I2S AVIP

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
uvm	Universal Verification Methodology
i2s	Inter IC Sound
avip	Accelerated verification intellectual property
hdl	Hardware descriptive language
hvl	Hardware verification language
bfm	Bus functional model
tlm	Transaction level modelling
sclk	Serial Clock
sd	Serial Data
ws	Word Select
tx	Transmitter
rx	Receiver
wsp	Word Select Period

Chapter 1

Introduction

1.1 I2S

Many digital audio systems are being introduced into the consumer audio market, including compact disc, digital audio tapes, digital sound processors, and digital TV-sound. The digital audio signals in these systems are being processed by several VLSI ICs, such as:

•A/D and D/A converters
•Digital signal processors
•Error correction for compact disc and digital recording
•Digital filters
•Digital input/output interfaces.

Standardized communication structures are vital for both the equipment and the IC manufacturer, because they increase system flexibility. To this end, we have developed the inter-IC sound (I2S) bus - a serial link especially for digital audio.

The bus has to handle only audio data. To minimize the number of pins required and to keep wiring simple, a 3-line serial bus is used consisting of a line for two time-multiplexed data channels, a word select line and a clock line.

There are 3 signals are used in I2S bus-

1. **SCLK/SCK** - Serial clock

2. **WS** (word select)/ **FS** (Frame select)/ **WCLK** (word clock) / **LRCLK** (left right clock)

3. **SD (**Serial data)

This protocol requires three connections between the devices on a bus: the clock signal (SCLK) for synchronizing communication partners, a word select (WS) line for switching between the left and right audio channels, and a dedicated data wire exclusively for transmitting audio information bit-by-bit.

1.2 Key Features

1. It has 8,16,24,32 Serial data bits for each sample.

2. It supports both mono channel and stereo channel modes of communication

3. It has different sample rates. (8 khz to 192 khz)
4. It supports the Left justified (Codec) mode.
5. It has 16-bit, 32-bit, 48-bit, or 64-bit Word select period.
6. Mechanism to enable Serial data transmission, WS generation and Serial clock generation.

1.3 Key features for the TODO

1. It supports the Right justified (Codec) mode and Phillips Standard mode.

1.4 Signals

1.4.1. SCLK

The SCK or Serial Clock is the first line of the I2S protocol which is also known as BCLK or bit clock line. SCLK is the clock signal used for synchronizing data transmission in I2S. It's often called BCLK (Bit Clock) because it’s responsible for transmitting one bit at a time per clock cycle. So, when you're transmitting a 16-bit audio sample for one channel (left or right), the BCLK signal will pulse 16 times to transmit all 16 bits.

For stereo audio (left and right channels), the SCLK continues to pulse for each bit of the word, with alternating words for left and right channels based on the Word Select (WS) signal.

Frequency = sample rate * bits for each channel * number of channels

1.4.2. WS

Word Select (WS) signal in I2S (Inter-IC Sound) is a channel selection signal refers to the two stereo audio channels (left and right channels). The WS signal is used to indicate which audio channel (left or right) the data corresponds to in each audio frame.

In Left justified (CODEC) mode:

If WS = 1 → Left channel. If WS = 0 → Right channel

The number of bits in SD should always be less than or equal to WS period to ensure that all the bits are transferred properly to appropriate channel. When the WS signal is HIGH, the first block of data (e.g., 16 bits, 24 bits, or 32 bits depending on the system) will represent the

left channel audio data. When the WS signal switches to LOW, the next block of data will represent the right channel audio data.

In mono audio, the same audio signal is played through both speakers, so everything sounds like it is coming from a single point, typically in the center.

In stereo audio, each channel can carry different signals. For example, the left channel might carry an instrument or voice on the left side of the audio mix, and the right channel might carry something different that sounds like it's coming from the right side. SCLK and WS will always be generated by the master or controller.

1.4.3. SD

The Serial Data or SD is the last wire where the payload is transmitted. So, it is very significant that the MSB is first transferred, because both the transmitter & receiver may include different word lengths. Thus, the transmitter or the receiver has to recognize how many bits are transmitted.

If the word length of the receiver is greater than the transmitter, then the word is largened (LSB bits are set to zero). If the word length of the receiver is less than the word length of the transmitter, then the LSB bits are ignored.

1.5 I2S Left Justified Mode of Operation

- In this format, the MSB of the audio sample is aligned with the first clock pulse after the WS signal changes.
- For Left-Justified, the Word Select changes when the MSB bit for current frame is available.
- Serial data is justified to the left, which means that if Word Select's half period is 32 bit long and only 24 bits are used for audio data, the first 24 bits will be used for audio and the remaining 8 bits must be set to zero.

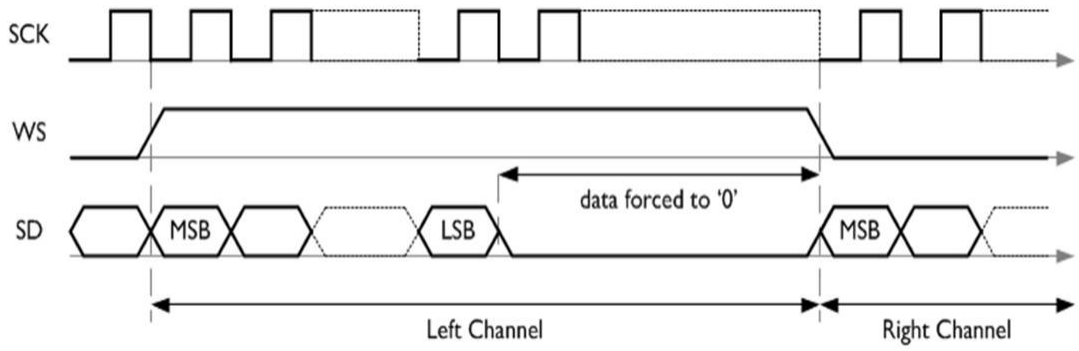

Fig 1.1 *I2S Left Justified mode of operation*

1.6 Application

1. I2S is primarily used to transfer digital audio signals between ICs within electronic devices. For example, it connects microcontrollers or digital signal processors (DSPs) to audio codecs, enabling efficient audio data processing and playback.

2. Many consumer electronics rely on I2S for audio data communication. Devices such as smartphones, tablets, and digital televisions utilize I2S to manage audio input and output, ensuring synchronized and high-quality sound reproduction.

3. Embedded systems and Internet of Things (IoT) devices often employ I2S to handle audio data. For example, smart home assistants use I2S to process voice commands, connecting microphones to processing units and subsequently to speakers for responses.

4. Electronic musical instruments and audio effect processors utilize I2S to transmit digital audio between components. This enables real-time sound processing and manipulation, essential for live performances and studio recordings.

Chapter 2

Architecture

2.1 I2S Testbench Architecture

The accelerated VIP has been divided into the two top modules HVL and HDL top as shown in Fig 2.1. The whole idea of using Accelerated VIP is to push the synthesizable part of the testbench into the separate top module along with the interface and it is named HDL TOP. The Un-synthesizable part is pushed into the HVL TOP providing the ability to run the longer tests quickly. This particular test bench can be used for the simulation as well as the emulation based on the mode of operation.

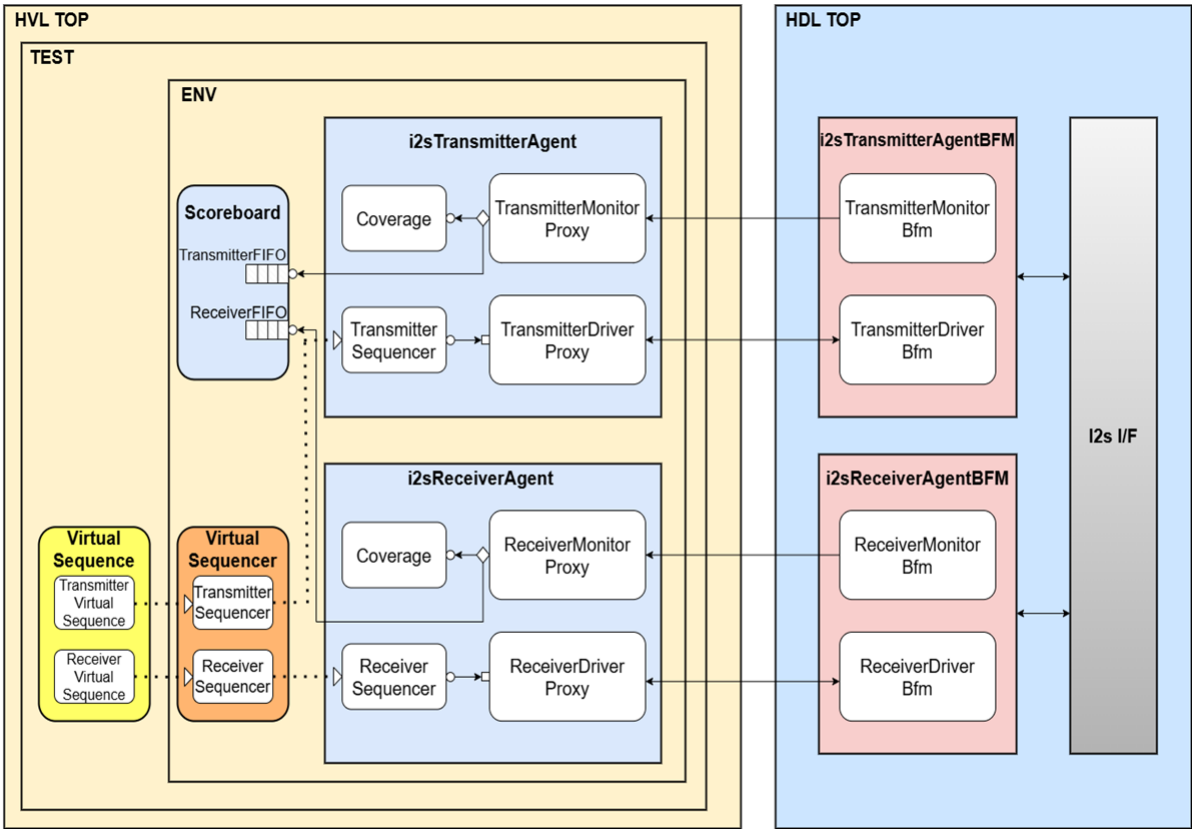

Fig 2.1 *I2S_AVIP Architecture*

HVL TOP has a Test which includes the ENV and virtual Sequence and ENV has both Proxy Agent Transmitter and Receiver Agent, Scoreboard and Virtual sequencer. Proxy Agent has Coverage, Monitor, Driver and Sequencer. The transactions flow from both the Transmitter virtual sequence and Receiver virtual sequence onto the I2S interface (I/F) through the Proxy Agent and Proxy Agent gets the data from Monitor BFM and uses the data to do checks using Scoreboard and Coverage.

HDL TOP consists of the design part, which is timed and synthesizable, Clock and reset signals are generated in the HDL TOP. Bus Functional Models (BFMs) i.e. synthesizable parts of drivers and monitors are present in HDL TOP, BFMs also have the back pointers to their proxy to call non-blocking methods which are defined in the proxy.

Chapter 3

Implementation

3.1 Pin Interface

Table 3.1 *I2S pins used in interface to external devices*

Signals	Source	Description
clk	Clock source	System Clock
rst	Reset Controller	Reset signal
sclk	Tx Master/ Rx Master	Serial Clock
ws	Tx Master/ Rx Master	Word Select
sd	Tx Master	Serial Data

3.2 Testbench Components

In this section, testbench components of the i2s - avip are discussed.

3.2.1 I2S HDL Top

HDL top is synthesizable, where generation of the clock and reset is done. Instantiation of the I2S interface handle, Transmitter agent bfm handle and receiver agent bfm handle is done as shown in Figure 3.1.

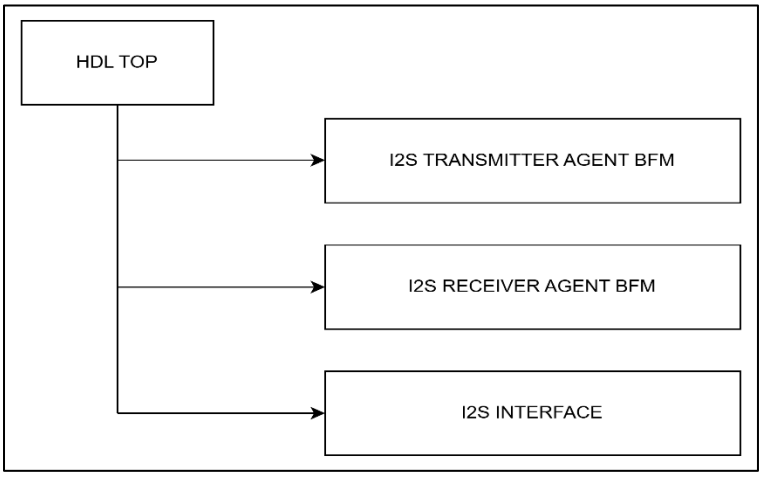

Fig 3.1 *HDL Top*

3.2.2 I2S Interface

Importing the global packages
Passing Signals: clk, rst
Declaration of signals: sclk, ws and sd as logic type

3.2.3 I2S Transmitter Agent BFM Module

Instantiates the below two interfaces here

1. I2S Transmitter driver bfm
2. I2S Transmitter monitor bfm

Instantiates the i2s transmitter assertions and binds it with the i2s transmitter monitor bfm handle and maps the signals of i2s transmitter assertions with the i2s interface signals. The i2s interface signals are passed to the i2s transmitter driver and monitor bfm in instantiations.

```
I2sTransmitterDriverBFM i2sTransmitterDriverBFM(.clk(i2sInterface.clk),
                                                .rst(i2sInterface.rst),
                                                .wsInput(i2sInterface.ws),
                                                .wsOutput(i2sInterface.wsOutput),
                                                .sclkInput(i2sInterface.sclk),
                                                .sclkOutput(i2sInterface.sclkOutput),
                                                .sd(i2sInterface.sd));
```

Fig 3.2 *I2S transmitter driver bfm instantiation in I2S transmitter agent bfm code snippet*

```
I2sTransmitterMonitorBFM i2sTransmitterMonitorBFM(.clk(i2sInterface.clk),
                                                  .rst(i2sInterface.rst),
                                                  .ws(i2sInterface.ws),
                                                  .wsOutput(i2sInterface.wsOutput),
                                                  .sclk(i2sInterface.sclk),
                                                  .sclkOutput(i2sInterface.sclkOutput),
                                                  .sd(i2sInterface.sd));
```

Fig 3.3 *I2S transmitter monitor bfm instantiation in I2S transmitter agent bfm code snippet*

Fig. 3.2 and 3.3 are the code snippets of instantiations of i2s transmitter driver and monitor bfm.

3.2.4 I2S Transmitter Driver BFM Interface

I2S Transmitter driver bfm is an interface where it will send the signals to the I2S interface. It has methods like genSclk, driveDataWhenTxMaster and driveDataWhenTxSlave which will be

called by the I2S Transmitter driver proxy which drives the Serial clock, Word Select and Serial data to the I2S interface. Fig 3.2 gives the reference of the instantiation of I2S Transmitter driver bfm.

3.2.5 I2S Transmitter Monitor BFM Interface

I2S Transmitter monitor bfm is an interface where it will get the signals from the I2S interface. It has methods like sampleData which will be called by the i2s transmitter monitor proxy which samples the Serial clock, Word Select and Serial data from the i2s interface. After sampling the data, the i2s transmitter monitor bfm interface sends the data to the i2s transmitter monitor proxy using the output port of sampleData task. Fig 3.3 gives the reference of the instantiation of i2s transmitter monitor bfm.

3.2.6 I2S Receiver Agent BFM Module

Instantiates the below two interfaces here

1. I2S Receiver driver bfm
2. I2S Receiver monitor bfm.

Instantiates the i2s receiver assertions and binds it with the i2s receiver monitor bfm handle and maps the signals of i2s receiver assertions with the i2s interface signals. The i2s interface signals are passed to the i2s receiver driver and monitor bfm in instantiations.

```
I2sReceiverDriverBFM i2sReceiverDriverBFM(.clk(i2sInterface.clk),
                                          .rst(i2sInterface.rst),
                                          .wsInput(i2sInterface.wsInput),
                                          .wsOutput(i2sInterface.wsOutput),
                                          .sclkInput(i2sInterface.sclkInput),
                                          .sclkOutput(i2sInterface.sclkOutput),
                                          .sd(i2sInterface.sd));
```

Fig 3.4 *I2S receiver driver bfm instantiation in I2S receiver agent bfm code snippet*

```
I2sReceiverMonitorBFM i2sReceiverMonitorBFM(.clk(i2sInterface.clk),
                                            .rst(i2sInterface.rst),
                                            .ws(i2sInterface.ws),
                                            .wsOutput(i2sInterface.wsOutput),
                                            .sclk(i2sInterface.sclk),
                                            .sclkOutput(i2sInterface.sclkOutput),
                                            .sd(i2sInterface.sd));
```

Fig 3.5 *I2S receiver monitor bfm instantiation in I2S receiver agent bfm code snippet*

Fig. 3.4 and 3.5 are the code snippets of instantiations of i2s receiver driver and monitor bfm.

3.2.7 I2S Receiver Driver BFM Interface

I2S Receiver driver bfm is an interface where it will send the signals to the I2S interface. It has methods like genSclk, drivePacket which will be called by the I2S Receiver driver proxy which drives the Serial clock, Word Select to the I2S interface. Fig 3.4 gives the reference of the instantiation of I2S Receiver driver bfm.

3.2.8 I2S Receiver Monitor BFM Interface

I2S Receiver monitor bfm is an interface where it will get the signals from the I2S interface. It has methods like samplePacket which will be called by the i2s Receiver monitor proxy which samples the Serial clock, Word Select and Serial Data from the i2s interface. After sampling the data, the i2s receiver monitor bfm interface sends the data to the i2s receiver monitor proxy using the output port of samplePacket task. Fig 3.5 gives the reference of the instantiation of i2s receiver monitor bfm.

3.2.9 I2S HVL TOP

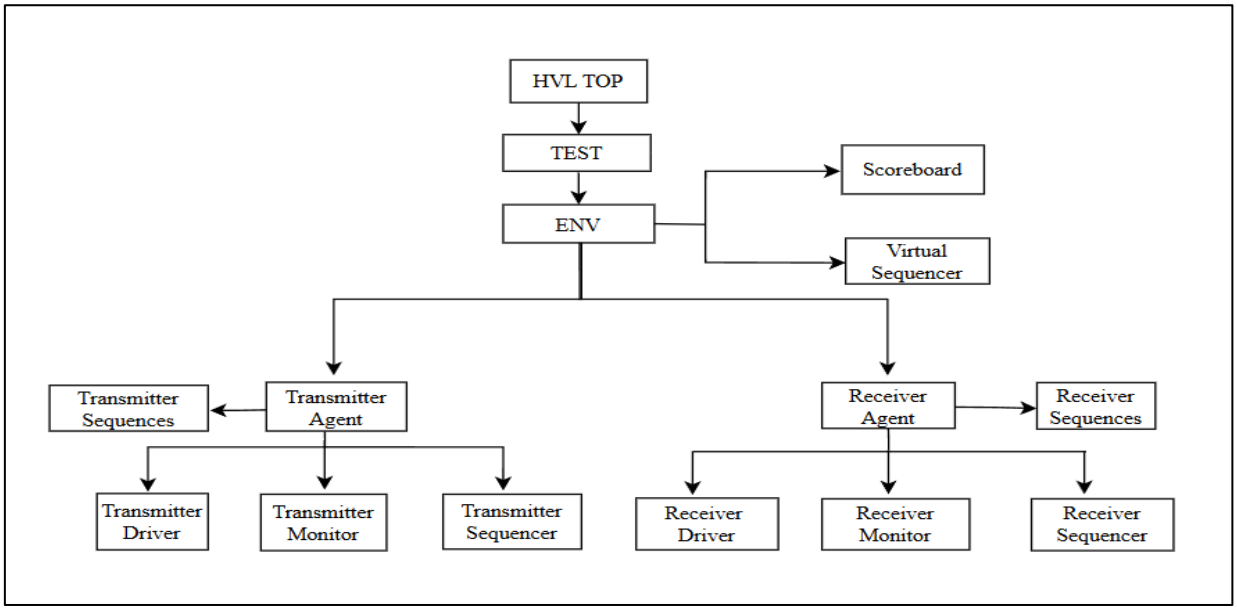

Fig 3.6 *HVL Top*

In top test is running by using the run_test("test_name") method, which will start the whole tb components.

