i>format_command_idx()</i>: Function which encapsulates a command record index.</li> </ul>
<b>send_command_to_vvc</b>	<p>Sends command to VVC and waits for ACK or timeout</p> <ul style="list-style-type: none"> <li>- Logs with <i>ID_UVVM_SEND_CMD</i> when sending to VVC</li> <li>- Logs with <i>ID_UVVM_CMD_ACK</i> when ACK or timeout occurs</li> </ul>
<b>Setting the command field</b>	<p>Sets target index and channel, and updates <i>shared_vvc_cmd</i> which is used to transport VVC commands from the central test bench sequencer to VVC.</p>



## 7.2 td\_vvc\_entity\_support\_pkg.vhd

The VVC support package contains procedures that are compiled into and used in the VVC. This includes initializers for the executor and interpreter, and the interpreter procedures called *interpreter\_\**, e.g. *interpreter\_await\_completion*. For more information about the *interpreter\_\** procedures, please see the Common\_VVC\_Methods under doc/. For more information about the other methods in this package, see the <name>\_vvc.vhd section in this document.

In addition to the procedures, the td\_vvc\_entity\_support\_pkg also contains types for VVC labels and executor results. The result array is also defined and its shared variable is instantiated in this package.

## 8 Additional Documentation

Additional documentation about UVVM and its features can be found under “uvvm\_vvc\_framework/doc/”.

### INTELLECTUAL PROPERTY

Disclaimer: This IP and any part thereof are provided “as is”, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with this IP.