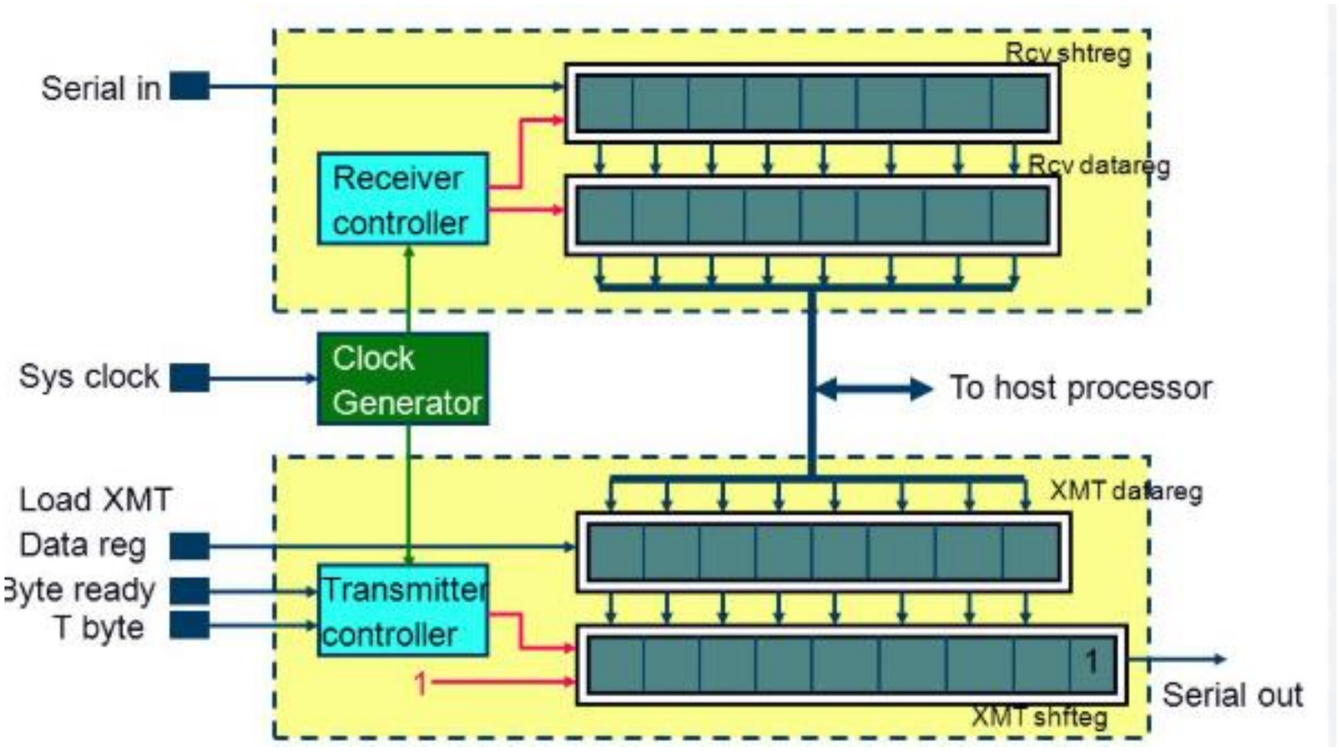
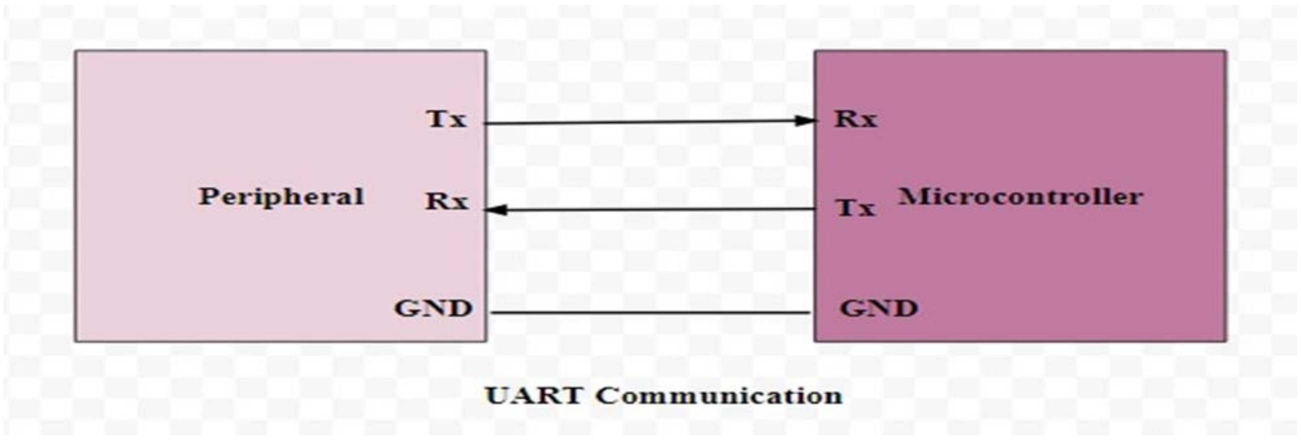


# UART VERIFICATION WITH UVM

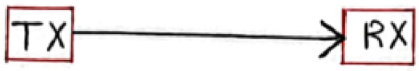


VLSI TECH
WITH ANOUSHKA

SEMICONDUCTOR

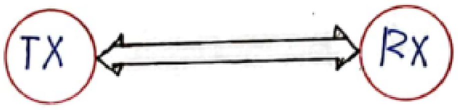
Types of Data Transmission :

1. Simplex



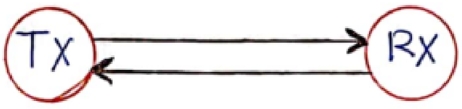
- One device transmits and other devices receive.
- Communication is possible in only 1 direction.
- Example : FM Radio

2. Half Duplex



- Two devices can send and receive from each others.
- Two way communication but only one at a time.
- Example : Walkie Talkie

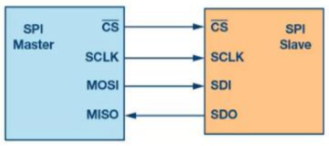
3. Full Duplex

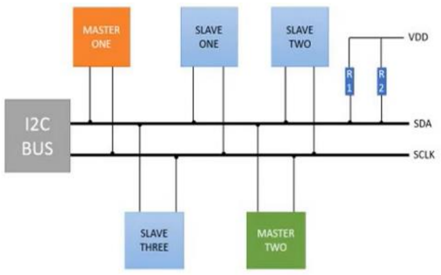


- Communication is possible in only 1 direction, both sides can transmit.
- Example : Phone call

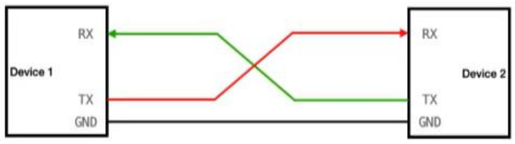
Synchronus

- Both sender and receiver share common clock (SPI,I2C)
- Device send clock → Master
- Device receive clock → Slave





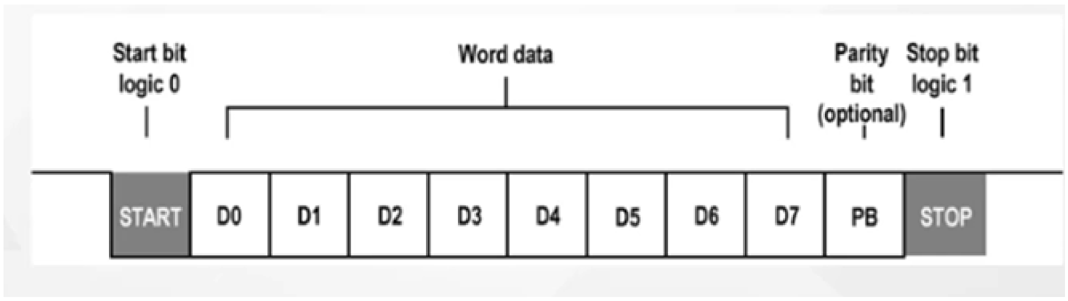
Asynchronus : Data is sent without clock.



- Point to point : Communicate between 2 devices (UART)
- Multi-drop : single transmitter and multiple receiver (SPI)
- Multi-point : channel is shared between devices. (I2C)

## UART

- Asynchronus communication Interface.
- Full/Half duplex.

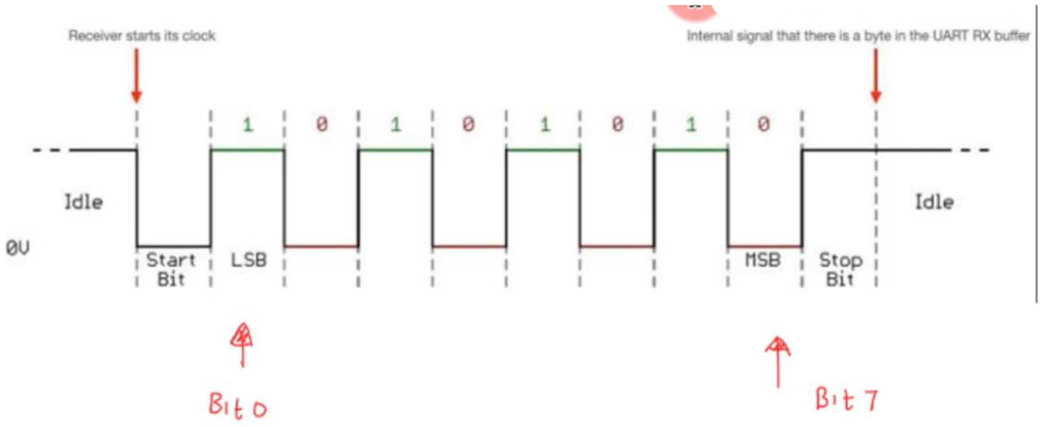


· In asynchronous communication, baud rate measures the speed of data transmission. Baud rate accounts for all bits sent, including start, stop, and parity bits.

. The actual data transfer speed, represented as bit rate (bps), indicates the amount of data transmitted from the sender to the receiver.

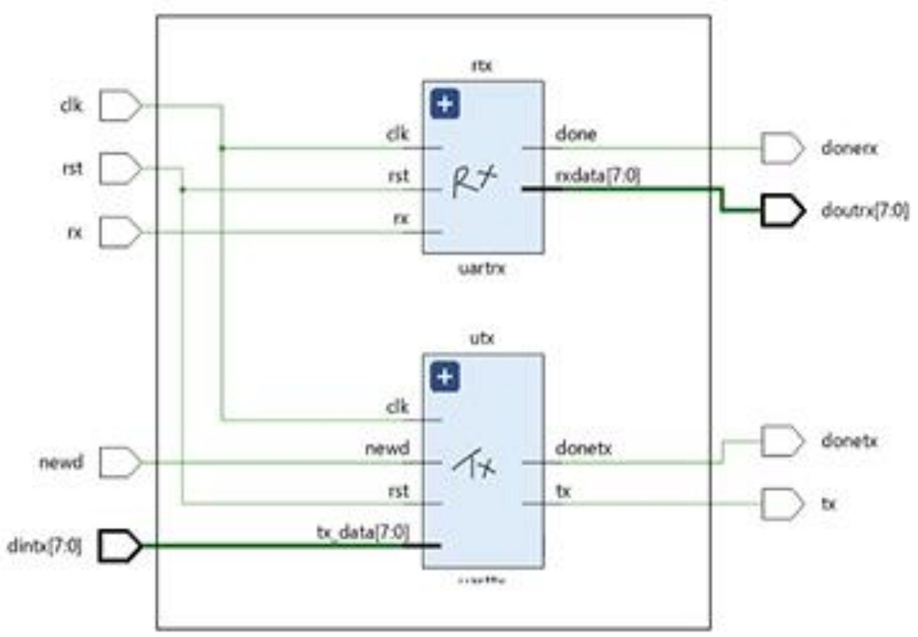
. UART speeds are expressed in bits per second (bit/s or bps).

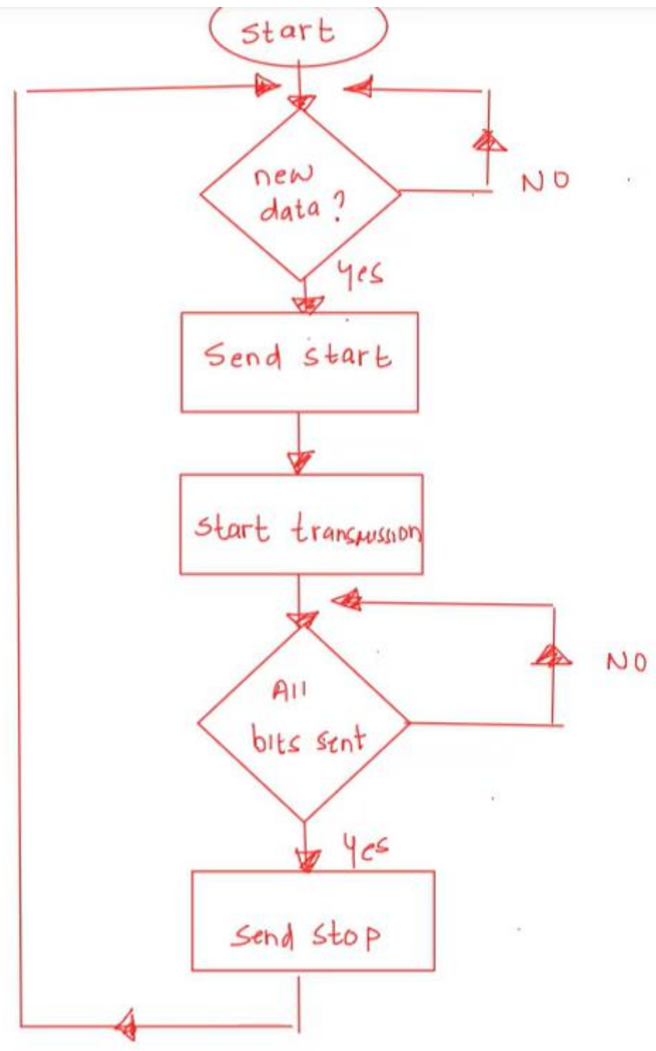
· Standard baud rates are as follows: 110, 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 460800, 921600, 1382400, 1843200, and 2764800 bit/s.


Receiver starts its clock
Internal signal that there is a byte in the UART RX buffer
1
0
1
0
1
0
1
0
Idle
0V
Start Bit
LSB
MSB
Stop Bit
Idle
Bit 0
Bit 7

| | | | |
|---|---|---|---|
| Start_bit (1-bit) | Data Frame 5 to 9 data bits | Parity_bits 0/1 bit | stop-bits 1/1.5/2 bits |

TOP


clk
rst
rx
newd
dintx[7:0]
rtx
clk
rst
rx
Rx
done
rxdata[7:0]
uartrx
utx
clk
newd
rst
tx_data[7:0]
Tx
donetx
tx
donerx
doutrx[7:0]
donetx
tx

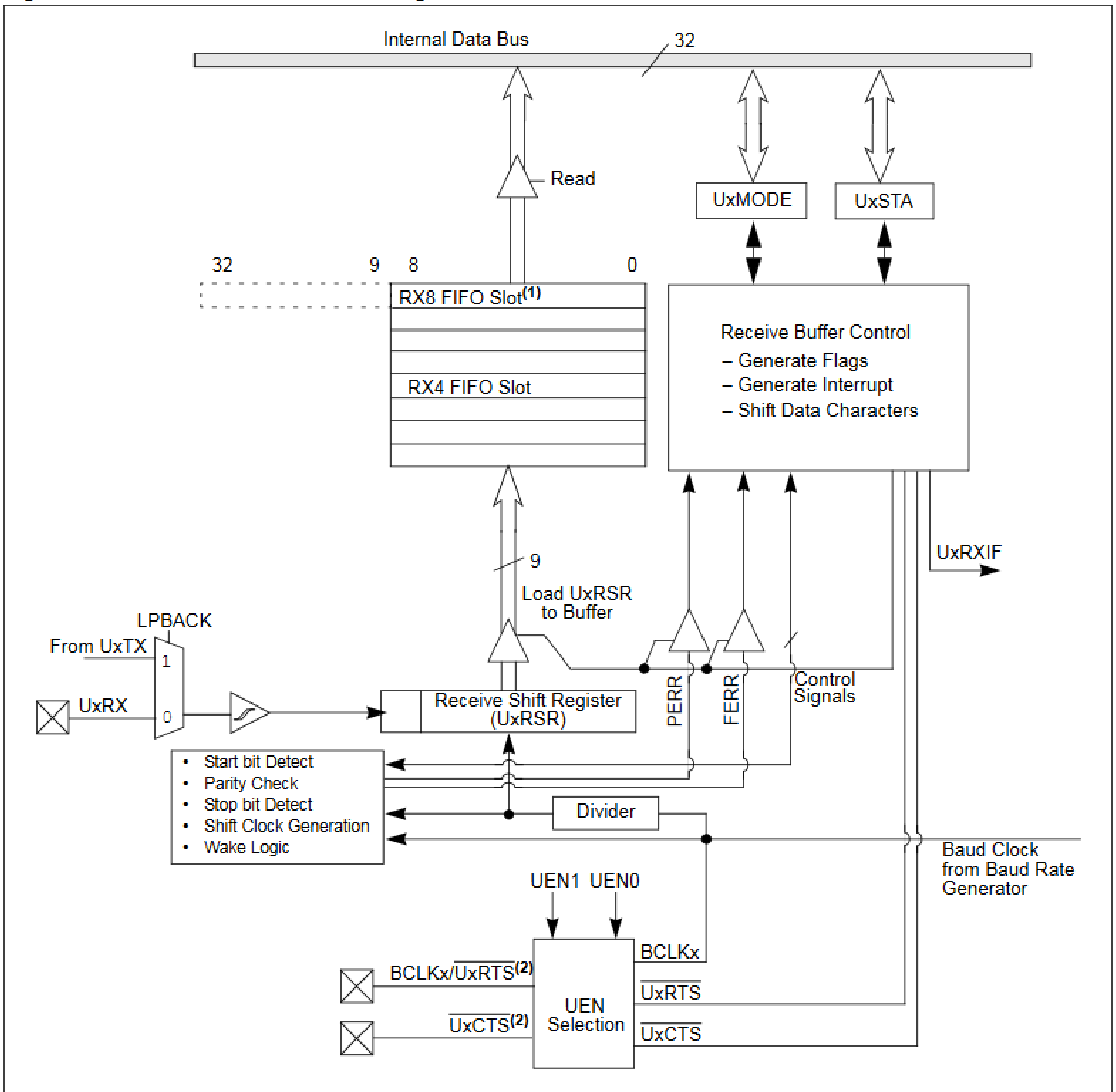

start
new data ?
NO
yes
Send start
Start transmission
All bits sent
NO
yes
Send stop

Since we want our UART to work on different baud rate, we need to make it capable of generating different clock at different baud rates.

First we will be understanding clock generation IP and then we will understand it's verification also.

Because doing this help us ensure clocks are working as expected.

**Figure 21-7: UART Receiver Block Diagram[1]**



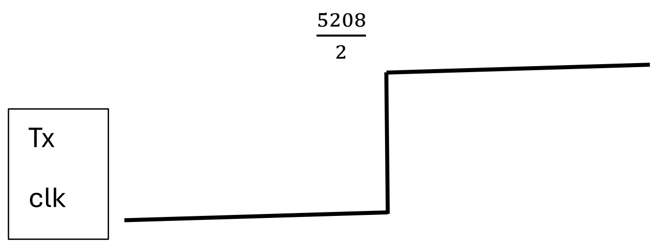
**Note 1:** Refer to the **"UART"** chapter in the specific device data sheet for availability of the 8-level-deep FIFO.