3.2.10 I2S Environment

Environment has the below components

a. i2SScoreboard

b. virtualSequencer
c. i2sTransmitterAgent
d. i2sReceiverAgent

In the build phase, the i2sEnvConfig handle will be called and create the memory for the above declared components.
In the connect phase, the i2sTransmitterMonitorProxy is connected to i2sScoreboard and i2sReceiverMonitorProxy to i2sScoreboard using analysis port and analysis fifo as shown in Fig 3.7.

3.2.11 I2S Scoreboard

A scoreboard is a verification component that contains checkers and verifies the functionality of a design. The scoreboard is implemented by extending the uvm_scoreboard.

The purpose of the scoreboard in the I2S-AVIP project is to

1. Compare the Word Select, Serial Data and Word Select Period from the Transmitter and Receiver.
2. Keep track of pass and failure rates identified in the comparison process
3. Report comparison success/failures result at the end of the simulation

The scoreboard consists of two analysis fifo's which receive the packets from the analysis port of the monitor class. Fig 3.7 shows the connection between the analysis port and analysis fifo.

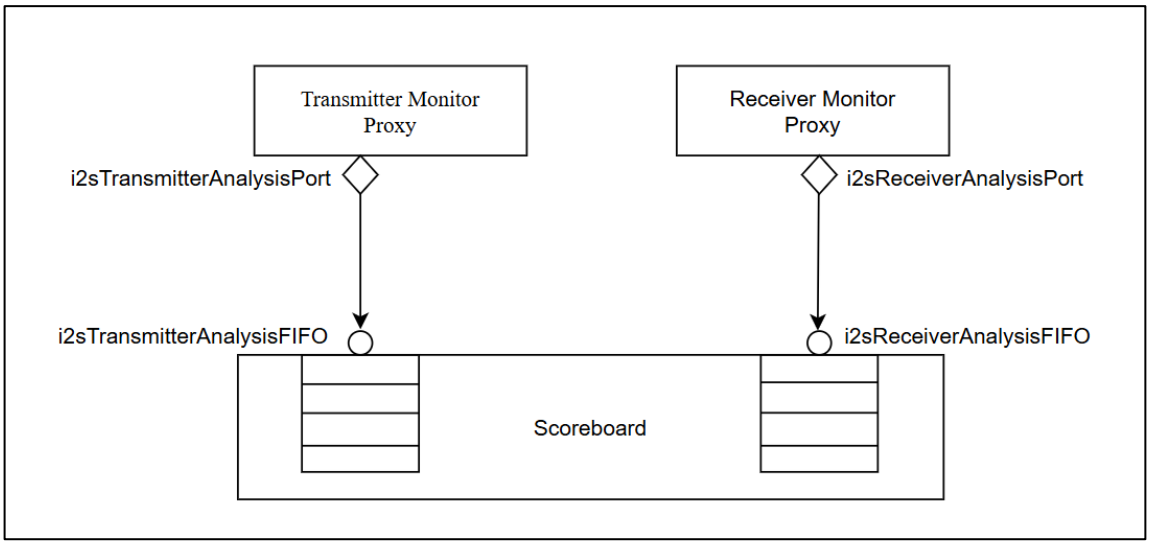

Fig 3.7 *Connection of the analysis ports of the monitor to the scoreboard analysis fifo*

In the monitor proxy class of transmitter and receiver, two analysis ports are declared. Fig 3.8 shows the declaration of the Transmitter analysis port and Receiver analysis port in the Transmitter monitor proxy and Receiver monitor proxy.

```
uvm_analysis_port #(I2sTransmitterTransaction) i2sTransmitterAnalysisPort;

uvm_analysis_port #(I2sReceiverTransaction)i2sReceiverAnalysisPort;
```

Fig 3.8 *shows the declaration of the Transmitter and Receiver analysis port in the Transmitter and Receiver monitor proxy*

In the scoreboard, two analysis fifo's are declared. Fig 3.9 shows the declaration of Transmitter analysis fifo and Receiver analysis fifo in the scoreboard.

```
uvm_tlm_analysis_fifo#(I2sTransmitterTransaction)i2sTransmitterAnalysisFIFO;

uvm_tlm_analysis_fifo#(I2sReceiverTransaction)i2sReceiverAnalysisFIFO;
```

Fig 3.9 *shows the declaration of Transmitter and Receiver analysis fifo in the scoreboard*

In the constructor, create objects for the two declared analysis fifo's. Fig 3.10 shows the creation of the Transmitter and Receiver analysis port.

```
function I2sScoreboard::new(string name = "I2sScoreboard", uvm_component parent = null);
  super.new(name, parent);

  i2sTransmitterAnalysisFIFO=new("i2sTransmitterAnalysisFIFO",this);
  i2sReceiverAnalysisFIFO=new("i2sReceiverAnalysisFIFO",this);

endfunction : new
```

Fig 3.10 *shows the creation of the Transmitter and Receiver analysis port*

In the connect phase of the environment class, the analysis port of both the Transmitter and Receiver monitor proxy class is connected to the analysis export of the Transmitter and Receiver fifo in the scoreboard. Fig 3.11 shows the connection made between the monitor analysis port and the scoreboard FIFOs in the connect phase of the env class.

```
i2sTransmitterAgent.i2sTransmitterMonitorProxy.i2sTransmitterAnalysisPort.connect(i2sScoreboard.i2sTransmitterAnalysisFIFO.analysis_export);

i2sReceiverAgent.i2sReceiverMonitorProxy.i2sReceiverAnalysisPort.connect(i2sScoreboard.i2sReceiverAnalysisFIFO.analysis_export);
```

Fig 3.11 *Connection done between the analysis port and analysis FIFO export in the env class*

In the run phase of the scoreboard, the get() method is used to get the data packet from the monitor write() method. Fig 3.12 shows the use of the get() method to get the transaction from the monitor analysis port.

```
task I2sScoreboard::run_phase(uvm_phase phase);
  super.run_phase(phase);
  `uvm_info(get_full_name(),"Inside i2s sb run phase",UVM_NONE)

  forever begin
    i2sTransmitterAnalysisFIFO.get(i2sTransmitterTransaction);
    `uvm_info(get_type_name(), $sformatf("after calling Transmitter analysis fifo get method"), UVM_HIGH);

    i2sReceiverAnalysisFIFO.get(i2sReceiverTransaction);
    `uvm_info(get_type_name(), $sformatf("after calling Receiver analysis fifo get method"), UVM_HIGH);
```

Fig 3.12 *Use of get method to get the packet from monitor analysis port*

The comparison of the WS, WSP, SD from the Transmitter monitor and Receiver monitor is done in the run phase. Fig 3.13 shows the comparison of the signals when transmitter WSP and receiver WSP are equal. Fig 3.14 shows the comparison of the signals when transmitter WSP is less than receiver WSP. Fig 3.15 shows the comparison of the signals when transmitter WSP is greater than receiver WSP.

```
if(i2sTransmitterTransaction.txWs == i2sReceiverTransaction.rxWs) begin

  if(i2sEnvConfig.i2sTransmitterAgentConfig.wordSelectPeriod == i2sEnvConfig.i2sReceiverAgentConfig.wordSelectPeriod) begin

    for(int i=0; i<i2sTransmitterTransaction.txSdLeftChannel.size(); i++) begin
       if(i2sTransmitterTransaction.txSdLeftChannel[i] == i2sReceiverTransaction.rxSdLeftChannel[i]) begin

        `uvm_info(get_type_name(),$sformatf("i2s Left channel serial data from transmitter and receiver is equal"),UVM_NONE);
        `uvm_info("SB_LeftChannel_serialData_MATCHED", $sformatf("Transmitter Left channel SerialData = %0h and Receiver left channel SerialData = %0h",
2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);
        leftChannelSerialDataComparisonSuccessCount++;
      end
      else
        begin
        `uvm_error(get_type_name(),$sformatf("i2s Left channel serialData from Transmitter and receiver is Not equal"));
        `uvm_info("SB_LeftChannel_serialData_MISMATCHED", $sformatf("Transmitter Left Channel SerialData = %0h and Receiver Left channel SerialData = %0
",i2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);

        leftChannelSerialDataComparisonFailedCount++;
      end
   end
  for(int i=0; i<i2sTransmitterTransaction.txSdRightChannel.size(); i++) begin
       if(i2sTransmitterTransaction.txSdRightChannel[i] == i2sReceiverTransaction.rxSdRightChannel[i]) begin

        `uvm_info(get_type_name(),$sformatf("i2s Right channel serial data from transmitter and receiver is equal"),UVM_NONE);
        `uvm_info("SB_RightChannel_serialData_MATCHED", $sformatf("Transmitter Right channel SerialData = %0h and Receiver Right channel SerialData = %0
",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);
        rightChannelSerialDataComparisonSuccessCount++;
      end
      else
        begin
        `uvm_error(get_type_name(),$sformatf("i2s Right channel serial Data from Transmitter and receiver is Not equal"));
        `uvm_info("SB_RightChannel_serialData_MISMATCHED", $sformatf("Transmitter Right Channel SerialData = %0h and Receiver Right Channel SerialData =
%0h",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);

        rightChannelSerialDataComparisonFailedCount++;
      end
    end
  end
```

Fig 3.13 *Comparison of the signals when Transmitter WSP = Receiver WSP*

```
   else if (i2sEnvConfig.i2sTransmitterAgentConfig.wordSelectPeriod < i2sEnvConfig.i2sReceiverAgentConfig.wordSelectPeriod) begin
      for(int i=0; i<i2sTransmitterTransaction.txSdLeftChannel.size(); i++) begin
         if(i2sTransmitterTransaction.txSdLeftChannel[i] == i2sReceiverTransaction.rxSdLeftChannel[i]) begin

          `uvm_info(get_type_name(),$sformatf("i2s Left channel serial data from transmitter and receiver is equal"),UVM_NONE);
          `uvm_info("SB_LeftChannel_serialData_MATCHED", $sformatf("Transmitter Left channel SerialData = %0h and Receiver left channel SerialData = %0h",
i2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);
          leftChannelSerialDataComparisonSuccessCount++;
        end
        else
          begin
          `uvm_error(get_type_name(),$sformatf("i2s Left channel serialData from Transmitter and receiver is Not equal"));
          `uvm_info("SB_LeftChannel_serialData_MISMATCHED", $sformatf("Transmitter Left Channel SerialData = %0h and Receiver Left channel SerialData = %0
h",i2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);
          leftChannelSerialDataComparisonFailedCount++;
        end
    end
    for(int i=0; i<i2sTransmitterTransaction.txSdRightChannel.size(); i++) begin
         if(i2sTransmitterTransaction.txSdRightChannel[i] == i2sReceiverTransaction.rxSdRightChannel[i]) begin

          `uvm_info(get_type_name(),$sformatf("i2s Right channel serial data from transmitter and receiver is equal"),UVM_NONE);
          `uvm_info("SB_RightChannel_serialData_MATCHED", $sformatf("Transmitter Right channel SerialData = %0h and Receiver Right channel SerialData = %0
h",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);
          rightChannelSerialDataComparisonSuccessCount++;
        end
        else
          begin
          `uvm_error(get_type_name(),$sformatf("i2s Right channel serialData from Transmitter and receiver is Not equal"));
          `uvm_info("SB_RightChannel_serialData_MISMATCHED", $sformatf("Transmitter Right Channel SerialData = %0h and Receiver Right Channel SerialData =
 %0h",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);

          rightChannelSerialDataComparisonFailedCount++;
       end
     end
    end
```

Fig 3.14 *Comparison of the signals when Transmitter WSP < Receiver WSP*

```
    else if (i2sEnvConfig.i2sTransmitterAgentConfig.wordSelectPeriod > i2sEnvConfig.i2sReceiverAgentConfig.wordSelectPeriod) begin
       for(int i=0; i<i2sTransmitterTransaction.txSdLeftChannel.size(); i++) begin
         if(i2sTransmitterTransaction.txSdLeftChannel[i] == i2sReceiverTransaction.rxSdLeftChannel[i]) begin

          `uvm_info(get_type_name(),$sformatf("i2s Left channel serial data from transmitter and receiver is equal"),UVM_NONE);
          `uvm_info("SB_LeftChannel_serialData_MATCHED", $sformatf("Transmitter Left channel SerialData = %0h and Receiver left channel SerialData = %0h",
i2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);
          leftChannelSerialDataComparisonSuccessCount++;
        end
        else
          begin
          `uvm_error(get_type_name(),$sformatf("i2s Left channel serialData from Transmitter and receiver is Not equal"));
          `uvm_info("SB_LeftChannel_serialData_MISMATCHED", $sformatf("Transmitter Left Channel SerialData = %0h and Receiver Left channel SerialData = %0
h",i2sTransmitterTransaction.txSdLeftChannel[i],i2sReceiverTransaction.rxSdLeftChannel[i]),UVM_NONE);
          leftChannelSerialDataComparisonFailedCount++;
        end
    end
    for(int i=0; i<i2sTransmitterTransaction.txSdRightChannel.size(); i++) begin
         if(i2sTransmitterTransaction.txSdRightChannel[i] == i2sReceiverTransaction.rxSdRightChannel[i]) begin

          `uvm_info(get_type_name(),$sformatf("i2s Right channel serial data from transmitter and receiver is equal"),UVM_NONE);
          `uvm_info("SB_RightChannel_serialData_MATCHED", $sformatf("Transmitter Right channel SerialData = %0h and Receiver Right channel SerialData = %0
h",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);
          rightChannelSerialDataComparisonSuccessCount++;
        end
        else
          begin
          `uvm_error(get_type_name(),$sformatf("i2s Right channel serialData from Transmitter and receiver is Not equal"));
          `uvm_info("SB_RightChannel_serialData_MISMATCHED", $sformatf("Transmitter Right Channel SerialData = %0h and Receiver Right Channel SerialData =
 %0h",i2sTransmitterTransaction.txSdRightChannel[i],i2sReceiverTransaction.rxSdRightChannel[i]),UVM_NONE);

          rightChannelSerialDataComparisonFailedCount++;
       end
     end

   end
  else
    `uvm_info(get_type_name(),$sformatf("i2s transmitter and receiver Word Selects are not equal"),UVM_NONE);

  end
endtask
```

Fig 3.15 *Comparison of the signals when Transmitter WSP > Receiver WSP*

Fig 3.16 explains the flow chart of the run phase in the scoreboard.

In the run phase, inside the forever loop, the scoreboard transmitter analysis fifo gets the transaction from the transmitter monitor analysis port using the get() method. Whenever the packet is received, firstly Word Select of transmitter and receiver are compared. If txWs and rxWs are equal, then compare txWSP and rxWSP. Based on that, Serial Data of the respective channel is compared.

