Accellera VHDL-TC Extensions-SC

Interfaces

Jim Lewis, SynthWorks

jim@synthworks.com

Version 0.1 Draft, 12 Jan 2006

Abstract

This paper covers the requirements for interfaces in VHDL, potential use models, and alternatives to implementing them.

Revision History

Version 0.1 12-Jan-2006, Jim Lewis. Initial work in progress draft.

1. Preface

This document is a work in progress. Some of the terminology in this area seems to be inconsistent and evolving, so I appologize in advance if I use a term in a way that is either different from what you have seen or just plain wrong.

2. Introduction

An interface is an abstract representation of the connectivity and communication between two or more objects. This abstract representation may be implemented with one or more language features. It is the intent of this paper to review what requirements could be, discuss potential use models, and discuss potential implementations. It should be noted that while an interface could be a specific language feature, it could also be a methodology that combines a set of features (existing, extended, and/or new).

Requirements from Accellera VHDL-TC enhancements spreadsheet:

> FT-17-1 More concise representation to specify complex interfaces (supporting for example transactors, complex buses) (perhaps by using record type with element modes)
>
> - should be considered with OO
> - should be considered with transaction
> - should be synthesizable
>
> FT-17-3 ensure that it can create a synthesizeable level of abstraction

Interfaces were discussed on the IEEE VHDL-200X working group reflector. The following list of potential requirements/features/properties/implementations were discussed as part of that effort:

1. Group related data items (SB1). Similar to a record?
2. Define behavior in a manner that can be reused within the design (SB2)
3. Support multiple levels of abstraction (signals vs. bus operations vs. groups of operations (init)) (SB3)
4. Support a “procedure like” call interface (SB3). Perhaps a procedure, but could be something else.
5. User of interface only needs to understand the call interface. Implementation could be serial or parallel transmission. (SB4)
6. Procedure interface does not block the caller (SB5)

7. Contractual relationship that defines the “procedures” required of an implementation. (SB6)
8. Support hierarchical connectivity between entities (SB7)
9. Interface should be extendable and/or parameterizable via generics (SB8)
10. Ability to specify delay at the interface (SB9)
11. Support limited access to data items such that the value can only be accessed by a method in the interface.
12. Interface should be configurable to select and/or remap any of the above during a given elaboration
13. An entity port list can have multiple interfaces (interfaces here being more like a composite type and not a separate IO list).
14. Support decomposition of the system (from system level to rtl level and/or ?gate-level?)
15. Support mixed models in system (system-level mixed with rtl-level of a given interface. ?gate-level?)
16. Support modeling of multiple independent operations
17. Avoid known quirks and issues with implementations in other languages

Before potential implementations can be discussed, use models must be considered. From the use models, important features should become more obvious.

3. Use Model

For use models, transaction models in testbenches and RTL hardware design will be considered.