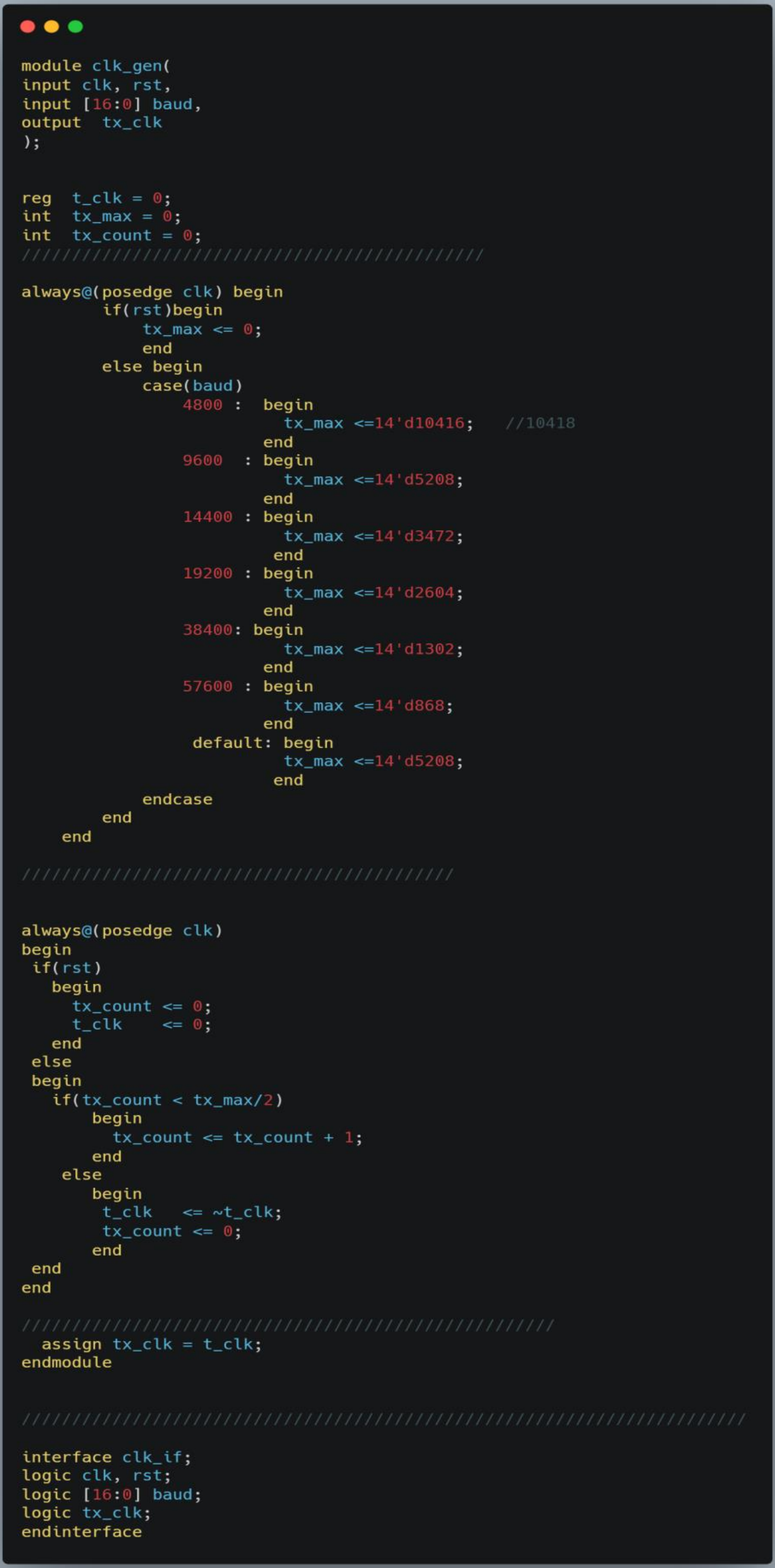
**2:** Refer to the **"Pin Diagrams"** section of the specific device data sheet for availability of the $\overline{UxRTS}$ and $\overline{UxCTS}$ pins.

$$count = \frac{F_{clk}}{F_{out}}$$

$$count = \frac{5 \times 10^6}{9600}$$

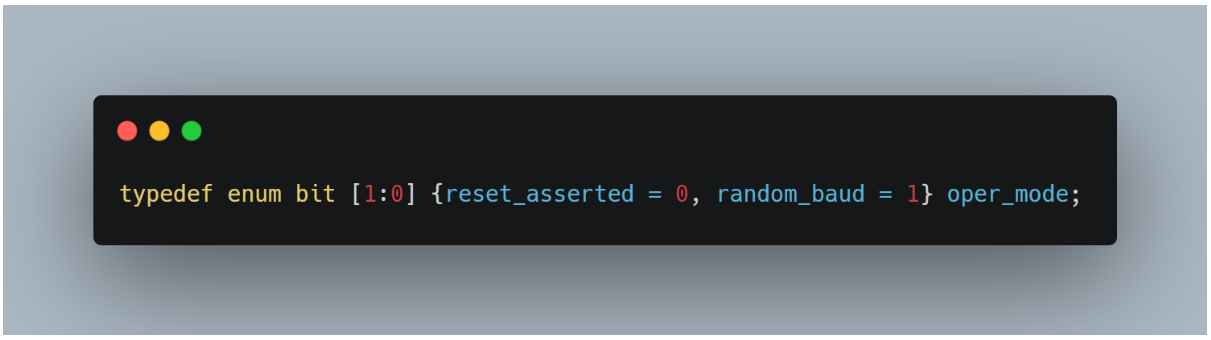
$$count = 5208$$





```
module clk_gen(
input clk, rst,
input [16:0] baud,
output  tx_clk
);


reg  t_clk = 0;
int  tx_max = 0;
int  tx_count = 0;
//////////////////////////////////////////////

always@(posedge clk) begin
        if(rst)begin
            tx_max <= 0;
            end
        else begin
            case(baud)
                4800 :  begin
                          tx_max <=14'd10416;   //10418
                        end
                9600  : begin
                          tx_max <=14'd5208;
                        end
                14400 : begin
                          tx_max <=14'd3472;
                         end
                19200 : begin
                          tx_max <=14'd2604;
                        end
                38400: begin
                          tx_max <=14'd1302;
                        end
                57600 : begin
                          tx_max <=14'd868;
                        end
                 default: begin
                          tx_max <=14'd5208;
                         end
            endcase
        end
    end

///////////////////////////////////////////


always@(posedge clk)
begin
 if(rst)
   begin
     tx_count <= 0;
     t_clk    <= 0;
   end
 else
 begin
   if(tx_count < tx_max/2)
       begin
         tx_count <= tx_count + 1;
       end
    else
       begin
        t_clk   <= ~t_clk;
        tx_count <= 0;
       end
 end
end

/////////////////////////////////////////////////////
  assign tx_clk = t_clk;
endmodule


////////////////////////////////////////////////////////////////////////

interface clk_if;
logic clk, rst;
logic [16:0] baud;
logic tx_clk;
endinterface
```

## Verifying Clock generator with UVM :

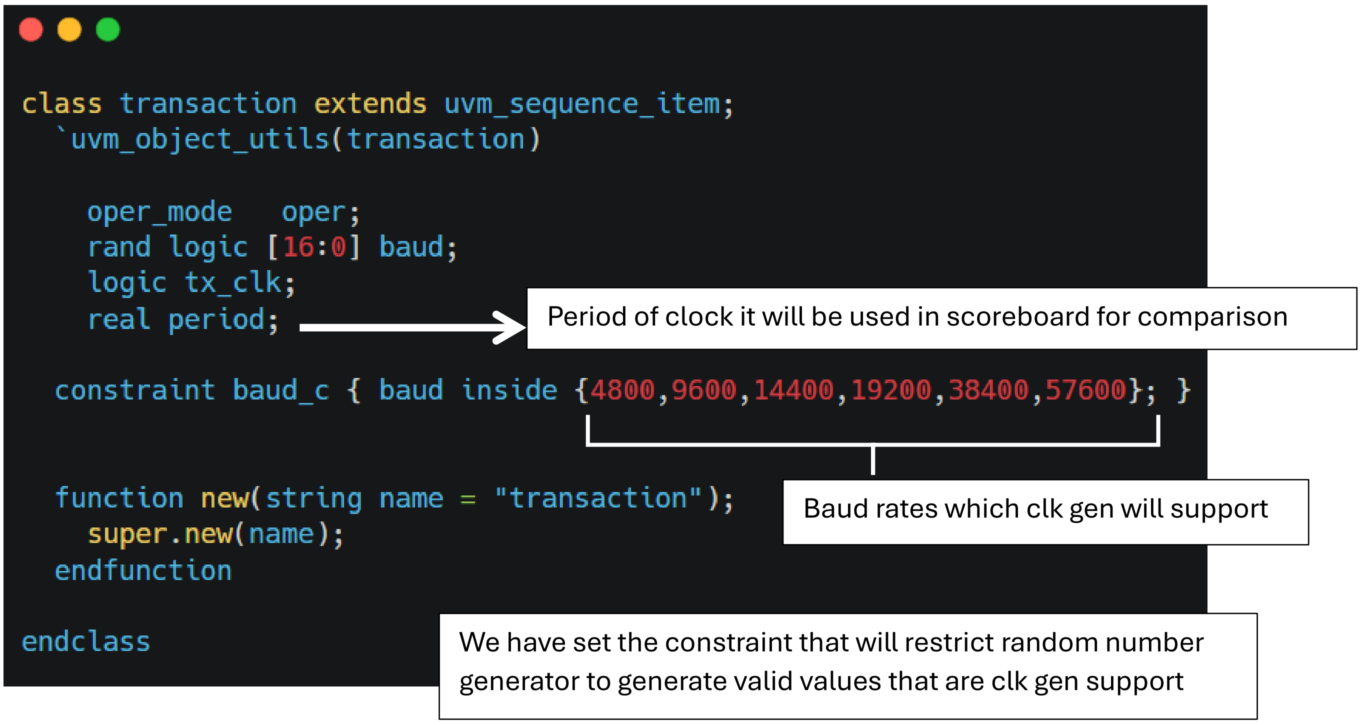


```
typedef enum bit [1:0] {reset_asserted = 0, random_baud = 1} oper_mode;
```

We declare enum bit variable to specify the mode we are targeting

1. Reset asserted → clock will stay predominantly at zero when user apply reset
2. Random_baud → applying random baud rate, to check clock is giving frequency as expected.

TRANSACTION CLASS



```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

    oper_mode   oper;
    rand logic [16:0] baud;
    logic tx_clk;
    real period;

  constraint baud_c { baud inside {4800,9600,14400,19200,38400,57600}; }


  function new(string name = "transaction");
    super.new(name);
  endfunction

endclass
```

Period of clock it will be used in scoreboard for comparison

Baud rates which clk gen will support

We have set the constraint that will restrict random number generator to generate valid values that are clk gen support

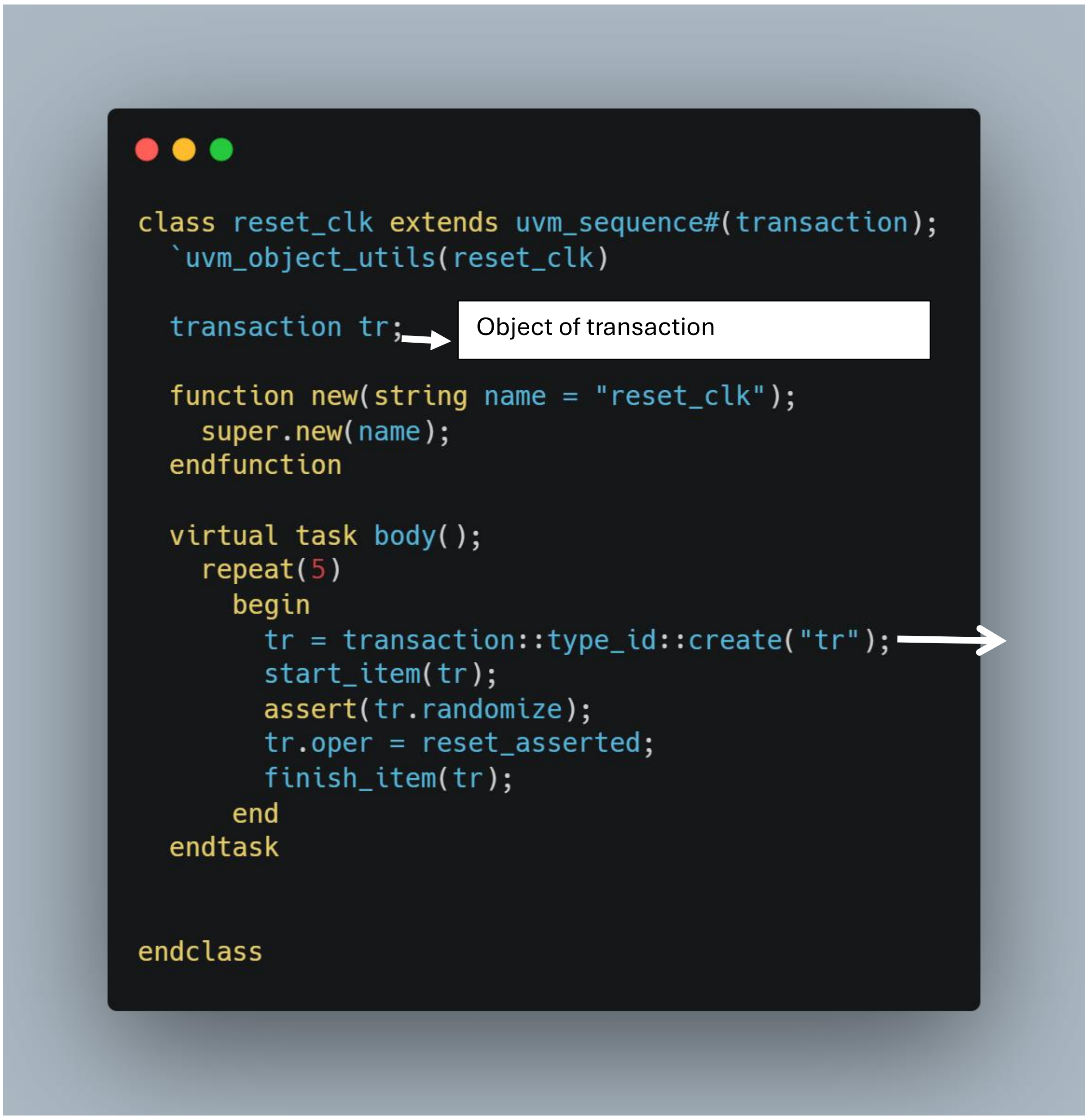
The **transaction** class extends uvm_sequence_item. A transaction represents a data packet or an action that moves between different components (like driver, monitor, etc.). It has:

- **oper_mode**: Defines operation type (like reset or random baud rate selection).
- **tx_clk**: Clock signal for data transmission.
- **period**: Time period between two clock edges.

SEQUENCE

The **reset_clk** sequence generates reset transactions repeatedly:

- tr.oper = reset_asserted;: Assigns the reset operation.
- This sequence is responsible for initiating a system reset for 5 clock cycles.

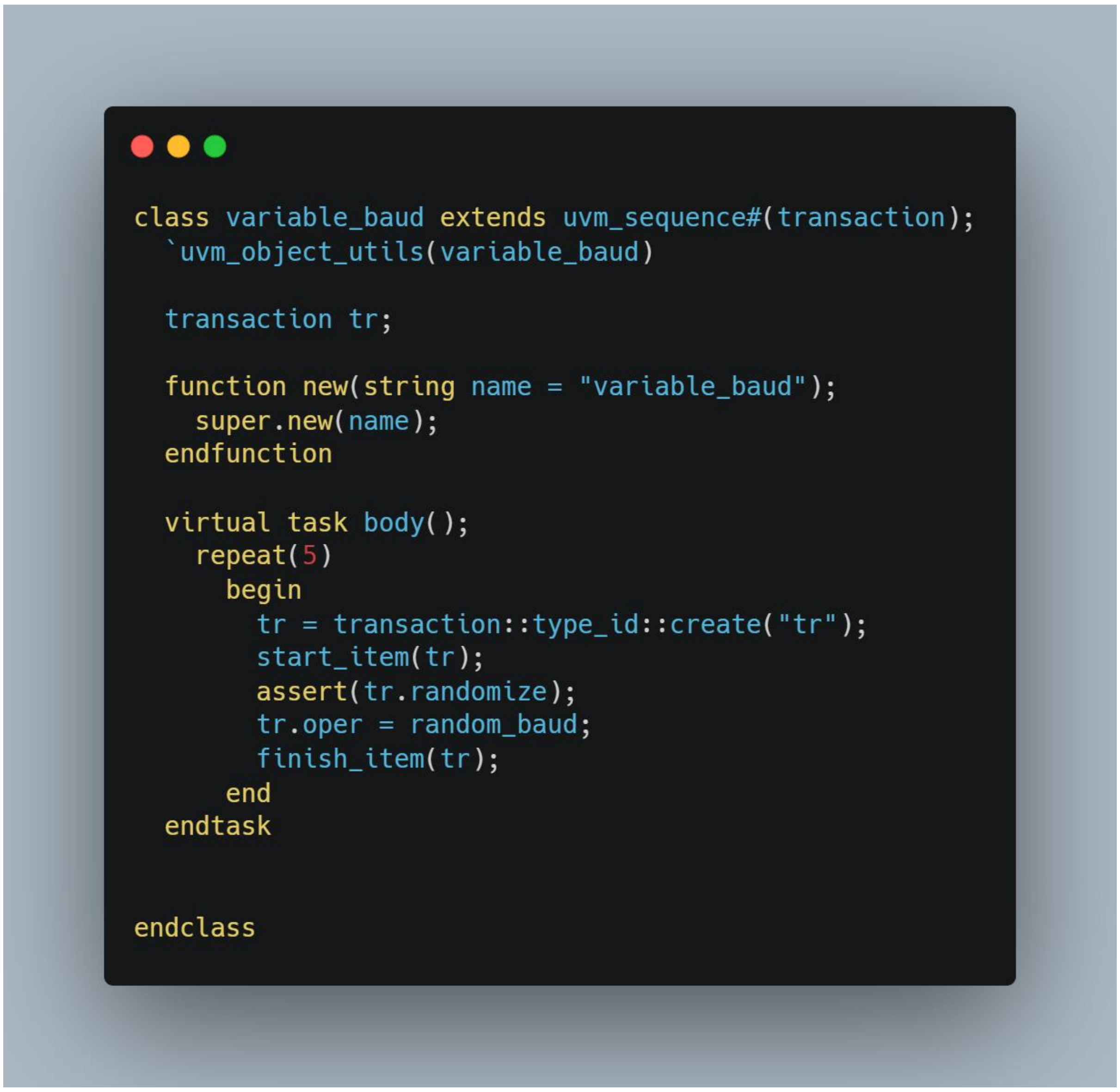


```
class reset_clk extends uvm_sequence#(transaction);
  `uvm_object_utils(reset_clk)

  transaction tr;

  function new(string name = "reset_clk");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.oper = reset_asserted;
        finish_item(tr);
      end
  endtask


endclass
```

The **variable_baud** sequence generates transactions that vary the baud rate:

- It randomizes the baud rate and assigns it using tr.oper = random_baud.