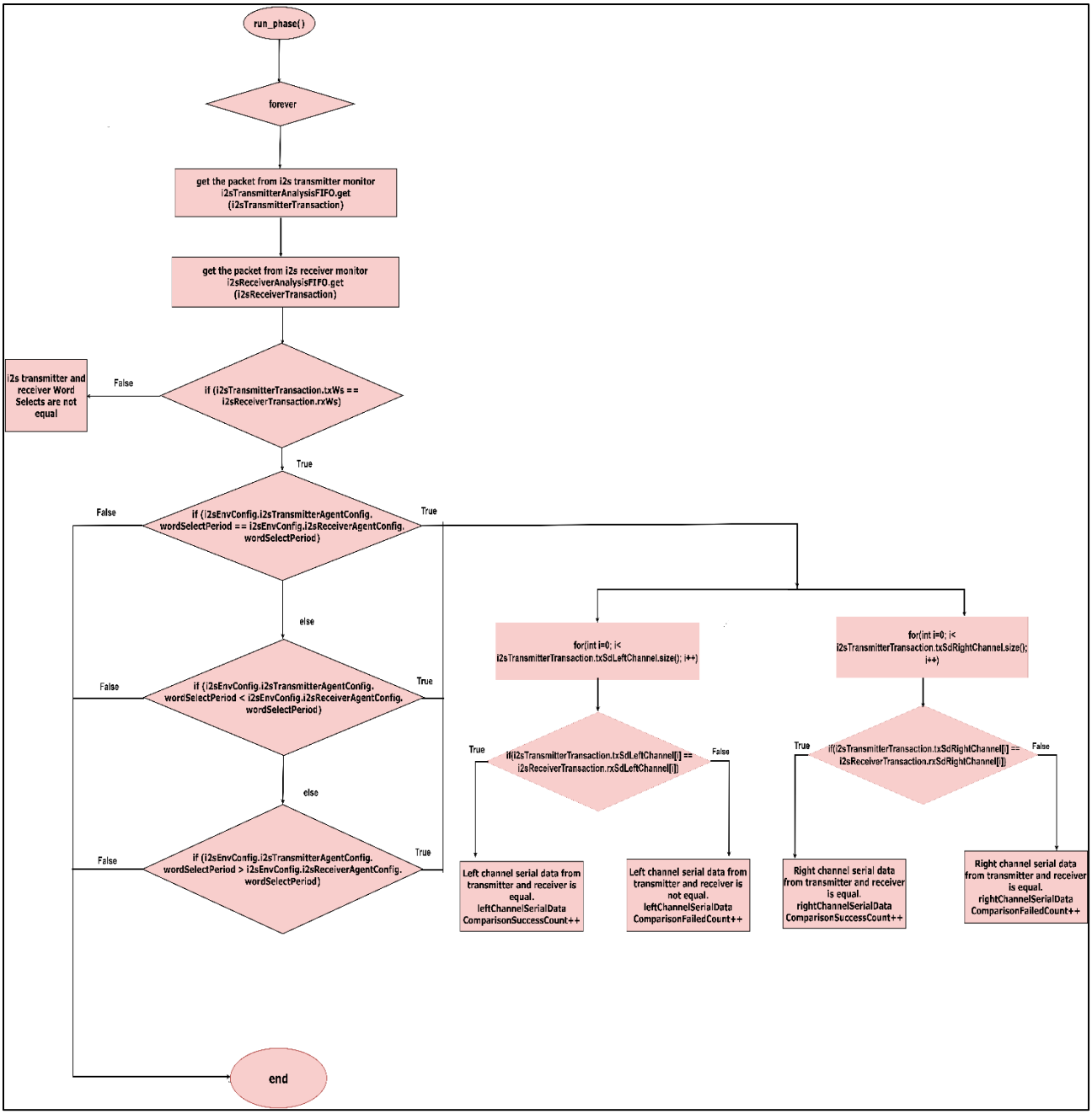

Fig 3.16 *Flow chart of the scoreboard Run phase*

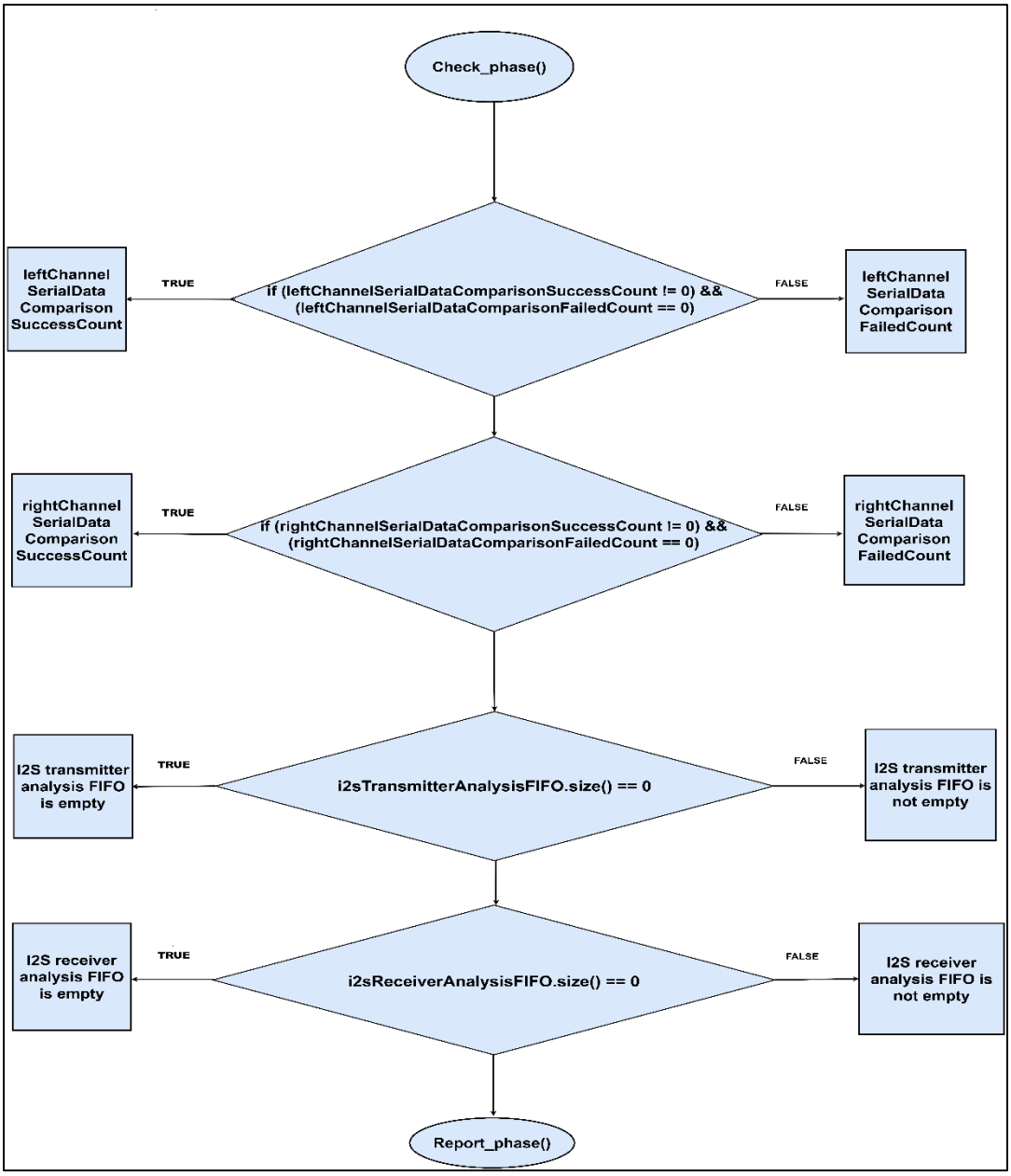

Fig 3.17 *Flow chart of the scoreboard check phase*

3.2.12 I2S Virtual Sequencer

In the virtual sequencer, declare the handles for environment configuration, transmitter sequencer and receiver sequencer. In the build phase, get the environment configuration through config_db and create the memory for transmitter sequencer and receiver sequencer.

3.2.13 I2S Transmitter Agent

I2S Transmitter agent component is a class extending from uvm_agent. It gets the I2sTransmitterAgentConfig and based on that we will create and connect the components. It creates the I2sTransmitterSequencer and I2sTransmitterDriverProxy only if the I2sTransmitterAgent is active which will depend on the value of is_active variable declared in the i2sTransmitterAgentConfig file. The I2sTransmitterCoverage is created in build_phase if the has_coverage variable is TRUE which is declared in the I2sTransmitterAgentConfig file. Please refer to Figure 3.16 for the I2S Transmitter agent build_phase code snippet.

The I2S Transmitter agent build phase has the creation of,

a. I2sTransmitterSequencer
b. I2sTransmitterDriverProxy
c. I2sTransmitterMonitorProxy
d. I2sTransmitterCoverage

```
function void I2sTransmitterAgent::build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(!uvm_config_db #(I2sTransmitterAgentConfig)::get(this,"","I2sTransmitterAgentConfig",i2sTransmitterAgentConfig))
  `uvm_fatal("config","cannot get the config m_cfg from uvm_config_db. Have u set it ?")

  i2sTransmitterMonitorProxy=I2sTransmitterMonitorProxy::type_id::create("i2sTransmitterMonitorProxy",this);

  if(i2sTransmitterAgentConfig.isActive==UVM_ACTIVE)

    begin
      i2sTransmitterDriverProxy=I2sTransmitterDriverProxy::type_id::create("i2sTransmitterDriverProxy",this);
      i2sTransmitterSequencer=I2sTransmitterSequencer::type_id::create("i2sTransmitterSequencer",this);
    end

  if(i2sTransmitterAgentConfig.hasCoverage)
   begin
    i2sTransmitterCoverage=I2sTransmitterCoverage::type_id::create("i2sTransmitterCoverage",this);
  end

  uvm_config_db#(I2sTransmitterAgentConfig)::set(uvm_root::get(), "*","I2sTransmitterAgentConfig",i2sTransmitterAgentConfig);
    `uvm_info(get_type_name(), $sformatf("\nI2S_TRANSMITTER_AGENT_CONFIG\n%s",
                i2sTransmitterAgentConfig.sprint()),UVM_LOW);
endfunction : build_phase
```

Fig 3.18 *I2S Transmitter agent build_phase code snippet.*

I2sTransmitterAgentConfig handles declared in the above created components will be mapped here in the connect phase. The I2sTransmitterDriverProxy and I2sTransmitterSequencer are connected using TLM ports if the I2sTransmitterAgent is active. The I2sTransmitterCoverage's analysis_export will be connected to the I2sTransmitterMonitorProxy's i2sTransmitterAnalysisPort in connect_phase for transaction analysis and coverage collection. Please refer to Figure 3.19 for the I2S Transmitter agent connect phase code snippet.

```
function void I2sTransmitterAgent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  if(i2sTransmitterAgentConfig.isActive==UVM_ACTIVE)
    begin
     i2sTransmitterDriverProxy.i2sTransmitterAgentConfig=i2sTransmitterAgentConfig;
      i2sTransmitterSequencer.i2sTransmitterAgentConfig=i2sTransmitterAgentConfig;
      i2sTransmitterDriverProxy.seq_item_port.connect(i2sTransmitterSequencer.seq_item_export);
  end

  if(i2sTransmitterAgentConfig.hasCoverage)
   begin
    i2sTransmitterMonitorProxy.i2sTransmitterAnalysisPort.connect(i2sTransmitterCoverage.analysis_export);
  end
  i2sTransmitterMonitorProxy.i2sTransmitterAgentConfig = i2sTransmitterAgentConfig;

endfunction : connect_phase
```

Fig 3.19 *I2S Transmitter agent connect phase code snippet*

3.2.14 I2S Transmitter Sequencer

I2sTransmitterSequencer component is a parameterized class of type I2sTransmitterTransaction, extending uvm_sequencer. I2sTransmitterSequencer sends the data from the I2sTransmitterSequences to the I2sTransmitterDriverProxy.

3.2.15 I2S Transmitter Driver Proxy

I2sTransmitterDriverProxy component is a parameterized class of type I2sTransmitterTransaction, extending uvm_driver. It gets the virtual handle of the I2S Transmitter driver bfm and using this handle we will call the bfm side methods.

As the I2S Transmitter driver bfm interface cannot access the class-based I2S Transmitter transaction data, so we have to convert that into struct data type. Similarly, it converts the I2s Transmitter configuration values into struct data type. I2s Transmitter driver proxy will call the converter class to convert the transmitter transaction packet and transmitter configuration packet into a struct data packet and struct configuration packet respectively (declared in i2s global package) and then will pass it to I2s Transmitter driver bfm using methods like genSclk, driveDataWhenTxMaster and driveDataWhenTxSlave declared in I2s Transmitter driver bfm.

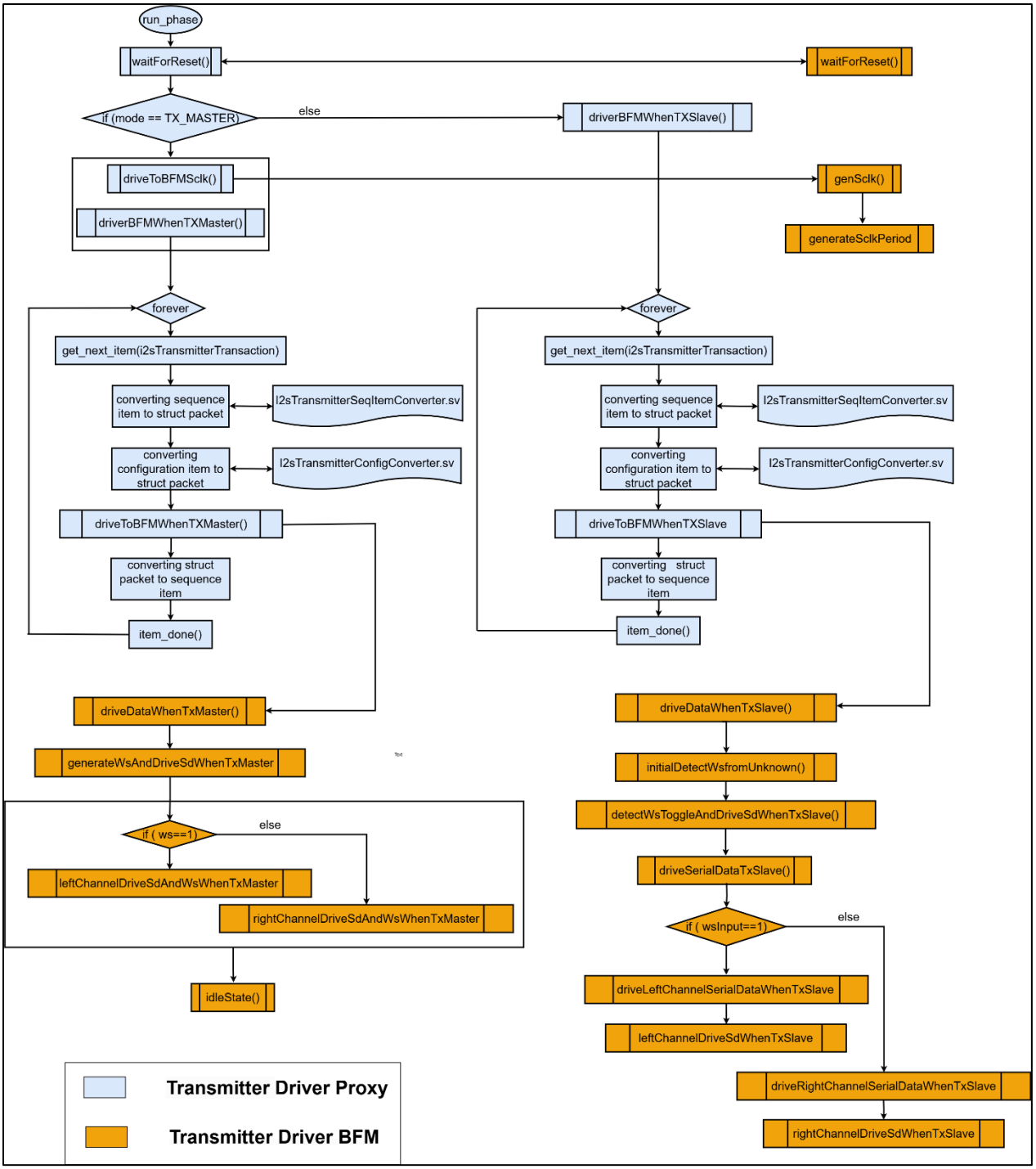

Fig 3.20 *Flowchart of communication between i2s Transmitter driver proxy and i2s Transmitter driver bfm*

```
task I2sTransmitterDriverProxy::run_phase(uvm_phase phase);
  super.run_phase(phase);

  `uvm_info(get_type_name(), "Running the transmitter Driver", UVM_NONE)

   i2sTransmitterDriverBFM.waitForReset();
   `uvm_info(get_type_name(), "I2S :: Reset detected", UVM_NONE);


     I2sTransmitterConfigConverter::fromTransmitterClass(i2sTransmitterAgentConfig, configStruct);
    `uvm_info(get_type_name(), $sformatf("IN DRIVER- Converted cfg struct\n%p",configStruct), UVM_NONE)

    `uvm_info("DEBUG", "IN DRIVER-Inside I2sTransmitterDriverProxy", UVM_NONE)

     if(configStruct.mode == TX_MASTER)

     begin

     fork
       driveToBFMSclk(configStruct);
       driverBFMWhenTXMaster(packetStruct,configStruct);
     join_any

     end

    else if(configStruct.mode == TX_SLAVE)
    begin
      driverBFMWhenTXSlave(packetStruct,configStruct);

    end
endtask : run_phase
```

Fig 3.21 *run phase of i2s Transmitter driver proxy code snippet*

```
task I2sTransmitterDriverProxy::driverBFMWhenTXMaster(inout i2sTransferPacketStruct packetStruct,input i2sTransferCfgStruct configStruct);

 forever
  begin
    seq_item_port.get_next_item(i2sTransmitterTransaction);

   `uvm_info(get_type_name(), $sformatf("IN DRIVER- Received i2sTransmitterTransaction\n%s",i2sTransmitterTransaction.sprint()), UVM_NONE)

   I2sTransmitterSeqItemConverter::fromTransmitterClass(i2sTransmitterTransaction, packetStruct);
   `uvm_info(get_type_name(), $sformatf("IN DRIVER- Converted i2sTransmitterTransaction to struct\n%p",packetStruct), UVM_NONE)

    I2sTransmitterConfigConverter::fromTransmitterClass(i2sTransmitterAgentConfig, configStruct);
   `uvm_info(get_type_name(), $sformatf("IN DRIVER- Converted cfg struct\n%p",configStruct), UVM_NONE)

    driveToBFMWhenTXMaster(packetStruct,configStruct);

   I2sTransmitterSeqItemConverter::toTransmitterClass(packetStruct,i2sTransmitterTransaction);
   `uvm_info(get_type_name(), $sformatf("IN DRIVER-After driving data to interface :: Received i2sTransmitterTransaction\n%s",i2sTransmitterTransaction.s
print()), UVM_NONE)

   seq_item_port.item_done();
 end

endtask:driverBFMWhenTXMaster
```

Fig 3.22 *driveBFMWhenTxMaster task in i2s Transmitter driver proxy code snippet*

```
 task I2sTransmitterDriverProxy::driverBFMWhenTXSlave(inout i2sTransferPacketStruct packetStruct,input i2sTransferCfgStruct configStruct);

 forever begin
   seq_item_port.get_next_item(i2sTransmitterTransaction);

  `uvm_info(get_type_name(), $sformatf("IN DRIVER- Received i2sTransmitterTransaction\n%s",i2sTransmitterTransaction.sprint()), UVM_NONE)

  I2sTransmitterSeqItemConverter::fromTransmitterClass(i2sTransmitterTransaction, packetStruct);
  `uvm_info(get_type_name(), $sformatf("IN DRIVER- Converted i2sTransmitterTransaction to struct\n%p",packetStruct), UVM_NONE)

   I2sTransmitterConfigConverter::fromTransmitterClass(i2sTransmitterAgentConfig, configStruct);
  `uvm_info(get_type_name(), $sformatf("IN DRIVER- Converted cfg struct\n%p",configStruct), UVM_NONE)

   driveToBFMWhenTXSlave(packetStruct,configStruct);

  I2sTransmitterSeqItemConverter::toTransmitterClass(packetStruct,i2sTransmitterTransaction);
  `uvm_info(get_type_name(), $sformatf("IN DRIVER-After driving data to interface :: Received i2sTransmitterTransaction\n%s",i2sTransmitterTransaction.s
print()), UVM_NONE)

  seq_item_port.item_done();

 end

 endtask:driverBFMWhenTXSlave
```

Fig 3.23 *driveBFMWhenTxSlave task in i2s Transmitter driver proxy code snippet*

3.2.16 I2S Transmitter Monitor Proxy

I2sTransmitterMonitorProxy component is a class extending uvm_monitor. It gets the virtual handle of the I2S Transmitter Monitor bfm and using this handle we will call the bfm side methods.

We have config struct packet and Transaction struct packet (declared in i2s global package) and then will pass it to the I2sTransmitterMonitorProxy using the sampleData method declared in i2s transmitter monitor bfm. As the i2s transmitter monitor bfm interface cannot access the class-based I2sTransmitterTransaction data, and the class cannot access the struct type data, so we have to convert that into the transaction data type. Similarly, it converts the I2sTransmitterAgentConfig Struct data type into a config data type.

