Stream
Model Independent Transaction
User Guide

User Guide for Release 2022.01

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1. Overview

The Stream Model Independent Transaction package (StreamTransactionPkg.vhd) defines a transaction interface (a record for communication between the test sequencer and the verification component) and transaction initiation procedures that are suitable for Stream Interfaces.

All verification components (VCs) that use this interface, support a common set of transactions (or API). Hence, a test writer who understands the transactions for one verification component will also understand the transactions of another verification component that also uses the same package. This makes the job of a verification engineer easier.

The main difference between verification components that support this package is the configuration of verification component interface specific features - such as error injection for a UART or configuration of TID, TDest, TUser, and TLast for AxiStream.

Having a shared set of transactions improves test case reuse between different verification components – a case where reuse is rarely possible with other verification methodologies.

1. Transmitters and Receivers

A Transmitter is the agent that sends information on a stream interface, and a Receiver is the agent that receives information on a stream interface. Terminology may vary from interface to interface, however, the use of transmitter and receiver for any given streaming interface is unambiguous.

2. Stream Transaction Interface

The stream transaction interface connects the test sequencer to a verification component. As such, it is the primary channel for information exchange between the two. This is done with the record type StreamRecType.

2.1 StreamRecType

```
type StreamRecType is record
  -- Handshaking controls
  --   Used by RequestTransaction in the Transaction Procedures
  --   Used by WaitForTransaction in the Verification Component
  --   RequestTransaction and WaitForTransaction are in osvvm.TbUtilPkg
  Rdy              : RdyType ;
  Ack              : AckType ;
  -- Transaction Type
  Operation        : StreamOperationType ;
  -- Data and Transaction Parameter to and from verification component
  DataToModel      : std_logic_vector_max_c ;
  ParamToModel     : std_logic_vector_max_c ;
  DataFromModel    : std_logic_vector_max_c ;
  ParamFromModel   : std_logic_vector_max_c ;
  -- BurstFifo
  BurstFifo        : ScoreboardIdType ;
  -- Verification Component Options Parameters - used by SetModelOptions
  IntToModel       : integer_max ;
  IntFromModel     : integer_max ;
  BoolToModel      : boolean_max ;
  BoolFromModel    : boolean_max ;
  TimeToModel      : time_max ;
  TimeFromModel    : time_max ;
  -- Verification Component Options Type
  Options          : integer_max ;
end record StreamRecType ;
```

One of the challenges of using a single record, such as StreamRecType, as an interface is dealing with multiple drivers on each record element. OSVVM does this giving each element a resolved type, such as bit_max, std_logic_vector_max_c, integer_max, time_max, and boolean_max. These are defined in the OSVVM package ResolutionPkg. These types allow the record to support multiple drivers and use resolution functions based on function maximum (return largest value).

The type StreamOperationType is discussed in VC section of this document.

2.2 BurstFifo is in the Interface

The BurstFifo is inside StreamRecType. This means it is easily accessible to both the verification component and the Test Sequencer (TestCtrl). The type, ScoreboardIdType, is a reference (though not an access type) to the scoreboard singleton data structure in ScoreboardGenericPkg. Using ScoreboardIdType allows the structure to be used as either a FIFO (by SendBurst or GetBurst) or a Scoreboard (by CheckBurst). The FIFO is std_logic_vector based and uses the ScoreboardPkg_slv instance from OsvvmLibraries/osvvm.

2.3 Usage of the Transaction Interface (StreamRecType)

The data and parameter fields of the StreamRecType are unconstrained. Unconstrained objects may be used on component/entity interfaces. The record fields need to be sized by the record signal that is mapped as the actual in the test harness of the testbench. Figure 1 shows the declaration StreamTxRec (which connects the AxiStreamTransmitter to TestCtrl) and StreamRxRec (which connects the AxiStreamReceiver to TestCtrl).

```
constant AXI_PARAM_WIDTH : integer :=
      TID'length + TDest'length + TUser'length + 1;

signal StreamTxRec, StreamRxRec : StreamRecType(
    DataToModel   (AXI_DATA_WIDTH-1  downto 0),
    ParamToModel  (AXI_PARAM_WIDTH-1 downto 0),
    DataFromModel (AXI_DATA_WIDTH-1  downto 0),
    ParamFromModel(AXI_PARAM_WIDTH-1 downto 0)
  ) ;
```

Figure 1. StreamRecType

3. Running the Stream Demo

The best way to learn is by trying things out as you go. In this step you will download OSVVM, build the libraries, and then run the demo. Code from the demo is shown in examples in the respective sections.

OSVVM is available on GitHub at https://github.com/OSVVM as a git repository or at https://osvvm.org/downloads as a ZIP file. Retrieve OSVVM from GitHub using git as shown in Figure 2. Note that the "—recursive" option is required since the OSVVM repositories are submodules of OsvvmLibraries. Submodules greatly simplify development and deployment of the libraries.

```
git clone --recursive https://github.com/OSVVM/OsvvmLibraries.git
```

Figure 2. Retrieving OSVVM from GitHub

Prior to starting the OSVVM scripting environment, create a directory named sim in which to run your simulations. Start your simulator and go to the sim directory. Once there, use the steps in Figure 3 to build the OSVVM Libraries (utility and verification component). These directions are supported in Mentor QuestaSim/ModelSim or Aldec RivieraPRO. Aldec's ActiveHDL, Synopsys' VCS, and Cadence's Xcelium are also supported but require a few extra steps. For these steps and additional details of the OSVVM scripting environment see Script_user_guide.pdf (in OsvvmLibraries/Documentation).

```
cd sim
source ../OsvvmLibraries/Scripts/StartUp.tcl
build  ../OsvvmLibraries
build  ../OsvvmLibraries/AXI4/AxiStream/RunDemoTests.pro
```

Figure 3. Building OSVVM and running the Demo

The intent of the OSVVM scripting is to make compiling and running your simulations independent of the simulator you are using.