Sequences help generate various test conditions. Here, we have a sequence for resetting the system and another for applying random baud rates.



```
class variable_baud extends uvm_sequence#(transaction);
  `uvm_object_utils(variable_baud)

  transaction tr;

  function new(string name = "variable_baud");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.oper = random_baud;
        finish_item(tr);
      end
  endtask


endclass
```

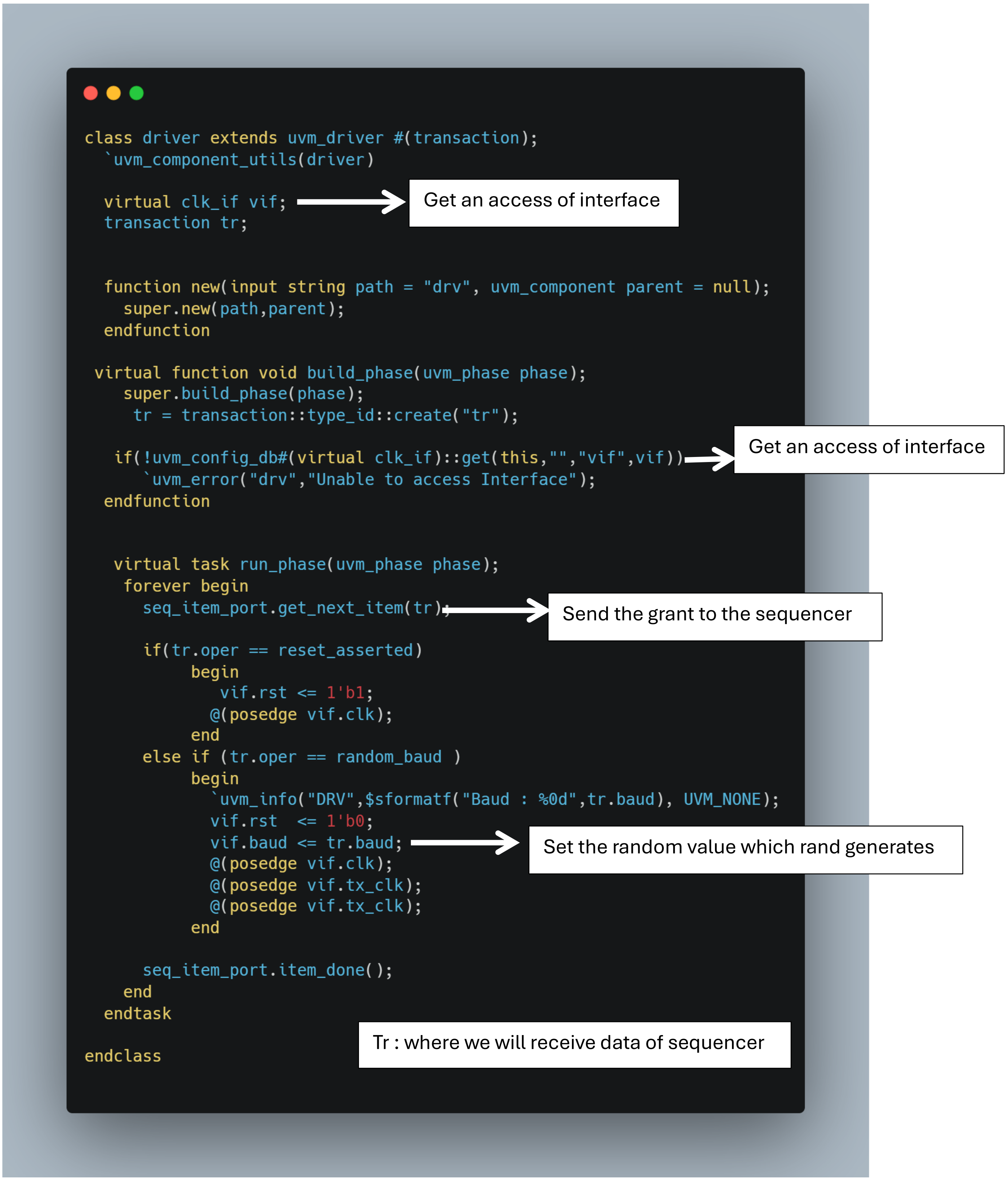
After this we will create the driver class, In driver depending upon type of mode we will apply respective signal.

The **driver** class drives the transactions onto the DUT (Design Under Test):

- **vif** is the virtual interface connecting the testbench to the DUT.
- Based on the transaction's oper_mode, the driver either asserts a reset or sends a baud rate to the DUT.

Key tasks include:

- **start_item()**: Signals the start of the transaction.
- **finish_item()**: Signals the end of the transaction.
- **seq_item_port.get_next_item()**: Fetches the next transaction from the sequencer.



```
class driver extends uvm_driver #(transaction);
  `uvm_component_utils(driver)

  virtual clk_if vif;
  transaction tr;


  function new(input string path = "drv", uvm_component parent = null);
    super.new(path,parent);
  endfunction

 virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     tr = transaction::type_id::create("tr");

   if(!uvm_config_db#(virtual clk_if)::get(this,"","vif",vif))
       `uvm_error("drv","Unable to access Interface");
  endfunction


   virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(tr);

      if(tr.oper == reset_asserted)
           begin
              vif.rst <= 1'b1;
             @(posedge vif.clk);
           end
      else if (tr.oper == random_baud )
           begin
             `uvm_info("DRV",$sformatf("Baud : %0d",tr.baud), UVM_NONE);
             vif.rst  <= 1'b0;
             vif.baud <= tr.baud;
             @(posedge vif.clk);
             @(posedge vif.tx_clk);
             @(posedge vif.tx_clk);
           end

      seq_item_port.item_done();
    end
  endtask

endclass
```

Get an access of interface
Get an access of interface
Send the grant to the sequencer
Set the random value which rand generates
Tr : where we will receive data of sequencer

The **monitor** observes transactions from the DUT without modifying them:

- It tracks system behavior (e.g., checking if reset is asserted or capturing baud rate).
- It captures the clock period and baud rate, then sends the transaction to the scoreboard.

- **uvm_config_db::get()**: This is used to retrieve the virtual interface from the UVM configuration database, which allows the monitor to access the DUT's signals.
- **real ton, real toff**: These variables store time values, used to calculate the time period (or clock frequency) by capturing clock edges.
- **virtual clk_if vif;**: This is the virtual interface that connects the monitor to the signals in the DUT. It provides access to the DUT's clock, reset, and baud rate signals.

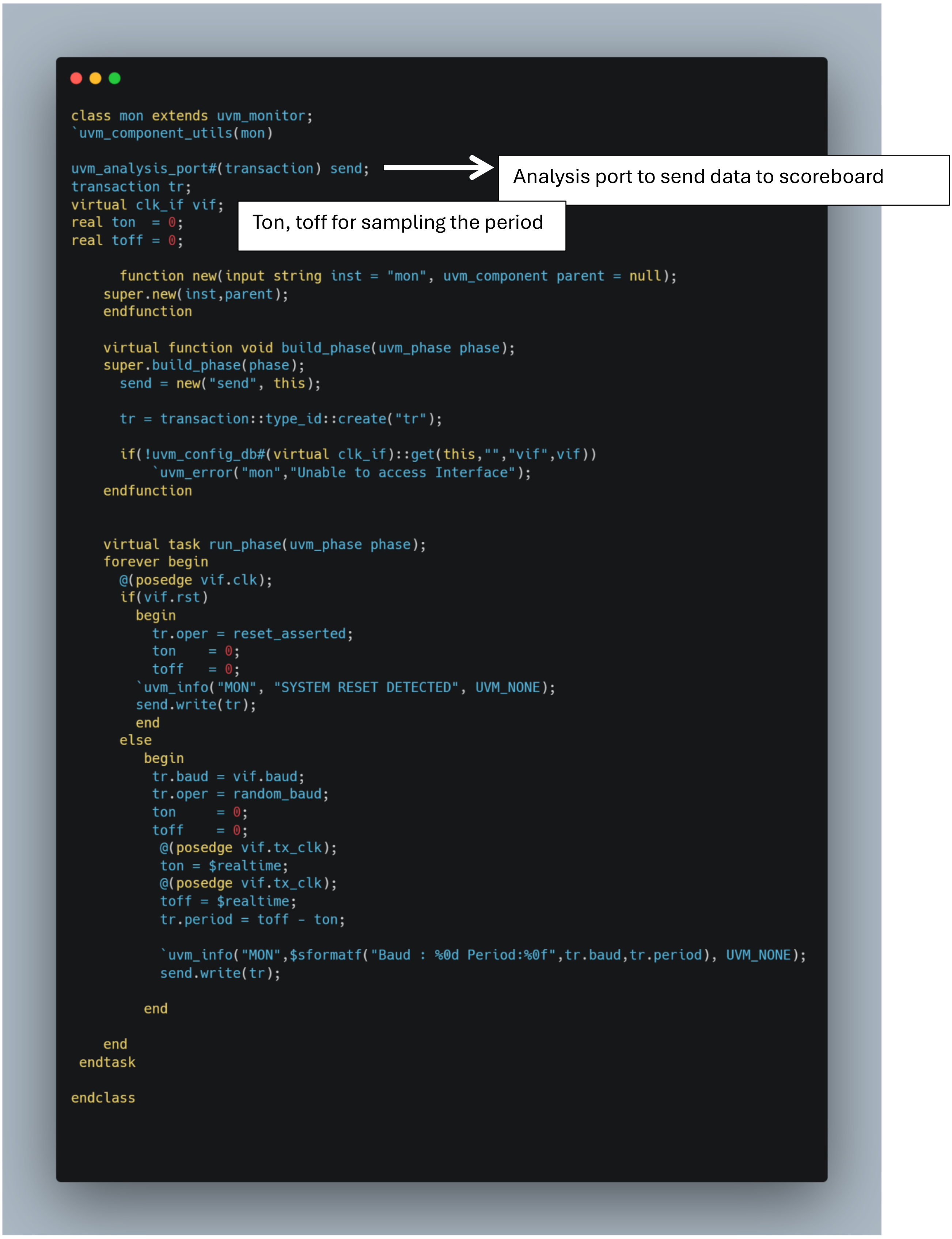
**run_phase**: This task continuously monitors the signals of interest (e.g., clk, rst, baud, tx_clk) and creates transactions accordingly.

 **If vif.rst is asserted**:

- A reset transaction is created (tr.oper = reset_asserted), and both ton and toff (used for timing) are reset to zero.
- It logs that the system reset was detected.
- The transaction is then sent to the analysis port (send.write(tr);), which forwards it to the scoreboard.

 **If vif.rst is not asserted**:

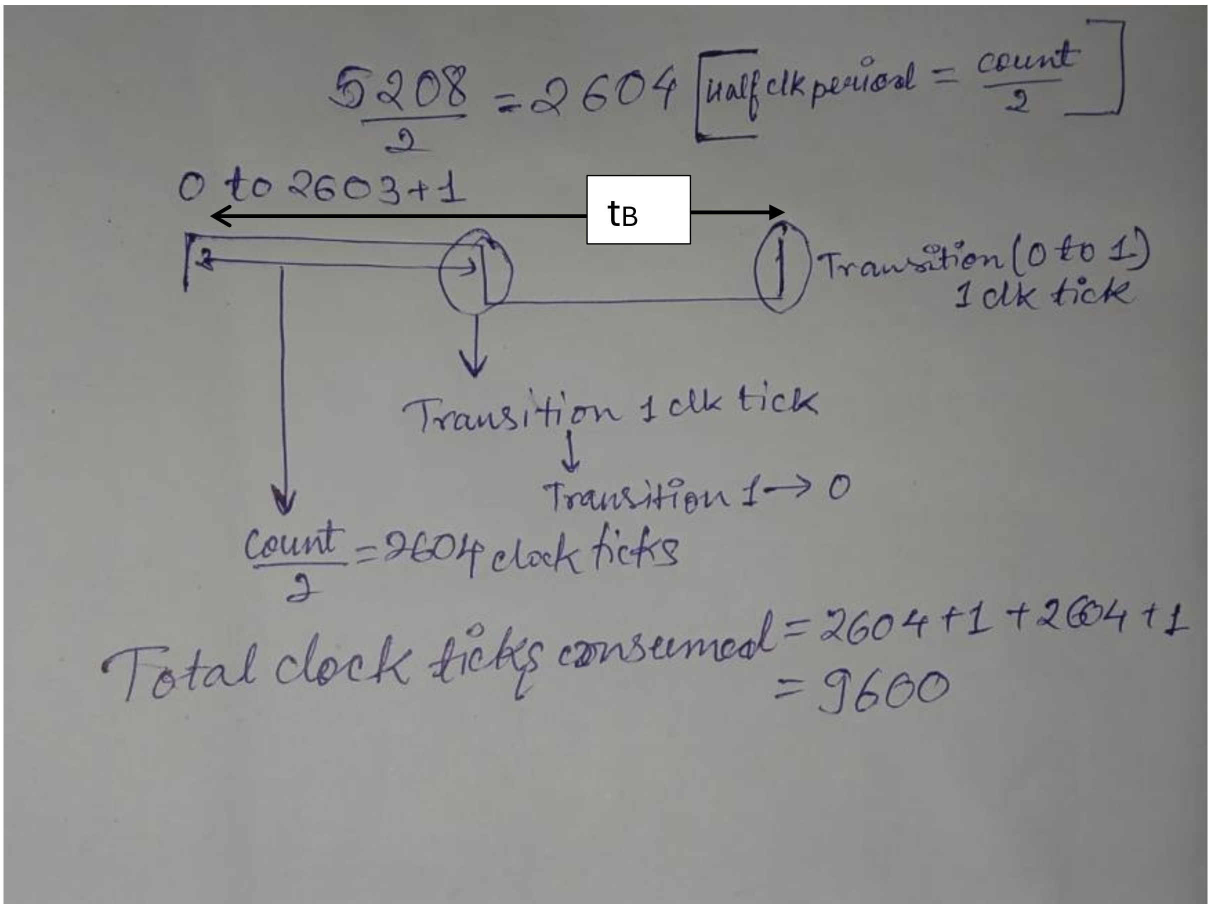
- The monitor captures the baud rate (tr.baud = vif.baud;) and notes that this is a random baud operation (tr.oper = random_baud).
- It calculates the period of the clock using ton and toff, based on two rising edges of the tx_clk.
- The transaction is filled with this information and sent to the analysis port (send.write(tr);).



Analysis port to send data to scoreboard

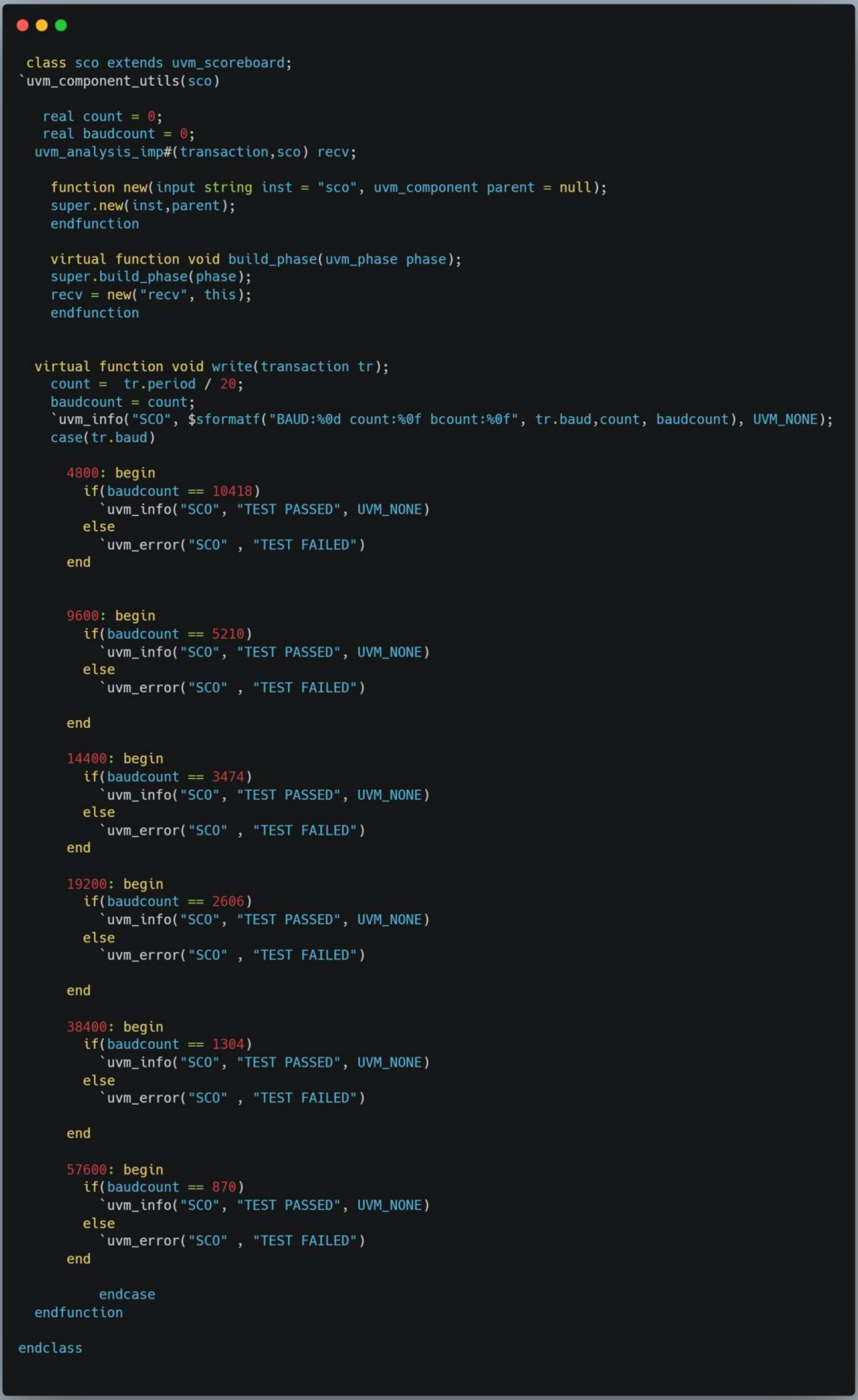
Ton, toff for sampling the period

```
class mon extends uvm_monitor;
`uvm_component_utils(mon)

uvm_analysis_port#(transaction) send;
transaction tr;
virtual clk_if vif;
real ton  = 0;
real toff = 0;

      function new(input string inst = "mon", uvm_component parent = null);
    super.new(inst,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
      send = new("send", this);

      tr = transaction::type_id::create("tr");

      if(!uvm_config_db#(virtual clk_if)::get(this,"","vif",vif))
          `uvm_error("mon","Unable to access Interface");
    endfunction


    virtual task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.clk);
      if(vif.rst)
        begin
          tr.oper = reset_asserted;
          ton     = 0;
          toff    = 0;
        `uvm_info("MON", "SYSTEM RESET DETECTED", UVM_NONE);
        send.write(tr);
        end
      else
         begin
          tr.baud = vif.baud;
          tr.oper = random_baud;
          ton      = 0;
          toff     = 0;
           @(posedge vif.tx_clk);
           ton = $realtime;
           @(posedge vif.tx_clk);
           toff = $realtime;
           tr.period = toff - ton;

           `uvm_info("MON",$sformatf("Baud : %0d Period:%0f",tr.baud,tr.period), UVM_NONE);
           send.write(tr);

         end

    end
 endtask

endclass
```


$\frac{5208}{2} = 2604$ [half clk period = $\frac{count}{2}$]
0 to 2603+1
$t_B$
2
1
1
Transition (0 to 1) 1 clk tick
Transition 1 clk tick
Transition 1 → 0
$\frac{count}{2}$ = 2604 clock ticks
Total clock ticks consumed = 2604 + 1 + 2604 + 1 = 9600

$$Count = \frac{Fclk}{Baud} = \frac{1/tf}{1/t_B}$$

These calculations will help us understand scoreboard



```
 class sco extends uvm_scoreboard;
`uvm_component_utils(sco)

   real count = 0;
   real baudcount = 0;
  uvm_analysis_imp#(transaction,sco) recv;

    function new(input string inst = "sco", uvm_component parent = null);
    super.new(inst,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    recv = new("recv", this);
    endfunction


  virtual function void write(transaction tr);
    count =  tr.period / 20;
    baudcount = count;
    `uvm_info("SCO", $sformatf("BAUD:%0d count:%0f bcount:%0f", tr.baud,count, baudcount), UVM_NONE);
    case(tr.baud)

      4800: begin
        if(baudcount == 10418)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")
      end


      9600: begin
        if(baudcount == 5210)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")

      end

      14400: begin
        if(baudcount == 3474)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")
      end

      19200: begin
        if(baudcount == 2606)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")

      end

      38400: begin
        if(baudcount == 1304)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")

      end

      57600: begin
        if(baudcount == 870)
          `uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          `uvm_error("SCO" , "TEST FAILED")
      end

          endcase
  endfunction

endclass
```
]

**Scoreboard (sco)**

The **Scoreboard** in UVM is a component that checks whether the output or behavior of the DUT matches the expected results. It is the central place for verifying correctness by comparing the actual data (sent from the monitor) with the expected data.

**uvm_analysis_imp#(transaction, sco) recv;**: This is an analysis import. The scoreboard uses this to receive transactions from the monitor. This is where the results sent by the monitor will be collected,

**write function**: This is the critical part of the scoreboard. It is called every time the monitor sends a transaction. The scoreboard evaluates this transaction to see if the DUT is behaving as expected.

**How the Monitor and Scoreboard Work Together:**

1. **Monitor**:
    - Observes the DUT's signals (like clock, reset, baud rate).
    - Packs these signals into transactions (including baud rate, operation mode, and clock period).
    - Sends the transactions to the scoreboard via the analysis port.
2. **Scoreboard**:
    - Receives the transactions from the monitor.
    - Compares the observed data with expected values (e.g., checking if the baud rate is correct).
    - Reports whether the test passes or fails.

**recv = new("recv", this);**: The analysis import port recv is instantiated. This port will connect to the monitor's analysis port, allowing it to receive transactions.

