**Basic UVM Testbench:**

The basic UVM testbench architecture is shown in figure 1. This test bench has been used to verify the basic functions of CAN protocol [3] or the Ethernet protocol. Testbench has the top module, test, environment, and the sequence files [15]. The details of the above-mentioned components are as follows.

**Top:**

The 16MHz clock is generated in top module. The reset generation is also done in the top module based on the specification whether the reset is active high reset or active low reset. The Design Under Verification (DUV) can be instantiated in the top. The interface is set in the configuration database and can be accessed anywhere in the test bench or the UVC by calling the get function from the configuration database. The get function fetch the interface and assigns it locally from the database. The run\_test () built-in function starts the uvm\_phases used in the components.

**Basic Test:**

Basic\_test is extended from the uvm\_test which is the built-in UVM class. UVM treats the uvm\_test as component, so the component is registered in the factory using the macro `uvm\_component\_utils. Env, agent\_cfg and sequence gets created using the create () function. Here the mode is set to ‘0’ to select the CAN UVC and the mode is set to ‘1’ to select the ethernet UVC. The mode variable is available in the agent configuration. The agent configuration is set in the configuration database after the specific configuration is set for the UVC. The object creation and setting in the database are done in the build\_phases. The run\_phases starts and drop the objections. The run\_phase of the test starts the specific sequence in the appropriate sequencer.

**Basic Env:**

Basic\_env is extended from the uvm\_env which is the built-in UVM class. UVM treats the uvm\_env as component, so the component is registered in the factory using the macro `uvm\_component\_utils. Basic\_agent of the UVC gets created using the create () function.

**Basic Sequence:**

Basic\_sequence is extended from the uvm\_sequence which is the built-in parameterized UVM class. UVM treats the uvm\_sequence as object, so the object is registered in the factory using the macro `uvm\_object\_utils. Basic\_txn of the UVC gets created using the create () function. The transaction got randomized and sent to the driver via sequencer. This operation is repeated for five times which can be automated by getting the values through the valueplusargs in basic\_test\_cfg. Basic\_test\_cfg is the configuration file which configures the variables of the UVM testbench.



Figure 1: Basic UVM Testbench Architecture

**3. Basic UVC:**

The architecture for the basic UVC is shown in Figure 2. The UVC is basically an IP based on how the driver and monitor is coded. UVC has UVM components as well as UVM objects. UVC has agent configuration, agent, sequencer, driver, monitor, sequence\_item and interface [16]. The detailed view of the above are explained below.

**Basic Agent Config:**

Basic agent configuration configures the specific UVC. Basic\_agent\_cfg is extended from the uvm\_object which is the built-in UVM class, so the basic\_agent\_cfg is registered in the factory using the macro `uvm\_object\_utils. In this scenario the UVC is used as CAN node by configuring the mode variable as ‘0’.

**Basic Transaction:**

Basic txn is the transaction which extends from the uvm\_sequence\_item. Uvm\_sequence\_item is child class of uvm\_object, so basic\_txn is registered in factory using the macro ‘uvm\_object\_utils. Here the fields of specific IP frame or the variables are randomized or packed based on the requirements. All the CAN frame fields are randomized using the rand keyword in system Verilog. The variables are sof-start of frame, b\_id-base identifier, e\_id-extended identifier, rtr-remote transmission request, ide-identifier extension, rsvd-reserved, dlc-data length code, data, crc-cyclic redundancy check, crc\_delimiter, ack-acknowledgment, ack\_delimiter, eof-end of frame and inter\_frame\_gap. Constraints are used in the sequence item to handle randomization properly. Say sop of the CAN protocol is always ‘0’, then we can use the constraints as “constraint c\_sof {sof == 1'b0;}” similarly all the sequence items may or may not use the constraints based on the protocol specification. Pack/Unpack build-in UVM functions can be used to pack the variable into frames and vice-versa. All the constraints, pack functions and unpack functions coded are not explained here, because that is not the intention of the paper.