GHDL can be run using tclsh. In windows, using MSYS2/MinGW64 start tclsh using "winpty tclsh".

4. Types of Transactions

A transaction may be either a directive or an interface transaction.

Directive transactions interact with the verification component without generating any transactions or interface waveforms. Examples of these are WaitForClock and GetAlertLogID.

An interface transaction results in interface signaling to the DUT. An interface transaction may be either blocking (such as Send or Get) or non-blocking (such as SendAsync and TryGet).

A blocking transaction is an interface transaction that does not does not return (complete) until the interface operation requested by the transaction has completed.

An asynchronous transaction is nonblocking interface transaction that returns before the transaction has completed - typically immediately and before the transaction has started. An asynchronous transaction has "Async" as part of its name.

A Try transaction is nonblocking interface transaction that checks to see if transaction information is available, such as read data, and if it is returns it. A Try transaction has "Try" as part of its name.

5. Basic Blocking Transactions

Basic blocking transactions dispatch a transaction to the verification component and wait for the verification component to finish the operation on the DUT interface before returning. These represent the minimal set of transactions that a verification component must implement.

In the discussion about blocking transactions, the examples for a particular set of transactions use example code from this test.

5.1 Transmitter: Send

Blocking Send Transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
------------------------------------------------------------
procedure Send (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure Send (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

5.2 Receiver: Get

Get is a blocking get transaction. Param, when present, is an extra parameter used by the verification component.

```
------------------------------------------------------------
procedure Get (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  variable  Param           : out   std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure Get (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  Data            : out   std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

5.3 Example

The code below transmits of 32 words.

```
AxiTransmitterProc : process
begin
  . . .
  log("Transmit 32 words") ;
  for I in 1 to 32 loop
    Send( StreamTxRec, X"0000_0000" + I ) ;
  end loop ;
```

The code below receives and checks the 32 words.

```
AxiReceiverProc: process
  variable RxData : std_logic_vector(31 downto 0);
begin
  . . .
  for I in 1 to 32 loop
    Get(StreamRxRec, RxData) ;
    AffirmIfEqual(RxData, X"0000_0000" + I, "RxData") ;
  end loop ;
```

6. Checking for Basic Blocking Transactions

6.1 Receiver: Check

Check is a blocking check transaction. Data is the expected value to be received. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure Check (
------------------------------------------------------------
```

```
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  Param           : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure Check (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

6.2 Example

The code below transmits of 32 words.

```
  AxiTransmitterProc : process
  begin
    . . .
    log("Transmit 32 words") ;
    for I in 1 to 32 loop
      Send( StreamTxRec, X"0000_1000" + I ) ;
    end loop ;
```

The code below checks the 32 words.

```
  AxiReceiverProc: process
  begin
    . . .
    for I in 1 to 32 loop
      Check(StreamRxRec, X"0000_1000" + I ) ;
    end loop ;
```

7. Basic Blocking Burst Transactions

7.1 Transmitter: SendBurst

SendBurst is a blocking send burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
  ------------------------------------------------------------
  procedure SendBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  NumFifoWords    : In    integer ;
    constant  Param           : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE   ) ;

  ------------------------------------------------------------
  procedure SendBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  NumFifoWords    : In    integer ;
```

```
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

7.2 Receiver: GetBurst

GetBurst is a blocking get burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for checking error injection.

```
  ------------------------------------------------------------
  procedure GetBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    variable  NumFifoWords    : inout integer ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure GetBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    variable  NumFifoWords    : inout integer ;
    variable  Param           : out   std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

7.3 Example

The code below transmits 32 words as a burst.

```
  AxiTransmitterProc : process
  begin
    . . .
    log("Send 32 word burst") ;
    for I in 1 to 32 loop
      Push( StreamTxRec.BurstFifo, X"0000_2000" + I  ) ;
    end loop ;
    SendBurst(StreamTxRec, 32) ;
```

The code below receives and checks the 32 word burst.

```
  AxiReceiverProc: process
    variable RxData : std_logic_vector(31 downto 0);
  begin
    . . .
    GetBurst(StreamRxRec, NumBytes) ;
    AffirmIfEqual(NumBytes, 32, "Receiver: 32 Received") ;
    for I in 1 to 32 loop
      RxData := Pop( StreamRxRec.BurstFifo ) ;
      AffirmIfEqual(RxData, X"0000_2000" + I , "RxData") ;
    end loop ;
```

8. Checking for Blocking Burst Transactions

8.1 Receiver: CheckBurst

CheckBurst is a blocking check burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for checking error injection.

```
------------------------------------------------------------
procedure CheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumFifoWords    : In    integer ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure CheckBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumFifoWords    : In    integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

8.2 Example

The code below transmits 32 words as a burst.

```
AxiTransmitterProc : process
begin
  . . .
  log("Send 32 word burst") ;
  for I in 1 to 32 loop
    Push( StreamTxRec.BurstFifo, X"0000_3000" + I ) ;
  end loop ;
  SendBurst(StreamTxRec, 32) ;
```

The code below checks the 32 word burst.