## Agent

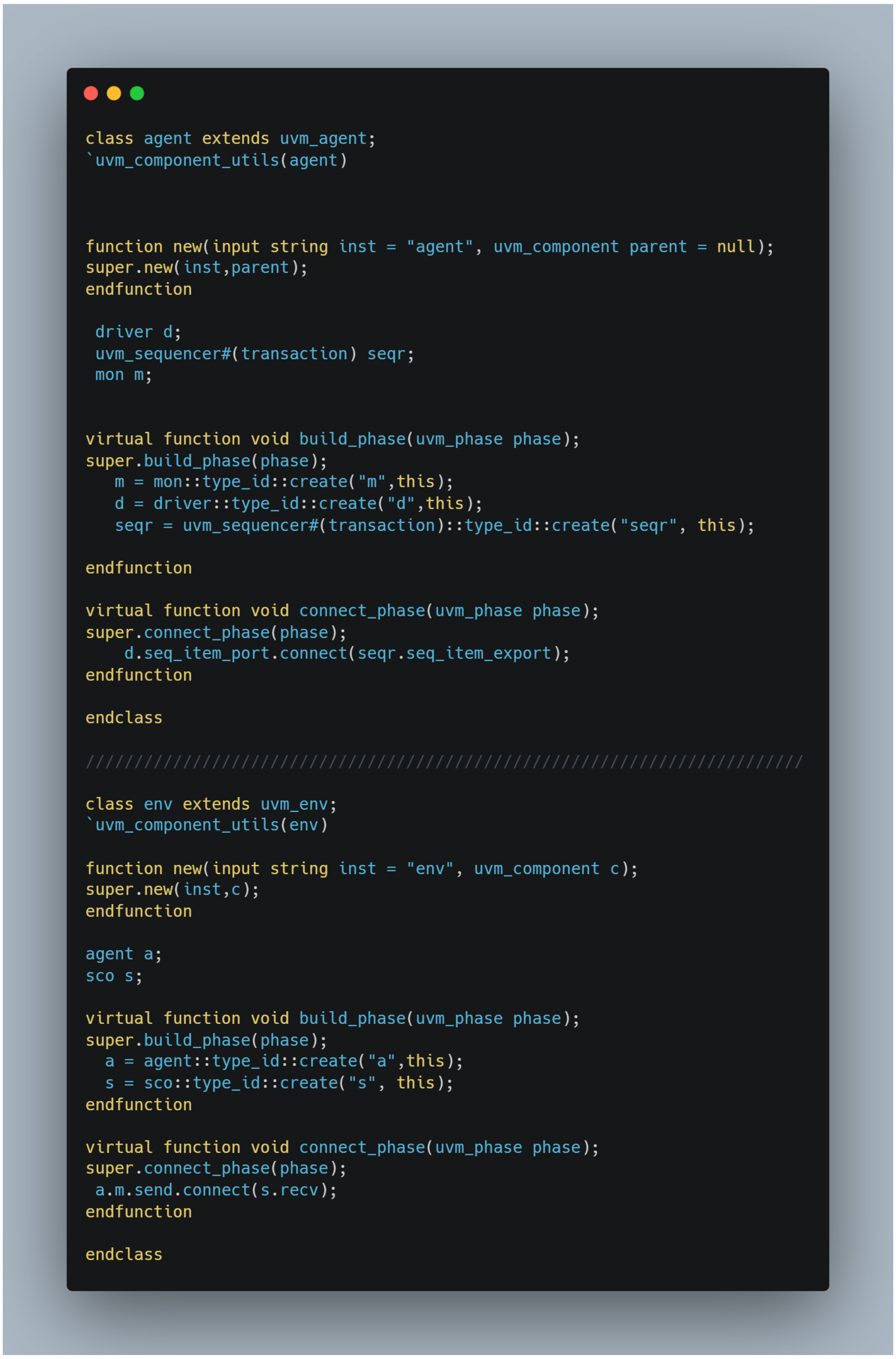
The **agent** brings together the driver, sequencer, and monitor:

- It instantiates these components and connects the sequencer to the driver.
- The monitor observes the signals and passes them to the scoreboard via the analysis port.
- **uvm_analysis_port#(transaction) ap;**: An analysis port (ap) is used to forward transactions collected by the monitor to higher-level components like the scoreboard or coverage collectors.

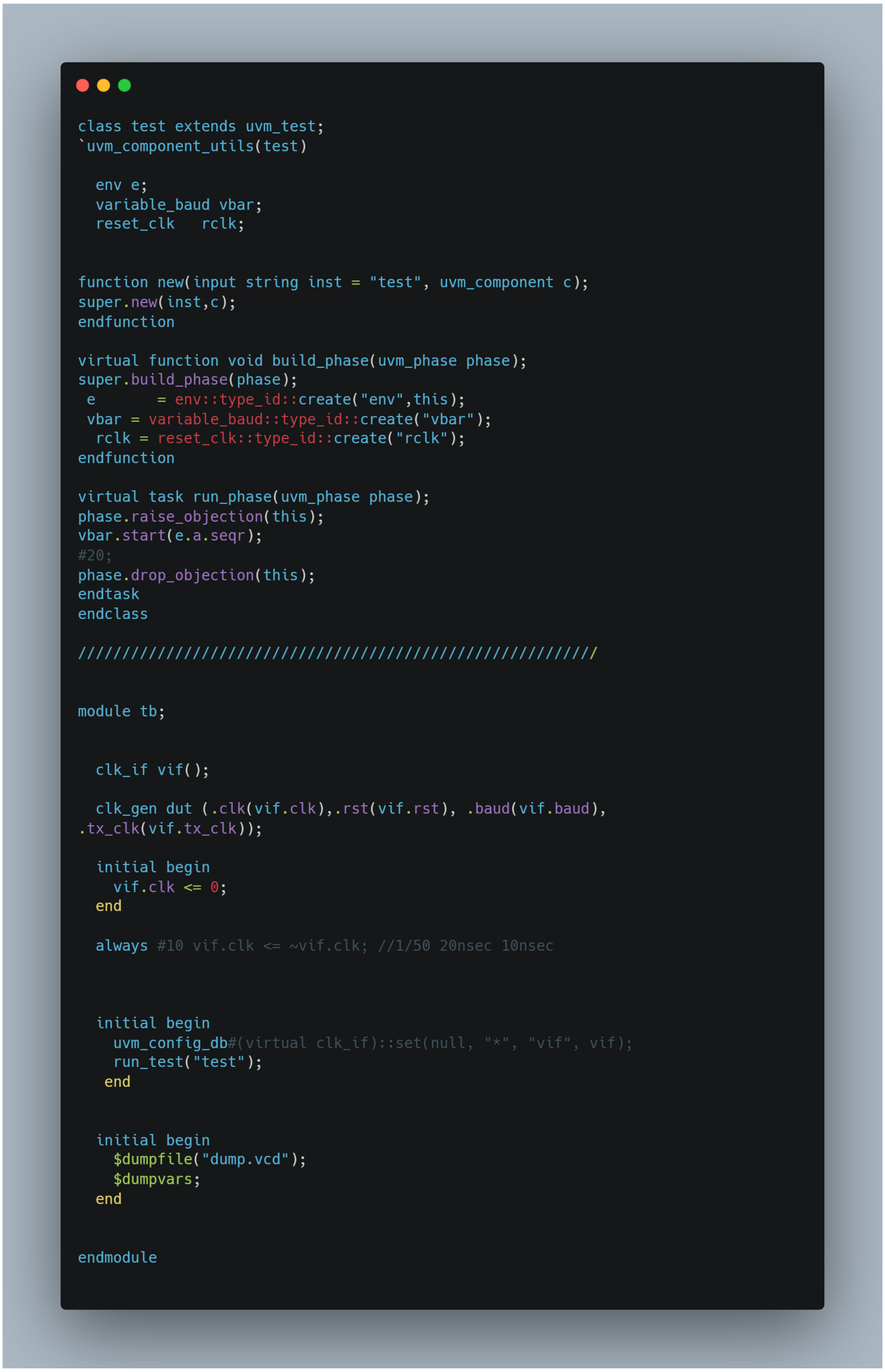
## Env

The **environment (env)** is the top-level component containing all other elements:

- The **connect_phase** connects the monitor's analysis port to the scoreboard's receive method, allowing transactions to flow from the monitor to the scoreboard for checking.
- **uvm_config_db::get()**: Similar to the agent, the environment retrieves the virtual interface from the UVM configuration database, making it accessible to both the agent and the scoreboard.
- **agent.monitor.ap.connect(scoreboard.recv)**: The environment connects the agent's monitor (which collects data) to the scoreboard. This allows the transactions gathered by the monitor to be sent to the scoreboard for checking.

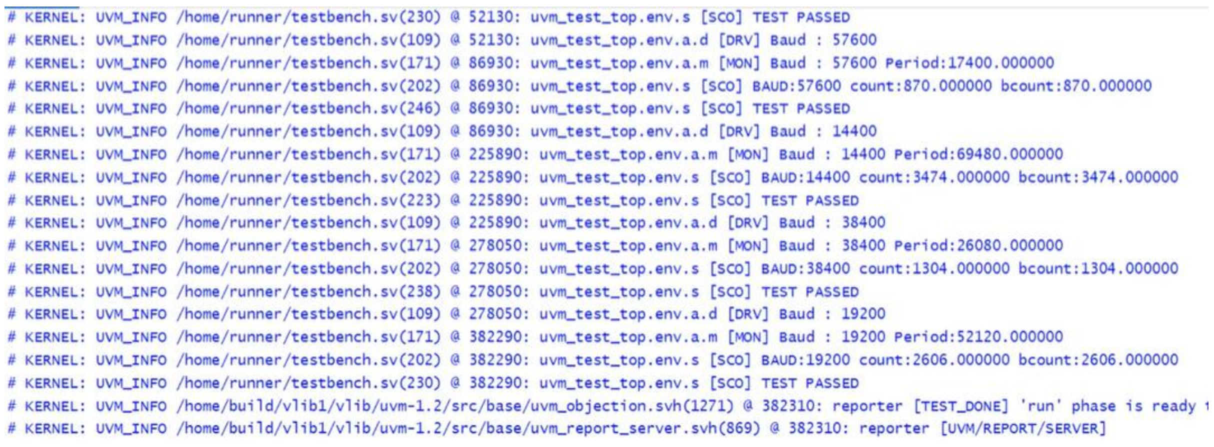


```
class agent extends uvm_agent;
`uvm_component_utils(agent)



function new(input string inst = "agent", uvm_component parent = null);
super.new(inst,parent);
endfunction

 driver d;
 uvm_sequencer#(transaction) seqr;
 mon m;


virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
   m = mon::type_id::create("m",this);
   d = driver::type_id::create("d",this);
   seqr = uvm_sequencer#(transaction)::type_id::create("seqr", this);

endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
    d.seq_item_port.connect(seqr.seq_item_export);
endfunction

endclass

//////////////////////////////////////////////////////////////////////////

class env extends uvm_env;
`uvm_component_utils(env)

function new(input string inst = "env", uvm_component c);
super.new(inst,c);
endfunction

agent a;
sco s;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  a = agent::type_id::create("a",this);
  s = sco::type_id::create("s", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
 a.m.send.connect(s.recv);
endfunction

endclass
```



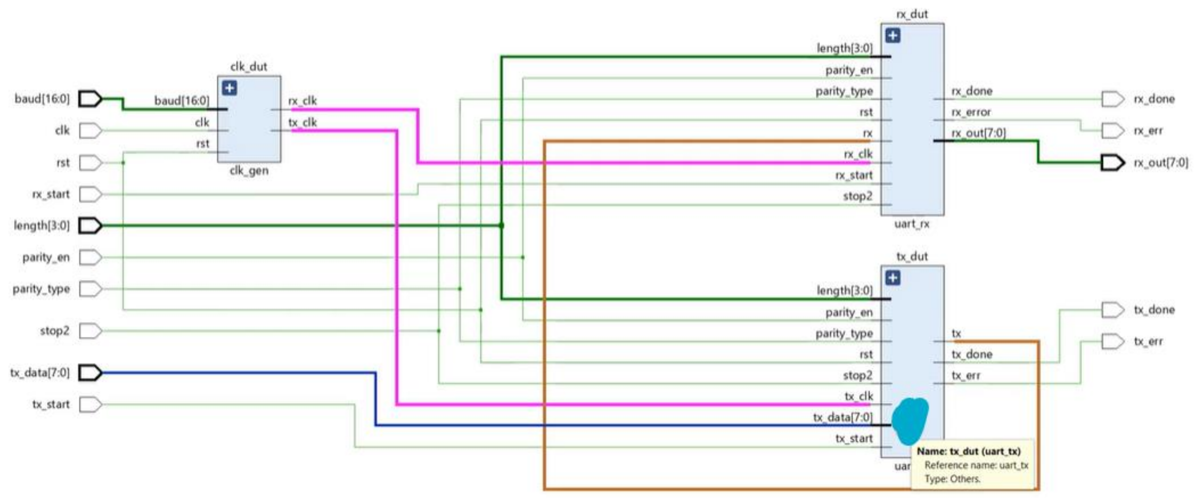
```
class test extends uvm_test;
`uvm_component_utils(test)

  env e;
  variable_baud vbar;
  reset_clk   rclk;


function new(input string inst = "test", uvm_component c);
super.new(inst,c);
endfunction

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
 e       = env::type_id::create("env",this);
 vbar = variable_baud::type_id::create("vbar");
  rclk = reset_clk::type_id::create("rclk");
endfunction

virtual task run_phase(uvm_phase phase);
phase.raise_objection(this);
vbar.start(e.a.seqr);
#20;
phase.drop_objection(this);
endtask
endclass

///////////////////////////////////////////////////////////


module tb;


  clk_if vif();

  clk_gen dut (.clk(vif.clk),.rst(vif.rst), .baud(vif.baud),
.tx_clk(vif.tx_clk));

  initial begin
    vif.clk <= 0;
  end

  always #10 vif.clk <= ~vif.clk; //1/50 20nsec 10nsec



  initial begin
    uvm_config_db#(virtual clk_if)::set(null, "*", "vif", vif);
    run_test("test");
   end


  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end


endmodule
```

## Output

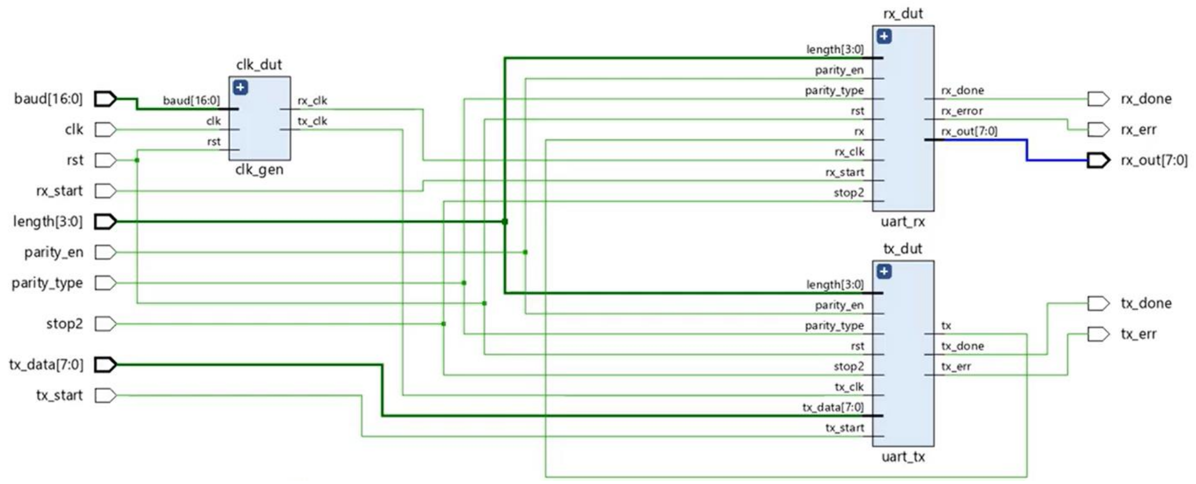


```
# KERNEL: UVM_INFO /home/runner/testbench.sv(230) @ 52130: uvm_test_top.env.s [SCO] TEST PASSED
# KERNEL: UVM_INFO /home/runner/testbench.sv(109) @ 52130: uvm_test_top.env.a.d [DRV] Baud : 57600
# KERNEL: UVM_INFO /home/runner/testbench.sv(171) @ 86930: uvm_test_top.env.a.m [MON] Baud : 57600 Period:17400.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(202) @ 86930: uvm_test_top.env.s [SCO] BAUD:57600 count:870.000000 bcount:870.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(246) @ 86930: uvm_test_top.env.s [SCO] TEST PASSED
# KERNEL: UVM_INFO /home/runner/testbench.sv(109) @ 86930: uvm_test_top.env.a.d [DRV] Baud : 14400
# KERNEL: UVM_INFO /home/runner/testbench.sv(171) @ 225890: uvm_test_top.env.a.m [MON] Baud : 14400 Period:69480.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(202) @ 225890: uvm_test_top.env.s [SCO] BAUD:14400 count:3474.000000 bcount:3474.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(223) @ 225890: uvm_test_top.env.s [SCO] TEST PASSED
# KERNEL: UVM_INFO /home/runner/testbench.sv(109) @ 225890: uvm_test_top.env.a.d [DRV] Baud : 38400
# KERNEL: UVM_INFO /home/runner/testbench.sv(171) @ 278050: uvm_test_top.env.a.m [MON] Baud : 38400 Period:26080.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(202) @ 278050: uvm_test_top.env.s [SCO] BAUD:38400 count:1304.000000 bcount:1304.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(238) @ 278050: uvm_test_top.env.s [SCO] TEST PASSED
# KERNEL: UVM_INFO /home/runner/testbench.sv(109) @ 278050: uvm_test_top.env.a.d [DRV] Baud : 19200
# KERNEL: UVM_INFO /home/runner/testbench.sv(171) @ 382290: uvm_test_top.env.a.m [MON] Baud : 19200 Period:52120.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(202) @ 382290: uvm_test_top.env.s [SCO] BAUD:19200 count:2606.000000 bcount:2606.000000
# KERNEL: UVM_INFO /home/runner/testbench.sv(230) @ 382290: uvm_test_top.env.s [SCO] TEST PASSED
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_objection.svh(1271) @ 382310: reporter [TEST_DONE] 'run' phase is ready
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(869) @ 382310: reporter [UVM/REPORT/SERVER]
```

## UART VERIFICATION



### Understanding UART System



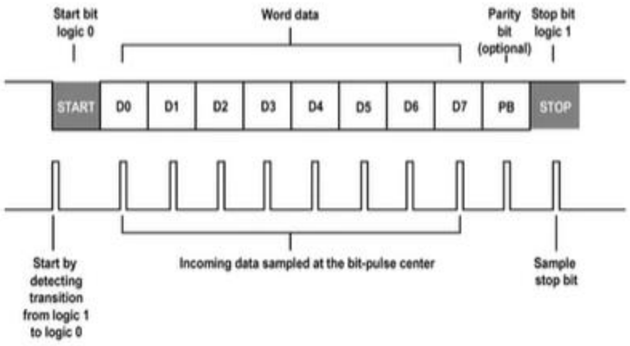
Here's a simple breakdown of how each part works:

**1. Clock Generation (clk_dut and clk_gen)**

- The **clk_gen** block takes a standard clock input and a reset signal to generate two different clock signals: one for the receiver and one for the transmitter. It also takes in a **baud rate** input (the speed at which data is transmitted) to ensure that the data is sent and received at the correct timing.
- The **clk_dut** block provides these clock signals (called **rx_clk** and **tx_clk**) to the receiver and transmitter, making sure both sections operate in sync with the right speed.

**2. UART Receiver (uart_rx)**

- The **uart_rx** block handles receiving data. It listens for incoming serial data, processes it, and converts it back into a format the system can use.
- It has several inputs to control how data is received:
    - **rx_start**: Signals the system to start receiving data.
    - **length**: Specifies how many bits the incoming data contains.We can have length between 5 to 8.



- 
    - **parity_en** and **parity_type**: These settings control whether error-checking parity bits are used, and whether it's odd or even parity.
    - **stop2**: Decides whether the data transmission ends with one or two stop bits (stop bits mark the end of data).
- Once the data is received, the **rx_done** signal lets the system know the reception is complete. If there's an error (like a parity or transmission error), the **rx_err** signal will flag it. The actual data that was received is passed to the output as **rx_out[7:0]**.

