Getting Started with TLM-2.0

A Series of Tutorials based on a set of Simple, Complete Examples

John Aynsley, Doulos, June 2008

Tutorial 2 – Response Status, DMI, and Debug Transport

Introduction

This second tutorial of the series explores the response status attribute of the generic payload, and also the direct memory and debug transport interfaces.

The Response Status of the Generic Payload

In tutorial example 1, when the target was unable to execute the transaction, it simply gave up by calling the SystemC report handler using SC_REPORT_ERROR(). This is acceptable, but there is a more structured way to handle the error, using the response status attribute of the generic payload. The response status is part of the transaction object, so can be checked by the initiator when the transaction is complete.

The default value of the response status attribute is TLM_INCOMPLETE_RESPONSE, indicating that the transaction has not reached the target or has not been executed by the target. If the transaction is executed successfully, the target should set the response status to TLM_OK_RESPONSE. If the transaction fails, the target can choose between a predefined set of error responses, as shown below:

```
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64    adr = trans.get_address() / 4;
  unsigned char*   ptr = trans.get_data_ptr();
  unsigned int     len = trans.get_data_length();
  unsigned char*   byt = trans.get_byte_enable_ptr();
  unsigned int     wid = trans.get_streaming_width();

  if (adr >= sc_dt::uint64(SIZE)) {
    trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
    return;
  }
```

The address error response should be used to indicate that the address is out-of-range or that the transaction failed because of the value of the address given in the transaction.

```
  if (byt != 0) {
    trans.set_response_status( tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE );
    return;
  }
```

The byte enable error response should be used to indicate either that the value of the byte enables in the transaction object caused an error at the target, or that the target does not support byte enables at all, as is the case in this example.

```
  if (len > 4 || wid < len) {
    trans.set_response_status( tlm::TLM_BURST_ERROR_RESPONSE );
    return;
  }
```

The burst error response should be used to indicate either that the value of the data length or the streaming width attribute in the transaction object caused an error at the target, or that the target does not support burst transfers or streaming bursts at all.

There is a fifth and final error response TLM_GENERIC_ERROR_RESPONSE, which can be used as a catch-all to indicate any kind of error in processing the transaction. A target can always fall back on the generic error response if it cannot or does not wish to set a more specific response.

When the initiator is checking the response status after the transaction is complete, the generic payload class provides a few convenience methods to make life easier:

```
socket->b_transport( *trans, delay );

if ( trans->is_response_error() )
{
  char txt[100];
  sprintf(txt, "Error from b_transport, response status = %s",
          trans->get_response_string().c_str());
  SC_REPORT_ERROR("TLM-2", txt);
}
```

is_response_ok and **is_response_error** are convenience methods that avoid the need to explicitly check the value of the enumeration. **get_response_string** is a method that returns the response as a text string, which can be convenient for printing out error messages.

Using the Direct Memory Interface

The purpose of the direct memory interface (DMI) is to speed up simulation by giving initiators a direct pointer to an area of memory in a target, thus bypassing the need to go through the transport interface for every single read and write transaction. DMI uses both forward and backward interfaces; remember, the forward path permits function calls from initiator socket to target socket, and the backward path permits function calls in the reverse direction. The forward DMI interface lets an initiator request a direct memory pointer from a target, and the backward DMI interface lets a target invalidate a DMI pointer previously given to an initiator.

We will start our examination of the DMI by looking at the method to get a new DMI pointer. This method is named **get_direct_mem_ptr**. It is called by the initiator along the forward path and is implemented by the target, a memory in our example. The memory uses the **simple_target_socket**, and as for **b_transport**, the target must register the implementation of the method with the socket. Otherwise, the simple socket would supply a default implementation that takes no action.

```
socket.register_get_direct_mem_ptr(this, &Memory::get_direct_mem_ptr);
```

The implementation of the method is as follows:

```
virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans,
                                 tlm::tlm_dmi& dmi_data)
{
  dmi_data.allow_read_write();
```

An initiator requests a DMI pointer for a particular address and for a particular mode of access, which usually means read access, write access, or both. The target must decide whether or not it can grant the kind of access being requested, and may even grant a higher level of access than requested. In this example, the target grants read/write access whatever the mode of the request. Ultimately the DMI transaction type is a template parameter, so applications can substitute their own modes of access where required. Of course, using a non-standard DMI transaction type will limit interoperability, just as substituting a non-standard type in place of the generic payload would limit interoperability when using the transport interface.

The target must now populate the DMI data object to describe the details of the access being given.

```
  dmi_data.set_dmi_ptr( reinterpret_cast<unsigned char*>( &mem[0] ) );
  dmi_data.set_start_address( 0 );
  dmi_data.set_end_address( SIZE*4-1 );
  dmi_data.set_read_latency( LATENCY );
  dmi_data.set_write_latency( LATENCY );

  return true;
}
```

The dmi_ptr is the actual direct memory pointer. This might not actually correspond to the requested address, because the target is free to grant any DMI region that encloses the requested address. It is usually desirable for the target to grant as large a region as possible. start_address and end_address describe the bounds of the DMI region from the point of view of the target, the entire memory contents in our example. read_latency and write_latency are estimates of the timing parameters for memory access, and may be used or ignored by the initiator depending on the degree of timing accuracy being modeled.

get_direct_mem_ptr returns true if it was able to provided a DMI pointer, or false otherwise.