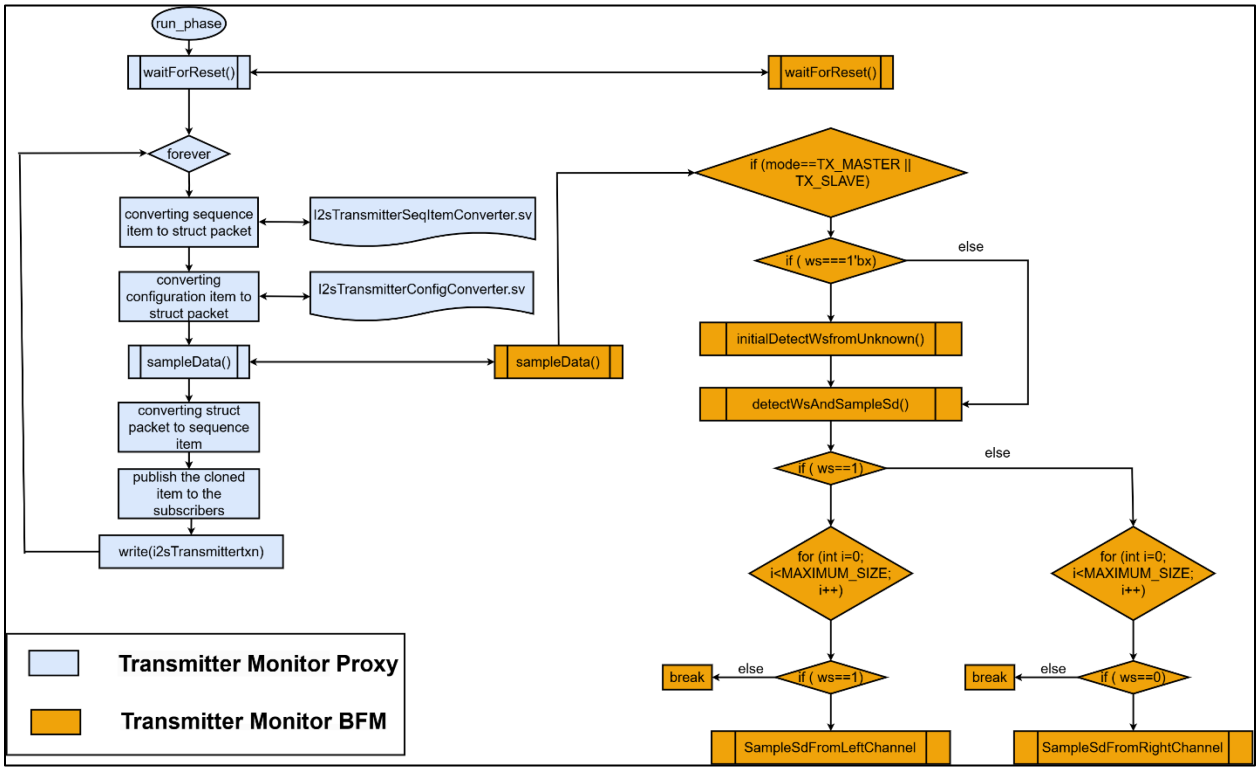

Fig 3.24 *Flowchart of communication between i2s Transmitter monitor proxy and i2s Transmitter monitor bfm*

```
task I2sTransmitterMonitorProxy::run_phase(uvm_phase phase);

  I2sTransmitterTransaction i2sTransmittertxn;

  `uvm_info(get_type_name(),"IN TRANSMITTER MONITOR: Running the Monitor Proxy", UVM_NONE)

  `uvm_info(get_type_name(), "IN TRANSMITTER MONITOR: Waiting for reset", UVM_NONE);
    i2sTransmitterMonitorBFM.waitForReset();
  `uvm_info(get_type_name(), "IN TRANSMITTER MONITOR: I2S: Reset detected", UVM_NONE);

  forever begin
     i2sTransferPacketStruct packetStruct;
     i2sTransferCfgStruct configStruct;

     I2sTransmitterSeqItemConverter::fromTransmitterClass(i2sTransmitterTransaction, packetStruct);
      `uvm_info(get_type_name(), $sformatf("IN TRANSMITTER MONITOR: Converted i2sTransmitterTransaction to struct\n%p",packetStruct), UVM_NONE)


     I2sTransmitterConfigConverter::fromTransmitterClass(i2sTransmitterAgentConfig, configStruct);

      `uvm_info(get_type_name(), $sformatf("IN TRANSMITTER MONITOR: Converted cfg struct\n%p",configStruct), UVM_NONE)

     i2sTransmitterMonitorBFM.sampleData(packetStruct,configStruct);

     I2sTransmitterSeqItemConverter::toTransmitterClass(packetStruct,i2sTransmitterTransaction);


     $cast(i2sTransmittertxn, i2sTransmitterTransaction.clone());
     `uvm_info(get_type_name(),$sformatf("IN TRANSMITTER MONITOR: Packet received from sample_data clone packet is \n %s",i2sTransmittertxn.sprint()),UVM_N
ONE)

     i2sTransmitterAnalysisPort.write(i2sTransmittertxn);

 end


endtask : run_phase
```

Fig 3.25 *run phase of i2s transmitter monitor proxy code snippet*

3.2.17 I2S Receiver Agent

I2S Receiver agent component is a class extending from uvm_agent. It gets the I2s ReceiverAgentConfig and based on that we will create and connect the components. It creates the I2sReceiverSequencer and I2sReceiverDriverProxy only if the I2sReceiverAgent is active which will depend on the value of is_active variable declared in the i2s ReceiverAgentConfig file. The I2sReceiverCoverage is created in build_phase if the has_coverage variable is TRUE which is declared in the I2sReceiverAgentConfig file. Please refer to Figure 3.26 for the I2S Receiver agent build_phase code snippet.

The I2S Receiver agent build phase has the creation of,

a. I2sReceiverSequencer
b. I2sReceiverDriverProxy
c. I2sReceiverMonitorProxy
d. I2sReceiverCoverage

```
function void I2sReceiverAgent::build_phase(uvm_phase phase);
  super.build_phase(phase);
  if(!uvm_config_db #(I2sReceiverAgentConfig)::get(this,"","I2sReceiverAgentConfig",i2sReceiverAgentConfig))
    `uvm_fatal("CONFIG","cannot get() the i2sReceiverAgentConfig from the uvm_config_db. have you set it?")

  i2sReceiverMonitorProxy=I2sReceiverMonitorProxy::type_id::create("i2sReceiverMonitorProxy",this);
    if(i2sReceiverAgentConfig.isActive==UVM_ACTIVE )
  begin
    $display("receiver isactive=%0s",i2sReceiverAgentConfig.isActive);
    i2sReceiverDriverProxy=I2sReceiverDriverProxy::type_id::create("i2sReceiverDriverProxy",this);
    i2sReceiverSequencer=I2sReceiverSequencer::type_id::create("i2sReceiverSequencer",this);

  end

  if(i2sReceiverAgentConfig.hasCoverage)
    begin
    i2sReceiverCoverage=I2sReceiverCoverage::type_id::create("i2sReceiverCoverage",this);
  end

  uvm_config_db#(I2sReceiverAgentConfig)::set(uvm_root::get(), "*","I2sReceiverAgentConfig",i2sReceiverAgentConfig);
    `uvm_info(get_type_name(), $sformatf("\nI2S_RECEIVER_AGENT_CONFIG\n%s",
                  i2sReceiverAgentConfig.sprint()),UVM_LOW);
endfunction : build_phase
```

Fig 3.26 *I2S Receiver agent build_phase code snippet.*

I2sReceiverAgentConfig handles declared in the above created components will be mapped here in the connect phase. The I2sReceiverDriverProxy and I2sReceiverSequencer are connected using TLM ports if the I2sReceiverAgent is active. The I2sReceiverCoverage's analysis_export will be connected to the I2sReceiverMonitorProxy's i2sReceiverAnalysisPort in connect_phase for transaction analysis and coverage collection.

Please refer to Figure 3.27 for the I2S Receiver agent connect phase code snippet.

```
function void I2sReceiverAgent::connect_phase(uvm_phase phase);

  if(i2sReceiverAgentConfig.isActive==UVM_ACTIVE)
  begin
    i2sReceiverDriverProxy.i2sReceiverAgentConfig=i2sReceiverAgentConfig;
    i2sReceiverSequencer.i2sReceiverAgentConfig=i2sReceiverAgentConfig;
    i2sReceiverDriverProxy.seq_item_port.connect(i2sReceiverSequencer.seq_item_export);
  end

  i2sReceiverMonitorProxy.i2sReceiverAgentConfig=i2sReceiverAgentConfig;

  if(i2sReceiverAgentConfig.hasCoverage)
    begin
    i2sReceiverMonitorProxy.i2sReceiverAnalysisPort.connect(i2sReceiverCoverage.analysis_export);
  end
endfunction : connect_phase
```

Fig 3.27 *I2S Receiver agent connect phase code snippet.*

3.2.18 I2S Receiver Sequencer

I2sReceiverSequencer component is a parameterized class of type I2sReceiverTransaction, extending uvm_sequencer. I2sReceiverSequencer sends the data from the I2sReceiverSequences to the I2sReceiverDriverProxy.

3.2.19 I2S Receiver Driver Proxy

I2sReceiverDriverProxy component is a parameterized class of type I2sReceiverTransaction, extending uvm_driver. It gets the virtual handle of the I2S Receiver driver bfm and using this handle we will call the bfm side methods.

As the I2S Receiver driver bfm interface cannot access the class-based I2S Receiver transaction data, so we have to convert that into struct data type. Similarly, it converts the I2s Receiver configuration values into struct data type. I2s Receiver driver proxy will call the converter class to convert the Receiver transaction packet and Receiver configuration packet into a struct data packet and struct configuration packet respectively (declared in i2s global package) and then will pass it to I2s Receiver driver bfm using methods like driveToBFMSclk and driverBFMWhenRXMaster declared in I2s Receiver driver bfm.

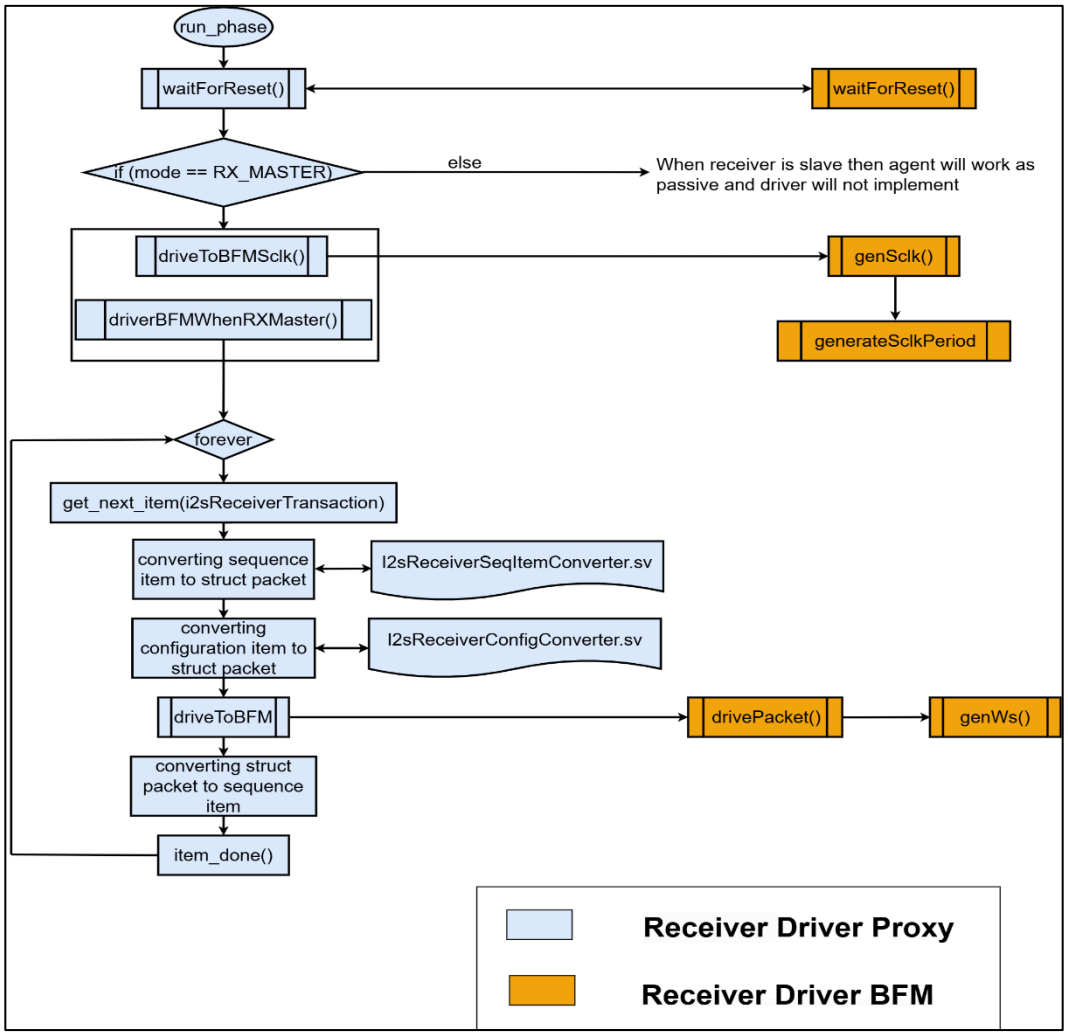

Fig 3.28 *Flowchart of communication between i2s Receiver driver proxy and i2s Receiver driver bfm*

```
task I2sReceiverDriverProxy::run_phase(uvm_phase phase);
  super.run_phase(phase);
   `uvm_info("START", "start of run phase in receiver Driver Proxy", UVM_HIGH);

  `uvm_info(get_type_name(), "Receiver Driver Proxy:: Waiting for reset", UVM_HIGH);
  i2sReceiverDriverBFM.waitForReset();
  `uvm_info(get_type_name(), "Receiver Driver Proxy :: Reset detected", UVM_HIGH);

   I2sReceiverConfigConverter::fromReceiverClass(i2sReceiverAgentConfig, configStruct);
    `uvm_info(get_type_name(), $sformatf("Converted cfg struct\n%p",configStruct), UVM_HIGH)

  if(configStruct.mode == RX_MASTER)
   begin

     fork

       driveToBFMSclk(configStruct);
       driverBFMWhenRXMaster(packetStruct,configStruct);
     join_any

   end
   else if(configStruct.mode == RX_SLAVE)
    begin
     `uvm_info(get_type_name(), $sformatf("Receiver will act as passive agent"), UVM_HIGH);
    end

 endtask : run_phase
```

Fig 3.29 *run phase of i2s Receiver driver proxy code snippet*

```
task I2sReceiverDriverProxy::driverBFMWhenRXMaster(inout i2sTransferPacketStruct packetStruct,input i2sTransferCfgStruct configStruct);
  `uvm_info("START", "Inside I2sReceiverDriverProxy", UVM_HIGH);

  forever begin

    seq_item_port.get_next_item(i2sReceiverTransaction);

    `uvm_info(get_type_name(), $sformatf("Received req\n%s",i2sReceiverTransaction.sprint()), UVM_HIGH)
    I2sReceiverSeqItemConverter::fromReceiverClass(i2sReceiverTransaction, packetStruct);
    `uvm_info(get_type_name(), $sformatf("Converted req struct\n%p",packetStruct), UVM_HIGH)
     I2sReceiverConfigConverter::fromReceiverClass(i2sReceiverAgentConfig, configStruct);
    `uvm_info(get_type_name(), $sformatf("Converted cfg struct\n%p",configStruct), UVM_HIGH)

    driveToBFM(packetStruct,configStruct);

    I2sReceiverSeqItemConverter::toReceiverClass(packetStruct,i2sReceiverTransaction);
    `uvm_info(get_type_name(), $sformatf("After :: Received req\n%s",i2sReceiverTransaction.sprint()), UVM_HIGH)

   seq_item_port.item_done();

  end
endtask
```

Fig 3.30 *driveBFMWhenRxSlave task in i2s Receiver driver proxy code snippet*

3.2.20 I2S Receiver Monitor Proxy

I2sReceiverMonitorProxy component is a class extending uvm_monitor. It gets the virtual handle of the I2S Receiver Monitor bfm and using this handle we will call the bfm side methods.

We have config struct packet and Transaction struct packet (declared in i2s global package) and then will pass it to the I2sReceiverMonitorProxy using the samplePacket method declared in i2s receiver monitor bfm. As the i2s receiver monitor bfm interface cannot access the class-based I2sReceiverTransaction data, and the class cannot access the struct type data, so we have to convert that into the transaction data type. Similarly, it converts the I2s ReceiverAgentConfig Struct data type into a config data type.

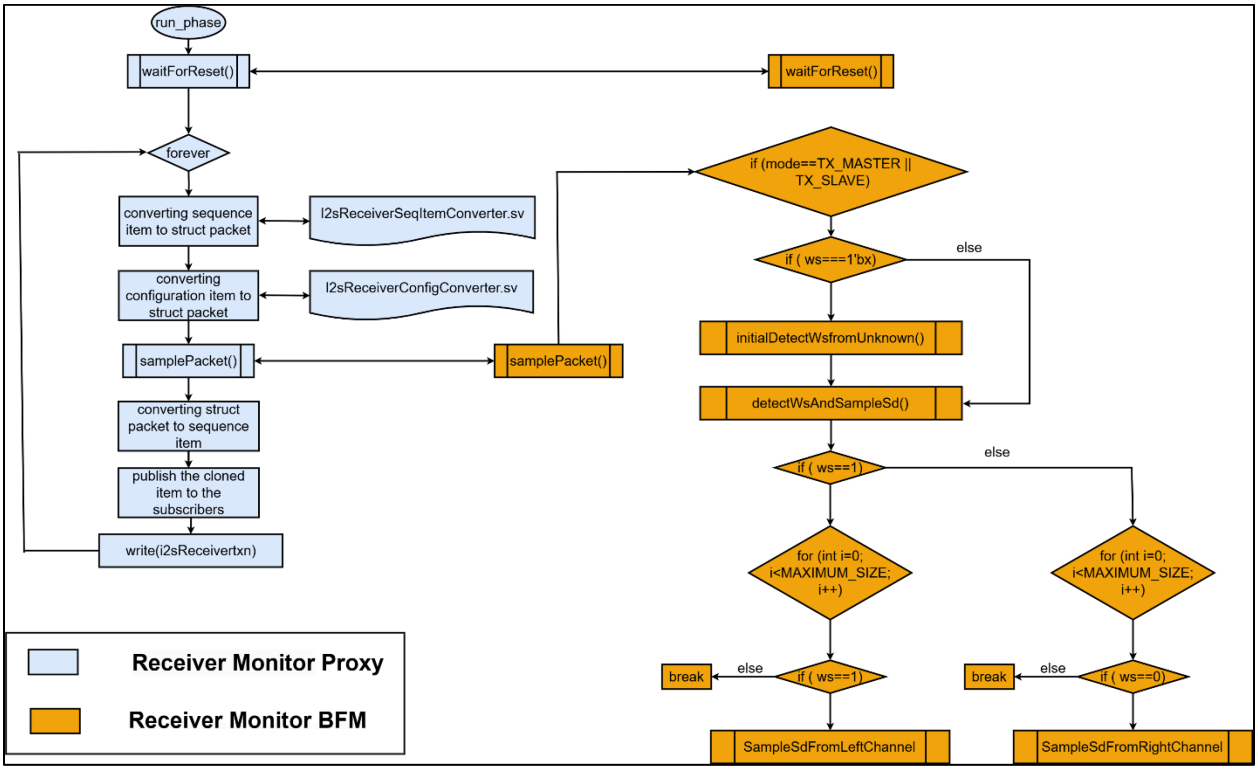

Fig 3.31 *Flowchart of communication between i2s Receiver monitor proxy and i2s Receiver monitor bfm*

```
task I2sReceiverMonitorProxy::run_phase(uvm_phase phase);

  I2sReceiverTransaction i2sReceiverTxn;

  `uvm_info(get_type_name(),"IN RECEIVER MONITOR: Running the Monitor Proxy", UVM_HIGH)

  `uvm_info(get_type_name(), "IN RECEIVER MONITOR: Waiting for reset", UVM_HIGH);
 i2sReceiverMonitorBFM.waitForReset();
  forever begin
    i2sTransferPacketStruct packetStruct;
    i2sTransferCfgStruct configStruct;

    I2sReceiverSeqItemConverter::fromReceiverClass(i2sReceiverTransaction, packetStruct);
    `uvm_info(get_type_name(), $sformatf("IN RECEIVER MONITOR: Converted req struct\n%p",packetStruct), UVM_HIGH)


    I2sReceiverConfigConverter::fromReceiverClass(i2sReceiverAgentConfig, configStruct);
    `uvm_info(get_type_name(), $sformatf("IN RECEIVER MONITOR: Converted cfg struct\n%p",configStruct), UVM_HIGH)

    i2sReceiverMonitorBFM.samplePacket(packetStruct,configStruct);

    I2sReceiverSeqItemConverter::toReceiverClass(packetStruct,i2sReceiverTransaction);


    $cast(i2sReceiverTxn, i2sReceiverTransaction.clone());
    `uvm_info(get_type_name(),$sformatf("IN RECEIVER MONITOR: Packet received from sample_data clone packet is  %s",i2sReceiverTxn.sprint()),UVM_HIGH)
    i2sReceiverAnalysisPort.write(i2sReceiverTxn);
end


endtask : run_phase
```

Fig 3.32 *run phase of i2s receiver monitor proxy code snippet*

3.2.21 UVM Verbosity

There are predefined UVM verbosity settings built into UVM (and OVM). These settings are included in the UVM src/uvm_object_globals.svh file and the settings are part of the enumerated

uvm_verbosity type definition. The settings actually have integer values that increment by 100 as shown below table

Table 3.2 *UVM verbosity Priorities*

Verbosity	**Default Value**
UVM_NONE	0(Highest Priority)
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500(Lowest Priority)

By default, when running a UVM simulation, all messages with verbosity settings of UVM_MEDIUM or lower (UVM_MEDIUM, UVM_LOW, and UVM_NONE) will print. Table 3.3 shows the Verbosity levels that have been used in this project.