**3. UART Transmitter (uart_tx)**

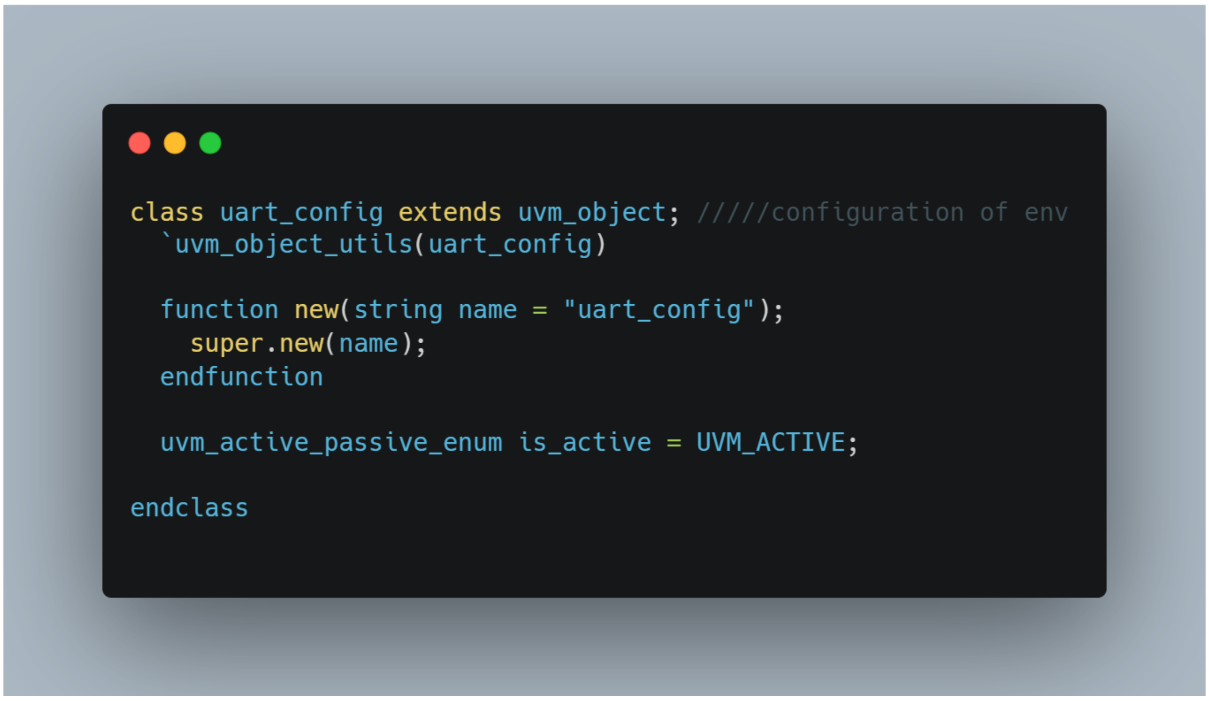
- The **uart_tx** block takes care of sending data. It converts the data from parallel (multiple bits sent at once) into serial format (one bit sent at a time).
- Inputs for the transmitter include:
    - **tx_start**: Triggers the start of data transmission.
    - **tx_data**: This is the actual 8-bit data that needs to be sent.
    - **length**, **parity_en**, **parity_type**, and **stop2**: Just like the receiver, these inputs allow for control over how many bits are sent, whether parity error checking is used, and how many stop bits to include.
- After the data is sent, the **tx_done** signal tells the system that the transmission is finished. If an error occurs during transmission, **tx_err** will flag it.

## UART VERIFICATION

### Enumerations (oper_mode)

- The typedef enum defines different operation modes for UART, with names like rand_baud_1_stop, length5wp, etc. Each name represents a different configuration for baud rate (speed), data length, parity, and stop bits.
- For example, rand_baud_1_stop = 0 refers to random baud rate with 1 stop bit.

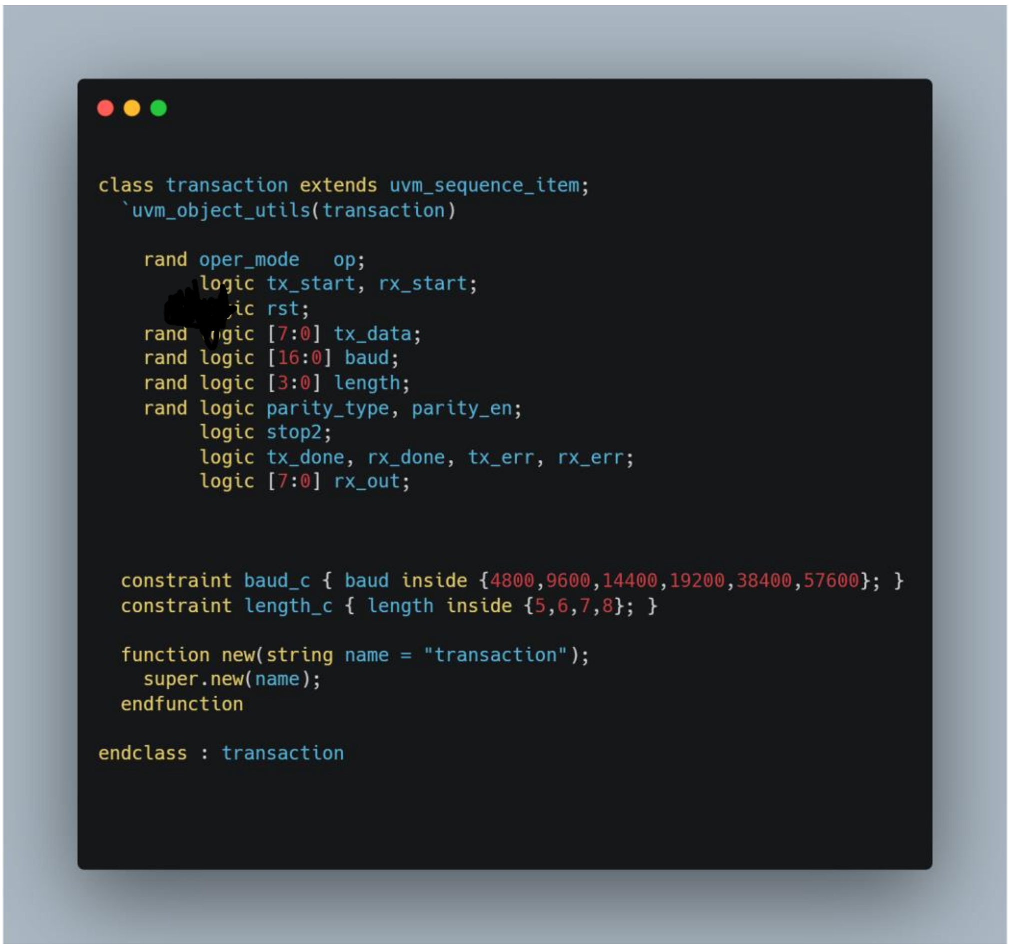
```
typedef enum bit [3:0]   {rand_baud_1_stop = 0, rand_length_1_stop = 1, length5wp = 2, length6wp = 3,
length7wp = 4, length8wp = 5, length5wop = 6, length6wop = 7, length7wop = 8, length8wop =
9,rand_baud_2_stop = 11, rand_length_2_stop = 12} oper_mode;
```



```
class uart_config extends uvm_object; /////configuration of env
  `uvm_object_utils(uart_config)

  function new(string name = "uart_config");
    super.new(name);
  endfunction

  uvm_active_passive_enum is_active = UVM_ACTIVE;

endclass
```

**Class uart_config**

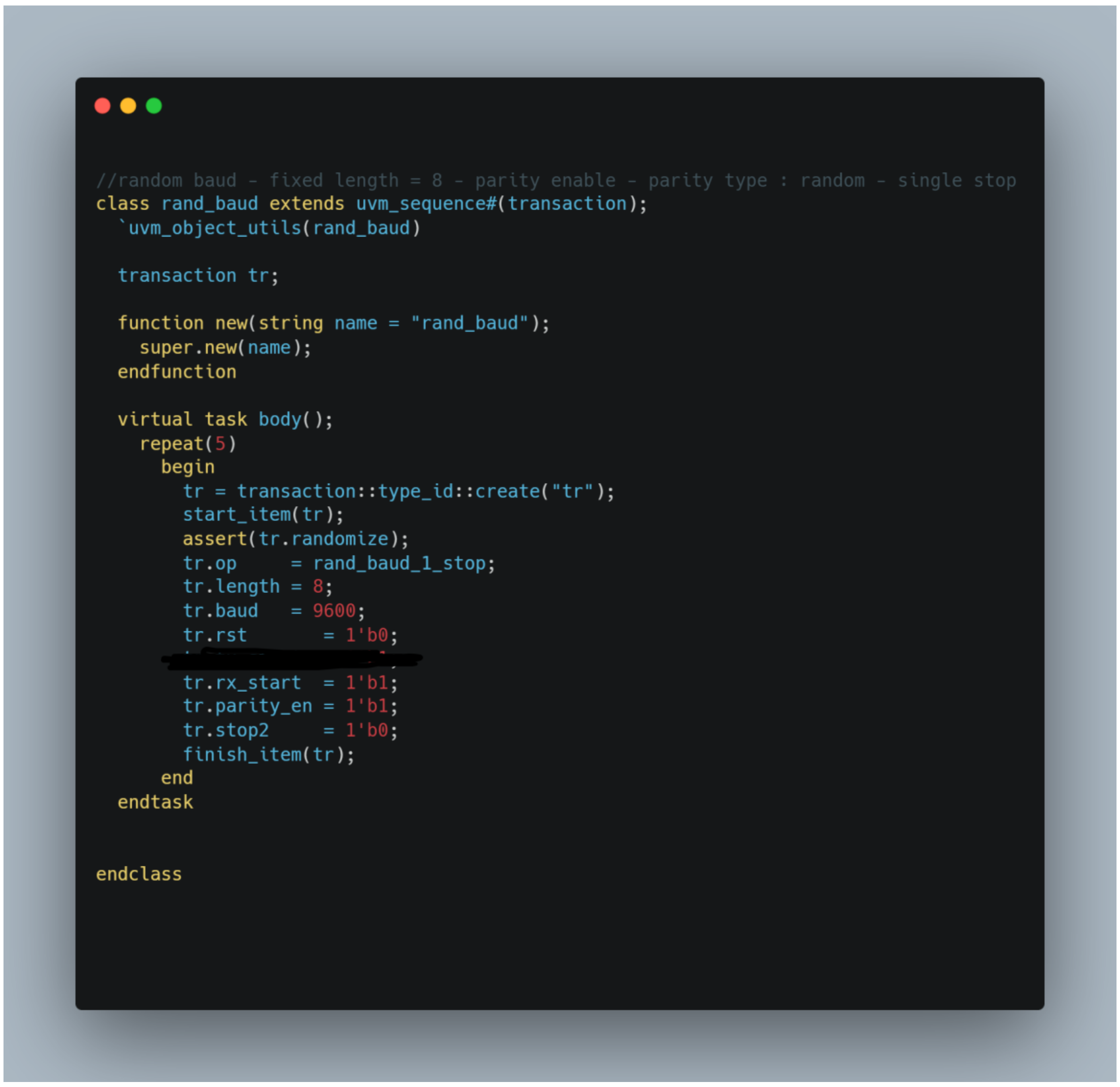
- This class is used to configure the UART (Universal Asynchronous Receiver-Transmitter) environment for the simulation.
- It extends uvm_object, meaning it inherits functionalities like reporting, copying, and comparisons.
- uvm_object_utils(uart_config): This macro helps register the class with UVM so UVM can automatically handle things like object creation.
- is_active = UVM_ACTIVE;: This sets the environment as active. An active environment means it drives the input signals to the design under test (DUT).



```
class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

    rand oper_mode   op;
         logic tx_start, rx_start;
    [redacted]ic rst;
    rand  ogic [7:0] tx_data;
    rand logic [16:0] baud;
    rand logic [3:0] length;
    rand logic parity_type, parity_en;
         logic stop2;
         logic tx_done, rx_done, tx_err, rx_err;
         logic [7:0] rx_out;



  constraint baud_c { baud inside {4800,9600,14400,19200,38400,57600}; }
  constraint length_c { length inside {5,6,7,8}; }

  function new(string name = "transaction");
    super.new(name);
  endfunction

endclass : transaction
```

**Classes for Randomized Sequences**

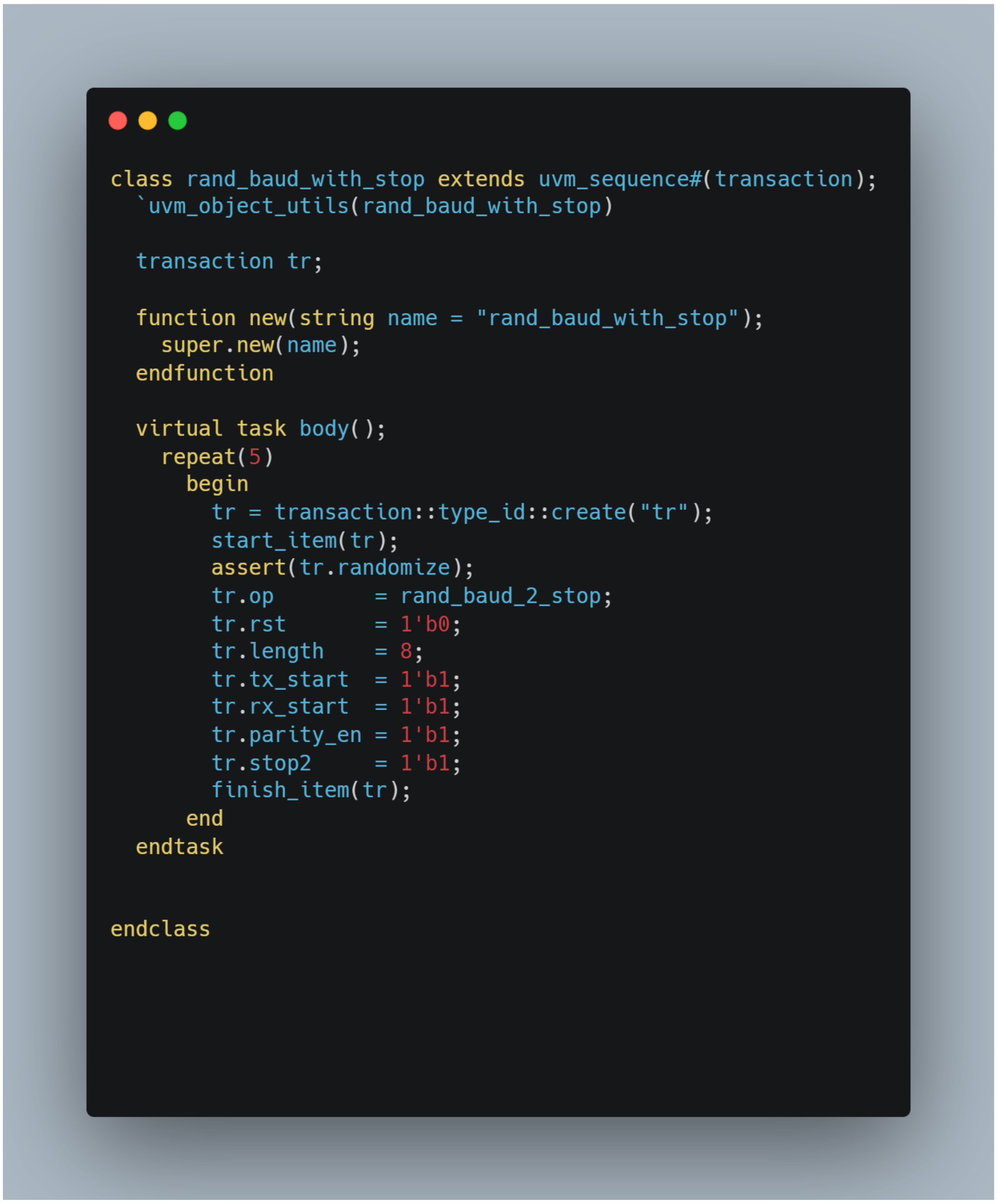
- Each class like rand_baud, rand_baud_with_stop, rand_baud_len5p, etc., represents different types of UART sequences with varying configurations.
- **rand_baud**: This sequence sets a fixed data length of 8 bits, a baud rate of 9600, enables parity, and uses a single stop bit.
- **rand_baud_with_stop**: Similar to rand_baud but uses two stop bits.
- **rand_baud_len5p**: Uses a 5-bit data length and enables parity.
- Each class repeats the sequence five times (repeat(5)), creating and randomizing a transaction object and setting its fields based on the specific sequence configuration.



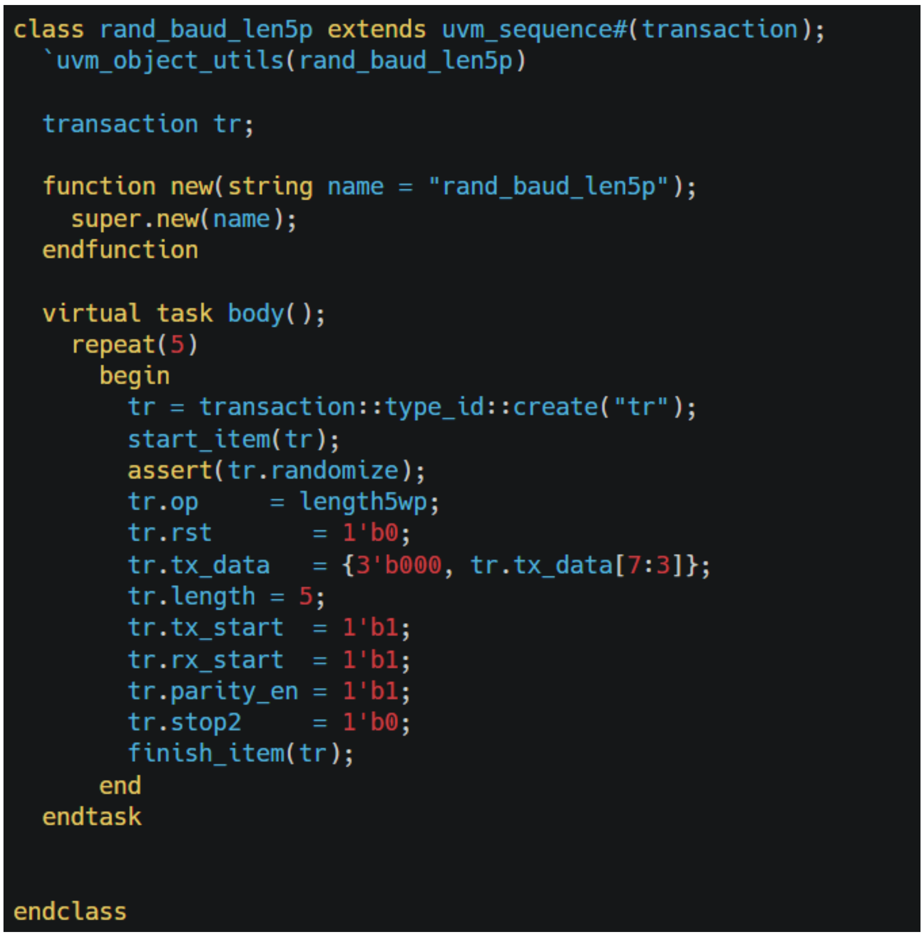
```
//random baud - fixed length = 8 - parity enable - parity type : random - single stop
class rand_baud extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud)

  transaction tr;

  function new(string name = "rand_baud");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op      = rand_baud_1_stop;
        tr.length = 8;
        tr.baud    = 9600;
        tr.rst       = 1'b0;
[redacted]
        tr.rx_start   = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2      = 1'b0;
        finish_item(tr);
      end
  endtask


endclass
```

Rand_baud → Length will be fixed baud will be random

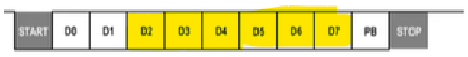
| | |
|---|---|
| Baud | : Random |
| Length | : Fixed |
| Parity | : Enable |
| Parity type | : Random |
| Stop | : Single |