```
AxiReceiverProc: process
  variable RxData : std_logic_vector(31 downto 0);
begin
  . . .
  for I in 1 to 32 loop
    Push( StreamRxRec.BurstFifo, X"0000_3000" + I  ) ;
  end loop ;
  CheckBurst(StreamRxRec, 32) ;
```

9. Patterns for Blocking Burst Transactions

9.1 Burst with a Vector of Words

9.1.1 Transmitter: SendBurstVector

```
------------------------------------------------------------
```

```
procedure SendBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant VectorOfWords  : In    slv_vector ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure SendBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant VectorOfWords  : In    slv_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.1.2 Receiver: CheckBurstVector

```
------------------------------------------------------------
procedure CheckBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant VectorOfWords  : In    slv_vector ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure CheckBurstVector (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant VectorOfWords  : In    slv_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.1.3 Example

The code below transmits 13 words as a burst.

```
AxiTransmitterProc : process
begin
  . . .
  log("SendBurstVector 13 word burst") ;
  SendBurstVector(StreamTxRec,
      (X"0000_4001", X"0000_4003", X"0000_4005", X"0000_4007", X"0000_4009",
       X"0000_4011", X"0000_4013", X"0000_4015", X"0000_4017", X"0000_4019",
       X"0000_4021", X"0000_4023", X"0000_4025") ) ;
```

The code below checks the 13 words.

```
AxiReceiverProc: process
  variable RxData : std_logic_vector(31 downto 0);
begin
  . . .
  CheckBurstVector(StreamRxRec,
```

```
    (X"0000_4001", X"0000_4003", X"0000_4005", X"0000_4007", X"0000_4009",
     X"0000_4011", X"0000_4013", X"0000_4015", X"0000_4017", X"0000_4019",
     X"0000_4021", X"0000_4023", X"0000_4025") ) ;
```

9.2 Burst with an Incrementing Pattern

9.2.1 Transmitter: SendBurstIncrement

```
------------------------------------------------------------
procedure SendBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure SendBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.2.2 Receiver: CheckBurstIncrement

```
------------------------------------------------------------
procedure CheckBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure CheckBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.2.3 Example

The code below transmits of 16 words.

```
AxiTransmitterProc : process
begin
  . . .
  log("SendBurstIncrement 16 word burst") ;
  SendBurstIncrement(StreamTxRec, X"0000_5000", 16) ;
```

The code below checks the 16 words.

```
AxiReceiverProc: process
  variable RxData : std_logic_vector(31 downto 0);
begin
  . . .
  CheckBurstIncrement(StreamRxRec, X"0000_5000", 16) ;
```

9.3 Burst with a Random Pattern

9.3.1 Transmitter: SendBurstRandom

```
------------------------------------------------------------
procedure SendBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure SendBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.3.2 Receiver: CheckBurstRandom

```
------------------------------------------------------------
procedure CheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

```
------------------------------------------------------------
procedure CheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.3.3 Example

The code below transmits of 16 words.

```
AxiTransmitterProc : process
begin
  . . .
  log("SendBurstRandom 24 word burst") ;
  SendBurstRandom(StreamTxRec, X"0000_6000", 24) ;
```

The code below checks the 16 words.

```
AxiReceiverProc: process
  variable RxData : std_logic_vector(31 downto 0);
begin
  . . .
  CheckBurstRandom(StreamRxRec, X"0000_6000", 24) ;
```

9.4 Burst with an Intelligent Coverage Random Pattern

9.4.1 Transmitter: SendBurstRandom

```
------------------------------------------------------------
procedure SendBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant CoverID        : In    CoverageIDType ;
  constant NumFifoWords   : In    integer ;
  constant FifoWidth      : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure SendBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant CoverID        : In    CoverageIDType ;
  constant NumFifoWords   : In    integer ;
  constant FifoWidth      : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.4.2 Receiver: CheckBurstRandom

```
------------------------------------------------------------
procedure CheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant CoverID        : In    CoverageIDType ;
  constant NumFifoWords   : In    integer ;
  constant FifoWidth      : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure CheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant CoverID        : In    CoverageIDType ;
  constant NumFifoWords   : In    integer ;
  constant FifoWidth      : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

9.4.3 Example

The code below transmits of 16 words.

```
AxiTransmitterProc : process
  variable CoverID : CoverageIdType ;
begin
  . . .
  log("SendBurstRandom 42 word burst") ;
  SendBurstRandom(StreamTxRec, CoverID, 42, 32) ;
```

The code below checks the 16 words.

```
AxiReceiverProc: process
  variable CoverID : CoverageIdType ;
begin
  . . .
  CheckBurstRandom (StreamRxRec, CoverID, 42, 32) ;
```

10. FIFO Fill and Check Patterns and Composite Bursts

FIFO fill patterns can be used in conjunction with SendBurst and CheckBurst to build burst transactions that are a composite of Vector, Increment, and Random patterns.   Likewise FIFO check patterns can be used with GetBurst in a similar fashion.

FIFO fill and check patterns are implemented in FifoFillPkg_slv.vhd (in the osvvm_common library).

