10 NOTES ON ALGORITHMIC MODELING INTERFACE AND PROGRAMMING GUIDE

This section is organized as an interface and programming guide for writing the executable code to be interfaced by the [Algorithmic Model] keyword described in Section 6c. Section 10 is structured as a reference document for the software engineer.

10.1 OVERVIEW

The algorithmic model of a Serializer-Deserializer (SERDES) transmitter or receiver consists of three functions: 'AMI_Init', 'AMI_GetWave' and 'AMI_Close'. The interfaces to these functions are designed to support three different phases of the simulation process: initialization, simulation of a segment of time, and termination of the simulation.

These functions ('AMI_Init', 'AMI_GetWave' and 'AMI_Close') should all be supplied in a single shared library, and their names and signatures must be as described in this section. If they are not supplied in the shared library named by the Executable sub-parameter, then they shall be ignored. This is acceptable so long as

1. The entire functionality of the model is supplied in the shared library.
2. All termination actions required by the model are included in the shared library.

The three functions can be included in the shared object library in one of the three following combinations:

Case 1: Shared library has AMI_Init, AMI_Getwave and AMI_Close.
Case 2: shared library has AMI_Init and AMI_Close.
Case 3: Shared library has only AMI_Init.

Please note that the function 'AMI_Init' is always required.

The interfaces to these functions are defined from three different perspectives. In addition to specifying the signature of the functions to provide a software coding perspective, anticipated application scenarios provide a functional and dynamic execution perspective, and a specification of the software infrastructure provides a software architecture perspective. Each of these perspectives is required to obtain interoperable software models.

10.2 APPLICATION SCENARIOS

10.2.1 LINEAR, TIME-INVARIANT EQUALIZATION MODEL

1. From the system netlist, the EDA platform determines that a given [Model] is described by an IBIS file.
2. From the IBIS file, the EDA platform determines that the [Model] is described at least in part by an algorithmic model, and that the AMI_Init function of that model returns an impulse response for that [Model].
3. The EDA platform loads the shared library containing the algorithmic model, and obtains the addresses of the AMI_Init, AMI_GetWave, and AMI_Close functions.
4. The EDA platform assembles the arguments for AMI_Init. These arguments include the impulse response of the channel driving the [Model], a handle for the dynamic memory used by the [Model], the parameters for configuring the [Model], and optionally the impulse responses of any crosstalk interferers.
5. The EDA platform calls AMI_Init with the arguments previously prepared.

6. AMI_Init parses the configuration parameters, allocates dynamic memory, places the address of the start of the dynamic memory in the memory handle, computes the impulse response of the block and passes the modified impulse response to the EDA tool. The new impulse response is expected to represent the filtered response.
7. The EDA platform completes the rest of the simulation/analysis using the impulse response from AMI_Init as a complete representation of the behavior of the given [Model].
8. Before exiting, the EDA platform calls AMI_Close, giving it the address in the memory handle for the [Model].
9. AMI_Close de-allocates the dynamic memory for the block and performs whatever other clean-up actions are required.
10. The EDA platform terminates execution.