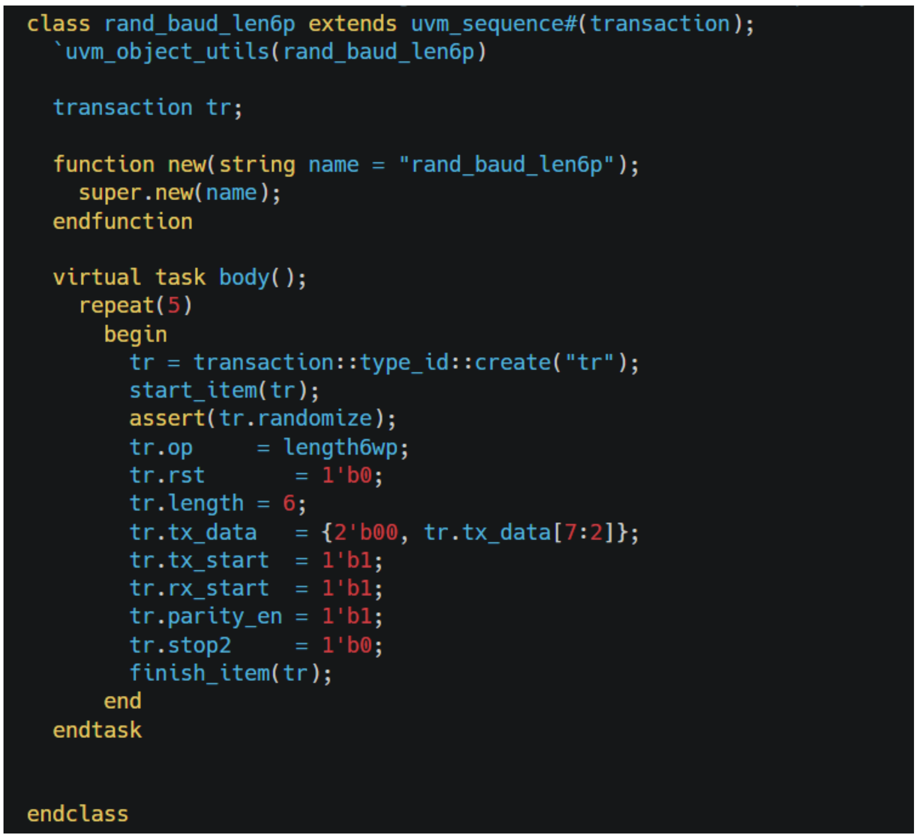


```
class rand_baud_with_stop extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_with_stop)

  transaction tr;

  function new(string name = "rand_baud_with_stop");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op        = rand_baud_2_stop;
        tr.rst       = 1'b0;
        tr.length    = 8;
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2     = 1'b1;
        finish_item(tr);
      end
  endtask


endclass
```

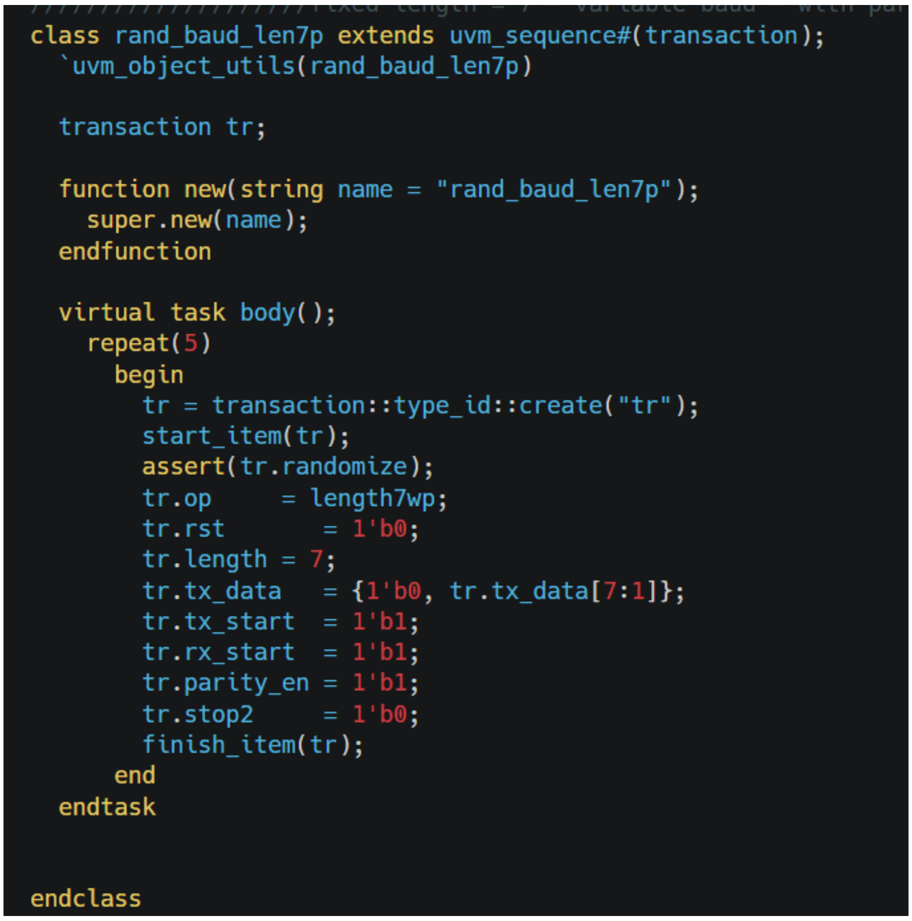


```
class rand_baud_len5p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len5p)

  transaction tr;

  function new(string name = "rand_baud_len5p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op     = length5wp;
        tr.rst       = 1'b0;
        tr.tx_data   = {3'b000, tr.tx_data[7:3]};
        tr.length = 5;
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2     = 1'b0;
        finish_item(tr);
      end
  endtask


endclass
```

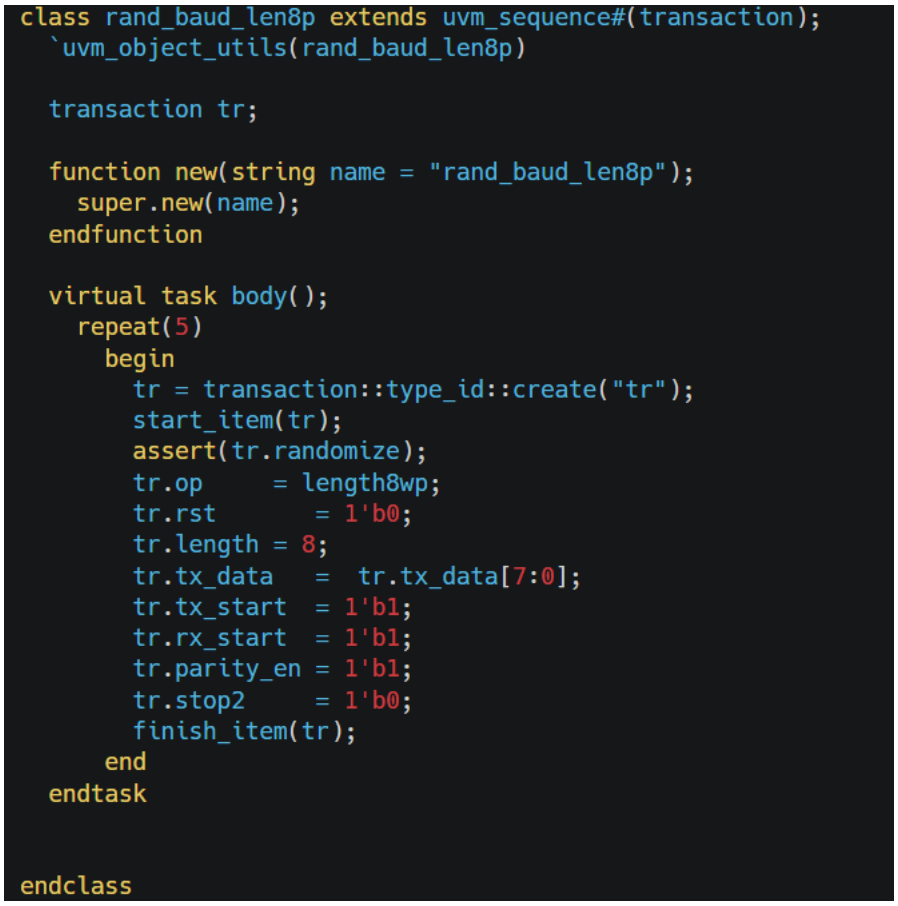




```
class rand_baud_len6p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len6p)

  transaction tr;

  function new(string name = "rand_baud_len6p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op     = length6wp;
        tr.rst       = 1'b0;
        tr.length = 6;
        tr.tx_data   = {2'b00, tr.tx_data[7:2]};
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2     = 1'b0;
        finish_item(tr);
      end
  endtask


endclass
```



```
class rand_baud_len7p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len7p)

  transaction tr;

  function new(string name = "rand_baud_len7p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op     = length7wp;
        tr.rst         = 1'b0;
        tr.length = 7;
        tr.tx_data   = {1'b0, tr.tx_data[7:1]};
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2     = 1'b0;
        finish_item(tr);
      end
  endtask


endclass
```



```
class rand_baud_len8p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len8p)

  transaction tr;

  function new(string name = "rand_baud_len8p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op     = length8wp;
        tr.rst         = 1'b0;
        tr.length = 8;
        tr.tx_data   =  tr.tx_data[7:0];
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
        tr.stop2     = 1'b0;
        finish_item(tr);
      end
  endtask


endclass
```

**rand_baud_len5p Sequence (Length = 5 bits, with Parity)**

This sequence sends UART frames where the data length is fixed at 5 bits, with parity enabled.

- **Purpose**: This sequence generates transactions for UART frames with 5-bit data, 1 start bit, 1 stop bit, and parity enabled. The baud rate is randomized for each frame.
- **Key Fields in the Transaction**:
    - tr.tx_data: The 5-bit data to be transmitted (upper bits are padded).
    - tr.length: Fixed to 5 bits.
    - tr.tx_start and tr.rx_start: Flags to indicate the start of transmission and reception.
    - tr.parity_en: Parity is enabled (used for error detection).
    - tr.stop2: Defines whether there are 1 or 2 stop bits (here, set to 1 stop bit).

**rand_baud_len6p Sequence (Length = 6 bits, with Parity)**

This sequence is almost identical to the rand_baud_len5p, except that the data length is 6 bits instead of 5.

- **Purpose**: Generates UART frames with 6-bit data, parity enabled, and 1 stop bit.
- **Key Difference**:
    - tr.length: Set to 6 bits.
    - tr.tx_data: Uses the upper 6 bits from tr.tx_data[7:2].

**rand_baud_len7p Sequence (Length = 7 bits, with Parity)**

Similarly, this sequence uses a data length of 7 bits. The fields are set accordingly to transmit 7-bit UART frames.

- **Purpose**: Generates UART frames with 7-bit data, parity, and 1 stop bit.
- **Key Difference**:

  - tr.length: Set to 7 bits.
  - tr.tx_data: Uses the upper 7 bits from tr.tx_data[7:1].

**rand_baud_len8p Sequence (Length = 8 bits, with Parity)**

This sequence handles the case where the data length is 8 bits, which is typical for UART communication.

- **Purpose**: Generates UART frames with 8-bit data, parity, and 1 stop bit.
- **Key Difference**:
  - tr.length: Set to 8 bits.
  - tr.tx_data: Uses the full 8 bits from tr.tx_data[7:0].

**Summary of the Sequences**

- Each sequence (rand_baud_len5p, rand_baud_len6p, rand_baud_len7p, rand_baud_len8p) defines UART communication with different fixed data lengths (5, 6, 7, or 8 bits).
- All sequences enable **parity** and use **1 stop bit**.
- These sequences generate random transactions with different configurations of data length and baud rate.
- The **repeat(5)** in the body() task means each sequence sends 5 transactions for testing.

## Driver

extends uvm_driver #(transaction) means that this class inherits all the functionality of the UVM driver, and it is parameterized by the transaction type, which will define the structure of the data being sent to the DUT.

The macro uvm_component_utils(driver) allows the UVM factory to create instances of this class and enables utilities like configuration and automation.

**vif** is a virtual interface handle. It is connected to the DUT's actual UART signals during simulation. This allows the driver to manipulate signals like rst, tx_data, baud, etc.

The driver retrieves the virtual interface vif from the UVM configuration database (uvm_config_db). This ensures the driver is connected to the DUT's interface.

If the interface cannot be accessed, an error is generated using the uvm_error macro.

**Reset DUT (reset_dut):**

This task asserts the reset signal (vif.rst = 1'b1) for 5 clock cycles. During this period, all the control signals like tx_start, rx_start, tx_data, etc., are initialized to their default (inactive) values.

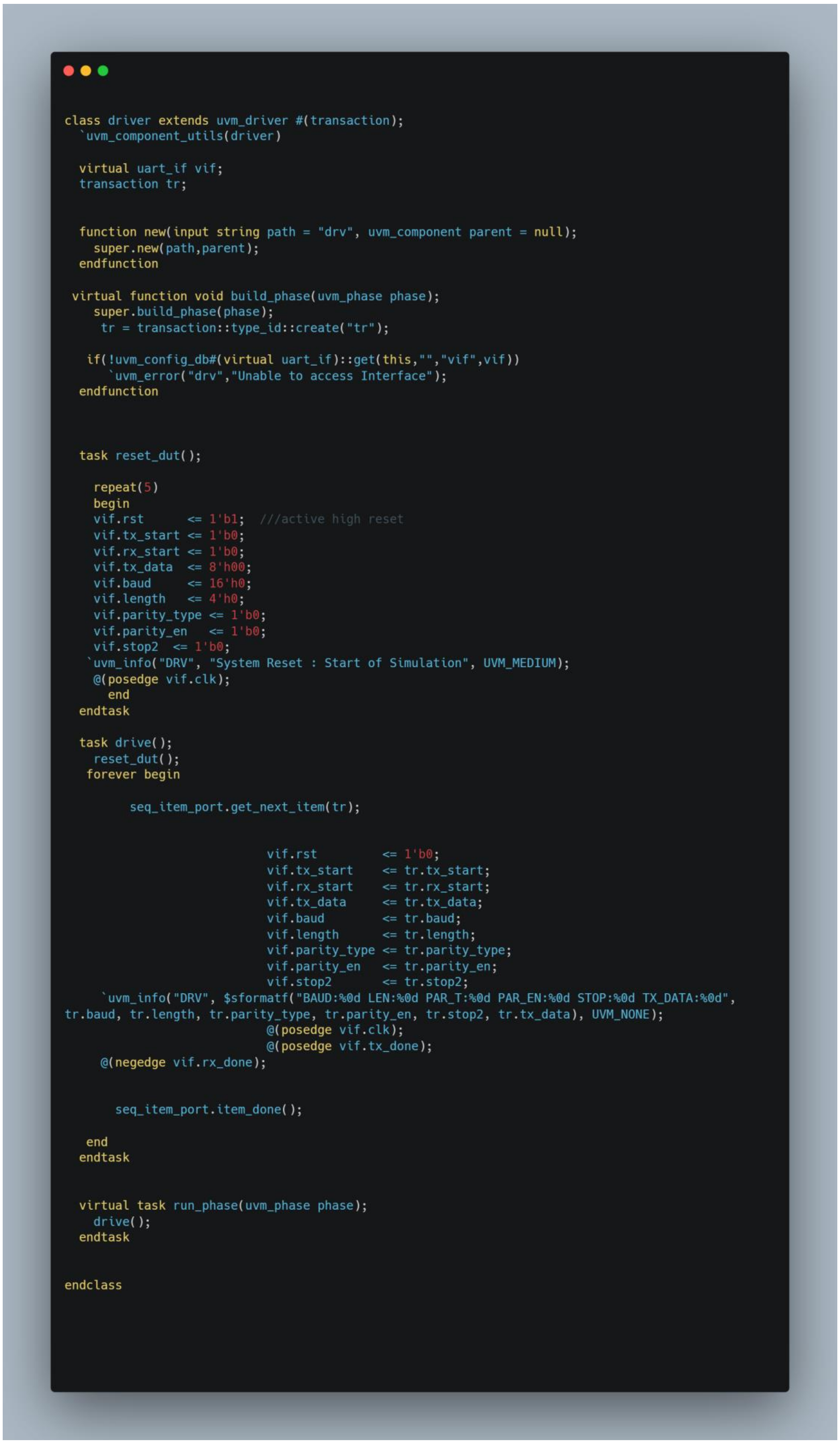
**Drive Task:**

The task assigns the values in the transaction object (tr) to the DUT's signals through the virtual interface (vif). These signals include tx_start, rx_start, tx_data, baud, etc.

**Run Phase**

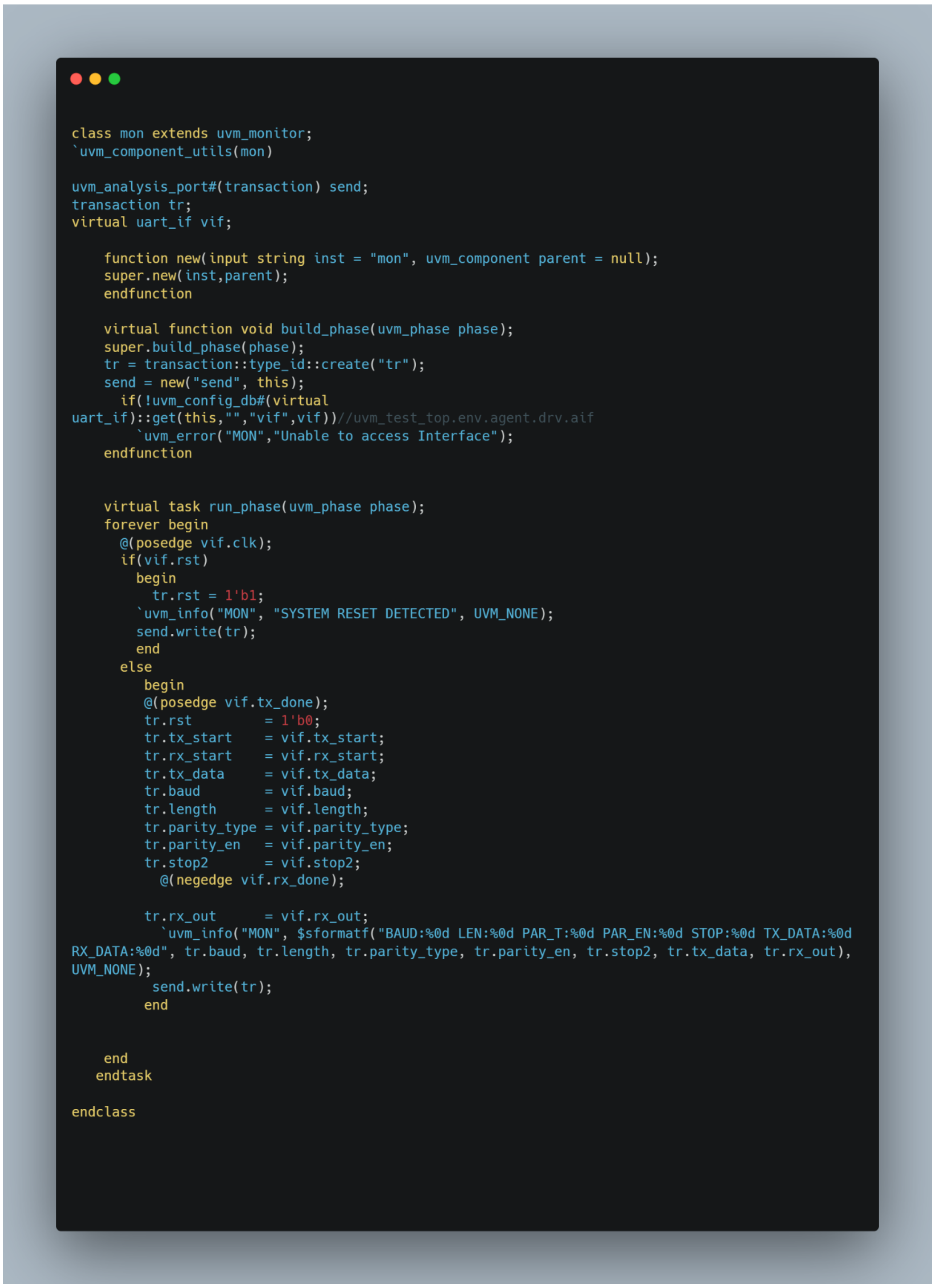
The run_phase is where the actual test begins. It simply calls the drive() task, starting the process of driving transactions on the DUT.

This task is automatically called by the UVM framework when the simulation reaches the run phase, making it the central point of operation for the driver.