Before we leave the target, there is one further refinement to consider. The target may signal to the initiator that it is able to support the direct memory interface using the *DMI hint* attribute of the generic payload. This can provide a simulation speedup for the initiator, because there is no point in the initiator making repeated calls to **get_direct_mem_ptr** if it can be told in advance that such calls are going to fail. Hence the **b_transport** method in our example makes the following call to set the DMI hint:

```
trans.set_dmi_allowed(true);
```

Now let's see how the initiator uses the DMI hint. After seeing the completion of a transaction, the initiator can check the DMI hint in the generic payload object, and if the hint is set, the initiator can request a DMI pointer from the target, knowing the request is likely to succeed:

```
tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
...
socket->b_transport( *trans, delay  );

if ( trans->is_dmi_allowed() )
{
  dmi_ptr_valid = socket->get_direct_mem_ptr( *trans, dmi_data );
}
```

The initiator calls **b_transport**, checks the DMI hint, then calls **get_direct_mem_ptr** and sets the flag **dmi_ptr_valid** to indicate a valid DMI pointer. Notice that the initiator is reusing the very same transaction object for both transport and direct memory interfaces, which improves the efficiency of the simulation. Subsequently, the initiator can use the DMI pointer to bypass the transport interface:

```
if (dmi_ptr_valid)
{
  if ( cmd == tlm::TLM_READ_COMMAND )
  {
    assert( dmi_data.is_read_allowed() );
    memcpy(&data, dmi_data.get_dmi_ptr() + i, 4);
    wait( dmi_data.get_read_latency() );
  }
  else if ( cmd == tlm::TLM_WRITE_COMMAND )
  {
    assert( dmi_data.is_write_allowed() );
    memcpy(dmi_data.get_dmi_ptr() + i, &data, 4);
    wait( dmi_data.get_write_latency() );
  }
}
else
{
  ...
  socket->b_transport(*trans, delay );
  ...
  if ( trans->is_dmi_allowed() )
    ...
}
```

Note the use of the DMI read_latency and write_latency parameters in the above code. When the initiator is using DMI, it honors the latencies passed with the **dmi_data** object.

This completes the description of the forward DMI interface. Now for the backward interface. The initiator must implement the **invalidate_direct_mem_ptr** method to wipe any existing pointers as requested by the target from time-to-time, and register this method with the simple initiator socket:

```
socket.register_invalidate_direct_mem_ptr(
       this, &Initiator::invalidate_direct_mem_ptr);
...
```

```
virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
                                       sc_dt::uint64 end_range)
{
  dmi_ptr_valid = false;
}
```

In this case the initiator ignores the bounds of the direct memory region, and simply invalidates the DMI pointer whatever.

Using the Debug Transport Interface

The purpose of the debug transport interface is to give an initiator the ability to read or write memory in the target without causing any side-effects and without simulation time passing. There are some similarities between DMI and debug, but the intent is very different. DMI is intended to speed-up simulation during normal transactions, whereas the debug transport interface is exclusively intended for debug.

There is only one debug transport interface, and that uses the forward path from initiator to target. The target must implement the **transport_dbg** method, and in the case of the simple target socket, must register the method with the socket. Otherwise, as for the direct memory interface, the simple socket would supply a default implementation that takes no action.

```
socket.register_transport_dbg(this, &Memory::transport_dbg);
...

virtual unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64    adr = trans.get_address() / 4;
  unsigned char*   ptr = trans.get_data_ptr();
  unsigned int     len = trans.get_data_length();

  unsigned int num_bytes = (len < SIZE - adr) ? len : SIZE - adr;

  if ( cmd == tlm::TLM_READ_COMMAND )
    memcpy(ptr, &mem[adr], num_bytes);
  else if ( cmd == tlm::TLM_WRITE_COMMAND )
    memcpy(&mem[adr], ptr, num_bytes);

  return num_bytes;
}
```

Notice that the debug transport interface is once again using the same transaction type, **tlm_generic_payload**. Compared to the transport interface, the debug transport interface only makes use of a restricted set of transaction attributes: the command, address, data pointer and data length. As you can see from the example above, the **transport_dbg** method is not obliged to read or write the given number of bytes, but should copy as many bytes as it is able. It is obliged to return the number of bytes actually copied. The implementation of this method is deliberately simple; it should not do anything other than copy bytes through the pointer in the transaction. Anything else would defeat the purpose.

Finally, we can see the initiator calling **transport_dbg** to dump out the memory contents:

```
trans->set_address(0);
```

```
trans->set_read();
trans->set_data_length(128);

unsigned char* data = new unsigned char[128];
trans->set_data_ptr(data);

unsigned int n_bytes = socket->transport_dbg( *trans );

for (unsigned int i = 0; i < n_bytes; i += 4)
{
  cout << "mem[" << i << "] = "
       << *(reinterpret_cast<unsigned int*>( &data[i] )) << endl;
}
```

You will find the source code for this second example in file tlm2_getting_started_2.cpp.