10.1 FIFO Fill Patterns

10.1.1 Fill with a Vector of Words

```
------------------------------------------------------------
procedure PushBurstVector (
-- Push each value in the VectorOfWords parameter into the FIFO.
-- FifoWidth must match the std_logic_vector parameter.
------------------------------------------------------------
  constant Fifo          : In    ScoreboardIdType ;
  constant VectorOfWords : In    slv_vector
) ;
------------------------------------------------------------
procedure PushBurstVector (
-- Push each value in the VectorOfWords parameter into the FIFO.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo           : in    ScoreboardIdType ;
  constant VectorOfWords  : in    integer_vector ;
  constant FifoWidth      : in    integer
) ;
```

10.1.2 Fill with an Incrementing Pattern

```
------------------------------------------------------------
procedure PushBurstIncrement (
-- Push Count number of values into FIFO.  The first value
-- pushed will be FirstWord and following values are one greater
-- than the previous one.
-- FifoWidth must match the std_logic_vector parameter.
------------------------------------------------------------
  constant Fifo           : In    ScoreboardIdType ;
  constant FirstWord      : In    std_logic_vector ;
  constant Count          : In    integer
) ;
------------------------------------------------------------
procedure PushBurstIncrement (
-- Push Count number of values into FIFO.  The first value
-- pushed will be FirstWord and following values are one greater
-- than the previous one.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant FirstWord    : in    integer ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.1.3 Fill with a Random Pattern

```
------------------------------------------------------------
procedure PushBurstRandom (
-- Push Count number of values into FIFO.  The first value
```

```
-- pushed will be FirstWord and following values are randomly generated
-- using the first value as the randomization seed.
-- FifoWidth must match the std_logic_vector parameter.
------------------------------------------------------------
  constant Fifo           : In    ScoreboardIdType ;
  constant FirstWord      : In    std_logic_vector ;
  constant Count          : In    integer
) ;
------------------------------------------------------------
procedure PushBurstRandom (
-- Push Count number of values into FIFO.  The first value
-- pushed will be FirstWord and following values are randomly generated
-- using the first value as the randomization seed.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant FirstWord    : in    integer ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.1.4 Fill with an Intelligent Coverage Random Pattern

```
------------------------------------------------------------
-- Experimental and Provisional
procedure PushBurstRandom (
-- Push Count number of values into FIFO.  Values are
-- randomly generated using the coverage model.
-- Only FifoWidth bits of each value will be pushed.
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant CoverID      : in    CoverageIdType ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.1.5 Example: Combining Patterns

The code below transmits of 42 words.

```
    log("Combining Patterns:  Vector, Increment, Random, Intelligent Coverage") ;
    PushBurstVector(StreamTxRec.BurstFifo,
        (X"0000_A001", X"0000_A003", X"0000_A005", X"0000_A007", X"0000_A009",
         X"0000_A011", X"0000_A013", X"0000_A015", X"0000_A017", X"0000_A019") ) ;
    PushBurstIncrement(StreamTxRec.BurstFifo, X"0000_A100", 10) ;
    PushBurstRandom   (StreamTxRec.BurstFifo, X"0000_A200", 6) ;
    CoverID := NewID("Cov1a") ;
    InitSeed(CoverID, 5) ; -- Get a common seed in both processes
    AddBins(CoverID, 1,
        GenBin(16#A000#, 16#A007#) & GenBin(16#A010#, 16#A017#) &
        GenBin(16#A020#, 16#A027#) & GenBin(16#A030#, 16#A037#)) ;
    PushBurstRandom(StreamTxRec.BurstFifo, CoverID, 16, 32) ;
```

```
    SendBurst(StreamTxRec, 42) ;
```

The code below checks the 42 words.

```
    PushBurstVector(StreamRxRec.BurstFifo,
        (X"0000_A001", X"0000_A003", X"0000_A005", X"0000_A007", X"0000_A009",
         X"0000_A011", X"0000_A013", X"0000_A015", X"0000_A017", X"0000_A019") ) ;
    PushBurstIncrement(StreamRxRec.BurstFifo, X"0000_A100", 10) ;
    PushBurstRandom(StreamRxRec.BurstFifo, X"0000_A200", 6) ;
    CoverID := NewID("Cov2a") ;
    InitSeed(CoverID, 5) ; -- Get a common seed in both processes
    AddBins(CoverID, 1,
        GenBin(16#A000#, 16#A007#) & GenBin(16#A010#, 16#A017#) &
        GenBin(16#A020#, 16#A027#) & GenBin(16#A030#, 16#A037#)) ;
    PushBurstRandom(StreamRxRec.BurstFifo, CoverID, 16, 32) ;
    CheckBurst(StreamRxRec, 42) ;
```

10.2 FIFO Check Patterns

10.2.1 Check with a Vector of Words

```
    ------------------------------------------------------------
  procedure CheckBurstVector (
  -- Check values from the FIFO against the values
  -- in the VectorOfWords parameter.
  -- Width of VectorOfWords(i) shall match the width of the Fifo
  ------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    constant VectorOfWords  : in    slv_vector
  ) ;

  ------------------------------------------------------------
  procedure CheckBurstVector (
  -- Check values from the FIFO against the values
  -- in the VectorOfWords parameter.
  -- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
  ------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    constant VectorOfWords  : in    integer_vector ;
    constant FifoWidth      : in    integer
  ) ;
```

10.2.2 Check with an Incrementing Pattern

```
  ------------------------------------------------------------
  procedure CheckBurstIncrement (
  -- Check values from the FIFO against the incrementing values
  -- that start with the value of the FirstWord.
  -- Width of FirstWord shall match the width of the Fifo
  ------------------------------------------------------------
    constant Fifo         : in    ScoreboardIdType ;
    constant FirstWord    : in    std_logic_vector ;
    constant Count        : in    integer
```

```
) ;