3.1. Transaction Based Testbench

3.1.1. Basic Transactions

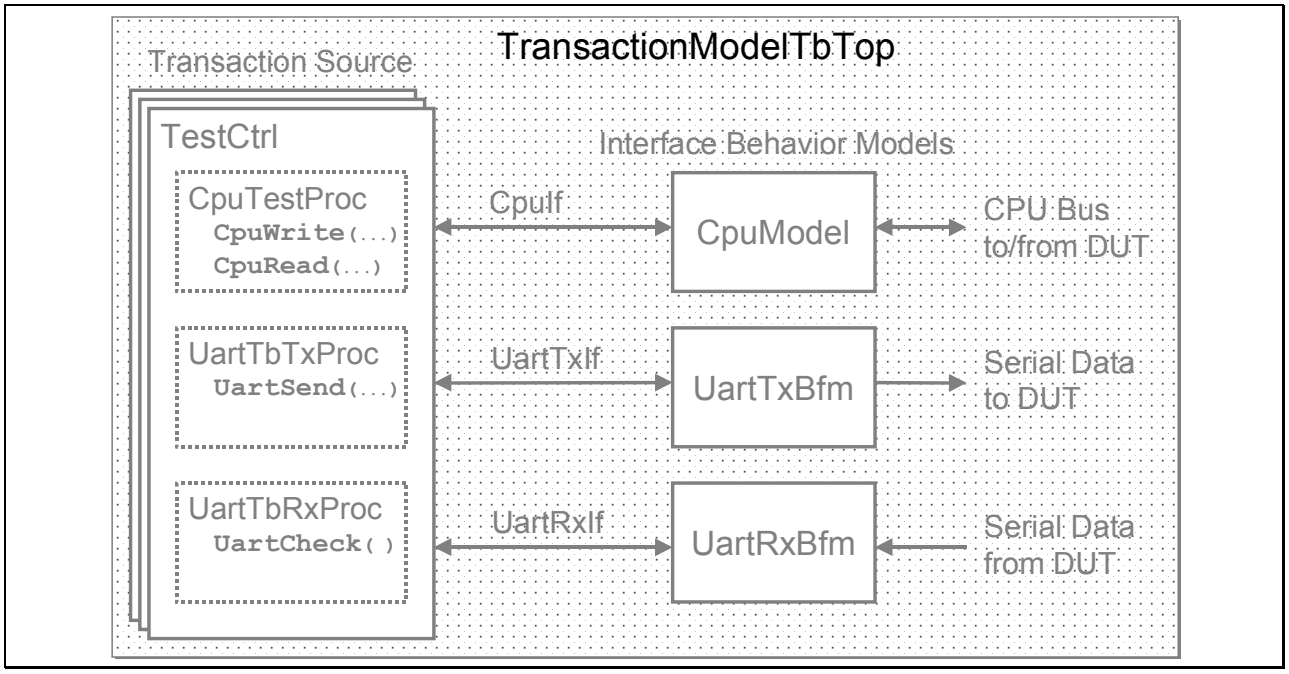
A transaction is an operation on a device interface. As such it could be a cpu read or a cpu write. A transaction is often either specified or created using a procedure. A test is created by calling transactions in a given sequence with specified values. For example,

```
CpuTestProc : process -- TestCtrl_UartRx1
  -- Declarations left out
begin
  wait until nReset = '1' ;

  CpuWrite(CpuRec, UART_DIVISOR_HIGH, X"0000");
  CpuWrite(CpuRec, UART_DIVISOR_LOW,  X"000A");
  CpuWrite(CpuRec, UART_CFG1, X"00" & "00" &
     PARITY_EVEN & STOP_BITS_1 & DATA_BITS_8);
  . . .
  CpuRead (CpuRec, UART_TX_INT_STAT,  DataO);

  SyncTo(SyncIn => UartTxRdy, SyncOut => CpuRdy);

  for I in 0 to 3 loop
    CpuPoll(CpuRec, UART_RX_INT_STAT, RX_DATA_VALID, '1') ;
    CpuReadCheck (CpuRec, UART_DATA,  X"4A" + I, true);
  end loop ;
  . . .
```

A simple approach to transactions is to implement the interface behavior in the procedure call. The limitation to this is that procedure is strictly sequential. While sequential code works well for some applications (like cpu read and cpu write) it does not work well for applications that require detailed statemachines (such as a UART receiver).

3.1.2. Transaction Based Bus Functional Models

In the approach shown below, each independent interface behavior is implemented in a separate interface behavior model (aka: transactor, server) and all of the transactions are grouped together in a single transaction source (aka: test control, client). The transaction source specifies sequences of transactions. The interface behavioral model creates the interface behavior specified in the transaction. The connection between the transaction source and interface behavior model can be represented as an interface (CpuIf, UartTxIf, UartRxIf).



The interface is used to abstractly carry the data items needed to support the transactions. Some interface elements will have a single driver and some will have multiple drivers. There is also a need for handshaking between the models. To support a wide range of modeling styles, the handshaking will need to support both blocking (such as read request and return with the read data) and non-blocking (accept the data from serial port now or it is gone). It may also need to support closed loop handshaking (ready-ready or ready-ack – perhaps only for blocking) and open-loop handshaking (event or edge based – perhaps only for non-blocking).