**Basic Interface:**

Interface is the bunch of signals. The single bit data signal used in the interface. Normally interface may have clocking blocks and modports. This UVC supports only serial interface protocol. Interface has virtual and physical interfaces to separate the dynamic and static worlds in a testbench. Objects created in the testbench environments are dynamic world and the module and DUT are considered as static world.

**Basic Driver:**

Basic driver class extends from uvm\_driver. Uvm\_driver is a component, so the same is registered in factory through the UVM macro `uvm\_component\_utils. Basically, the transaction variables are get converted into signals in driver. Agent cfg and the basic interface are got from the configuration database in build phase via the uvm\_config\_db method. After reset the sequence item is put from the sequencer in the run phase. Once the randomize values are get from the sequencer then the variables are packed and formed as frame in driver. The frame is split into serial data and get transferred for every clock. The inter\_frame\_gap i.e. the idle cycles are transferred after sending the entire frame serially.

**Basic Monitor:**

Basic Monitor class extends from uvm\_monitor. uvm\_monitor is a component, so the same is registered in factory through the UVM macro `uvm\_component\_utils. Basically, the signals from interface gets converted into transaction variables in monitor. Agent cfg and the basic interface gets from the configuration database in build phase via the uvm\_config\_db method. After reset the monitor monitors signals and gets collected in the run phase. The collected data are packed to form the frame in monitor. The frame gets unpacked and assigned it to the specific sequence items in transactions. Finally, the transactions may transfer to the scoreboard for data checking or for coverage via the analysis port.

**Basic Agent:**

Basic agent class extends from uvm\_agent. uvm\_agent is a component, so the same is registered in factory through the UVM macro `uvm\_component\_utils. Agent creates the UVC architecture. The sequencer, driver and monitor get created in the agent through the create function of UVM in build phase. The connection between the sequencer and driver through sequence\_item\_port and the connection between the monitor and scoreboard/coverage collector via analysis port are done in the connect phase.

Figure 2: Basic UVC Architecture

The testbench is compiled and simulated using the QUESTA simulator. The results of this basic UVC used as CAN in Figure 3,4 and 5.

The fragments of log file which shows the CAN frame:

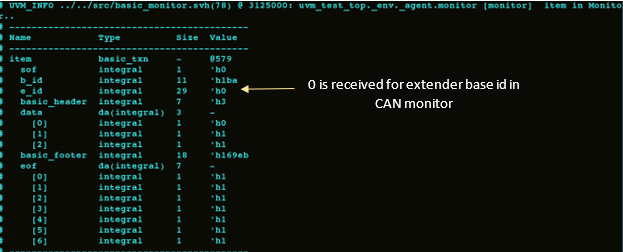


Figure 3: CAN transaction from Monitor

CAN frame format is captured in Figure 3. The transactions which are transmitted from the CAN bus is sampled in the monitor and logged the same in the simulation log file. CAN frame fields monitored are SOF, base or extended identifier, header and footer part. Finally, the completion of the CAN frame is identified by receiving the 7 sequential 1’s.

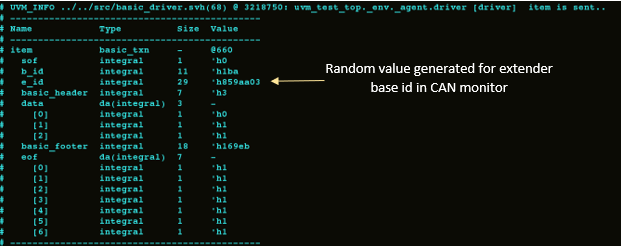


Figure 4: CAN transaction from Driver

CAN frame format transmitted in the CAN bus is shown in Figure 4. The transactions are already explained for Figure 3, but there is one main difference between the driver and monitor transactions. Even though the extended base identified is generated randomly in the driver, CAN DUT does not transmit the same in the CAN bus. So, CAN monitor samples zero for the e\_id frame field.

Wave for the basic UVC used as CAN protocol:

A picture containing object, indoor, person

Description automatically generated

Figure 5: CAN Waveform

Figure 5 shows the clock, reset and data values sent over the CAN bus. All the frame fields are transmitted serially in the CAN bus.