Table 3.3 *Descriptions of each Verbosity level*

Verbosity	**Description**
UVM_NONE	UVM_NONE is level 0 and should be used to reduce report verbosity to a bare minimum of vital simulation regression suite messages.
UVM_LOW	UVM_LOW is level 100 and should be used to reduce report verbosity and only shows important messages
UVM_MEDIUM	UVM_MEDIUM is level 200 and should be used as the default $display command. If the verbosity isn't selected then, these messages will print by default as UVM_MEDIUM. This verbosity setting should not be used for any debugging messages or for standard test-passing messages.
UVM_HIGH	UVM_HIGH is level 300 and should be used to increase report verbosity by showing both failing and passing transaction information, but does not show annoying UVM phase status information after it has been established that the UVM phases are working properly
UVM_FULL	UVM_FULL is level 400 and should be used to increase report verbosity by showing UVM phase status information as well as both failing and passing transaction information.

Chapter 4

Directory Structure

4.1 Package content

The package structure diagram navigates users to find out the file locations, where it is located and which folder.

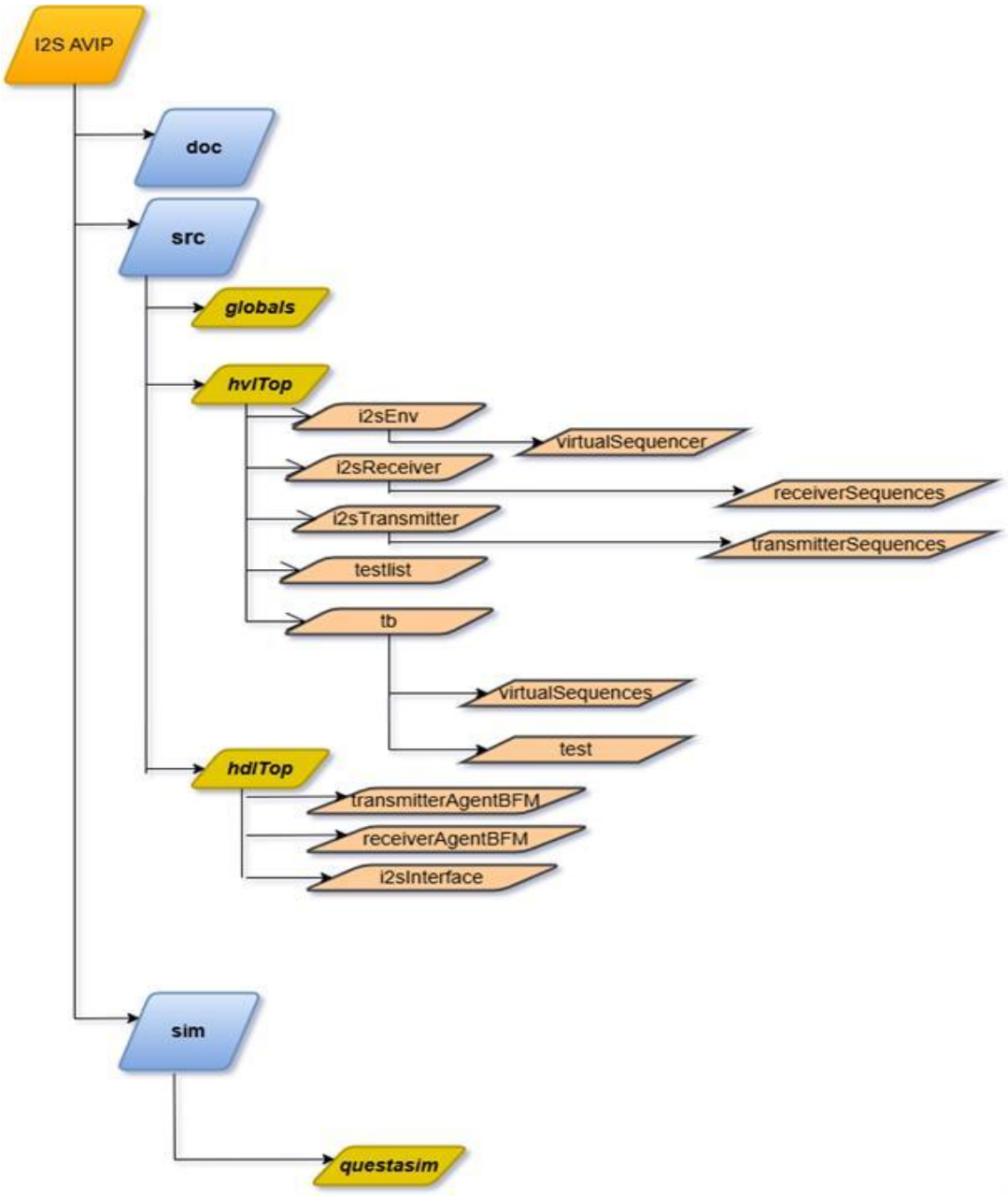

Fig 4.1 *Package Structure of I2S_AVIP*

Table 4.1 *Directory Path*

Directory	Description
i2s_avip/doc	Contains test bench architecture and components description and verification plan and assertion plan
i2s_avip/sim	Contains all simulating tools and I2sCompile.f file which contain all directories and compiling files
i2s_avip/src/globals	Contains globals package parameters (names,modes)
i2s_avip/src/hdl_top	Contains all bfm files (transmitterAgentBFM, receiverAgentBFM, i2sInterface)
i2s_avip/src/hdl_top/transmitterAgentBFM	Contains assertions file, transmitter agent, driver and monitor BFM files
i2s_avip/src/hdl_top/receiverAgentBFM	Contains assertions file, receiver agent, driver and monitor BFM files
i2s_avip/src/hdl_top/i2sInterface	Contains i2s interface file
i2s_avip/src/hvl_top	Contains all tb component folders (i2sTransmitter, i2sEnv, i2sReceiver, tb, testlist)
i2s_avip/src/hvl_top/tb	Contains test and virtual sequences folders
i2s_avip/src/hvl_top/tb/test	Contains all the test files
i2s_avip/src/hvl_top/tb/virtual sequences	Contains all the virtual sequences files
i2s_avip/src/hvl_top/testlist	Contains testlist for regression test
i2s_avip/src/hvl_top/i2sTransmitter	Contains transmitter agent, agent config, config converter, coverage, driver proxy, monitor proxy, package, sequence item converter, sequencer, transaction and transmitter sequences folder
i2s_avip/src/hvl_top/i2sTransmitter /transmitterSequences	Contains all transmitter sequence files
i2s_avip/src/hvl_top/i2sReceiver	Contains receiver agent, agent config, config converter, coverage, driver proxy, monitor proxy, package, sequence item converter, sequencer, transaction and receiver sequences folder
i2s_avip/src/hvl_top/i2sReceiver/receiver Sequences	Contains all receiver sequence files
i2s_avip/src/hvl_top/i2sEnv	Contains environment along with its config file, package file, Scoreboard file and virtual sequencer folder
i2s_avip/src/hvl_top/i2sEnv/virtualSequencer	Contains virtual sequencer file

Chapter 5

Configurations

5.1 Global package variables

The global variables declared in the global package files are given in this chapter

Table 5.1 *Global package variables*

Name	Type	Description
DATA_WIDTH	parameter	Sets the maximum width of the data packet. Default value = 8
MAXIMUM_SIZE	parameter	Sets the maximum size of the Serial Data Default value = 4
WS_DEFAULT	parameter	Sets the default value of WS. WS_DEFAULT=1'bx
hasCoverageEnum	enum	Enables coverage. TRUE=1'b1 FALSE=1'b0
numOfChannelsEnum	enum	Sets the number of channels used. MONO= 1 STEREO= 2
modeTypeEnum	enum	Used to select mode. TX_MASTER=2'b00 TX_SLAVE=2'b01 RX_MASTER=2'b10 RX_SLAVE=2'b11
clockrateFrequencyEnum	enum	Used to select clock rate frequency. KHZ_8=8000 KHZ_16=16000 KHZ_24=24000 KHZ_32=32000 KHZ_48=48000 KHZ_96=96000 KHZ_192=192000
wordSelectPeriodEnum	enum	Used to select word select period. WS_PERIOD_2_BYTE = 16, WS_PERIOD_4_BYTE=32, WS_PERIOD_6_BYTE=48, WS_PERIOD_8_BYTE=64
noofBitsTransferEnum	enum	Used to select number of bits transfer. BITS_8 = 8,

		BITS_16 =16, BITS_24 =24, BITS_32 =32
dataTransferDirectionEnum	enum	Contains direction of data transfer MSB_FIRST=1'b0; LSB_FIRST=1'b1
i2sStateEnum	enum	Contains the states of our protocol RESET_DEACTIVATED, RESET_ACTIVATED, IDLE, LEFT_CHANNEL, RIGHT_CHANNEL
i2sTransferCfgStruct	struct	Contains transfer configuration contents bit[1:0]mode; int clockratefrequency; int wordSelectPeriod; int clockPeriod; int sclkFrequency; bit Sclk; int numOfChannels; bit dataTransferDirection;

Configuration used

1.Environment Configuration

2.Transmitter Agent Configuration

3.Receiver Agent Configuration

5.2 Transmitter Agent configuration

Table 5.2 *TransmitterAgentConfig*

Name	**Type**	**Description**
isActive	enum	Tells whether UVM Transmitter agent is active or passive.
hasCoverage	enum	Used for enabling transmitter agent coverage
mode	enum	Used to select mode on which protocol should work
clockratefrequency	enum	Defines the clock rate frequency of the protocol

wordSelectPeriod	enum	Tells the Word Select Period of transmitter.
numOfChannels	enum	Used to indicate the number of channels used.
dataTransferDirection	enum	Tells us the direction of the data transfer
Sclk	bit	Tells the initial value of Sclk
clockPeriod	int	Used to determine the clock period of the serial clock
sclkFrequency	int	Used to define the frequency of sclk generated

5.3 Receiver Agent Configuration

Table 5.3 *ReceiverAgentConfig*

Name	**Type**	**Description**
isActive	enum	Tells whether UVM Receiver agent is active or passive.
hasCoverage	enum	Used for enabling Receiver agent coverage
mode	enum	Used to select mode on which protocol should work
clockratefrequency	enum	Defines the clock rate frequency of the protocol
wordSelectPeriod	enum	Tells the Word Select Period of receiver.
numOfChannels	enum	Used to indicate the number of channels used.
dataTransferDirection	enum	Tells us the direction of the data transfer
Sclk	bit	Tells the initial value of Sclk
clkperiod	int	Used to determine the clock period of the serial clock
sclkFrequency	int	Used to define the frequency of sclk generated
delayfortxSd	int	Used to define the delay between WS and start of SD.

5.4 Environment Configuration

Table 5.4 *EnvConfig*

Name	Type	Description
hasScoreboard	bit	Tells if it has scoreboard or not
hasVirtualSequencer	bit	Tells if it has virtual sequencer or not
i2sTransmitterAgentConfig	I2sTransmitterAgentConfig	Contains the configurations used in transmitter agent
i2sReceiverAgentConfig	I2sReceiverAgentConfig	Contains the configurations used in receiver agent

Chapter 6

Verification Plan

6.1 Verification plan

Verification plan is an important step in Verification flow, it defines the plan of an entire project and verifies the different scenarios to achieve the test plan.

A Verification plan defines what needs to be verified in Design under test(DUT) and then drives the verification strategy. As an example, the verification plan may define the features that a system has and these may get translated into the coverage metrics that are set.

6.2 Template of Verification Plan

For more information of Verification Plan click below link:

I2S_Verification_Plan - Google Sheets

SI NO.	FEATURE	SUB- FEATURE	DESCRIPTION	Testcase Name
1.1	Data Transmission (Transmitter and Receiver Word Length are Equal)	8bit	Data transfer to send are 8bits	I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest
		16bit	Data transfer to send are 16bits	I2sWriteOperationWith16bitdataTxMasterRxSlaveWith8khzTest I2sWriteOperationWith16bitdataRxMasterTxSlaveWith48khzTest
		24 bit	Data transfer to send are 24bits	I2sWriteOperationWith24bitdataTxMasterRxSlaveWith24khzTest I2sWriteOperationWith24bitdataRxMasterTxSlaveWith16khzTest
		32bit	Data transfer to send are 32bits	I2sWriteOperationWith32bitdataTxMasterRxSlaveWith192khzTest I2sWriteOperationWith32bitdataRxMasterTxSlaveWith96khzTest
1.2	Data Transmission (Transmitter and Receiver Word Length are Unequal)	Receiver word length > Transmitter word length	Padding the LSBs with zeros	I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP16bitWith8khzTest I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP16bitWith16khzTest I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP16bitWith96khzTest I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP32bitWith192khzTest I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP32bitWith24khzTest I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP48bitWith48khzTest
		Receiver word length < Transmitter word length	Ignoring the LSBs	I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP64bitWith96khzTest I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP64bitWith192khzTest I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP64bitWith32khzTest I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP32bitWith48khzTest I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP48bitWith16khzTest I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP48bitWith192khzTest

Fig 6.1 *Verification plan template*

6.3 Sections for different test Scenarios

6.3.1 Directed tests

These directed tests provide explicit stimulus to the design inputs, run the design in simulation, and check the behavior of the design against expected results.

These tests describe the different combinations of number of bits transfer (8/16/24/32) with respect to Word Select Period (16/32/48/64) for different frequencies.

Table 6.1 *Checking coverage closure for No of bits transfers w.r.t. WSP for different frequencies*

S. No.	Modes	No of bits transfer	Test names	Description
1	TxMaster-Rxslave	8 bit	I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest	Checking the 8 bit data transfer with TXWSP=16 and RXWSP=16
2	RxMaster-TxSlave	8 bit	I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest	Checking the 8 bit data transfer with TXWSP=16 and RXWSP=16
3	TxMaster-Rxslave	16 bit	I2sWriteOperationWith16bitdataTxMasterRxSlaveWith8khzTest	Checking the 16 bit data transfer with TXWSP=32 and RXWSP=32
4	RxMaster-TxSlave	16 bit	I2sWriteOperationWith16bitdataRxMasterTxSlaveWith48khzTest	Checking the 16 bit data transfer with TXWSP=32 and RXWSP=32
5	TxMaster-Rxslave	24 bit	I2sWriteOperationWith24bitdataTxMasterRxSlaveWith24khzTest	Checking the 24 bit data transfer with TXWSP=48 and RXWSP=48
6	RxMaster-TxSlave	24 bit	I2sWriteOperationWith24bitdataRxMasterTxSlaveWith16khzTest	Checking the 24 bit data transfer with TXWSP=48 and RXWSP=48
7	TxMaster-Rxslave	32 bit	I2sWriteOperationWith32bitdataTxMasterRxSlaveWith192khzTest	Checking the 32 bit data transfer with TXWSP=64 and RXWSP=64
8	RxMaster-TxSlave	32 bit	I2sWriteOperationWith32bitdataRxMasterTxSlaveWith96khzTest	Checking the 32 bit data transfer with TXWSP=64 and RXWSP=64
9	RxMaster-TxSlave	8 bit	I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP16bitWith8khzTest	Checking the 8 bit data transfer with TXWSP=16 and RXWSP=32
10	RxMaster-TxSlave	8 bit	I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP16bitWith16khzTest	Checking the 8 bit data transfer with TXWSP=16 and RXWSP=48

11	RxMaster-TxSlave	8 bit	I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP16bitWith96khzTest	Checking the 8 bit data transfer with TXWSP=16 and RXWSP=64
12	RxMaster-TxSlave	16 bit	I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP32bitWith192khzTest	Checking the 16 bit data transfer with TXWSP=32 and RXWSP=48
13	RxMaster-TxSlave	16 bit	I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP32bitWith24khzTest	Checking the 16 bit data transfer with TXWSP=32 and RXWSP=64
14	RxMaster-TxSlave	24 bit	I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP48bitWith48khzTest	Checking the 24 bit data transfer with TXWSP=48 and RXWSP=64
15	RxMaster-TxSlave	32 bit	I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP64bitWith96khzTest	Checking the 32 bit data transfer with TXWSP=64 and RXWSP=32
16	RxMaster-TxSlave	32 bit	I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP64bitWith192khzTest	Checking the 32 bit data transfer with TXWSP=64 and RXWSP=48
17	RxMaster-TxSlave	32 bit	I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP64bitWith32khzTest	Checking the 32 bit data transfer with TXWSP=64 and RXWSP=16
18	RxMaster-TxSlave	16 bit	I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP32bitWith48khzTest	Checking the 16 bit data transfer with TXWSP=32 and RXWSP=16
19	RxMaster-TxSlave	24 bit	I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP48bitWith16khzTest	Checking the 24 bit data transfer with TXWSP=48 and RXWSP=16
20	RxMaster-TxSlave	24 bit	I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP48bitWith192khzTest	Checking the 24 bit data transfer with TXWSP=48 and RXWSP=32

Chapter 7

Assertion Plan

7.1 Assertion Plan overview

Assertion plan is an important step in verification flow, which validates the behavior of design at every instance.

7.1.1 What are assertions?

- An assertion specifies the behavior of the system.
- Piece of verification code that monitors a design implementation for compliance with the specifications.
- Directive to a verification tool that the tool should attempt to prove/assume/count a given property using formal methods.
- Helps in detecting functional bugs early in the design cycle.
- Can be used in simulation, formal verification, and emulation environments.

7.1.2 Why do we use it?

- Assertions are primarily used to validate the behavior of a design.
- Assertions can be used to provide functional coverage and to flag that input stimulus, which is used for validation, does not conform to assumed requirements.
- Assertions are used to find more bugs and source the bugs faster.

7.1.3 Benefits of Assertions

- Improves observability of the design.
- Improves debugging of the design.
- Improves documentation of the design.
- Enable formal verification by proving or checking properties automatically.