The interface behavior model receives transactions from the transaction source via the interface and creates appropriate signaling to the UUT. Implementing the interface behavior model as an entity facilitates modeling behavior that requires multiple processes (complex statemachines). In addition there are items that need to be modeled (such as assertions and protocol checkers) that are convenient to group in the same model.

Grouping all transactions into a single transaction source facilitates creating interactions between different models. Most interactions can be controlled by signals local to the transaction source rather than by signals that must go through hierarchical levels of the design. Note that the transaction source has one interface per interface behavior model. Hence an entity port list can contain multiple interfaces (interface here being more like a composite type and not a separate IO list).

A further benefit of this modeling methodology is that if a model in the system needs to change, such as changing the CpuModel from an X86 type model to a 68000, only the model and the UUT need to change. Many or all of the transactions specified in the transaction source remain unchanged.

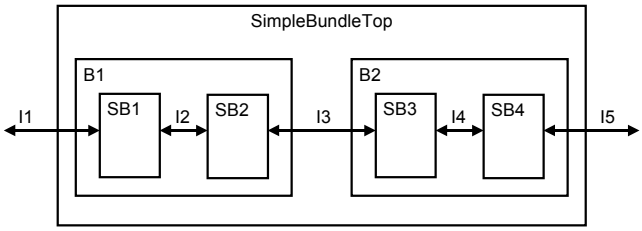
Details of an approach that uses this methodology and implements the interface as a record with elements that are based on std_logic is available in the second half of the paper titled, "Accelerating Verification Through Pre-Use of System-Level Testbench Components" that is available at: http://www.synthworks.com/papers/index.htm.

3.2. RTL Design

3.2.1. Simple Bundles

Connectivity of RTL functions consists of one or more signal objects. A bundle is a simplified form of an interface that allows this connectivity to be referenced as a single object. As such it is a composite similar in some ways to a record. One difference is that since a bundle is a port, there must be a method to resolve the IO direction of each element. Different elements may have a different IO direction.

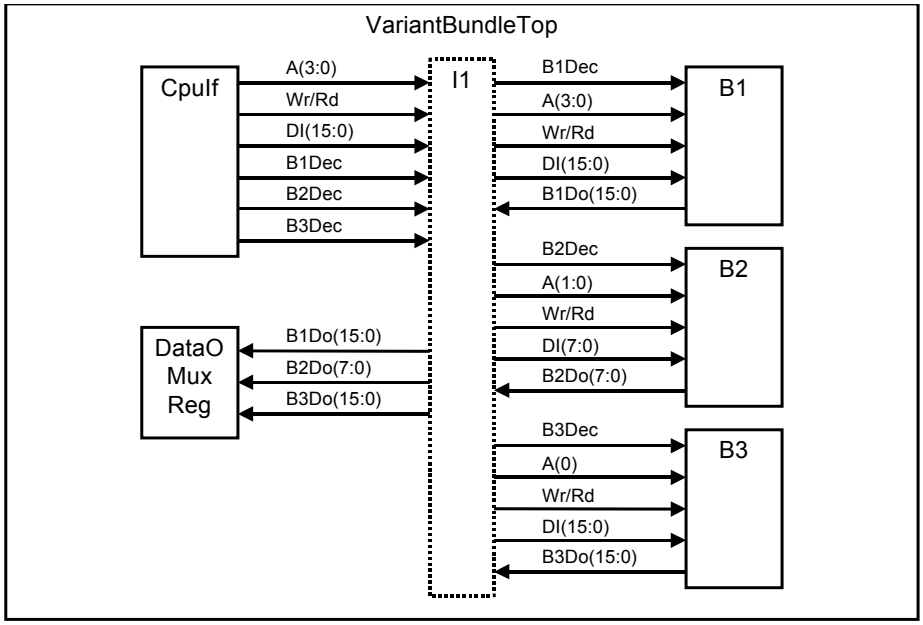
A simple model for connectivity using bundles is shown below. SimpleBundleTop is the top level of the design. B1 and B2 are subblocks of SimpleBundleTop. SB1 and SB2 are subblocks of B1. SB3 and SB4 are subblocks of B2. Each connection (labled I1, I2, I3, I4, and I5) is a bundle.