------------------------------------------------------------
procedure CheckBurstIncrement (
-- Check values from the FIFO against the incrementing values
-- that start with the value of the FirstWord.
-- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant FirstWord    : in    integer ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.2.3 Check with a Random Pattern

```
------------------------------------------------------------
procedure CheckBurstRandom (
-- Check values from the FIFO against the random values
-- that are generated using the value of the FirstWord and
-- NumFifoWords as the randomization seeds.
-- Width of FirstWord shall match the width of the Fifo
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant FirstWord    : in    std_logic_vector ;
  constant Count        : in    integer
) ;

------------------------------------------------------------
procedure CheckBurstRandom (
-- Check values from the FIFO against the random values
-- that are generated using the value of the FirstWord and
-- Count as the randomization seeds.
-- Each value of VectorOfWords shall be converted to FifoWidth bits wide.
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant FirstWord    : in    integer ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.2.4 Check with an Intelligent Coverage Random Pattern

```
------------------------------------------------------------
-- Experimental and Provisional
procedure CheckBurstRandom (
------------------------------------------------------------
  constant Fifo         : in    ScoreboardIdType ;
  constant CoverID      : in    CoverageIdType ;
  constant Count        : in    integer ;
  constant FifoWidth    : in    integer := 8
) ;
```

10.2.5 Example: Combining Patterns

The code below transmits of 42 words.

```
log("Combining Patterns:  Vector, Increment, Random, Intelligent Coverage") ;
PushBurstVector(StreamTxRec.BurstFifo,
    (X"0000_A001", X"0000_A003", X"0000_A005", X"0000_A007", X"0000_A009",
     X"0000_A011", X"0000_A013", X"0000_A015", X"0000_A017", X"0000_A019") ) ;
PushBurstIncrement(StreamTxRec.BurstFifo, X"0000_A100", 10) ;
PushBurstRandom   (StreamTxRec.BurstFifo, X"0000_A200", 6) ;
CoverID := NewID("Cov1a") ;
InitSeed(CoverID, 5) ; -- Get a common seed in both processes
AddBins(CoverID, 1,
    GenBin(16#A000#, 16#A007#) & GenBin(16#A010#, 16#A017#) &
    GenBin(16#A020#, 16#A027#) & GenBin(16#A030#, 16#A037#)) ;
PushBurstRandom(StreamTxRec.BurstFifo, CoverID, 16, 32) ;
SendBurst(StreamTxRec, 42) ;
```

The code below checks the 42 words.

```
PushBurstVector(StreamRxRec.BurstFifo,
    (X"0000_A001", X"0000_A003", X"0000_A005", X"0000_A007", X"0000_A009",
     X"0000_A011", X"0000_A013", X"0000_A015", X"0000_A017", X"0000_A019") ) ;
PushBurstIncrement(StreamRxRec.BurstFifo, X"0000_A100", 10) ;
PushBurstRandom(StreamRxRec.BurstFifo, X"0000_A200", 6) ;
CoverID := NewID("Cov2a") ;
InitSeed(CoverID, 5) ; -- Get a common seed in both processes
AddBins(CoverID, 1,
    GenBin(16#A000#, 16#A007#) & GenBin(16#A010#, 16#A017#) &
    GenBin(16#A020#, 16#A027#) & GenBin(16#A030#, 16#A037#)) ;
PushBurstRandom(StreamRxRec.BurstFifo, CoverID, 16, 32) ;
CheckBurst(StreamRxRec, 42) ;
```

10.3 FIFO Pop Burst

10.3.1 PopBurstVector

PopBurstVector returns the contents of the FIFO as a vector.   The vector can either be an array of std_logic_vector (slv_vector) or integer_vector.

```
------------------------------------------------------------
procedure PopBurstVector (
-- Pop values from the FIFO into the VectorOfWords parameter.
-- Width of VectorOfWords(i) shall match the width of the Fifo
------------------------------------------------------------
  constant Fifo           : in    ScoreboardIdType ;
  variable VectorOfWords  : out   slv_vector
) ;

------------------------------------------------------------
procedure PopBurstVector (
-- Pop values from the FIFO into the VectorOfWords parameter.
-- Each value popped will be FifoWidth bits wide.
```

```
----------------------------------------------------------------
    constant Fifo           : in    ScoreboardIdType ;
    variable VectorOfWords  : out   integer_vector
  ) ;
```

10.3.2 Example: PopBurstVector

The code below transmits of 42 words.

```
    log("SendBurstVector 5 word burst") ;
    SendBurstVector(StreamTxRec,
        (X"0000_C001", X"0000_C003", X"0000_C005", X"0000_C007", X"0000_C009") ) ;

    log("SendBurstVector 5 word burst") ;
    PushBurstVector(StreamTxRec.BurstFifo,
        (16#D001#, 16#D003#, 16#D005#, 16#D007#, 16#D009#), 32 ) ;
    SendBurst(StreamTxRec, 5) ;
```

The code below checks the 42 words.

```
    GetBurst(StreamRxRec, NumBytes) ;
    PopBurstVector(StreamRxRec.BurstFifo, slvBurstVector) ;
    AffirmIf(slvBurstVector =
        (X"0000_C001", X"0000_C003", X"0000_C005", X"0000_C007", X"0000_C009"),
        "slvBurstVector = C001, C003, C005, C007, C009") ;