7.2 Template of Assertion Plan

Template for Assertion plan is done in an excel sheet and refer to link below:

I2S_Assertion_Plan - Google Sheets

7.3 Transmitter Assertion Conditions

7.3.1 SdZeroWhenReset

```
23   property SdZeroWhenReset();
24    @(posedge clk)
25       (rst==0) |-> sd==0;
26   endproperty
27
28  TX_SD_ZER0_WHEN_RESET :assert property(SdZeroWhenReset)
29    $info("TX_SD_ZER0_WHEN_RESET: ASSERTED");
30    else
31    $error("TX_SD_ZER0_WHEN_RESET:NOT_ASSERTED");
```

Fig 7.1 *Transmitter SdZeroWhenReset Assertion*

The SdZeroWhenReset property is evaluated as follows:

- Initially, it will check for posedge of clk.
- If (rst==0) then at the same clock cycle sd should be zero.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.3.2 wsNotUnknown

```
33   property wsNotUnknown();
34       @(posedge sclk) disable iff (!rst)
35          ($changed(ws) && !($isunknown(ws))) |=> ($stable(ws) until $changed(ws));
36    endproperty
37
38  TX_WS_NOT_UNKNOWN: assert property (wsNotUnknown)
39   $info("TX_WS_NOT_UNKNOWN : ASSERTED");
40   else
41     $error("TX_WS_NOT_UNKNOWN : NOT_ASSERTED");
```

Fig 7.2 *Transmitter wsNotUnknown Assertion*

The wsNotUnknown property is evaluated as follows:

- Initially, it will check for posedge of sclk, and the property is disabled if rst is low.
- If ws changes its value and ws does not have unknown value, then from next clock cycle ws should be stable until it changes.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised

7.3.3 sdNotUnknown

```
43   property sdNotUnknown();
44      @(posedge sclk) disable iff (!rst)
45         $changed(ws)  |-> (!($isunknown(sd)) until $changed(ws));
46
47   endproperty
48
49   TX_SD_NOT_UNKNOWN: assert property (sdNotUnknown)
50   $info("TX_SD_NOT_UNKNOWN: ASSERTED");
51   else
52     $error("TX_SD_NOT_UNKNOWN : NOT_ASSERTED");
```

Fig 7.3 *Transmitter sdNotUnknown Assertion*

The sdNotUnknown property is evaluated as follows:

- Initially, it will check for posedge of sclk, and the property is disabled if rst is low.
- If ws changes its value, then from next clock cycle sd should have known value until ws changes.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.4 Receiver Assertion Conditions

7.4.1 SdZeroWhenReset

```
18   property SdZeroWhenReset();
19    @(posedge clk)
20       (rst==0) |-> sd==0;
21   endproperty
22   RX_SD_ZER0_WHEN_RESET :assert property(SdZeroWhenReset)
23    $info("RX_SD_ZER0_WHEN_RESET: ASSERTED");
24    else
25    $error("RX_SD_ZER0_WHEN_RESET:NOT ASSERTED");
26
```

Fig 7.4 *Receiver sdZeroWhenReset Assertion*

The SdZeroWhenReset property is evaluated as follows:

- Initially, it will check for posedge of clk.
- If (rst==0) then at the same clock cycle sd should be zero.

- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.4.2 wsNotUnknown

```
28   property wsNotUnknown();
29       @(posedge sclk) disable iff (!rst)
30           ($changed(ws) && !($isunknown(ws))) |=> ($stable(ws) until $changed(ws));
31   endproperty
32
33   RX_WS_NOT_UNKNOWN: assert property (wsNotUnknown)
34   $info("RX_WS_NOT_UNKNOWN: ASSERTED");
35   else
36     $error("RX_WS_NOT_UNKNOWN : NOT ASSERTED");
```

Fig 7.5 *Receiver wsNotUnknown Assertion*

The wsNotUnknown property is evaluated as follows:

- Initially, it will check for posedge of clk, and the property is disabled if rst is low.
- If ws changes its value and ws does not have unknown value, then from next clock cycle ws should be stable until it changes.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

7.4.3 sdNotUnknown

```
38   property sdNotUnknown();
39      @(posedge sclk) disable iff (!rst)
40           $changed(ws)  |-> (!($isunknown(sd)) until $changed(ws));
41   endproperty
42
43   RX_SD_NOT_UNKNOWN: assert property (sdNotUnknown)
44   $info("RX_SD_NOT_UNKNOWN: ASSERTED");
45   else
46     $error("RX_SD_NOT_UNKNOWN : NOT ASSERTED");
```

Fig 7.6 *Receiver sdNotUnknown Assertion*

The sdNotUnknown property is evaluated as follows:

- Initially, it will check for posedge of clk, and the property is disabled if rst is low.

- If ws changes its value, then from next clock cycle sd should have known value until ws changes.
- If the above alignment conditions hold, then the property is true. Otherwise, the property will fail, and an error message is raised.

Chapter 8

Coverage Plan

8.1 Template of Coverage Plan

Template for Coverage plan is done in an excel sheet and refer to link below:

I2S Coverage Plan

8.2 Functional Coverage

- Functional coverage is the coverage data generated from the user defined functional coverage model and assertions usually written in System Verilog. During simulation, the simulator generates functional coverage based on the stimulus. Looking at the functional coverage data, one can identify the portions of the DUT [Features] verified. Also, it helps us to target the DUT features that are unverified.
- The reason for switching to functional coverage is that we can create the bins manually as per our requirement while in the code coverage it is generated by the system itself.

8.3 Uvm_Subscriber

This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection "subscribes" this component to any transactions emitted by the connected analysis port. Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

```
virtual class  uvm_subscriber #(type T=int) extends uvm_component;
typedef uvm_subscriber #(T) this_type;

// Port: analysis_export
//
// This export provides access to the write method, which derived subscribers
// must implement.

uvm_analysis_imp #(T, this_type) analysis_export;

// Function: new
//
// Creates and initializes an instance of this class using the normal
// constructor arguments for <uvm_component>: ~name~ is the name of the
// instance, and ~parent~ is the handle to the hierarchical parent, if any.

function new (string name, uvm_component parent);
  super.new(name, parent);
  analysis_export = new("analysis_imp", this);
endfunction

// Function: write
//
// A pure virtual method that must be defined in each subclass. Access
// to this method by outside components should be done via the
// analysis_export.

 pure virtual function void write(T t);

 endclass
```

Fig 8. 1 *uvm_subscriber*

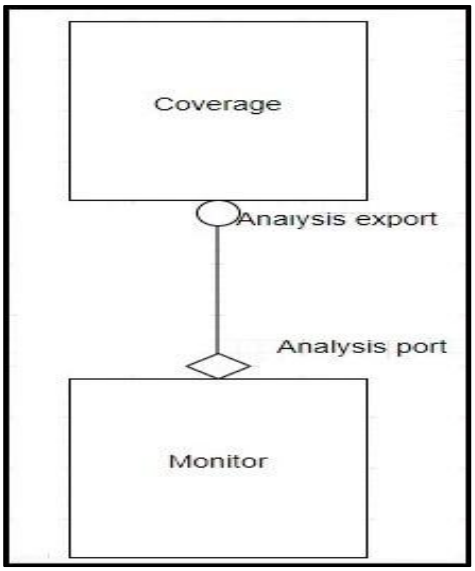

Fig 8.2 *Monitor and coverage connection*

8.3.1 Analysis Export

This export provides access to the write method, which derived subscribers must implement.

8.3.2 Write function

The write function is to process the incoming transactions.

```
154 function void I2sTransmitterCoverage::write(I2sTransmitterTransaction t);
155   `uvm_info(get_type_name(), $sformatf("Before Calling the Sample Method"),UVM_HIGH);
156    i2sTransmitterTransactionCovergroup.sample(i2sTransmitterAgentConfig,t);
157   `uvm_info(get_type_name(), $sformatf("After Calling the Sample Method"),UVM_HIGH);
158 endfunction: write
```

Fig 8.3 *Write function*

8.4 Covergroup

```
11   covergroup i2sTransmitterTransactionCovergroup with function sample (I2sTransmitterAgentConfig i2sTransmitterAgentConfig,I2sTransmitterTransaction i2sTransmitterTransaction);
12  option.per_instance = 1;
```

Fig 8.4 *Covergroup*

The above red mark points in Figure covergroup is explained below:-

1. With function sample: - It is used to pass a variable to covergroup.

2. Parameter based on which the coverpoint is generated.

3. Per Instance Coverage - 'option.per_instance'

In your test bench, you might have instantiated coverage_group multiple times. By default, System Verilog collects all the coverage data from all the instances. You might have more than one generator and they might generate different streams of transaction. In this case you may want to see separate reports. Using this option, you can keep track of coverage for each instance.

3.1. option.per_instance=1

Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report.

8.5 Bucket

In this we create the single bin for the multiple values i.e.

```
87 NUMOFBITSTRANSFER_TX_CP : coverpoint i2sTransmitterTransaction.txNumOfBitsTransfer{
88   option.comment = "Num of Bits Transfer";
89   bins BITS_8  = {8};
90   bins BITS_16 = {16};
91   bins BITS_24 = {24};
92   bins BITS_32 = {32};
93   illegal_bins BINS_INVALID={[33:$]};
94 }
```

Fig 8.5 *Bucket*

- In the above point we can see bins for values of number of bits transfer as 8,16,24 and 32.

8.6 Coverpoints

There we created the bins based on the serial data left channel range.

```
32 LEFTCHANNELSERIALDATARANGE_0_CP : coverpoint i2sTransmitterTransaction.txSdLeftChannel[0] {
33   option.comment = "left channel serial data value range ";
34   bins SD_LEFT_CHANNEL_0_LOW_VALID_RANGE = {[0:50]};
35   bins SD_LEFT_CHANNEL_0_MID_VALID_RANGE = {[51:200]};
36   bins SD_LEFT_CHANNEL_0_HIGH_VALID_RANGE = {[201:255]};
37  }
```

Fig 8.6 *Coverpoints*

8.7 Cross Coverpoints

Cross allows keeping track of information which is received simultaneous on more than one cover point. Cross coverage is specified using the cross construct.

Cross coverage between coverpoints NUMOFBITSTRANSFER_TX_CP and WORDSELECT_TX_CP

```
109  WORDSELECTPERIOD_TX_CP : coverpoint i2sTransmitterAgentConfig.wordSelectPeriod{
110   option.comment = " WORD SELECT PERIOD";
111
112    bins WS_PERIOD_2_BYTE = {16};
113    bins WS_PERIOD_4_BYTE = {32};
114    bins WS_PERIOD_6_BYTE = {48};
115    bins WS_PERIOD_8_BYTE = {64};
116    illegal_bins WS_PERIOD_INVALID= {[65:$]};
117 }
118
119 NUMOFBITSTRANSFER_TX_X_WORD_SELECT_TX_CP:cross NUMOFBITSTRANSFER_TX_CP,WORDSELECT_TX_CP;
```

Fig 8. 7 *Cross Coverpoints*

8.7.1 Illegal bins

```
87 NUMOFBITSTRANSFER_TX_CP : coverpoint i2sTransmitterTransaction.txNumOfBitsTransfer{
88   option.comment = "Num of Bits Transfer";
89   bins BITS_8  = {8};
90   bins BITS_16 = {16};
91   bins BITS_24 = {24};
92   bins BITS_32 = {32};
93   illegal_bins BINS_INVALID={[33:$]};
```

Fig 8.8 *Illegal bins*

Illegal bins are used when we don't want to have the particular value e.g. - we don't want to have the number of bits transfer more than 32.

8.8 Creation of the covergroup

```
133 function I2sTransmitterCoverage::new(string name = "I2sTransmitterCoverage", uvm_component parent = null);
134   super.new(name, parent);
135   i2sTransmitterTransactionCovergroup=new();
136 endfunction : new
```

Fig 8.9 *Creation of covergroup*

In this function the creation of the covergroup is done with the new as shown in the figure

8.9 Sampling of the covergroup

In this the sampling of the covergroup is done in the write function as shown below

```
154 function void I2sTransmitterCoverage::write(I2sTransmitterTransaction t);
155   `uvm_info(get_type_name(), $sformatf("Before Calling the Sample Method"),UVM_HIGH);
156    i2sTransmitterTransactionCovergroup.sample(i2sTransmitterAgentConfig,t);
157   `uvm_info(get_type_name(), $sformatf("After Calling the Sample Method"),UVM_HIGH);
158 endfunction: write
```

Fig 8.10 *Sampling of the covergroup*

8.10 Checking for the coverage

1. Make Compile
2. Make simulate
3. Open the log file

```
Testname: I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest
Log file path: I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest.log
Waveform: vsim -view I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/waveform.wlf &
Coverage report: firefox I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/html_cov_report/index.html &
```

Fig 8.11 *Simulation log file path*

4.Search for the coverage (There will be the full coverage) in the log file.

5.To check the individual coverage bins hit open the coverage report as shown: -

```
Testname: I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest
Log file path: I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest.log
Waveform: vsim -view I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/waveform.wlf &
Coverage report: firefox I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest/html_cov_report/index.html &
```

Fig 8.12 *Coverage report path*

Then new html window will open

Coverage Summary by Type:

Total Coverage:					40.49%	**50.23%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	80	80	0	1	100.00%	**100.00%**
Statements	2613	1635	978	1	62.57%	**62.57%**
Branches	1809	456	1353	1	25.20%	**25.20%**
FEC Conditions	4	0	4	1	0.00%	**0.00%**
Toggles	1304	178	1126	1	13.65%	**13.65%**
Assertions	6	6	0	1	100.00%	**100.00%**

Fig 8.13 *HTML window showing all coverage*

Here click on the covergroup there we can see the per instance created and inside that each coverpoint with bins is present there.

Covergroup type:

i2sTransmitterTransactionCovergroup

Summary	Total Bins	Hits	Hit %
Coverpoints	44	44	100.00%
Crosses	8	8	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CLOCKFREQUENCY_TX_CP	7	7	0	100.00%	100.00%	100.00%
LEFTCHANNELSERIALDATARANGE_0_CP	3	3	0	100.00%	100.00%	100.00%
LEFTCHANNELSERIALDATARANGE_1_CP	3	3	0	100.00%	100.00%	100.00%
LEFTCHANNELSERIALDATARANGE_2_CP	3	3	0	100.00%	100.00%	100.00%
LEFTCHANNELSERIALDATARANGE_3_CP	3	3	0	100.00%	100.00%	100.00%
NUMOFBITSTRANSFER_TX_CP	4	4	0	100.00%	100.00%	100.00%
NUMOFCHANNELS_TX_CP	2	2	0	100.00%	100.00%	100.00%
RIGHTCHANNELSERIALDATARANGE_0_CP	3	3	0	100.00%	100.00%	100.00%
RIGHTCHANNELSERIALDATARANGE_1_CP	3	3	0	100.00%	100.00%	100.00%
RIGHTCHANNELSERIALDATARANGE_2_CP	3	3	0	100.00%	100.00%	100.00%
RIGHTCHANNELSERIALDATARANGE_3_CP	3	3	0	100.00%	100.00%	100.00%
SERIALCLOCK_TX_CP	1	1	0	100.00%	100.00%	100.00%
WORDSELECT_TX_CP	2	2	0	100.00%	100.00%	100.00%
WORDSELECTPERIOD_TX_CP	4	4	0	100.00%	100.00%	100.00%

Fig 8.14 *All coverpoints present in the Transmitter Covergroup*

Covergroup type:

i2sReceiverTransactionCovergroup

Summary	Total Bins	Hits	Hit %
Coverpoints	20	20	100.00%
Crosses	8	8	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CLOCKFREQUENCY_RX_CP	7	7	0	100.00%	100.00%	100.00%
NUMOFBITSTRANSFER_RX_CP	4	4	0	100.00%	100.00%	100.00%
NUMOFCHANNELS_RX_CP	2	2	0	100.00%	100.00%	100.00%
SERIALCLOCK_RX_CP	1	1	0	100.00%	100.00%	100.00%
WORDSELECT_RX_CP	2	2	0	100.00%	100.00%	100.00%
WORDSELECTPERIOD_RX_CP	4	4	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
NUMOFBITSTRANSFER_RX_CP_X_WORD_SELECT_RX_CP	8	8	0	100.00%	100.00%	100.00%

Fig 8.15 *All coverpoints present in the Receiver Covergroup*

Coverpoint: NUMOFBITSTRANSFER_TX_CP

Search:

Bin Name	At Least	Hits
illegal_bin BINS_INVALID	--	0
BITS_8	1	20
BITS_16	1	38
BITS_24	1	28
BITS_32	1	30

Fig 8.16 *Individual Coverpoint Hit*

Chapter 9

Test cases

9.1 Test Flow

In the test, there is virtual sequence and in virtual sequence, sequences are there, sequence_item get started in sequences, sequences will start in virtual sequence and virtual sequence will start in Test.

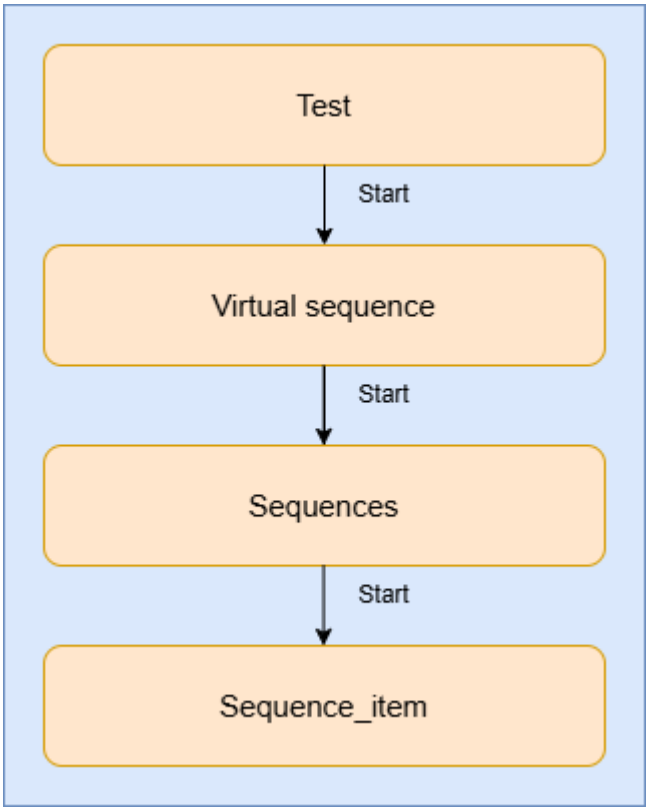

Fig 9.1 *Test flow*

9.2 I2S Test Cases Flowchart

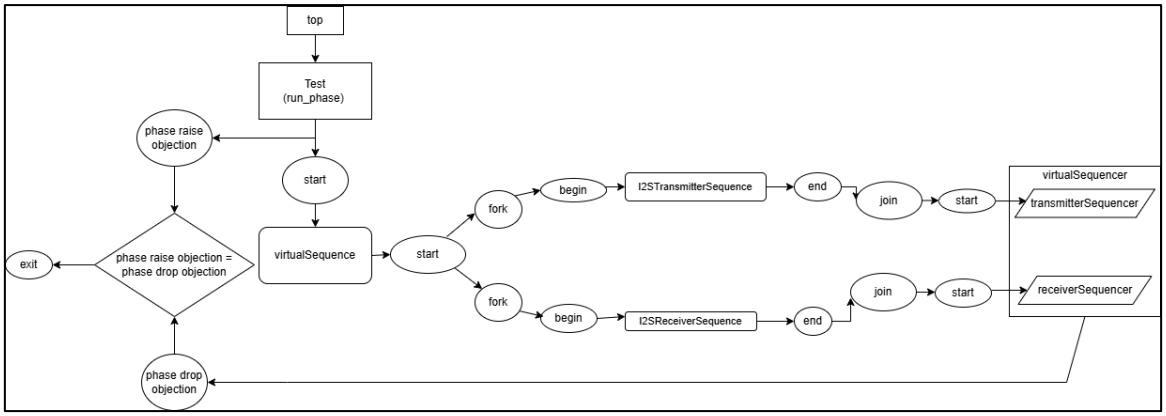

Fig 9.2 *I2S test cases flow chart*

9.3 Transaction

Table 9.1 *Transaction Signals*

Variables	Type	Description
txWs	logic	Indicates whether data should go for left channel or right channel
txSdLeftChannel	bit	Carries each byte of left channel serial data from transmitter to receiver
txSdRightChannel	bit	Carries each byte of right channel serial data from transmitter to receiver
txNumOfBitsTransfer	enum	Indicates number of bits transfer 8/16/24/32

9.3.1 I2S Transmitter Transaction

```
13    constraint txSdLeftChannelSize{soft txSdLeftChannel.size() == txNumOfBitsTransfer/DATA_WIDTH; }
14    constraint txSdRightChannelSize{soft txSdRightChannel.size() == txNumOfBitsTransfer/DATA_WIDTH; }
15
```

Fig 9.3 *Constraints of I2S Transmitter transaction*

Table 9.2 *Describing constraints of I2STransmitterTransaction*

Constraint	Description
txSdLeftChannelSize	The constraint ensures that SdLeftChannelSize is equal to numofbitstransfer divided by DATA_WIDTH.
txSdRightChannelSize	The constraint ensures that SdRightChannelSize is equal to numofbitstransfer divided by DATA_WIDTH.