Within a bundle at any subblock, a given element may be mode in, out, or inout. Bundle elements typically have different IO modes at opposite ends of the interface. An element of bundle I2 that is mode “in” at SB1 will be mode “out” at SB2. Bundle elements typically have same IO modes as the bundle is propagated up or down through the hierarchy. An element of bundle I3 that is mode “in” at SB2 will be mode “in” at B1.

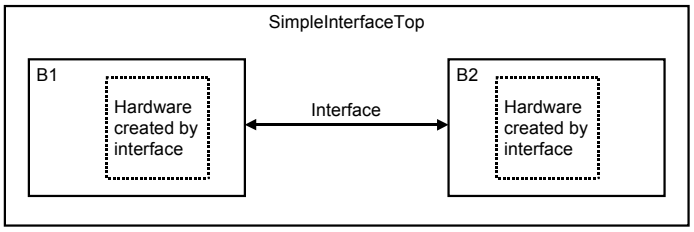
3.2.2. Variant Bundles

The simple view of bundles is that every block connects to every element of the bundle. This is not always the case. The following example shows potential connectivity of addressable logic. Note that in this case a subblock does not either drive or read some elements of the interface. In fact some subblocks do not use all bits of some array elements (note A and DI at B1, B2, and B3). From a synthesis standpoint, having unused elements on a composite is problematic - they show up as unconnected ports to backend tools. As a result, a method to specify partial connectivity to a bundle is desirable.



3.2.3. Simple Interfaces

A simple interface packages subprograms with bundles. A block (such as B1) calls a subprogram to create the interface behavior. Hence the interface behavior is hidden from B1. Note that since B1 calls the subprogram, the actual hardware will be created in block B1. The hardware created by the subprogram call is shown with dotted lines below to indicate that most likely it does not create any hierarchical boundaries.

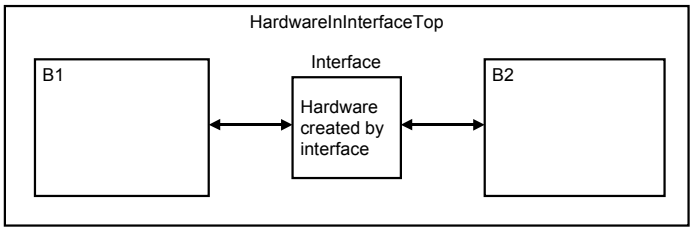


Bus behavior is a sequence of actions. A sequence of actions implies multiple clocks. If a subprogram were to implement a sequence of actions, it would require multiple waits containing clock. This is a coding style that IEEE 1076.6-2004 calls an implicit statemachine. This coding style will work for some simple interfaces, however, it could get challenging to implement a statemachine with many independent cycles. Even straightforward things like a UART receiver can be difficult.

This methodology seems to work well for testbenches and system level design, however, it would only be usable for very simple RTL designs.

3.2.4. Concurreny in Interfaces

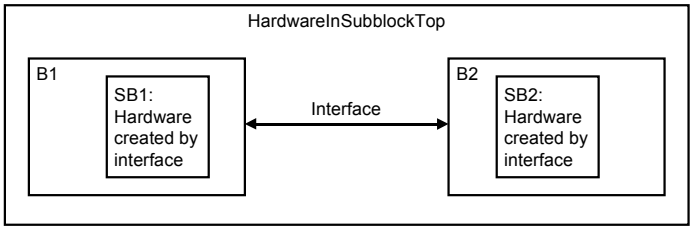
Hardware creation requires a concurrent region. Other languages suggest that hardware creation can be done in the interface as shown below.



If an interface construct is to create hardware, it would have to have a concurrent region. If it does, would it be a primary unit? Would it have a secondary unit?