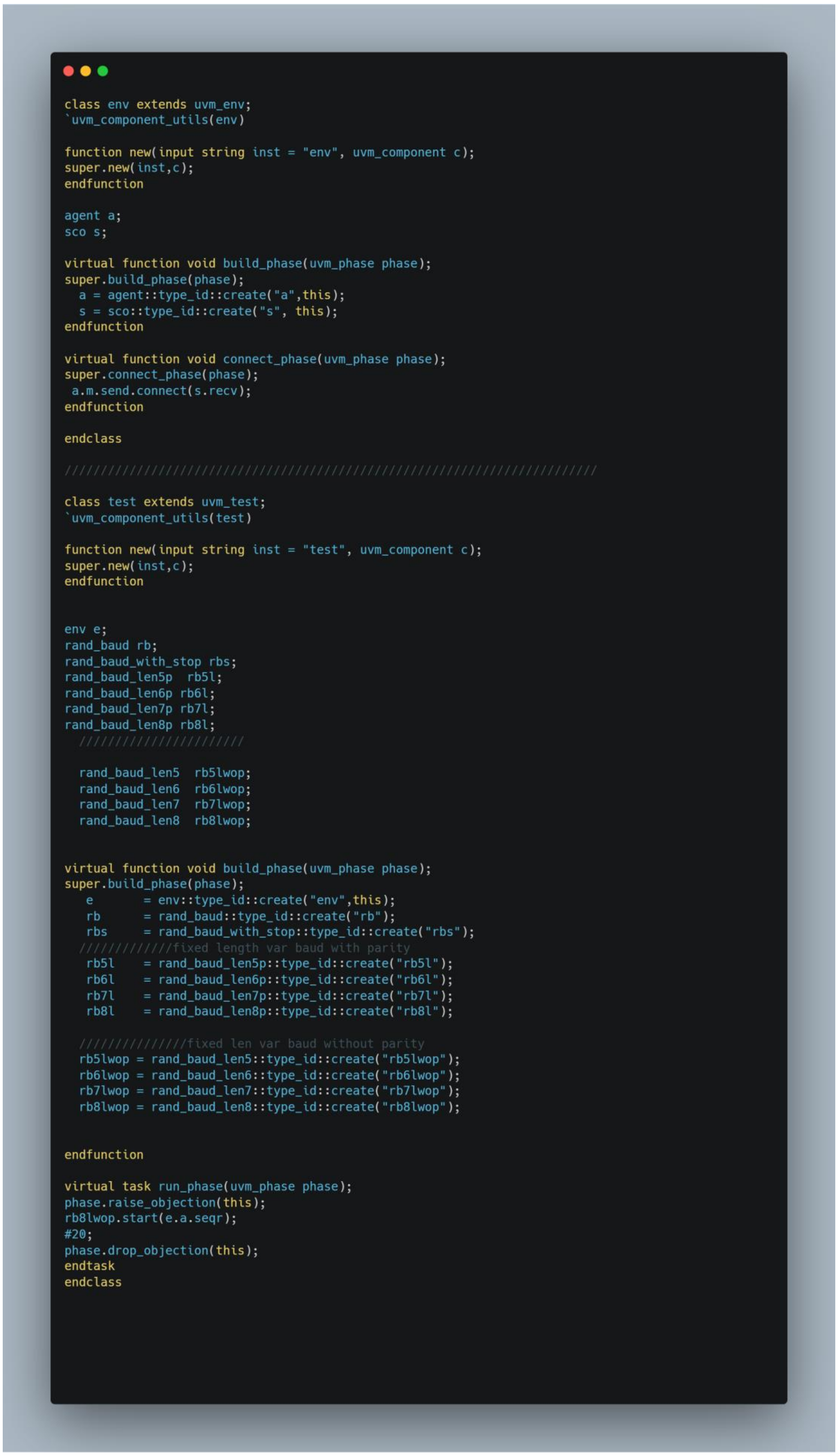


```
class driver extends uvm_driver #(transaction);
  `uvm_component_utils(driver)

  virtual uart_if vif;
  transaction tr;


  function new(input string path = "drv", uvm_component parent = null);
    super.new(path,parent);
  endfunction

 virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     tr = transaction::type_id::create("tr");

   if(!uvm_config_db#(virtual uart_if)::get(this,"","vif",vif))
      `uvm_error("drv","Unable to access Interface");
  endfunction



  task reset_dut();

    repeat(5)
    begin
    vif.rst      <= 1'b1;  ///active high reset
    vif.tx_start <= 1'b0;
    vif.rx_start <= 1'b0;
    vif.tx_data  <= 8'h00;
    vif.baud     <= 16'h0;
    vif.length   <= 4'h0;
    vif.parity_type <= 1'b0;
    vif.parity_en  <= 1'b0;
    vif.stop2  <= 1'b0;
   `uvm_info("DRV", "System Reset : Start of Simulation", UVM_MEDIUM);
    @(posedge vif.clk);
      end
  endtask

  task drive();
    reset_dut();
   forever begin

         seq_item_port.get_next_item(tr);


                            vif.rst         <= 1'b0;
                            vif.tx_start    <= tr.tx_start;
                            vif.rx_start    <= tr.rx_start;
                            vif.tx_data     <= tr.tx_data;
                            vif.baud        <= tr.baud;
                            vif.length      <= tr.length;
                            vif.parity_type <= tr.parity_type;
                            vif.parity_en   <= tr.parity_en;
                            vif.stop2       <= tr.stop2;
     `uvm_info("DRV", $sformatf("BAUD:%0d LEN:%0d PAR_T:%0d PAR_EN:%0d STOP:%0d TX_DATA:%0d",
tr.baud, tr.length, tr.parity_type, tr.parity_en, tr.stop2, tr.tx_data), UVM_NONE);
                            @(posedge vif.clk);
                            @(posedge vif.tx_done);
     @(negedge vif.rx_done);


       seq_item_port.item_done();

   end
  endtask


  virtual task run_phase(uvm_phase phase);
    drive();
  endtask


endclass
```

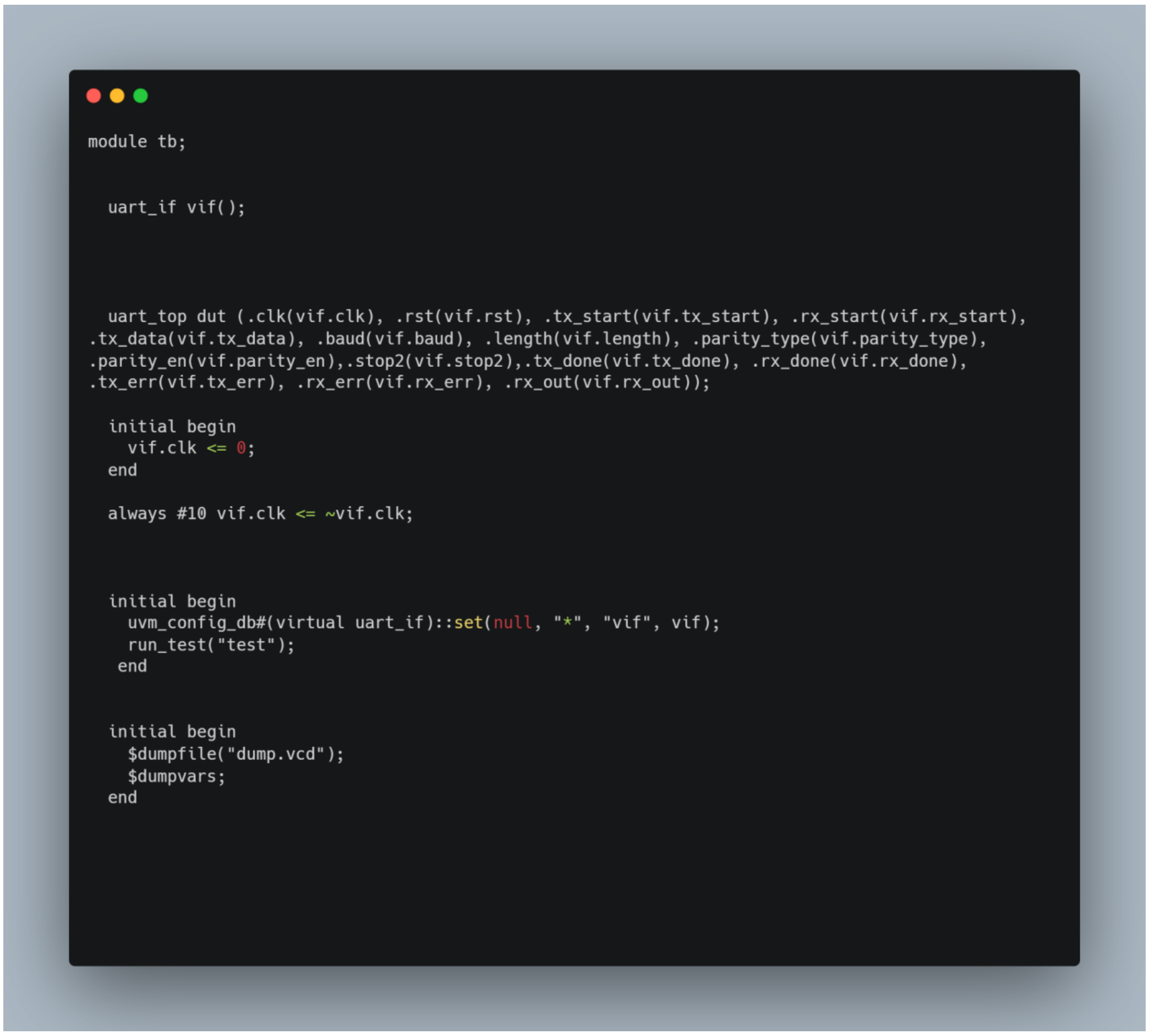
## Monitor



```
class mon extends uvm_monitor;
`uvm_component_utils(mon)

uvm_analysis_port#(transaction) send;
transaction tr;
virtual uart_if vif;

    function new(input string inst = "mon", uvm_component parent = null);
    super.new(inst,parent);
    endfunction

    virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    tr = transaction::type_id::create("tr");
    send = new("send", this);
      if(!uvm_config_db#(virtual
uart_if)::get(this,"","vif",vif))//uvm_test_top.env.agent.drv.aif
        `uvm_error("MON","Unable to access Interface");
    endfunction


    virtual task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.clk);
      if(vif.rst)
        begin
          tr.rst = 1'b1;
        `uvm_info("MON", "SYSTEM RESET DETECTED", UVM_NONE);
        send.write(tr);
        end
      else
         begin
         @(posedge vif.tx_done);
         tr.rst         = 1'b0;
         tr.tx_start    = vif.tx_start;
         tr.rx_start    = vif.rx_start;
         tr.tx_data     = vif.tx_data;
         tr.baud        = vif.baud;
         tr.length      = vif.length;
         tr.parity_type = vif.parity_type;
         tr.parity_en   = vif.parity_en;
         tr.stop2       = vif.stop2;
           @(negedge vif.rx_done);

         tr.rx_out      = vif.rx_out;
           `uvm_info("MON", $sformatf("BAUD:%0d LEN:%0d PAR_T:%0d PAR_EN:%0d STOP:%0d TX_DATA:%0d
RX_DATA:%0d", tr.baud, tr.length, tr.parity_type, tr.parity_en, tr.stop2, tr.tx_data, tr.rx_out),
UVM_NONE);
          send.write(tr);
         end


    end
   endtask

endclass
```

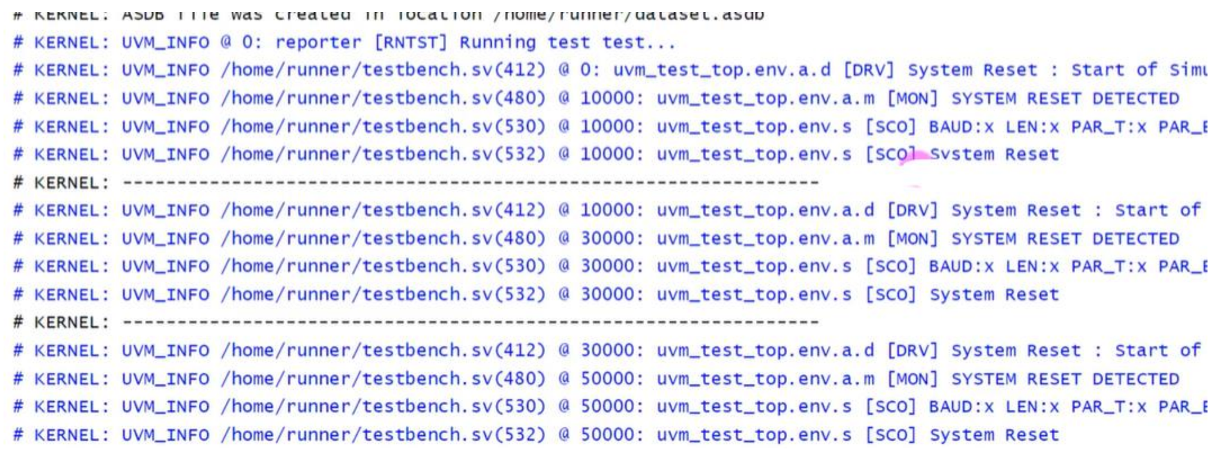


```
class env extends uvm_env;
`uvm_component_utils(env)

function new(input string inst = "env", uvm_component c);
super.new(inst,c);
endfunction

agent a;
sco s;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  a = agent::type_id::create("a",this);
  s = sco::type_id::create("s", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
 a.m.send.connect(s.recv);
endfunction

endclass

//////////////////////////////////////////////////////////////////////////

class test extends uvm_test;
`uvm_component_utils(test)

function new(input string inst = "test", uvm_component c);
super.new(inst,c);
endfunction


env e;
rand_baud rb;
rand_baud_with_stop rbs;
rand_baud_len5p  rb5l;
rand_baud_len6p rb6l;
rand_baud_len7p rb7l;
rand_baud_len8p rb8l;
  //////////////////////

  rand_baud_len5  rb5lwop;
  rand_baud_len6  rb6lwop;
  rand_baud_len7  rb7lwop;
  rand_baud_len8  rb8lwop;


virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
   e       = env::type_id::create("env",this);
   rb      = rand_baud::type_id::create("rb");
   rbs     = rand_baud_with_stop::type_id::create("rbs");
  /////////////fixed length var baud with parity
   rb5l    = rand_baud_len5p::type_id::create("rb5l");
   rb6l    = rand_baud_len6p::type_id::create("rb6l");
   rb7l    = rand_baud_len7p::type_id::create("rb7l");
   rb8l    = rand_baud_len8p::type_id::create("rb8l");

  //////////////fixed len var baud without parity
  rb5lwop = rand_baud_len5::type_id::create("rb5lwop");
  rb6lwop = rand_baud_len6::type_id::create("rb6lwop");
  rb7lwop = rand_baud_len7::type_id::create("rb7lwop");
  rb8lwop = rand_baud_len8::type_id::create("rb8lwop");


endfunction

virtual task run_phase(uvm_phase phase);
phase.raise_objection(this);
rb8lwop.start(e.a.seqr);
#20;
phase.drop_objection(this);
endtask
endclass
```



```
module tb;


  uart_if vif();



  uart_top dut (.clk(vif.clk), .rst(vif.rst), .tx_start(vif.tx_start), .rx_start(vif.rx_start),
.tx_data(vif.tx_data), .baud(vif.baud), .length(vif.length), .parity_type(vif.parity_type),
.parity_en(vif.parity_en),.stop2(vif.stop2),.tx_done(vif.tx_done), .rx_done(vif.rx_done),
.tx_err(vif.tx_err), .rx_err(vif.rx_err), .rx_out(vif.rx_out));

  initial begin
    vif.clk <= 0;
  end

  always #10 vif.clk <= ~vif.clk;



  initial begin
    uvm_config_db#(virtual uart_if)::set(null, "*", "vif", vif);
    run_test("test");
   end


  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end
```

## Output



```
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: UVM_INFO @ 0: reporter [RNTST] Running test test...
# KERNEL: UVM_INFO /home/runner/testbench.sv(412) @ 0: uvm_test_top.env.a.d [DRV] System Reset : Start of Simu
# KERNEL: UVM_INFO /home/runner/testbench.sv(480) @ 10000: uvm_test_top.env.a.m [MON] SYSTEM RESET DETECTED
# KERNEL: UVM_INFO /home/runner/testbench.sv(530) @ 10000: uvm_test_top.env.s [SCO] BAUD:x LEN:x PAR_T:x PAR_E
# KERNEL: UVM_INFO /home/runner/testbench.sv(532) @ 10000: uvm_test_top.env.s [SCO] System Reset
# KERNEL: ----------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/testbench.sv(412) @ 10000: uvm_test_top.env.a.d [DRV] System Reset : Start of
# KERNEL: UVM_INFO /home/runner/testbench.sv(480) @ 30000: uvm_test_top.env.a.m [MON] SYSTEM RESET DETECTED
# KERNEL: UVM_INFO /home/runner/testbench.sv(530) @ 30000: uvm_test_top.env.s [SCO] BAUD:x LEN:x PAR_T:x PAR_E
# KERNEL: UVM_INFO /home/runner/testbench.sv(532) @ 30000: uvm_test_top.env.s [SCO] System Reset
# KERNEL: ----------------------------------------------------------------
# KERNEL: UVM_INFO /home/runner/testbench.sv(412) @ 30000: uvm_test_top.env.a.d [DRV] System Reset : Start of
# KERNEL: UVM_INFO /home/runner/testbench.sv(480) @ 50000: uvm_test_top.env.a.m [MON] SYSTEM RESET DETECTED
# KERNEL: UVM_INFO /home/runner/testbench.sv(530) @ 50000: uvm_test_top.env.s [SCO] BAUD:x LEN:x PAR_T:x PAR_E
# KERNEL: UVM_INFO /home/runner/testbench.sv(532) @ 50000: uvm_test_top.env.s [SCO] System Reset
```

```
`include "uvm_macros.svh"
        import uvm_pkg::*;



        typedef enum bit [1:0] {reset_asserted = 0, random_baud = 1} oper_mode;

        //////////////////////////////////////////////////////////////////////

        class transaction extends uvm_sequence_item;
          `uvm_object_utils(transaction)

            oper_mode   oper;
            rand logic [16:0] baud;
            logic tx_clk;
            real period;

          constraint baud_c { baud inside {4800,9600,14400,19200,38400,57600}; }



          function new(string name = "transaction");
            super.new(name);
          endfunction


        endclass
        //////////////////////////////////////////////////////////


        class reset_clk extends uvm_sequence#(transaction);
          `uvm_object_utils(reset_clk)
```

```
 transaction tr;

 function new(string name = "reset_clk");
  super.new(name);
 endfunction

 virtual task body();
  repeat(5)
    begin
     tr = transaction::type_id::create("tr");
     start_item(tr);
     assert(tr.randomize);
     tr.oper = reset_asserted;
     finish_item(tr);
    end
 endtask


endclass


////////////////////////////////////////////////////////////////////////
class variable_baud extends uvm_sequence#(transaction);
 `uvm_object_utils(variable_baud)

 transaction tr;

 function new(string name = "variable_baud");
  super.new(name);
 endfunction
 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.oper = random_baud;
    finish_item(tr);
   end
 endtask


endclass

////////////////////////////////////////////////////////////////////



class driver extends uvm_driver #(transaction);
 `uvm_component_utils(driver)

 virtual clk_if vif;
 transaction tr;



 function new(input string path = "drv", uvm_component parent = null);
  super.new(path,parent);
 endfunction

 virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
   tr = transaction::type_id::create("tr");
 if(!uvm_config_db#(virtual clk_if)::get(this,"","vif",vif))
    ` uvm_error("drv","Unable to access Interface");
 endfunction



 virtual task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(tr);

    if(tr.oper == reset_asserted)
        begin
          vif.rst <= 1'b1;
         @(posedge vif.clk);
        end
    else if (tr.oper == random_baud )
        begin
         ` uvm_info("DRV",$sformatf("Baud : %0d",tr.baud), UVM_NONE);
         vif.rst  <= 1'b0;
         vif.baud <= tr.baud;
         @(posedge vif.clk);
         @(posedge vif.tx_clk);
         @(posedge vif.tx_clk);
        end

    seq_item_port.item_done();
  end
 endtask

endclass
```

```
////////////////////////////////////////////////////////////
class mon extends uvm_monitor;
`uvm_component_utils(mon)

uvm_analysis_port#(transaction) send;
transaction tr;
virtual clk_if vif;
real ton  = 0;
real toff = 0;

   function new(input string inst = "mon", uvm_component parent = null);
  super.new(inst,parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
   send = new("send", this);

   tr = transaction::type_id::create("tr");

   if(!uvm_config_db#(virtual clk_if)::get(this,"","vif",vif))
     `uvm_error("mon","Unable to access Interface");
  endfunction



  virtual task run_phase(uvm_phase phase);
  forever begin
   @(posedge vif.clk);
   if(vif.rst)
    begin
     tr.oper = reset_asserted;
```

```
        ton    = 0;
        toff   = 0;
       ` uvm_info("MON", "SYSTEM RESET DETECTED", UVM_NONE);
       send.write(tr);
       end
     else
       begin
        tr.baud = vif.baud;
        tr.oper = random_baud;
        ton     = 0;
        toff    = 0;
         @(posedge vif.tx_clk);
         ton = $realtime;
         @(posedge vif.tx_clk);
         toff = $realtime;
         tr.period = toff - ton;

         ` uvm_info("MON",$sformatf("Baud : %0d Period:%0f",tr.baud,tr.period),
UVM_NONE);
         send.write(tr);

       end

    end
 endtask

endclass

//////////////////////////////////////////////////////////////////////////////////////
 class sco extends uvm_scoreboard;
 ` uvm_component_utils(sco)
 real count = 0;

 real baudcount = 0;

uvm_analysis_imp#(transaction,sco) recv;


  function new(input string inst = "sco", uvm_component parent = null);

  super.new(inst,parent);

  endfunction


  virtual function void build_phase(uvm_phase phase);

  super.build_phase(phase);

  recv = new("recv", this);

  endfunction



virtual function void write(transaction tr);

  count =  tr.period / 20;

  baudcount = count;

  `uvm_info("SCO", $sformatf("BAUD:%0d count:%0f bcount:%0f", tr.baud,count,
baudcount), UVM_NONE);

  case(tr.baud)


    4800: begin

      if(baudcount == 10418)

        `uvm_info("SCO", "TEST PASSED", UVM_NONE)

      else

        `uvm_error("SCO" , "TEST FAILED")

    end



    9600: begin
      if(baudcount == 5210)
        `uvm_info("SCO", "TEST PASSED", UVM_NONE)
      else
        `uvm_error("SCO" , "TEST FAILED")

    end

    14400: begin
      if(baudcount == 3474)
        `uvm_info("SCO", "TEST PASSED", UVM_NONE)
      else
        `uvm_error("SCO" , "TEST FAILED")
    end

    19200: begin
      if(baudcount == 2606)
        `uvm_info("SCO", "TEST PASSED", UVM_NONE)
      else
        `uvm_error("SCO" , "TEST FAILED")

    end

    38400: begin
      if(baudcount == 1304)
        `uvm_info("SCO", "TEST PASSED", UVM_NONE)
      else
        `uvm_error("SCO" , "TEST FAILED")

    end

    57600: begin
```

```
        if(baudcount == 870)
          ` uvm_info("SCO", "TEST PASSED", UVM_NONE)
        else
          ` uvm_error("SCO" , "TEST FAILED")
      end


        endcase
   endfunction


endclass

////////////////////////////////////////////////////////////////////////////////




class agent extends uvm_agent;
` uvm_component_utils(agent)




function new(input string inst = "agent", uvm_component parent = null);
super.new(inst,parent);
endfunction


 driver d;
 uvm_sequencer#(transaction) seqr;
 mon m;



virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
```

```
  m = mon::type_id::create("m",this);
  d = driver::type_id::create("d",this);
  seqr = uvm_sequencer#(transaction)::type_id::create("seqr", this);

endfunction

virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
   d.seq_item_port.connect(seqr.seq_item_export);
endfunction

endclass

//////////////////////////////////////////////////////////////////////////

class env extends uvm_env;
` uvm_component_utils(env)

function new(input string inst = "env", uvm_component c);
super.new(inst,c);
endfunction

agent a;
sco s;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
 a = agent::type_id::create("a",this);
 s = sco::type_id::create("s", this);
endfunction
```

```
virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
 a.m.send.connect(s.recv);
endfunction

endclass
```