10.2.2 Nonlinear, and / or Time-variant Equalization Model

1. From the system netlist, the EDA platform determines that a given block is described by an IBIS file.
2. From the IBIS file, the EDA platform determines that the block is described at least in part by an algorithmic model.
3. The EDA platform loads the shared library or shared object file containing the algorithmic model, and obtains the addresses of the AMI_Init, AMI_GetWave, and AMI_Close functions.
4. The EDA platform assembles the arguments for AMI_Init. These arguments include the impulse response of the channel driving the block, a handle for the dynamic memory used by the block, the parameters for configuring the block, and optionally the impulse responses of any crosstalk interferers.
5. The EDA platform calls AMI_Init with the arguments previously prepared.
6. AMI_Init parses the configuration parameters, allocates dynamic memory and places the address of the start of the dynamic memory in the memory handle. AMI_Init may also compute the impulse response of the block and pass the modified impulse response to the EDA tool. The new impulse response is expected to represent the filtered response.
7. A long time simulation may be broken up into multiple time segments. For each time segment, the EDA platform computes the input waveform to the [Model] for that time segment. For example, if a million bits are to be run, there can be 1000 segments of 1000 bits each, i.e., one time segment comprises 1000 bits.
8. For each time segment, the EDA platform calls the AMI_GetWave function, giving it the input waveform and the address in the dynamic memory handle for the block.
9. The AMI_GetWave function computes the output waveform for the block. In the case of a transmitter, this is the Input voltage to the receiver. In the case of the receiver, this is the voltage waveform at the decision point of the receiver.
10. The EDA platform uses the output of the receiver AMI_GetWave function to complete the simulation/analysis.
11. Before exiting, the EDA platform calls AMI_Close, giving it the address in the memory handle for the block.
12. AMI_Close de-allocates the dynamic memory for the block and performs whatever other clean-up actions are required.
13. The EDA platform terminates execution.

10.2.3 Reference system analysis flow

System simulations will commonly involve both TX and RX algorithmic models, each of which may perform filtering in the AMI_Init call, the AMI_Getwave call, or both. Since both LTI and non-LTI behavior can be modeled with algorithmic models, the manner in which models are evaluated can affect simulation results. The following steps are defined as the reference simulation flow. Other methods of calling models and processing results may be employed, but the final simulation waveforms are expected to match the waveforms produced by the reference simulation flow.

The steps in this flow are chained, with the input to each step being the output of the step that preceded it.

- Step 1. The simulation platform obtains the impulse response for the analog channel. This represents the combined impulse response of the transmitter's analog output, the channel and the receiver's analog front end. This impulse response represents the transmitter's output characteristics without filtering, for example, equalization.
- Step 2. The output of Step 1 is presented to the TX model's AMI_Init call. If Use_Init_Output for the TX model is set to True, the impulse response returned by the TX AMI_Init call is passed onto Step 3. If Use_Init_Output for the TX model is set to False, the same impulse response passed into Step 2 is passed on to Step 3.
- Step 3. The output of Step 2 is presented to the RX model's AMI_Init call. If Use_Init_Output for the RX model is set to True, the impulse response returned by the RX AMI_Init call is passed onto Step 4. If Use_Init_Output for the RX model is set to False, the same impulse response passed into Step 3 is passed on to Step 4.
- Step 4. The simulation platform takes the output of Step 3 and combines (for example by convolution) the input bitstream and a unit pulse to produce an analog waveform.
- Step 5. The output of Step 4 is presented to the TX model's AMI_Getwave call. If the TX model does not include an AMI_Getwave call, this step is a pass-through step, and the input to Step 5 is passed directly to step 6.
- Step 6. The output of Step 5 is presented to the RX model's AMI_Getwave call. If the RX model does not include an AMI_Getwave call, this step is a pass-through step, and the input to Step 6 is passed directly to Step 7.
- Step 7. The output of Step 6 becomes the simulation waveform output at the RX decision point, which may be post-processed by the simulation tool.

Steps 4 though 7 can be called once or can be called multiple times to process the full analog waveform. Splitting up the full analog waveform into mulitple calls minimizes the memory requirement when doing long simulations, and allows AMI_Getwave to return model status every so many bits. Once all blocks of the input waveform have been processed, TX AMI_Close and RX AMI_close are called to perform any final processing and release allocated memory.

10.3 FUNCTION SIGNATURES

Some introductory thoughts can go here,such as explaining that all arguments are required even if they are null pointers, etc...