    GetBurst(StreamRxRec, NumBytes) ;
    PopBurstVector(StreamRxRec.BurstFifo, intBurstVector) ;
    AffirmIf(intBurstVector =
        (16#D001#, 16#D003#, 16#D005#, 16#D007#, 16#D009#),
        "slvBurstVector = D001, D003, D005, D007, D009") ;
```

11. Basic Nonblocking Transactions

Nonblocking transactions are either asynchronous transactions or try transactions. Asynchronous transactions return before the transaction has been completed. Try transactions check to see if data is available and return it if it is. If data is available, the Available parameter is set to TRUE, otherwise, it is FALSE.

Nonblocking transactions are useful creating different time relationships needed for split transaction interfaces, such as AXI4 Full where Write Address, Write Data, Write Response, Read Address, and Read Data are all independent of each other. Nonblocking transactions are also needed when a test sequencer dispatches transactions from a single process.

If you are just getting started with Stream Model Independent Transactions, this section largely parallels the blocking transactions, so it is recommended to skip ahead to Directive Transactions.

11.1 Transmitter: SendAsync

SendAsync is an asynchronous / non-blocking send transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
----------------------------------------------------------------
procedure SendAsync (
----------------------------------------------------------------
```

```
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  Param           : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure SendAsync (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

11.2 Receiver: TryGet

TryGet is a non-blocking try get transaction. If Data is available, get it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
  ------------------------------------------------------------
  procedure TryGet (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    variable  Data            : out   std_logic_vector ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure TryGet (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    variable  Data            : out   std_logic_vector ;
    variable  Param           : out   std_logic_vector ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

12. Checking for Basic Nonblocking Transactions

12.1 Receiver: TryCheck

TryCheck is a non-blocking try check transaction If Data is available, check it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
  ------------------------------------------------------------
  procedure TryCheck (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  Data            : in    std_logic_vector ;
    constant  Param           : in    std_logic_vector ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
```

```
) ;

------------------------------------------------------------
procedure TryCheck (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  Data            : in    std_logic_vector ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

13. Nonblocking Burst Transactions

13.1 Transmitter: SendBurstAsync

SendBurstAsync is an asynchronous / non-blocking send burst transaction. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for error injection.

```
------------------------------------------------------------
procedure SendBurstAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumFifoWords    : In    integer ;
  constant  Param           : in    std_logic_vector ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure SendBurstAsync (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  NumFifoWords    : In    integer ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;
```

13.2 Receiver: TryGetBurst

TryGetBurst is a non-blocking try get burst transaction. If Data is available, get it and return available TRUE, otherwise Return Available FALSE. Param, when present, is an extra parameter used by the verification component. The UART verification component uses Param for received error status.

```
------------------------------------------------------------
procedure TryGetBurst (
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  NumFifoWords    : inout integer ;
  variable  Available       : out   boolean ;
  constant  StatusMsgOn     : in    boolean := FALSE
) ;

------------------------------------------------------------
procedure TryGetBurst (
```

```
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    variable  NumFifoWords    : inout integer ;
    variable  Param           : out   std_logic_vector ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

14. Nonblocking Check Burst Transactions

14.1 Receiver: TryCheckBurst

TryCheckBurst is a non-blocking try check burst transaction. Param, when present, is an extra parameter used by the verification component. If BURST Data is available, check it and return available TRUE, otherwise Return Available FALSE. The UART verification component uses Param for checking error injection.

```
  ------------------------------------------------------------
  procedure TryCheckBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  NumFifoWords    : In    integer ;
    constant  Param           : in    std_logic_vector ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;

  ------------------------------------------------------------
  procedure TryCheckBurst (
  ------------------------------------------------------------
    signal    TransactionRec  : inout StreamRecType ;
    constant  NumFifoWords    : In    integer ;
    variable  Available       : out   boolean ;
    constant  StatusMsgOn     : in    boolean := FALSE
  ) ;
```

15. Patterns for Nonblocking Burst Transactions

15.1 Burst with a Vector of Words

15.1.1 Transmitter: SendBurstVectorAsync

```
  ------------------------------------------------------------
  procedure SendBurstVectorAsync (
  ------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant VectorOfWords  : In    slv_vector ;
    constant Param          : In    std_logic_vector ;
    constant StatusMsgOn    : In    boolean := false
  ) ;

  ------------------------------------------------------------
  procedure SendBurstVectorAsync (
```

```
--------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant VectorOfWords  : In    slv_vector ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

15.1.2 Receiver: TryCheckBurstVector

```
--------------------------------------------------------------
  procedure TryCheckBurstVector (
--------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant VectorOfWords  : In    slv_vector ;
    constant Param          : In    std_logic_vector ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
  )  ;

--------------------------------------------------------------
  procedure TryCheckBurstVector (
--------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant VectorOfWords  : In    slv_vector ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

15.2 Burst with an Incrementing Pattern

15.2.1 Transmitter: SendBurstIncrementAsync

```
--------------------------------------------------------------
  procedure SendBurstIncrementAsync (
--------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant FirstWord      : In    std_logic_vector ;
    constant NumFifoWords   : In    integer ;
    constant Param          : In    std_logic_vector ;
    constant StatusMsgOn    : In    boolean := false
  ) ;

--------------------------------------------------------------
  procedure SendBurstIncrementAsync (
--------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant FirstWord      : In    std_logic_vector ;
    constant NumFifoWords   : In    integer ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

15.2.2 Receiver: TryCheckBurstIncrement

```
------------------------------------------------------------
procedure TryCheckBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  variable Available      : Out   boolean ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure TryCheckBurstIncrement (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  variable Available      : Out   boolean ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

15.3 Burst with a Random Pattern

15.3.1 Transmitter: SendBurstRandomAsync

```
------------------------------------------------------------
procedure SendBurstRandomAsync (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant Param          : In    std_logic_vector ;
  constant StatusMsgOn    : In    boolean := false
) ;

------------------------------------------------------------
procedure SendBurstRandomAsync (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

15.3.2 Receiver: TryCheckBurstRandom

```
------------------------------------------------------------
procedure TryCheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant FirstWord      : In    std_logic_vector ;
  constant NumFifoWords   : In    integer ;
```

```
    constant Param          : In    std_logic_vector ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
  ) ;