## UART Verification

```
`include "uvm_macros.svh"
 import uvm_pkg::*;

/////////////build the seq for random length with and without priority

////////////////////////////////////////////////////////////////////////////////
class uart_config extends uvm_object; /////configuration of env
  `uvm_object_utils(uart_config)

  function new(string name = "uart_config");
    super.new(name);
  endfunction

  uvm_active_passive_enum is_active = UVM_ACTIVE;

endclass

/////////////////////////////////////////////////////////
```

```
typedef enum bit [3:0]   {rand_baud_1_stop = 0, rand_length_1_stop = 1,
length5wp = 2, length6wp = 3, length7wp = 4, length8wp = 5, length5wop = 6,
length6wop = 7, length7wop = 8, length8wop = 9,rand_baud_2_stop = 11,
rand_length_2_stop = 12} oper_mode;



class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

   rand oper_mode   op;
       logic tx_start, rx_start;
       logic rst;
   rand logic [7:0] tx_data;
   rand logic [16:0] baud;
   rand logic [3:0] length;
   rand logic parity_type, parity_en;
       logic stop2;
       logic tx_done, rx_done, tx_err, rx_err;
       logic [7:0] rx_out;




  constraint baud_c { baud inside {4800,9600,14400,19200,38400,57600}; }
  constraint length_c { length inside {5,6,7,8}; }

  function new(string name = "transaction");
   super.new(name);
  endfunction

endclass : transaction
```

```
////////////////////////////////////////////////////////////////////



///////////////////random baud - fixed length = 8 - parity enable - parity type :
random - single stop
class rand_baud extends uvm_sequence#(transaction);
 ` uvm_object_utils(rand_baud)

 transaction tr;

 function new(string name = "rand_baud");
  super.new(name);
 endfunction

 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op    = rand_baud_1_stop;
    tr.length = 8;
    tr.baud   = 9600;
    tr.rst     = 1'b0;
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b1;
    tr.stop2    = 1'b0;
```

```
      finish_item(tr);
    end
  endtask



endclass
///////////////////random baud - fixed length = 8 - parity enable - parity type :
random - two stop
//////////////////////////////////////////////////////////
class rand_baud_with_stop extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_with_stop)

  transaction tr;

  function new(string name = "rand_baud_with_stop");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op       = rand_baud_2_stop;
        tr.rst      = 1'b0;
        tr.length    = 8;
        tr.tx_start  = 1'b1;
        tr.rx_start  = 1'b1;
        tr.parity_en = 1'b1;
      tr.stop2    = 1'b1;
      finish_item(tr);
    end
  endtask



endclass

/////////////////////////////////////////////////////////
////////////////////fixed length = 5 - variable baud - with parity
class rand_baud_len5p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len5p)

  transaction tr;

  function new(string name = "rand_baud_len5p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
        assert(tr.randomize);
        tr.op     = length5wp;
        tr.rst      = 1'b0;
        tr.tx_data   = {3'b000, tr.tx_data[7:3]};
        tr.length = 5;
      tr.tx_start  = 1'b1;
      tr.rx_start  = 1'b1;
      tr.parity_en = 1'b1;
      tr.stop2     = 1'b0;
      finish_item(tr);
    end
  endtask


endclass




///////////////////////////////////////////////////////
//////////////////fixed length = 6 - variable baud - with parity
class rand_baud_len6p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len6p)

  transaction tr;

  function new(string name = "rand_baud_len6p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
      begin
        tr = transaction::type_id::create("tr");
        start_item(tr);
      assert(tr.randomize);
      tr.op     = length6wp;
      tr.rst      = 1'b0;
      tr.length = 6;
      tr.tx_data   = {2'b00, tr.tx_data[7:2]};
      tr.tx_start  = 1'b1;
      tr.rx_start  = 1'b1;
      tr.parity_en = 1'b1;
      tr.stop2     = 1'b0;
      finish_item(tr);
    end
  endtask


endclass

//////////////////////////////////////////////////////////
///////////////////fixed length = 7 - variable baud - with parity
class rand_baud_len7p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len7p)

  transaction tr;

  function new(string name = "rand_baud_len7p");
    super.new(name);
  endfunction

  virtual task body();
    repeat(5)
    begin
      tr = transaction::type_id::create("tr");
      start_item(tr);
      assert(tr.randomize);
      tr.op     = length7wp;
      tr.rst      = 1'b0;
      tr.length = 7;
      tr.tx_data   = {1'b0, tr.tx_data[7:1]};
      tr.tx_start  = 1'b1;
      tr.rx_start  = 1'b1;
      tr.parity_en = 1'b1;
      tr.stop2    = 1'b0;
      finish_item(tr);
    end
  endtask



endclass


////////////////////////////////////////////////////////
///////////////////fixed length = 8 - variable baud - with parity
class rand_baud_len8p extends uvm_sequence#(transaction);
  `uvm_object_utils(rand_baud_len8p)


  transaction tr;


  function new(string name = "rand_baud_len8p");
    super.new(name);
  endfunction
```

```
 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op    = length8wp;
    tr.rst     = 1'b0;
    tr.length = 8;
    tr.tx_data   =  tr.tx_data[7:0];
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b1;
    tr.stop2    = 1'b0;
    finish_item(tr);
   end
 endtask


endclass

////////////////////////////////////////////////////////////
////////////////////fixed length = 5 - variable baud - without parity
class rand_baud_len5 extends uvm_sequence#(transaction);
 `uvm_object_utils(rand_baud_len5)

 transaction tr;
 function new(string name = "rand_baud_len5");
  super.new(name);
 endfunction

 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op     = length5wop;
    tr.rst      = 1'b0;
    tr.length = 5;
    tr.tx_data   = {3'b000, tr.tx_data[7:3]};
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b0;
    tr.stop2     = 1'b0;
    finish_item(tr);
   end
 endtask


endclass


////////////////////////////////////////////////////////
/////////////////////fixed length = 6- variable baud - without parity
class rand_baud_len6 extends uvm_sequence#(transaction);
```

```
 `uvm_object_utils(rand_baud_len6)

 transaction tr;

 function new(string name = "rand_baud_len6");
  super.new(name);
 endfunction

 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op     = length6wop;
    tr.rst      = 1'b0;
    tr.length = 6;
    tr.tx_data   = {2'b00, tr.tx_data[7:2]};
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b0;
    tr.stop2     = 1'b0;
    finish_item(tr);
   end
 endtask


endclass
```

```
////////////////////////////////////////////////////////
///////////////////fixed length = 7- variable baud - without parity
class rand_baud_len7 extends uvm_sequence#(transaction);
 `uvm_object_utils(rand_baud_len7)


 transaction tr;


 function new(string name = "rand_baud_len7");
  super.new(name);
 endfunction


 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op    = length7wop;
    tr.rst     = 1'b0;
    tr.length = 7;
    tr.tx_data   = {1'b0, tr.tx_data[7:1]};
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b0;
    tr.stop2    = 1'b0;
    finish_item(tr);
   end
 endtask
```

```
endclass



////////////////////////////////////////////////////////
////////////////////fixed length = 8 - variable baud - without parity
class rand_baud_len8 extends uvm_sequence#(transaction);
 `uvm_object_utils(rand_baud_len8)

 transaction tr;

 function new(string name = "rand_baud_len8");
  super.new(name);
 endfunction

 virtual task body();
  repeat(5)
   begin
    tr = transaction::type_id::create("tr");
    start_item(tr);
    assert(tr.randomize);
    tr.op    = length8wop;
    tr.rst     = 1'b0;
    tr.length = 8;
    tr.tx_data   =  tr.tx_data[7:0];
    tr.tx_start  = 1'b1;
    tr.rx_start  = 1'b1;
    tr.parity_en = 1'b0;
    tr.stop2    = 1'b0;
```

```
      finish_item(tr);
    end
  endtask


endclass

//////////////////////////////////////////////////////////////////////////////////////




class driver extends uvm_driver #(transaction);
  ` uvm_component_utils(driver)

  virtual uart_if vif;
  transaction tr;


  function new(input string path = "drv", uvm_component parent = null);
    super.new(path,parent);
  endfunction

 virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     tr = transaction::type_id::create("tr");

   if(!uvm_config_db#(virtual uart_if)::get(this,"","vif",vif))
      ` uvm_error("drv","Unable to access Interface");
```

```
endfunction




task reset_dut();


  repeat(5)
  begin
  vif.rst     <= 1'b1;  ///active high reset
  vif.tx_start <= 1'b0;
  vif.rx_start <= 1'b0;
  vif.tx_data  <= 8'h00;
  vif.baud    <= 16'h0;
  vif.length   <= 4'h0;
  vif.parity_type <= 1'b0;
  vif.parity_en   <= 1'b0;
  vif.stop2  <= 1'b0;
  `uvm_info("DRV", "System Reset : Start of Simulation", UVM_MEDIUM);
  @(posedge vif.clk);
    end
endtask


task drive();
  reset_dut();
 forever begin


     seq_item_port.get_next_item(tr);
```

```
                    vif.rst       <= 1'b0;
                    vif.tx_start   <= tr.tx_start;
                    vif.rx_start   <= tr.rx_start;
                    vif.tx_data    <= tr.tx_data;
                    vif.baud      <= tr.baud;
                    vif.length     <= tr.length;
                    vif.parity_type <= tr.parity_type;
                    vif.parity_en   <= tr.parity_en;
                    vif.stop2      <= tr.stop2;
    `uvm_info("DRV", $sformatf("BAUD:%0d LEN:%0d PAR_T:%0d PAR_EN:%0d
STOP:%0d TX_DATA:%0d", tr.baud, tr.length, tr.parity_type, tr.parity_en, tr.stop2,
tr.tx_data), UVM_NONE);
                    @(posedge vif.clk);
                    @(posedge vif.tx_done);
   @(negedge vif.rx_done);



     seq_item_port.item_done();


  end
 endtask



 virtual task run_phase(uvm_phase phase);
  drive();
 endtask



endclass
```

```
////////////////////////////////////////////////////////////////////////////////////////////

class mon extends uvm_monitor;
` uvm_component_utils(mon)

uvm_analysis_port#(transaction) send;
transaction tr;
virtual uart_if vif;

  function new(input string inst = "mon", uvm_component parent = null);
  super.new(inst,parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  tr = transaction::type_id::create("tr");
  send = new("send", this);
   if(!uvm_config_db#(virtual
uart_if)::get(this,"","vif",vif))//uvm_test_top.env.agent.drv.aif
     ` uvm_error("MON","Unable to access Interface");
  endfunction



  virtual task run_phase(uvm_phase phase);
  forever begin
   @(posedge vif.clk);
   if(vif.rst)
    begin
     tr.rst = 1'b1;
    ` uvm_info("MON", "SYSTEM RESET DETECTED", UVM_NONE);
```

```
      send.write(tr);
      end
     else
       begin
       @(posedge vif.tx_done);
       tr.rst       = 1'b0;
       tr.tx_start    = vif.tx_start;
       tr.rx_start    = vif.rx_start;
       tr.tx_data     = vif.tx_data;
       tr.baud       = vif.baud;
       tr.length     = vif.length;
       tr.parity_type = vif.parity_type;
       tr.parity_en   = vif.parity_en;
       tr.stop2      = vif.stop2;
         @(negedge vif.rx_done);


       tr.rx_out     = vif.rx_out;
         `uvm_info("MON", $sformatf("BAUD:%0d LEN:%0d PAR_T:%0d
PAR_EN:%0d STOP:%0d TX_DATA:%0d RX_DATA:%0d", tr.baud, tr.length,
tr.parity_type, tr.parity_en, tr.stop2, tr.tx_data, tr.rx_out), UVM_NONE);
        send.write(tr);
       end



   end
  endtask


endclass
//////////////////////////////////////////////////////////////////////////////////////////
```

```
class sco extends uvm_scoreboard;
` uvm_component_utils(sco)

 uvm_analysis_imp#(transaction,sco) recv;
 bit [31:0] arr[32] = '{default:0};
 bit [31:0] addr    = 0;
 bit [31:0] data_rd = 0;



  function new(input string inst = "sco", uvm_component parent = null);
  super.new(inst,parent);
  endfunction

  virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  recv = new("recv", this);
  endfunction


 virtual function void write(transaction tr);
   ` uvm_info("SCO", $sformatf("BAUD:%0d LEN:%0d PAR_T:%0d PAR_EN:%0d
STOP:%0d TX_DATA:%0d RX_DATA:%0d", tr.baud, tr.length, tr.parity_type,
tr.parity_en, tr.stop2, tr.tx_data, tr.rx_out), UVM_NONE);
  if(tr.rst == 1'b1)
    ` uvm_info("SCO", "System Reset", UVM_NONE)
  else if(tr.tx_data == tr.rx_out)
    ` uvm_info("SCO", "Test Passed", UVM_NONE)
  else
    ` uvm_info("SCO", "Test Failed", UVM_NONE)
    $display("------------------------------------------------------------");
    endfunction

endclass



//////////////////////////////////////////////////////////////////////////////////////////



class agent extends uvm_agent;
` uvm_component_utils(agent)

  uart_config cfg;

function new(input string inst = "agent", uvm_component parent = null);
super.new(inst,parent);
endfunction

 driver d;
 uvm_sequencer#(transaction) seqr;
 mon m;



virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
   cfg =  uart_config::type_id::create("cfg");
   m = mon::type_id::create("m",this);

  if(cfg.is_active == UVM_ACTIVE)
  begin
  d = driver::type_id::create("d",this);
  seqr = uvm_sequencer#(transaction)::type_id::create("seqr", this);
  end



endfunction


virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
 if(cfg.is_active == UVM_ACTIVE) begin
  d.seq_item_port.connect(seqr.seq_item_export);
 end
endfunction


endclass


//////////////////////////////////////////////////////////////////////////////////


class env extends uvm_env;
` uvm_component_utils(env)


function new(input string inst = "env", uvm_component c);
super.new(inst,c);
endfunction


agent a;
sco s;
```

```
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  a = agent::type_id::create("a",this);
  s = sco::type_id::create("s", this);
endfunction


virtual function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
 a.m.send.connect(s.recv);
endfunction


endclass


////////////////////////////////////////////////////////////////////////


class test extends uvm_test;
` uvm_component_utils(test)


function new(input string inst = "test", uvm_component c);
super.new(inst,c);
endfunction



env e;
rand_baud rb;
rand_baud_with_stop rbs;
rand_baud_len5p  rb5l;
rand_baud_len6p rb6l;
rand_baud_len7p rb7l;
```

```
rand_baud_len8p rb8l;
 //////////////////////


 rand_baud_len5  rb5lwop;
 rand_baud_len6  rb6lwop;
 rand_baud_len7  rb7lwop;
 rand_baud_len8  rb8lwop;



virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
  e      = env::type_id::create("env",this);
  rb     = rand_baud::type_id::create("rb");
  rbs    = rand_baud_with_stop::type_id::create("rbs");
 /////////////fixed length var baud with parity
  rb5l   = rand_baud_len5p::type_id::create("rb5l");
  rb6l   = rand_baud_len6p::type_id::create("rb6l");
  rb7l   = rand_baud_len7p::type_id::create("rb7l");
  rb8l   = rand_baud_len8p::type_id::create("rb8l");

 ///////////////fixed len var baud without parity
 rb5lwop = rand_baud_len5::type_id::create("rb5lwop");
 rb6lwop = rand_baud_len6::type_id::create("rb6lwop");
 rb7lwop = rand_baud_len7::type_id::create("rb7lwop");
 rb8lwop = rand_baud_len8::type_id::create("rb8lwop");



endfunction
```

```
virtual task run_phase(uvm_phase phase);

phase.raise_objection(this);

rb8lwop.start(e.a.seqr);

#20;

phase.drop_objection(this);

endtask

endclass


//////////////////////////////////////////////////////////////////

module tb;



 uart_if vif();






 uart_top dut (.clk(vif.clk), .rst(vif.rst), .tx_start(vif.tx_start), .rx_start(vif.rx_start),
.tx_data(vif.tx_data), .baud(vif.baud), .length(vif.length),
.parity_type(vif.parity_type),
.parity_en(vif.parity_en),.stop2(vif.stop2),.tx_done(vif.tx_done),
.rx_done(vif.rx_done), .tx_err(vif.tx_err), .rx_err(vif.rx_err), .rx_out(vif.rx_out));


 initial begin

  vif.clk <= 0;

 end


 always #10 vif.clk <= ~vif.clk;
 initial begin
  uvm_config_db#(virtual uart_if)::set(null, "*", "vif", vif);
  run_test("test");
 end


 initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
 end


endmodule
```