- Written functions for do_copy, do_compare, do_print methods, $casting is used to copy the data member values and compare the data member values and by using a printer, printing the I2sTransmitterTransaction signals.

```
30 function void I2sTransmitterTransaction::do_copy (uvm_object rhs);
31   I2sTransmitterTransaction i2sTransmitterTransactionCopyObj;
32
33   if(!$cast(i2sTransmitterTransactionCopyObj,rhs)) begin
34     `uvm_fatal("do_copy","cast of the rhs object failed")
35   end
36   super.do_copy(rhs);
37
38   txWs = i2sTransmitterTransactionCopyObj.txWs;
39   txSdLeftChannel = i2sTransmitterTransactionCopyObj.txSdLeftChannel;
40   txSdRightChannel = i2sTransmitterTransactionCopyObj.txSdRightChannel;
41   txNumOfBitsTransfer = i2sTransmitterTransactionCopyObj.txNumOfBitsTransfer;
42
43 endfunction : do_copy
```

Fig 9.4 *do_copy method of Transmitter Transaction*

```
45 function bit  I2sTransmitterTransaction::do_compare (uvm_object rhs,uvm_comparer comparer);
46   I2sTransmitterTransaction i2sTransmitterTransactionCopyObj;
47
48   if(!$cast(i2sTransmitterTransactionCopyObj,rhs)) begin
49   `uvm_fatal("FATAL_I2S_TRANSMITTER_SEQ_ITEM_DO_COMPARE_FAILED","cast of the rhs object failed")
50   return 0;
51   end
52
53   return super.do_compare(rhs,comparer) &&
54   txWs == i2sTransmitterTransactionCopyObj.txWs &&
55   txSdLeftChannel == i2sTransmitterTransactionCopyObj.txSdLeftChannel &&
56   txSdRightChannel == i2sTransmitterTransactionCopyObj.txSdRightChannel &&
57   txNumOfBitsTransfer == i2sTransmitterTransactionCopyObj.txNumOfBitsTransfer;
58   endfunction : do_compare
```

Fig 9.5 *do_compare method of Transmitter Transaction*

```
61 function void I2sTransmitterTransaction::do_print(uvm_printer printer);
62   super.do_print(printer);
63
64   printer.print_field($sformatf("WORD SELECT"),this.txWs,1,UVM_DEC);
65
66   foreach(txSdLeftChannel[i]) begin
67   printer.print_field($sformatf("LEFT CHANNEL SERIALDATA[%0d]",i),this.txSdLeftChannel[i],$bits(txSdLeftChannel[i]),UVM_BIN);
68   end
69   foreach(txSdRightChannel[i]) begin
70   printer.print_field($sformatf("RIGHT CHANNEL SERIALDATA[%0d]",i),this.txSdRightChannel[i],$bits(txSdRightChannel[i]),UVM_BIN);
71   end
72
73   printer.print_field($sformatf("NO_OF_BITS_TRANSFER"),this.txNumOfBitsTransfer,$bits(txNumOfBitsTransfer),UVM_DEC);
74
75 endfunction : do_print
```

Fig 9.6 *do_print method of Transmitter Transaction*

9.3.2 I2S Receiver Transaction

```
25 function void I2sReceiverTransaction::do_copy (uvm_object rhs);
26   I2sReceiverTransaction i2sReceiverTransactionCopyObj;
27
28   if(!$cast(i2sReceiverTransactionCopyObj,rhs)) begin
29     `uvm_fatal("do_copy","cast of the rhs object failed")
30   end
31   super.do_copy(rhs);
32
33   rxWs = i2sReceiverTransactionCopyObj.rxWs;
34   rxSdLeftChannel = i2sReceiverTransactionCopyObj.rxSdLeftChannel;
35   rxSdRightChannel= i2sReceiverTransactionCopyObj.rxSdRightChannel;
36   rxNumOfBitsTransfer = i2sReceiverTransactionCopyObj.rxNumOfBitsTransfer;
37
38 endfunction : do_copy
```

Fig 9.7 *do_copy method of Receiver Transaction*

```
40 function bit  I2sReceiverTransaction::do_compare (uvm_object rhs,uvm_comparer comparer);
41   I2sReceiverTransaction i2sReceiverTransactionCopyObj;
42
43   if(!$cast(i2sReceiverTransactionCopyObj,rhs)) begin
44   `uvm_fatal("FATAL_I2S_RECEIVER_SEQ_ITEM_DO_COMPARE_FAILED","cast of the rhs object failed")
45   return 0;
46   end
47
48   return super.do_compare(rhs,comparer) &&
49   rxWs == i2sReceiverTransactionCopyObj.rxWs &&
50   rxSdLeftChannel == i2sReceiverTransactionCopyObj.rxSdLeftChannel &&
51   rxSdRightChannel== i2sReceiverTransactionCopyObj.rxSdRightChannel &&
52   rxNumOfBitsTransfer == i2sReceiverTransactionCopyObj.rxNumOfBitsTransfer;
53 endfunction : do_compare
```

Fig 9.8 *do_compare method of Receiver Transaction*

```
56 function void I2sReceiverTransaction::do_print(uvm_printer printer);
57   super.do_print(printer);
58
59   printer.print_field($sformatf("WORD SELECT"),this.rxWs,1,UVM_BIN);
60   foreach(rxSdLeftChannel[i]) begin
61   printer.print_field($sformatf("LEFT CHANNEL SERIALDATA[%0d]",i),this.rxSdLeftChannel[i],$bits(rxSdLeftChannel[i]),UVM_BIN);
62   end
63
64  foreach(rxSdRightChannel[i]) begin
65   printer.print_field($sformatf("RIGHT CHANNEL SERIALDATA[%0d]",i),this.rxSdRightChannel[i],$bits(rxSdRightChannel[i]),UVM_BIN);
66   end
67  printer.print_field($sformatf("NO_OF_BITS_TRANSFER"),this.rxNumOfBitsTransfer,$bits(rxNumOfBitsTransfer),UVM_DEC);
68
69 endfunction : do_print
```

Fig 9.9 *do_print method of Receiver Transaction*

9.4 Sequences

A UVM Sequence is an object that contains a behavior for generating stimulus. A sequence generates a series of sequence_item's and sends it to the driver via sequencer. Sequence is written by extending the uvm_sequence.

9.4.1 Methods

Table 9.3 *Sequence Methods*

Method	**Description**
new	Creates and initializes a new sequence object
start_item	This method will send the request item to the sequencer, which will forward it to the driver
req.randomize()	Generate the transaction(seq_item).
finish_item	Wait for acknowledgement or response

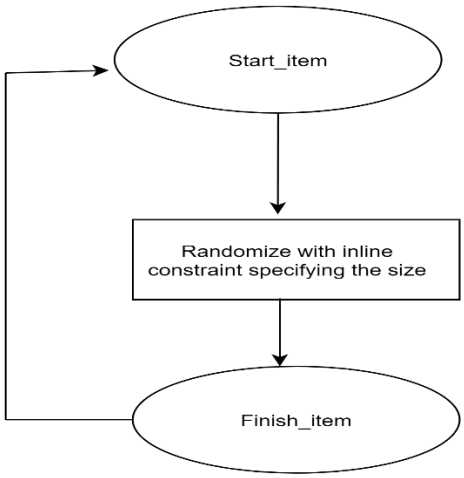

Fig 9.10 *Flow chart for sequence methods*

Table 9.4 *Describing transmitter and receiver sequences*

Sections	**Transmitter Sequence**	**Receiver Sequence**	**Description**
BaseSequence	I2sTransmitterBaseSeq	I2sReceiverBaseSeq	Base class is extended from uvm_sequence and parameterized with transaction (I2sTransmitterTransaction, I2sReceiverTransaction)
Data transfers	I2sTransmitterSequence	I2sReceiverSequence	Extended from base sequence. Based on a request from the driver, the task will drive the transactions. In between start_ item and finish_ item, the task body randomizes the transaction object (req) with inline constraints, assigning values from sequence variables, and reports a fatal error if randomization fails.

In the transmitter sequence body, req is created, and start_item(req) initiates the sequence. The transaction (req) is then randomized with inline constraints, assigning values from sequence variables (transmitter signals), followed by finish_item(req) to complete the sequence.

```
23 task I2sTransmitterWrite8bitTransferSeq::body();
24   super.body();
25
26   start_item(i2sTransmitterTransaction);
27   if(!i2sTransmitterTransaction.randomize() with {
28                              txWs == txWsSeq;
29                               foreach(txSdLeftChannelSeq[i]){
30                                  txSdLeftChannel[i]  == txSdLeftChannelSeq[i]};
31                               foreach(txSdRightChannelSeq[i]){
32                                  txSdRightChannel[i]  == txSdRightChannelSeq[i]};
33                               txNumOfBitsTransfer  == txNumOfBitsTransferSeq;
34                                }) begin
35       `uvm_error(get_type_name(), "Randomization failed")
36  end
37
38   foreach(i2sTransmitterTransaction.txSdLeftChannel[i]) begin
39     $display("Left Channel SD[%0d]=%b",i,i2sTransmitterTransaction.txSdLeftChannel[i]);
40    end
41   foreach(i2sTransmitterTransaction.txSdRightChannel[i]) begin
42     $display("Right Channel SD[%0d]=%b",i,i2sTransmitterTransaction.txSdRightChannel[i]);
43    end
44
45   i2sTransmitterTransaction.print();
46   finish_item(i2sTransmitterTransaction);
47
48 endtask:body
```

Fig 9.11 *I2sTransmitterWrite8bitTransferSeq body method*

```
12   constraint txSdLeftChannelSeq_c{soft txSdLeftChannelSeq.size() == txNumOfBitsTransferSeq/DATA_WIDTH; }
13   constraint txSdRightChannelSeq_c{soft txSdRightChannelSeq.size() == txNumOfBitsTransferSeq/DATA_WIDTH; }
```

Fig 9.12 *Constraints of I2sTransmitterWrite8BitTransferSeq*

```
21 task I2sReceiverWrite8bitTransferSeq::body();
22   super.body();
23   start_item(req);
24   if(!req.randomize() with {rxWs == rxWsSeq;
25                             }) begin
26       `uvm_error(get_type_name(), "Randomization failed")
27   end
28   req.print();
29   finish_item(req);
30
31 endtask:body
```

Fig 9.13 *I2sReceiverWrite8BitTransferSeq body method*

9.5 Virtual sequences

A virtual sequence is a container to start multiple sequences on different sequencers in the environment. This virtual sequence is usually executed by a virtual sequencer which has handles

to actual sequencers. This need for a virtual sequence arises when you require different sequences to be run on different environments.

9.5.1 Virtual sequence base class

Virtual sequence base class is extended from uvm_sequence and parameterized with uvm_transaction. Declaring p_sequencer as macro, handles virtual sequencer and transmitter, receiver sequencer and environment config.

```
class I2sVirtualBaseSeq extends uvm_sequence#(uvm_sequence_item);
  `uvm_object_utils(I2sVirtualBaseSeq)

   `uvm_declare_p_sequencer(I2sVirtualSequencer)

   I2sTransmitterSequencer  i2sTransmitterSequencer;
   I2sReceiverSequencer   i2sReceiverSequencer;
   I2sEnvConfig i2sEnvConfig;

   extern function new(string name="I2sVirtualBaseSeq");
   extern task body();
endclass:I2sVirtualBaseSeq
```

Fig 9.14 *Virtual base sequence*

In virtual sequence body method, Getting the env configurations and Dynamic casting of p_sequencer and m_sequencer. Connect the transmitter sequencer and receiver sequencer in sequencer with local transmitter sequencer and receiver sequencer.

```
task I2sVirtualBaseSeq::body();
  if(!uvm_config_db#(I2sEnvConfig) ::get(null,get_full_name(),"I2sEnvConfig",i2sEnvConfig)) begin
  `uvm_fatal(get_type_name(),"cannot get() env_cfg from uvm_config_db.Have you set() it?")
  end

  //dynamic casting of p_sequencer and m_sequencer
  if(!$cast(p_sequencer,m_sequencer))begin
  `uvm_error(get_full_name(),"Virtual sequencer pointer cast failed")
  end

  //connecting transmitter sequencer and receiver sequencer present in p_sequencer to local transmitter sequencer and receiver sequencer
  i2sTransmitterSequencer = p_sequencer.i2sTransmitterSequencer;
  i2sReceiverSequencer    = p_sequencer.i2sReceiverSequencer;

endtask:body
```

Fig 9.15 *Virtual base sequence body*

In the virtual sequence body method, when transmitter acts as master and receiver as slave creating transmitter sequence handle and starts the transmitter sequence within repeat statement as shown in Fig 9.15.

When transmitter acts as slave and receiver as master, creating transmitter and receiver sequence handles and starts the transmitter and receiver sequences within fork join and transmitter sequence within repeat statement as shown in Fig 9.16.

```
task I2sVirtual8bitWriteOperationTxMasterRxSlaveSeq::body();

repeat(2)
 begin
 i2sTransmitterWrite8bitTransferSeq = I2sTransmitterWrite8bitTransferSeq::type_id::create("i2sTransmitterWrite8bitTransferSeq");
  `uvm_info(get_type_name(), $sformatf("Inside Body Seq start I2sVirtual8bitWriteOperationTxMasterRxSlaveSeq"), UVM_NONE);

  if(!i2sTransmitterWrite8bitTransferSeq.randomize() with {txWsSeq==1;
                                                           txNumOfBitsTransferSeq == (p_sequencer.i2sTransmitterSequencer.i2sTransmitterAgentConfig.wordSelectPeriod/2);
                                                           }) begin

      `uvm_error(get_type_name(), "Randomization failed : Inside I2sTransmitterWrite8bitTransferSeq")
 end


`uvm_info(get_type_name(), "Attempting to start the virtual sequence", UVM_NONE);
i2sTransmitterWrite8bitTransferSeq.start(p_sequencer.i2sTransmitterSequencer);

 end
endtask : body
```

Fig 9.16 *I2sVirtual8bitWriteOperationTxMasterRxSlave sequence body*

```
task I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq::body();

repeat(2)
 begin

  i2sReceiverWrite8bitTransferSeq = I2sReceiverWrite8bitTransferSeq::type_id::create("i2sReceiverWrite8bitTransferSeq");
  i2sTransmitterWrite8bitTransferSeq = I2sTransmitterWrite8bitTransferSeq::type_id::create("i2sTransmitterWrite8bitTransferSeq");

  `uvm_info(get_type_name(), $sformatf("Inside Body Seq Start: I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq"), UVM_NONE);

 if(!i2sReceiverWrite8bitTransferSeq.randomize() with {rxWsSeq==1;
                                                       }) begin
      `uvm_error(get_type_name(), "Randomization failed : Inside I2sReceiverWrite8bitTransferSeq")
  end


 if (!i2sTransmitterWrite8bitTransferSeq.randomize() with {txNumOfBitsTransferSeq == (p_sequencer.i2sTransmitterSequencer.i2sTransmitterAgentConfig.wordSelectPeriod/2);
                                                           }) begin
    `uvm_error(get_type_name(), "Randomization failed: Inside I2sTransmitterWrite8bitTransferSeq")
  end


  fork

    begin
  `uvm_info(get_type_name(), "Starting Receiver Sequence", UVM_LOW);

      i2sReceiverWrite8bitTransferSeq.start(p_sequencer.i2sReceiverSequencer);
    end
    begin
        `uvm_info(get_type_name(), "Starting Transmitter Sequence", UVM_LOW);
      i2sTransmitterWrite8bitTransferSeq.start(p_sequencer.i2sTransmitterSequencer);
     end
  join

  `uvm_info(get_type_name(), "Fork_join Completed",UVM_NONE);
end

endtask : body
```

Fig 9.17 *I2sVirtual8bitWriteOperationRxMasterTxSlave sequence body*

Table 9.5: *Describing virtual sequences*

Virtual Sequences	Description
I2sVirtual8bitWriteOperationTxMasterRxSlaveSeq	Inside the I2sVirtual8bitWriteOperationTxMasterRxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq	Inside the I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual16bitWriteOperationTxMasterRxSlaveSeq	Inside the I2sVirtual16bitWriteOperationTxMasterRxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual16bitWriteOperationRxMasterTxSlaveSeq	Inside the I2sVirtual16bitWriteOperationRxMasterTxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual24bitWriteOperationRxMasterTxSlaveSeq	Inside the I2sVirtual24bitWriteOperationRxMasterTxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual24bitWriteOperationTxMasterRxSlaveSeq	Inside the I2sVirtual24bitWriteOperationTxMasterRxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual32bitWriteOperationRxMasterTxSlaveSeq	Inside the I2sVirtual32bitWriteOperationRxMasterTxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2

I2sVirtual32bitWriteOperationTxMasterRxSlaveSeq	Inside the I2sVirtual32bitWriteOperationTxMasterRxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP16bitSeq	Inside the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP16bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP16bitSeq	Inside the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP16bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP16bitSeq	Inside the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP16bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP32bitSeq	Inside the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP32bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP32bitSeq	Inside the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP32bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP48bitSeq	Inside the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP48bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline

	constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP64bitSeq	Inside the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP64bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP64bitSeq	Inside the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP64bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP64bitSeq	Inside the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP64bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP32bitSeq	Inside the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP32bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP48bitSeq	Inside the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP48bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP48bitSeq	Inside the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP48bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer as wordSelectPeriod/2
I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP32bitSeq	Inside the I2sVirtualRandomWriteOperationTxMasterRxSl

	aveWithTxWSP32bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer less than or equal to wordSelectPeriod/2
I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP48bitSeq	Inside the I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP48bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer less than or equal to wordSelectPeriod/2
I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP64bitSeq	Inside the I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP64bitSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer less than or equal to wordSelectPeriod/2
I2sVirtualRandomWriteOperationRxMasterTxSlaveSeq	Inside the I2sVirtualRandomWriteOperationTxMasterRxSlaveSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize ws as 1 and NumOfBitsTransfer less than or equal to wordSelectPeriod/2
I2sVirtualWriteOperationDataTransferErrorSeq	Inside the I2sVirtualWriteOperationDataTransferErrorSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize NumOfBitsTransfer less than or equal to wordSelectPeriod/2
I2sVirtualWriteOperationWithInvalidWSPErrorSeq	Inside the I2sVirtualWriteOperationDataTransferErrorSeq extending from base class. Declaring handles for transmitter and receiver sequences, using inline constraints to randomize NumOfBitsTransfer equal to wordSelectPeriod/2

9.6 Test Cases

The uvm_test class defines the test scenario and verification goals.

A) In base test, declaring the handles for environment config and environment class.

```
 4 class I2sBaseTest extends uvm_test;
 5   `uvm_component_utils(I2sBaseTest)
 6
 7   I2sEnv i2sEnv;
 8   I2sEnvConfig i2sEnvConfig;
 9   I2sVirtualBaseSeq i2sVirtualBaseSeq;
10
11   extern function new(string name = "I2sBaseTest", uvm_component parent = null);
12   extern virtual function void build_phase(uvm_phase phase);
13   extern virtual function void setupEnvConfig();
14   extern virtual function void setupTransmitterAgentConfig();
15   extern virtual function void setupReceiverAgentConfig();
16   extern virtual function void end_of_elaboration_phase(uvm_phase phase);
17   extern virtual task run_phase(uvm_phase phase);
18 endclass : I2sBaseTest
```

Fig 9.18 *Base test*

B) In build phase, calling the setupEnvConfig and constructing the environment handle.