Function: **AMI_Init**

Required: Yes

Declaration:

```
long AMI_Init (double *impulse_matrix,
               long row_size,
               long aggressors,
               double sample_interval,
               double bit_time,
               char *AMI_parameters_in,
               char **AMI_parameters_out,
               void **AMI_memory_handle,
               char **msg)
```

Arguments:

impulse_matrix

‘impulse_matrix’ is the channel impulse response matrix. The impulse values are in volts and are uniformly spaced in time. The sample spacing is given by the parameter ‘sample_interval’.
The impulse_matrix is stored in a single dimensional array of floating point numbers which is formed by concatenating the columns of the impulse response matrix, starting with the first column and ending with the last column. The matrix elements can be retrieved/identified using

impulse_matrix[idx] = element (row, col)
idx = col * number_of_rows + row
row – row index , ranges from 0 to row_size-1
col – column index, ranges from 0 to aggressors

The first column of the impulse_matrix is the impulse response for the primary channel. The rest are the impulse responses from aggressor drivers to the victim receiver.

The AMI_Init function may return a modified impulse response by modifying the first column of impulse_matrix. If the impulse response is modified, the new impulse response is expected to represent the filtered response. The number of items in the matrix should remain unchanged.

The aggressor columns of the matrix should not be modified.

row_size

The number of rows in the impulse_matrix.

aggressors

The number of aggressors in the impulse_matrix.

sample_interval

This is the sampling interval of the impulse_matrix. Sample_interval is usually a fraction of the highest data rate (lowest bit_time) of the device. Example:

Sample_interval = (lowest_bit_time/64)

bit_time

The bit time or unit interval (UI) of the current data, e.g., 100 ps, 200 ps etc. The shared library may use this information along with the impulse_matrix to initialize the filter coefficients.

AMI_parameters (_in and _out)

Memory for AMI_parameters_in is allocated and de-allocated by the EDA platform. The memory pointed to by AMI_parameters_out is allocated and de-allocated by the model. This is a pointer to a string. All the input from the IBIS AMI parameter file are passed using a string that been formatted as a parameter tree.

Examples of the tree parameter passing are:

```
(dll
  (tx
    (taps 4)
    (spacing sync)
  )
)
```

and

```
(root
  (branch1
    (leaf1 value1)
    (leaf2 value2)
    (branch2
      (leaf3 value3)
      (leaf4 value4)
    )
    (leaf5 value5 value6 value7)
  )
)
```

The syntax for this string is:

1. Neither names nor individual values can contain white space characters.
2. Parameter name/value pairs are always enclosed in parentheses, with the value separated from the name by white space.
3. A parameter value in a name/value pair can be either a single value or a list of values separated by whitespace.
4. Parameter name/value pairs can be grouped together into parameter groups by starting with an open parenthesis followed by the group name followed by the concatenation of one or more name/value pairs followed by a close parenthesis.
5. Parameter name/values pairs and parameter groups can be freely intermixed inside a parameter group.
6. The top level parameter string must be a parameter group.
7. White space is ignored, except as a delimiter between the parameter name and value.
8. Parameter values can be expressed either as a string literal, decimal number or in the standard ANCI 'C' notation for floating point numbers (e.g., 2.0e-9). String literal values are delimited using a double quote (") and no double quotes are allowed inside the string literals.
9. A parameter can be assigned an array of values by enclosing the parameter name and the array of values inside a single set of parentheses, with the parameter name and the individual values all separated by white space.

The modified BNF specification for the syntax is:

```
<tree>:
  <branch>

<branch>:
  ( <branch name> <leaf list> )

<leaf list>:
  <branch>
  <leaf>
  <leaf list> <branch>
  <leaf list> <leaf>

<leaf>:
  ( <parameter name> whitespace <value list> )

<value list>:
  <value>
  <value list> whitespace <value>
<value>:
  <string literals>
  <decimal number>
  <decimal number>e<exponent>
  <decimal number>E<exponent>
```

AMI_memory_handle

Used to point to local storage for the algorithmic block being modeled and shall be passed back during the AMI_GetWave calls. e.g. a code snippet may look like the following:

```
   my_space = allocate_space( sizeof_space );
   status = store_all_kinds_of_things( my_space );
   *serdes_memory_handle = my_space;
```

The memory pointed to by AMI_handle is allocated and de-allocated by the model.

msg (optional)

Provides descriptive, textual message from the algorithmic model to the EDA platform. It must provide a character string message that can be used by EDA platform to update log file or display in user interface.