  ------------------------------------------------------------
  procedure TryCheckBurstRandom (
  ------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant FirstWord      : In    std_logic_vector ;
    constant NumFifoWords   : In    integer ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

15.4 Burst with an Intelligent Coverage Random Pattern

15.4.1 Transmitter: SendBurstRandomAsync

```
  ------------------------------------------------------------
  procedure SendBurstRandomAsync (
  ------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant CoverID        : In    CoverageIDType ;
    constant NumFifoWords   : In    integer ;
    constant FifoWidth      : In    integer ;
    constant Param          : In    std_logic_vector ;
    constant StatusMsgOn    : In    boolean := false
  ) ;

  ------------------------------------------------------------
  procedure SendBurstRandomAsync (
  ------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant CoverID        : In    CoverageIDType ;
    constant NumFifoWords   : In    integer ;
    constant FifoWidth      : In    integer ;
    constant StatusMsgOn    : In    boolean := false
  ) ;
```

15.4.2 Receiver: TryCheckBurstRandom

```
  ------------------------------------------------------------
  procedure TryCheckBurstRandom (
  ------------------------------------------------------------
    signal   TransactionRec : InOut StreamRecType ;
    constant CoverID        : In    CoverageIDType ;
    constant NumFifoWords   : In    integer ;
    constant FifoWidth      : In    integer ;
    constant Param          : In    std_logic_vector ;
    variable Available      : Out   boolean ;
    constant StatusMsgOn    : In    boolean := false
```

```
) ;

------------------------------------------------------------
procedure TryCheckBurstRandom (
------------------------------------------------------------
  signal   TransactionRec : InOut StreamRecType ;
  constant CoverID        : In    CoverageIDType ;
  constant NumFifoWords   : In    integer ;
  constant FifoWidth      : In    integer ;
  variable Available      : Out   boolean ;
  constant StatusMsgOn    : In    boolean := false
) ;
```

16. Directive Transactions

Directive transactions interact with the verification component without generating any transactions or interface waveforms. These transactions are supported by all verification components.

```
------------------------------------------------------------
procedure WaitForTransaction (
--  Wait until pending (transmit) or next (receive) transaction(s) complete
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType
) ;

------------------------------------------------------------
procedure WaitForClock (
-- Wait for NumberOfClocks number of clocks
-- relative to the verification component clock
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  constant  WaitCycles      : in    natural := 1
) ;

------------------------------------------------------------
procedure GetTransactionCount (
-- Get the number of transactions handled by the model.
------------------------------------------------------------
  signal    TransactionRec   : inout StreamRecType ;
  variable  TransactionCount : out   integer
) ;

------------------------------------------------------------
procedure GetAlertLogID (
-- Get the AlertLogID from the verification component.
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  AlertLogID      : out   AlertLogIDType
) ;

------------------------------------------------------------
procedure GetErrorCount (
-- Error reporting for testbenches that do not use OSVVM AlertLogPkg
```

```
-- Returns error count.  If an error count /= 0, also print errors
------------------------------------------------------------
  signal    TransactionRec  : inout StreamRecType ;
  variable  ErrorCount      : out   natural
) ;
```

17. Burst FIFOs and Burst Mode Controls

The burst FIFOs hold bursts of data that is to be sent to or received from the interface. The burst FIFO can be configured in the modes defined for StreamFifoBurstModeType. Currently these modes defined as a subtype of integer, shown below. The intention is to facilitate model specific extensions without the need to define separate transactions.

```
subtype StreamFifoBurstModeType is integer ;

-- Word mode indicates the burst FIFO contains interface words.
-- The size of the word may either be interface specific (such as
-- a UART which supports up to 8 bits) or be interface instance specific
-- (such as AxiStream which supports interfaces sizes of 1, 2, 4, 8,
-- 16, ... bytes)
constant STREAM_BURST_WORD_MODE       : StreamFifoBurstModeType := 0 ;

-- Word + Param mode indicates the burst FIFO contains interface
-- words plus a parameter.   The size of the parameter is also either
-- interface specific (such as the OSVVM UART, which uses 3 bits -
-- one bit for each of parity, stop, and break error injection) or
-- interface instance specific (such as AxiStream which uses the Param
-- field to hold TUser).  AxiStream TUser may be different size for
-- different applications.
constant STREAM_BURST_WORD_PARAM_MODE : StreamFifoBurstModeType  := 1 ;

-- Byte mode is experimental and may be removed in a future revision.
-- Byte mode indicates that the burst FIFO contains bytes.
-- The verification component assembles interface words from the bytes.
-- This allows transfers to be conceptualized in an interface independent
-- manner.
constant STREAM_BURST_BYTE_MODE       : StreamFifoBurstModeType := 2 ;
```

SetBurstMode and GetBurstMode are directive transactions that configure the burst mode into one of the modes defined in for StreamFfifoBurstModeType.

```
------------------------------------------------------------
procedure SetBurstMode (
------------------------------------------------------------
  signal   TransRec    : InOut StreamRecType ;
  constant OptVal      : In    StreamFifoBurstModeType
) ;

------------------------------------------------------------
procedure GetBurstMode (
------------------------------------------------------------
  signal   TransRec    : InOut StreamRecType ;
  variable OptVal      : Out   StreamFifoBurstModeType
```

```
) ;
```

18. Set and Get Model Options

Model operations are directive transactions that are used to configure the verification component. They can either be used directly or with a model specific wrapper around them - see the AxiStream User Guide for examples.

```
------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  constant OptVal          : In    boolean
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  constant OptVal          : In    integer
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  constant OptVal          : In    std_logic_vector
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  constant OptVal          : In    time
) ;

------------------------------------------------------------
procedure SetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  variable OptVal          : Out   boolean
) ;
```

```
------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  variable OptVal          : Out   integer
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  variable OptVal          : Out   std_logic_vector
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer ;
  variable OptVal          : Out   time
) ;

------------------------------------------------------------
procedure GetModelOptions (
------------------------------------------------------------
  signal   TransactionRec  : InOut StreamRecType ;
  constant Option          : In    integer
) ;
```

19.Verification Components and Stream Model Independent Transactions

19.1 StreamOperationType

StreamOperationType is an enumerated type that indicates to the verification component type of transaction that is being dispatched. Being an enumerated type, it allows the determination of the operation in the simulator's waveform window. Table 4 shows the correlation between StreamOperationType values and the transaction name.