C) Inside setupEnvConfig function, constructing the environment config class handle. With the help of this I2sEnvironmentConfig handle all the required fields in the I2S_AVIP config class have been set up with respective values and then calling the setupTransmitterAgentConfig and setupReceiverAgentConfig functions.

```
31 function void I2sBaseTest::setupEnvConfig();
32   i2sEnvConfig.hasScoreboard = 1;
33   i2sEnvConfig.hasVirtualSequencer = 1;
34
35   uvm_config_db #(I2sEnvConfig)::set(this,"*","I2sEnvConfig",i2sEnvConfig);
36   `uvm_info(get_type_name(),$sformatf("i2sEnvConfig = \n %0p", i2sEnvConfig.sprint()),UVM_NONE)
37   setupTransmitterAgentConfig();
38   setupReceiverAgentConfig();
39 endfunction: setupEnvConfig
```

Fig 9.19 *Setup Environment Config*

In setupTransmitterAgentConfig function, i2sTransmitterAgentConfig class handle which is in i2sEnvConfig class has been constructed with the help of this handle all the required fields(hasCoverage, is_active) in i2sTransmitterAgentConfig class has been setup.

```
41 function void I2sBaseTest::setupTransmitterAgentConfig();
42     i2sEnvConfig.i2sTransmitterAgentConfig=I2sTransmitterAgentConfig::type_id::create("I2sTransmitterAgentConfig", this);
43     i2sEnvConfig.i2sTransmitterAgentConfig.isActive     = uvm_active_passive_enum'(UVM_ACTIVE);
44     i2sEnvConfig.i2sTransmitterAgentConfig.hasCoverage  = hasCoverageEnum'(TRUE);
45     i2sEnvConfig.i2sTransmitterAgentConfig.mode  = modeTypeEnum'(TX_MASTER);
46     i2sEnvConfig.i2sTransmitterAgentConfig.dataTransferDirection  = dataTransferDirectionEnum'(MSB_FIRST);
47
48     uvm_config_db #(I2sTransmitterAgentConfig)::set(this,"*","I2sTransmitterAgentConfig",i2sEnvConfig.i2sTransmitterAgentConfig);
49 endfunction: setupTransmitterAgentConfig
```

Fig 9.20 *Transmitter Agent Config setup*

D) In setupReceiverAgentConfig function, i2sReceiverAgentConfig class handle which is in i2sEnvConfig class has been constructed with the help of this handle all the required fields(hasCoverage, is_active) in i2sReceiverAgentConfig class has been setup followed by the end of the elaboration phase used to print the topology.

```
51 function void I2sBaseTest::setupReceiverAgentConfig();
52     i2sEnvConfig.i2sReceiverAgentConfig=I2sReceiverAgentConfig::type_id::create("I2sReceiverAgentConfig", this);
53     i2sEnvConfig.i2sReceiverAgentConfig.isActive     = uvm_active_passive_enum'(UVM_ACTIVE);
54     i2sEnvConfig.i2sReceiverAgentConfig.hasCoverage  = hasCoverageEnum'(TRUE);
55     i2sEnvConfig.i2sReceiverAgentConfig.mode  = modeTypeEnum'(RX_SLAVE);
56     i2sEnvConfig.i2sReceiverAgentConfig.dataTransferDirection  = dataTransferDirectionEnum'(MSB_FIRST);
57
58     uvm_config_db #(I2sReceiverAgentConfig)::set(this,"*","I2sReceiverAgentConfig",i2sEnvConfig.i2sReceiverAgentConfig);
59 endfunction: setupReceiverAgentConfig
```

Fig 9.21 *Receiver Agent Config setup*

Extend the “I2sWriteOperationWith” particular bit transfer test from base test and declare virtual sequence handle then create virtual sequence in test, and start the virtual sequence in run_phase, raise and drop objection.

```
 4 class I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest extends I2sBaseTest;
 5   `uvm_component_utils(I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest)
 6
 7   I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq i2sVirtual8bitWriteOperationRxMasterTxSlaveSeq;
 8
 9   extern function new(string name = "I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest", uvm_component parent = null);
10   extern virtual task run_phase(uvm_phase phase);
11   extern virtual function void setupTransmitterAgentConfig();
12   extern virtual function void setupReceiverAgentConfig();
13
14 endclass : I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest
```

Fig 9.22 *Example for I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest*

```
39 task I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest::run_phase(uvm_phase phase);
40
41   i2sVirtual8bitWriteOperationRxMasterTxSlaveSeq = I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq::type_id::create("i2sVirtual8bitWriteOperationRxMasterTxSlaveSeq");
42   `uvm_info(get_type_name(), $sformatf("Inside run_phase I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest"), UVM_LOW);
43
44   phase.raise_objection(this);
45   i2sVirtual8bitWriteOperationRxMasterTxSlaveSeq.start(i2sEnv.i2sVirtualSequencer);
46   #10;
47   phase.drop_objection(this);
48 endtask : run_phase
```

Fig 9.23 *Run phase of I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest*

Table 9.6: *Tests*

Test Names	Description
I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest	Extend test from base test and created the I2sVirtual8bitWriteOperationTxMasterRxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with transmitter as Master and Receiver as Slave with 48khz frequency.
I2sWriteOperationWith16bitdataTxMasterRxSlaveWith8khzTest	Extend test from base test and created the I2sVirtual16bitWriteOperationTxMasterRxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master and Receiver as Slave with 8khz frequency.
I2sWriteOperationWith24bitdataTxMasterRxSlaveWith24khzTest	Extend test from base test and created the I2sVirtual24bitWriteOperationTxMasterRxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master and Receiver as Slave with 24khz frequency
I2sWriteOperationWith32bitdataTxMasterRxSlaveWith192khzTest	Extend test from base test and created the I2sVirtual32bitWriteOperationTxMasterRxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master and Receiver as Slave with 192khz frequency
I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest	Extend test from base test and created the I2sVirtual8bitWriteOperationRxMasterTxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Receiver as Master with 8khz frequency
I2sWriteOperationWith16bitdataRxMasterTxSlaveWith48khzTest	Extend test from base test and created the I2sVirtual16bitWriteOperationRxMasterTxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with

	Transmitter as Slave and Receiver as Master with 48khz frequency
I2sWriteOperationWith24bitdataRxMasterTxSlaveWith16khzTest	Extend test from base test and created the I2sVirtual24bitWriteOperationRxMasterTxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Receiver as Master with 16khz frequency
I2sWriteOperationWith32bitdataRxMasterTxSlaveWith96khzTest	Extend test from base test and created the I2sVirtual32bitWriteOperationRxMasterTxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Receiver as Master with 96khz frequency
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP16bitWith8khzTest	Extend test from base test and created the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP16bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave with Word Select Period as 16bits and Receiver as Master with Word Select Period as 32 bits with 8khz frequency
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP16bitWith16khzTest	Extend test from base test and created the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP16bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 16bits and Receiver as Master with Word Select Period as 48 bits with 16khz frequency
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP16bitWith96khzTest	Extend test from base test and created the I2sVirtual8bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP16bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 16bits and

	Receiver as Master with Word Select Period as 64bits with 96khz frequency
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP32bitWith192khzTest	Extend test from base test and created the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP32bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 32bits and Receiver as Master with Word Select Period as 48 bits with 192khz frequency
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP32bitWith24khzTest	Extend test from base test and created the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP32bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 32 bits and Receiver as Master with Word Select Period as 64 bits with 24khz frequency
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP48bitWith48khzTest	Extend test from base test and created the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP64bitTxWSP48bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 48 bits and Receiver as Master with Word Select Period as 64 bits with 48khz frequency
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP64bitWith96khzTest	Extend test from base test and created the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP64bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 64 bits and Receiver as Master with Word Select Period as 32 bits with 96khz frequency
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP64bitWith192khzTest	Extend test from base test and created the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP48bitTxWSP64bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter

	as Slave and Word Select Period as 48 bits and Receiver as Master with Word Select Period as 64 bits with 192khz frequency
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP64bitWith32khzTest	Extend test from base test and created the I2sVirtual32bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP64bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 64 bits and Receiver as Master with Word Select Period as 16 bits with 32khz frequency
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP32bitWith48khzTest	Extend test from base test and created the I2sVirtual16bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP32bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 32 bits and Receiver as Master with Word Select Period as 16 bits with 48khz frequency
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP48bitWith16khzTest	Extend test from base test and created the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP16bitTxWSP48bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 48 bits and Receiver as Master with Word Select Period as 16 bits with 16khz frequency
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP48bitWith192khzTest	Extend test from base test and created the I2sVirtual24bitWriteOperationRxMasterTxSlaveWithRxWSP32bitTxWSP48bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Word Select Period as 48 bits and Receiver as Master with Word Select Period as 32 bits with 192khz frequency
I2sWriteOperationRandomTxMasterRxSlaveWithTxWSP32bitWith16khzTest	Extend test from base test and created the I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP32bitSeq virtual sequence handle and starting the sequences in between

	phase raise and drop objection and enforcing a write operation with Transmitter as Master with Word Select Period as 32 bits and Receiver as Slave with 16khz frequency
I2sWriteOperationRandomTxMasterRxSlaveWithTxWSP48bitWith96khzTest	Extend test from base test and created the I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP48bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master with Word Select Period as 48 bits and Receiver as Slave with 96khz frequency
I2sWriteOperationRandomTxMasterRxSlaveWithTxWSP64bitWith32khzTest	Extend test from base test and created the I2sVirtualRandomWriteOperationTxMasterRxSlaveWithTxWSP64bitSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master with Word Select Period as 64 bits and Receiver as Slave with 32khz frequency
I2sWriteOperationRandomRxMasterTxSlaveWith32khzTest	Extend test from base test and created the I2sVirtualRandomWriteOperationRxMasterTxSlaveSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Slave and Receiver as Master with 32khz frequency
I2sWriteOperationDataTransferErrorTest	Extend test from base test and created the I2sVirtualWriteOperationDataTransferErrorSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master and Receiver as Slave with 48khz frequency and passing LSB of Serial data first.
I2sWriteOperationWithInvalidWSPErrorTest	Extend test from base test and created the I2sVirtualWriteOperationWithInvalidWSPErrorSeq virtual sequence handle and starting the sequences in between phase raise and drop objection and enforcing a write operation with Transmitter as Master and Receiver as Slave

	with 48khz frequency with invalid Word select period as 128.

9.7 Testlists

Regression list for I2S

Table 9.7: *Testlists*

Test Names	Description
I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khzTest	Checks for an 8bitdataTxMasterRxSlave Write operation with frequency 48khz.
I2sWriteOperationWith16bitdataTxMasterRxSlaveWith8khzTest	Checks for an 16bitdataTxMasterRxSlave Write operation with frequency 8khz.
I2sWriteOperationWith24bitdataTxMasterRxSlaveWith24khzTest	Checks for an 24bitdataTxMasterRxSlave Write operation with frequency 24khz.
I2sWriteOperationWith32bitdataTxMasterRxSlaveWith192khzTest	Checks for an 32bitdataTxMasterRxSlave Write operation with frequency 192khz.
I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khzTest	Checks for an 8bitdataRxMasterTxSlave Write operation with frequency 8khz.
I2sWriteOperationWith16bitdataRxMasterTxSlaveWith48khzTest	Checks for an 16bitdataRxMasterTxSlave Write operation with frequency 48khz.
I2sWriteOperationWith24bitdataRxMasterTxSlaveWith16khzTest	Checks for an 24bitdataRxMasterTxSlave Write operation with frequency 16khz.
I2sWriteOperationWith32bitdataRxMasterTxSlaveWith96khzTest	Checks for an 32bitdataRxMasterTxSlave Write operation with frequency 96khz.
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP16bitWith8khzTest	Checks for an 8bitdataRxMasterTxSlaveWith Rx Word Select Period as 32bit and Tx Word Select Period and16bit Write operation with frequency 16khz.
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP16bitWith16khzTest	Checks for an 8bitdataRxMasterTxSlaveWith Rx Word Select Period as 32bit and Tx Word Select Period and16bit Write operation with frequency 16khz.
I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP16bitWith96khzTest	Checks for an 8bitdataRxMasterTxSlaveWith Rx Word Select Period as 64bit and Tx Word Select Period and16bit Write operation with frequency 96khz.
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP32bitWith192khzTest	Checks for an 16bitdataRxMasterTxSlaveWith Rx Word Select Period as 48bit and Tx Word Select

	Period and32bit Write operation with frequency 192khz.
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP32bitWith24khzTest	Checks for an 16bitdataRxMasterTxSlaveWith Rx Word Select Period as 64bit and Tx Word Select Period and 32bit Write operation with frequency 24khz.
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP48bitWith48khzTest	Checks for an 24bitdataRxMasterTxSlaveWith Rx Word Select Period as 64bit and Tx Word Select Period and 48bit Write operation with frequency 48khz.
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP64bitWith96khzTest	Checks for an 32bitdataRxMasterTxSlaveWith Rx Word Select Period as 32bit and Tx Word Select Period and 64bit Write operation with frequency 96khz.
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP48bitTxWSP64bitWith192khzTest	Checks for an 32bitdataRxMasterTxSlaveWith Rx Word Select Period as 48bit and Tx Word Select Period and 64bit Write operation with frequency 192khz.
I2sWriteOperationWith32bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP64bitWith32khzTest	Checks for an 32bitdataRxMasterTxSlaveWith Rx Word Select Period as 16bit and Tx Word Select Period and 64bit Write operation with frequency 32khz.
I2sWriteOperationWith16bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP32bitWith48khzTest	Checks for an 16bitdataRxMasterTxSlaveWith Rx Word Select Period as 16bit and Tx Word Select Period and 32bit Write operation with frequency 48khz
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP16bitTxWSP48bitWith16khzTest	Checks for an 24bitdataRxMasterTxSlaveWith Rx Word Select Period as 16bit and Tx Word Select Period and 48bit Write operation with frequency 16khz
I2sWriteOperationWith24bitdataRxMasterTxSlaveWithRxWSP32bitTxWSP48bitWith192khzTest	Checks for an 24bitdataRxMasterTxSlaveWith Rx Word Select Period as 32bit and Tx Word Select Period and 48bit Write operation with frequency 192khz
I2sWriteOperationRandomTxMasterRxSlaveWithTxWSP32bitWith16khzTest	Checks for an RandomTxMasterRxSlave with Tx Word Select Period as 32bit Write operation with frequency 16khz.

I2sWriteOperationRandomTxMasterRxSlave WithTxWSP48bitWith96khzTest	Checks for an RandomTxMasterRxSlave with Tx Word Select Period as 48bit Write operation with frequency 96khz.
I2sWriteOperationRandomTxMasterRxSlave WithTxWSP64bitWith32khzTest	Checks for an RandomTxMasterRxSlave with Tx Word Select Period as 64bit Write operation with frequency 32khz.
I2sWriteOperationRandomRxMasterTxSlave With32khzTest	Checks for an RandomTxMasterRxSlave Write operation with frequency 32khz.

Chapter 10

Simulation Results and Waveforms

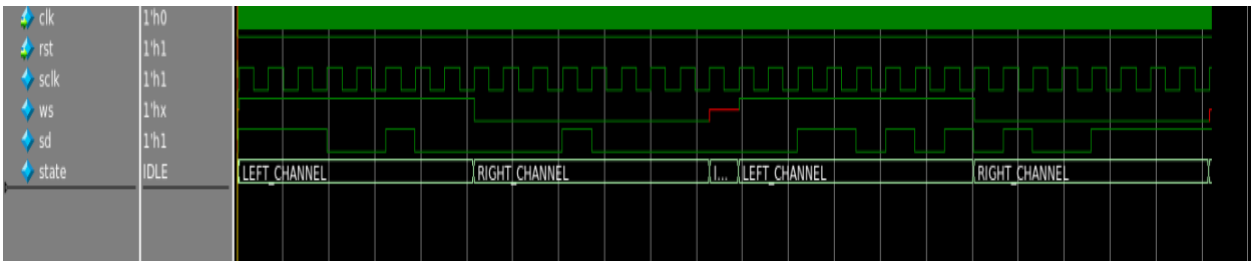

Fig 10.1 *I2sWriteOperationWith8bitdataTxMasterRxSlaveWith48khz*

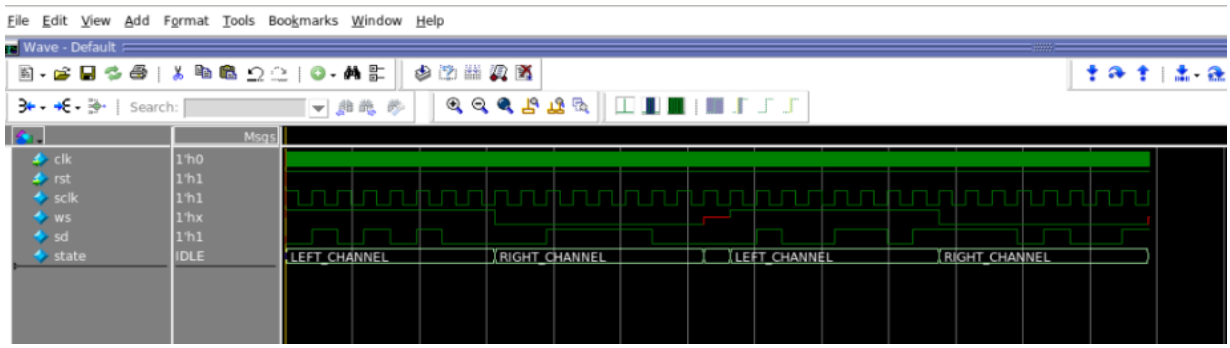

Fig 10.2 *I2sWriteOperationWith8bitdataRxMasterTxSlaveWith8khz*

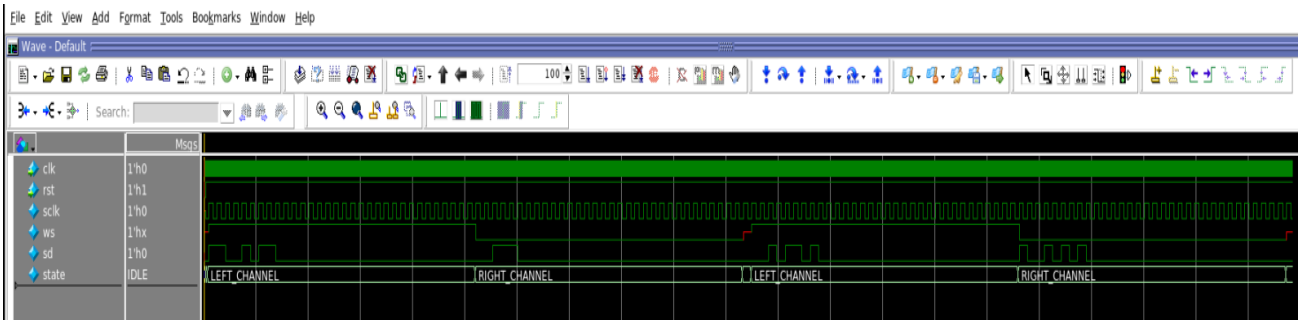

Fig 10.3 *I2sWriteOperationWith8bitdataRxMasterTxSlaveWithRxWSP64bitTxWSP16bitWith96khz*

Chapter 11

References

I2S Protocol Specification