Return Value

1 for success
0 for failure

Definition: Tells the world how wonderful AMI is.

Usage Rules: Use with caution. The garbage in, garbage out principles apply.

Other Notes: This is to describe how simple things can be made more complicated.

Examples:

```
(make up your own)
```

Function: **AMI_GetWave**

Required: No

Declaration:
```
long AMI_GetWave (double *wave,
                  long wave_size,
                  double *clock_times,
                  char **AMI_parameters_out,
                  void *AMI_memory)
```

Arguments:

wave

A vector of a time domain waveform, sampled uniformly at an interval specified by the ‘sample_interval’ specified during the init call. The wave is both input and output. The EDA platform provides the wave. The algorithmic model is expected to modify the waveform in place by applying a filtering behavior, for example, an equalization function, being modeled in the AMI_Getwave call.

Depending on the EDA platform and the analysis/simulation method chosen, the input waveform could include many components. For example, the input waveform could include:

- The waveform for the primary channel only.

- The waveform for the primary channel plus crosstalk and amplitude noise.
- The output of a time domain circuit simulator such as SPICE.

It is assumed that the electrical interface to either the driver or the receiver is differential. Therefore, the sample values are assumed to be differential voltages centered nominally around zero volts. The algorithmic model's logic threshold may be non-zero, for example to model the differential offset of a receiver; however that offset will usually be small compared to the input or output differential voltage.

The output waveform is expected to be the waveform at the decision point of the receiver (that is, the point in the receiver where the choice is made as to whether the data bit is a "1" or a "0"). It is understood that for some receiver architectures, there is no one circuit node which is the decision point for the receiver. In such a case, the output waveform is expected to be the equivalent waveform that would exist at such a node were it to exist.

wave_size

Number of samples in the waveform vector.

clock_times

Vector to return clock times. The clock times are referenced to the start of the simulation (the first AMI_GetWave call). The time is always greater than or equal to zero. The last clock is indicated by putting a value of -1 at the end of clocks for the current wave sample. The clock_time vector is allocated by the EDA platform and is guaranteed to be greater than the number of clocks expected during the AMI_GetWave call. The clock times are the times at which clock signal at the output of the clock recovery loop crosses the logic threshold. It is to be assumed that the input data signal is sampled at exactly one half clock period after a clock time.

AMI_parameters_out (optional)

A handle to a 'tree string' as described in 1.3.1.2.6. This is used by the algorithmic model to return dynamic information and parameters. The memory for this string is to be allocated and deleted by the algorithmic model.

AMI_memory

This is the memory which was allocated during the init call.

Return Value

1 for success
0 for failure

Definition: Tells the world how wonderful AMI is.

Usage Rules: Use with caution. The garbage in, garbage out principles apply.

Other Notes: This is to describe how simple things can be made more complicated.

Examples:

```
(make up your own)
```

Function: **AMI_Close**

Required: Sometimes

Declaration: `long AMI_Close(void * AMI_memory)`

Arguments:

AMI_memory

Same as for AMI_GetWave. See Section 3.2.2.5.

Return Value

1 for success
0 for failure

Definition: Tells the world how wonderful AMI is.

Usage Rules: Use with caution. The garbage in, garbage out principles apply.

Other Notes: This is to describe how simple things can be made more complicated.

Examples:

```
(make up your own)
```

10.4 CODE SEGMENT EXAMPLES

```
extern long AMI_GetWave (wave, wave_size, clock_times, AMI_memory);

  my_space = AMI_memory;

   clk_idx=0;
   time = my_space->prev_time + my_space->sample_interval;
   for(i=0; i<wave_size; i++)
     {
     wave = filterandmodify(wave, my_space);
     if (clock_times && found_clock (my_space, time))
       clock_times[clk_idx++] = getclocktime (my_space, time);
     time += my_space->sample_interval;
     }
  clock_times[clk_idx] = -1;   //terminate the clock vector
  Return 1;
```