An alternative to an interface containing a concurrent region is to use a separate entity/architecture pair to implement the concurrent portion of the interface. The interface entity would have an interface to each connected block (B1 and B2 above). In this approach, the main function subprograms in each interface would be to hand off data values to the interface entity. This could be by a simplified protocol. While it may be possible to implement the simple hardware handshaking elements in the interface, there may also be some benefit from having a library of handshaking components for this purpose. Using components allows different handshaking elements to be interchanged using configurations. For example at the system level view there may be only one clock and handshaking without registers can be accomplished. As the system evolves, the block that implements a function may have a different clock from the interface and a more sophisticated component can be selected. Would the handshaking components be instantiated in the block or the interface? Does it matter?

Another alternate view is for the interface hardware to be created as a subblock of B1 and B2 as shown below.



In this case, the SB1 is a subblock of B1 and is implemented as an entity/architecture. The required features of the interface between B1 and B2 is similar to a bundle. By having a standardized interface between B1 and SB1 and abstracting the connection as an interface/bundle, multiple different types of interfaces can be swapped in and out.

3.2.5. Limitations

For RTL design, bundles seem to offer a large amount of value. It is not clear how much value an interface methodology/feature would bring. In order to be able to interchange different blocks (USB, Firewire, ethernet) the information provided to each interface must be identical. This means the interface cannot be externally configurable. If the interfaces are externally configurable, the information exchanged is no longer the same and interchanging blocks is no longer a simple exchange one for the other.

4. Current Capabilities

At a bare minimum an interface is a composite with a method to group the subprograms together. To some degree, this can be done using a record for the composite and a package to group the record with the subprograms.

4.1.1. Single Record + Package

Using current language features, an interface could be a record and subprograms defined in a package. Since all of the record elements are not in the same direction, the record object must be an inout of an entity. As a result, all record elements must be of a resolved subtype (based on std_logic). This is shown in the following example:

```
type UartTbRecType is record
  CmdRdy             : std_logic ;
  CmdAck             : std_logic ;
  Data               : std_logic_vector (7 downto 0);
  StatusMode         : unsigned ( 3 downto 0) ;
  TbErrCnt           : unsigned (15 downto 0) ;
  UartBaudPeriod     : unsigned (31 downto 0) ;
  NumDataBits        : unsigned ( 2 downto 0) ;
  ParityMode         : unsigned ( 2 downto 0) ;
  NumStopBits        : std_logic ;
end record ;
```

To resolve drivers, all of the record elements are initialized to ‘Z’. Typically this is done with a constant, such as the one that follows.

```
constant InitTbUartTbRec : UartTbRecType := (
    CmdRdy           => 'Z',
    CmdAck           => 'Z',
    Data             => (others => 'Z'),
    StatusMode       => (others => 'Z'),
    TbErrCnt         => (others => 'Z'),
    UartBaudPeriod   => (others => 'Z'),
    NumDataBits      => (others => 'Z'),
    ParityMode       => (others => 'Z'),
    NumStopBits      => (others => 'Z')
) ;
```

In this approach, supporting subprograms are needed for handshaking through the record interface. These are somewhat generic in that any interface that uses the same method for handshaking can use them. In addition, supporting subprograms are also needed for copying values to the record at the start of a transaction and out of the record at the end of a transaction. These would be specific to a particular interface as they have characteristics that are based on the interface itself (address, data, and operation type).

4.1.2. Two Records

There are also transaction-based testbench approaches that use two records: one for "in" and the other for "out". As a result, there are no drivers to resolve. Since this is not a single element, it is not a true interface, but it is "interface like". When using multiple records, both records must be specified on the interface and with every subprogram call (transaction initiation).

5. Implementation Considerations

5.1. Composites: Resolving Values vs. Specifying Element Direction

An interface at the minimum is a composite object. The composite must pass through a port in an entity. It has individual elements that can be either in, out, or inout with respect to the current design. Hence, a method to resolve drivers or specify element direction is needed. The following four methods will be considered:

- Require all elements to be resolved types
- Pass an undriven value up through the hierarchy
- Last driven value wins semantics
- Specify the mode (direction) of each element of a composite

5.1.1. Require All Elements to be Resolved Subtypes

This approach uses resolved subtypes to allow multiple drivers on an interface object.