AddressBusOperationType Value	Transmitter Transaction Name	Receiver Transaction Name
WAIT_FOR_CLOCK	WaitForClock	WaitForClock
WAIT_FOR_TRANSACTION	WaitForTransaction	WaitForTransaction
GET_TRANSACTION_COUNT	GetTransactionCount	GetTransactionCount
GET_ALERTLOG_ID	GetAlertLogID	GetAlertLogID
SET_BURST_MODE	SetBurstMode	SetBurstMode
GET_BURST_MODE	GetBurstMode	GetBurstMode

AddressBusOperationType Value	Transmitter Transaction Name	Receiver Transaction Name
SET_MODEL_OPTIONS	SetModelOptions	SetModelOptions
GET_MODEL_OPTIONS	GetModelOptions	GetModelOptions
SEND	Send	
SEND_ASYNC	SendAsync	
SEND_BURST	SendBurst	
SEND_BURST_ASYNC	SendBurstAsync	
GET		Get
TRY_GET		TryGet
GET_BURST		GetBurst
TRY_GET_BURST		TryGetBurst
CHECK		Check
TRY_CHECK		TryCheck
CHECK_BURST		CheckBurst
TRY_CHECK_BURST		TryCheckBurst

Figure 4. Correlation between StreamOperationType and the transaction name

These values are used in the transaction dispatcher of the verification component to determine which transaction was called and to appropriately handle the information in the record.

19.2 Verification Component Support Functions

Verification component support functions help decode the operation value (StreamOperationType) to determine properties about the operation. It is recommended that a verification component use these when they apply.

```
------------------------------------------------------------
function IsTry (
-- True when this transaction is an asynchronous or try transaction.
------------------------------------------------------------
  constant Operation     : in StreamOperationType
) return boolean ;

------------------------------------------------------------
function IsCheck (
-- True when this transaction is a check transaction.
------------------------------------------------------------
  constant Operation     : in StreamOperationType
) return boolean ;
```

19.3 Using the Burst FIFOs in the VC

19.3.1 Initializing Burst FIFOs

The burst FIFOs need to be initialized. A good place to do this is in the transaction dispatcher of the verification components. Figure 5 shows the declaration of a BurstFifo. The BurstFifo is accessed as a record element of TransRec.

```
TransactionDispatcher : process
 . . .
begin
  wait for 0 ns ;
  TransRec.BurstFifo <= NewID("RxBurstFifo", ModelID) ;
  wait for 0 ns ;
```

Figure 5. BurstFifo Initialization

19.3.2 Accessing Burst FIFOs

The Burst Fifos support basic FIFO operations. These are shown in Figure 6.

```
Push(TransRec.BurstFifo, Data) ;
Check(TransRec.BurstFifo, Data) ;
Data := Pop(TransRec.BurstFifo) ;
```

Figure 6. Making the BurstFifos visible in the test sequencer (TestCtrl)

19.3.3 Packing and Unpacking the FIFO

The burst FIFOs can be configured to be either byte width or word width (matching the interface size). The following procedures (from FifoFillPkg_slv.vhd) are used to transform byte width data in the burst FIFO to/from the verification component interface width.

```
------------------------------------------------------------
procedure PopWord (
-- Pop a word to the VC from a byte in the BurstFifo
-- Used to assemble bytes in the FIFO to form words in the VC
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  variable Valid             : out   boolean ;
  variable Data              : out   std_logic_vector ;
  variable BytesToSend       : inout integer ;
  constant ByteAddress       : in    natural := 0
) ;
------------------------------------------------------------
procedure PushWord (
-- Push a word from the VC to a byte in the BurstFifo
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  variable Data              : in    std_logic_vector ;
  constant DropUndriven      : in    boolean := FALSE ;
  constant ByteAddress       : in    natural := 0
) ;
------------------------------------------------------------
procedure CheckWord (
```

```
-- Compare a word in the VC to bytes in the BurstFifo
------------------------------------------------------------
  constant Fifo              : in    ScoreboardIDType ;
  variable Data              : in    std_logic_vector ;
  constant DropUndriven      : in    boolean := FALSE ;
  constant ByteAddress       : in    natural := 0
) ;
```

20. About the OSVVM Model Independent Transactions

OSVVM Model Independent Transactions were developed and are maintained by Jim Lewis of SynthWorks VHDL Training. These evolved from methodology and packages developed for SynthWorks' VHDL Testbenches and verification class. They are part of the Open Source VHDL Verification Methodology (OSVVM) model library (osvvm_common), which brings leading edge verification techniques to the VHDL community.

Please support OSVVM by purchasing your VHDL training from SynthWorks.

21. About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus years of design, teaching, and problem-solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.

22. References

[1] Jim Lewis, VHDL Testbenches and Verification, student manual for SynthWorks' class.