In this method, all composite objects are inout on entity ports. Each element must be of a resolved subtype (such as std_logic). The composite is initialized so that all elements get a non-driving value ('Z'). To drive a value across the interface, only one driver can be providing an active (non-Z) value at a time. An example of a record and initialization value is shown in the section titled "Single Record + Package".

This approach is limiting in that it requires a resolved type that has a representation for non-driven (such as 'Z' from std_logic). As a result elements that would be better to be represented as integer (such as TbErrCnt) and time (UartBaudPeriod) must be converted to either unsigned or std_logic_vector.

5.1.2. Pass an Undriven Value Through the Hierarchy

This is an approach to work around having to use explicitly resolved subtypes.

In this method, all composite objects are inout on entity ports. Elements of a composite that are not driven by a given model shall be made undriven (?through initialization, explicit assignment, or disconnect?). It may be necessary to either mark the subtype of the element (with an indication to use a default resolution method) or create a new object class that permits this (interface). The intent is that an undriven composite element behaves as if it were marked with mode "in".

In many ways this abuses the sense of inout. As a result, if a new interface object class is created, perhaps it should not require a mode at all.

5.1.3. Last Driven Value Wins

This is another approach to work around having to use resolved subtypes.

Create a new object class that maintains delta cycle assignment semantics of signals (so it has events), but rather than have multiple drivers, it only retains the last value assigned (similar to a variable). In this case, when a new value is deposited on an element of a composite object, that value becomes the effective value of the element.

5.1.4. Specify the Mode (direction) of Each Element of a Composite

This approach specifies the direction of each element of a composite. As such it avoids the need for resolved subtypes (except when the element is inout) and avoids the abuse of the mode "inout". This can be accomplished in a similar manner to System Verilog modports.

5.2. Packaging / Structure

There are often subprograms that are associated with operations on the composite object that is part of the interface. A couple of options for this are:

- Package + Record
- Class

What is the difference between a package and a class?

Class:

- Inherits from a base class.
- If an externally defined extended class is visible then definitions inherited from base class are visible.
- If an externally defined extended class is visible then the base class is also visible?
- Subprograms defined in the extended class hide homographs in the base class.
- ?If a subprogram that is defined only in the base class is called for an extended class and that subprogram in turn calls a subprogram that is defined in both the extended class and the base class, then the subprogram from the ?extended? (or is it ?base?) class is called.
- Is a class a type? Can it be overloaded outside of the class definition?
- Could a class be a simple extension of a record declaration?

Package:

- Visibility of declarations in a package is gained by referencing the package
- Package can combine multiple extensions on a single type.
  Consider a basic math package such as numeric_std. In a separate package overloading can be provided for sin/cos functions. In another separate package overloading for square root can be provided. A program can reference all the packages and get the accumulated subprograms.

6. Interface Implementation

6.1. Base-line Implementation

Interfaces are a methodology. As such to implement them, it is permissible to leverage existing constructs. Since this revision effort has a goal of minimizing additional keywords, it seems reasonable to first try to use or extend current language features.

This proposal extends records to create a simple interface. Concurrency in this approach will be implemented in an entity/architecture. The record allows definition of a composite type. Records are extended with OO features using a use clause and specifying a type. IO direction of elements is declared using a port declaration. Records are extended to allow subprograms to be declared within the construct. The syntax definition is as follows:

```
record_type_definition ::=
  record
      element_or_extension_declaration
      {element_or_extension_declaration}

      {port_or_subprogram_declaration}

    end record [ record_type_simple_name ]

element_or_extension::= element_declaration | extension_declaration

element_declaration ::= identifier_list : element_subtype_definition ;

element_subtype_definition ::= subtype_indication

extension_declaration ::= use identifier_list ;

identifier_list ::= identifier { , identifier }

port_or_subprogram_declaration ::= record_port_declaration | subprogram_body

record_port_declaration ::=
    port identifier [use identifier] ( record_port_list ) ;

record_port_list ::= interface_list

interface_list ::=  interface_element { ; interface_element }
interface_element ::= interface_declaration

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration
    | interface_file_declaration
    | use record_identifier.record_port_identifier
```

The following example adapts the transaction-based interface implemented with a single record and a package. Note that the types used in the record have been changed to something that is easier to work with.

```
type RdyRdyHandShaking is record
  CmdRdy             : std_logic ;
  CmdAck             : std_logic ;

  Port SourcePort (
    CmdRdy     : out std_logic ;
    CmdAck     : in std_logic
  ) ;

Port ModelPort (
    CmdRdy     : in std_logic ;
    CmdAck     : out std_logic
  ) ;

  procedure SourceRdy ( Rec : SourcePort RdyRdyHandShaking) is
      . . .
  begin
      . . .
  end procedure SourceRdy ;

  procedure ModelRdy ( Rec : ModelPort  RdyRdyHandShaking) is
     . . .
  begin
     . . .
  end procedure ModelRdy;

end record RdyRdyHandShaking ;


type UartTbRecType is record
  use RdyRdyHandShaking ;

  Data               : std_logic_vector (7 downto 0);
  StatusMode         : StatusMode_EnumType ;
  TbErrCnt           : integer ;
  UartBaudPeriod     : time ;
  NumDataBits        : integer ;
  ParityMode         : ParityMode_EnumType
  NumStopBits        : integer ;

  Port UartTbSourcePort (
    Use RdyRdyHandShaking.SourcePort ;
    Data             : inout std_logic_vector (7 downto 0) := (others = 'Z') ;
    StatusMode       : out StatusMode_EnumType ;
    TbErrCnt         : out integer ;
    UartBaudPeriod   : out time ;
    NumDataBits      : out integer ;
    ParityMode       : out ParityMode_EnumType
    NumStopBits      : out integer
  ) ;
```

```
Port UartTbModelPort (
    Use RdyRdyHandShaking.ModelPort ;
    Data              : inout std_logic_vector (7 downto 0) := (others = 'Z') ;
    StatusMode        : in StatusMode_EnumType ;
    TbErrCnt          : in integer ;
    UartBaudPeriod    : in time ;
    NumDataBits       : in integer ;
    ParityMode        : in ParityMode_EnumType
    NumStopBits       : in integer
  ) ;

procedure UartTransmit ( Rec : UartTbSourcePort UartTbRecType, Data :
std_logic_vector(7 downto 0) ) is
      . . .
 begin
      . . .
  end procedure UartTransmit;

end record ;
```

6.2. To be resolved

The following is a list of todo and/or items that need further consideration:

- Needs an example showing the solution to issues in variant bundles
- Is use of "use" ok or should a alternate be used such as "extends"?
- Should there be record bodies?
  Current proposal indirectly suggests packages should allow subprogram bodies.
- Should a class be a primary unit and, hence, cannot be an extension of a record?
- Support limited access to data items such that the value can only be accessed by a method in the interface.
  Alternatives seem to be either a keyword such as "private" or a "record body".
- Contractual relationship that defines the "procedures" required of an implementation. (SB6)
  How do we create virtual subprograms? Include a subprogram declaration in a record and require that to use a record type in a declaration, the record must an implementation for all subprograms.
- Including assertions with the interface. Assertions could be included in a separate entity/architecture that has the record as an in so it can monitor all values in the record.
- Interoperability with other transaction based approaches (SystemC)
- Interface should be extendable and/or parameterizable via generics (SB8)
- Ability to specify delay at the interface (SB9)
  Do we want this? Alternatives, make it an optional part of a record port declaration or create a delay declaration. If it was made part of a record port declaration, perhaps entities could be extended in a similar way. I am not sure I want this as delays are often more complicated than one value. For example, propagation delay may have different values depending on whether it is rising or falling. It may also be unclocked and be dependent on several different signals (such as RAM DataOut being dependent on Address setup, chip select, and output enable).
- How do we configure a design that contains bundles and/or interfaces?
  The composite type for the interface will need to be passed as a type generic to an entity. Some how this type generic needs to be based on a base class (with virtual/abstract subprograms). Any type that gets mapped to the type generic would need to be derived from the base class in order to be valid.
- Can we merge the interface concept with the OO and constrained random features?
- Many other things that escape me at this point. ☺