Section 11
Inter-Language Function Calls

11.1 Introduction (informative)

This section defines special declarations that can be used to allow SystemVerilog code to call C functions (SV-to-C direction) or to allow C code to call SystemVerilog functions (C-to-SV direction).

11.2 Mapping SystemVerilog function argument types to C function argument types

TBD.

11.3 SV-to-C function calls

```
extern_function_declaration ::=      // from Annex ???
        extern [attribute_instance] named_function_proto ["cname"] ;

attribute_instance ::= (* attr_spec {, attr_spec } *) // from Annex A.9.1

attr_spec ::=
          attr_name = constant_expression
        | attr_name

attr_name ::= identifier

Pre-defined values for identifier: pure | context | queueable
```

Syntax 11-1 External C function declaration syntax (excerpt from Annex A)

External C function declarations declare that a legal SystemVerilog *named_function_proto* function prototype (see section 10.3, Annex A.2.6) is externally implemented in C and is callable in the same way that a native SystemVerilog function would be called.

*Author's Note: Should we change the **extern** keyword to **import** ? This would make it more symmetrical with **export** for the C-to-SV direction.*

The quoted *cname* is an optional C alias for the function. If present, the external C function would actually have this name but all SystemVerilog code that calls the function would refer to it as the *function_identifier* (see section 10.3) in the *named_function_proto*. If the *cname* is absent, the C function name must match exactly with the *function_identifier.* In this case, the *function_identifier* would be restricted to legal C identifier syntax.

External C function declarations can be decorated with a number of pre-defined attributes. The attributes use the standard SystemVerilog attribute syntax that is listed in Annex A.9.1. For the specific case of external C function declarations the allowable values for the attribute *identifier* are **pure**, **context**, and **queueable**. These attributes are discussed in the following sub-sections.

Author's Note: I changed attributes to follow standard SystemVerilog attribute syntax as per chapter 6. This avoids pollution of keyword space by making attributes pre-standardized identifiers rather than reserved keywords. A concern was raised at the general SystemVerilog face to face meeting that keyword pollution is getting to be a problem.

11.3.1 Pure function calls

Pure function calls are attributed with the **pure** identifier. Calling pure function calls must have no side effects manifested in the SystemVerilog space of the design. The function's outputs must be purely a function of its inputs.

11.3.2 Simple static call bindings

The simplest type of SV-to-C function call name binding is, static name match. In this case the C function is a free standing function and is associated with no instance specific context.

In this simple usage the user is required to do only the following:

- Define a C function in C code.
- Declare that function in SystemVerilog code using the *extern_function_declaration* syntax described above.
- Call the function from anywhere in the SystemVerilog code within the scope of the function declaration and within the semantic requirements of all SystemVerilog function calls (described in section 10.3).

The user will be required to compile the C code into an object file that is linked with the SystemVerilog code. For static call binding, linking must be done based on symbol name match and argument profiles. If aliasing is used, the linked function name must match the *cname*. Otherwise it must match exactly with the *function_identifier.* Aliasing can be used in cases where the SystemVerilog function name is not a legal C identifier.

Here are examples of external C function declarations and the associated function calls from SystemVerilog:

Declare an external pure callable C function the SystemVerilog side:

```
    extern (* pure *) integer MyCFunc( input integer portID );
```

Define the function on the C side:

```
   int MyCFunc( int portID ){
       return locallyMapped( portID );
   }
```

It can be called from the SystemVerilog side as follows:

```
   always @( clock ) begin
       if( reset ) begin
           ..
       end
       else begin
           if( state == READY ) begin
               mappedID <= MyCFunc( portID ); // Call to C.
               state <= WAITING;
           end
           ...
       end
   end
```

Example 11-1 Pure C function callable from SystemVerilog

Declare an external pure callable void aliased C function on the SystemVerilog side:

```
   extern (* pure *) void MapPortID(
      input integer portID, output integer mappedID ) “MyCFunc”;
```

Define the function on the C side:

```
    void MyCFunc( int portID, int *mappedID ){
       *mappedID = locallyMapped( portID );
   }
```

It can be called from the SystemVerilog side as follows:

```
   always @( clock ) begin
       if( reset ) begin
           ..
       end
       else begin
           if( state == READY ) begin
               MapID( portID, mappedID ); // Call to C.
               state <= WAITING;
           end
           ...
       end
   end
```

Example 11-2 Pure void aliased C function callable from SystemVerilog

11.3.3 Context sensitive static call bindings

Notice that in both of the examples above, the C functions are free standing functions that can be called from SystemVerilog. In this case the C functions have no way of knowing any information about the module instance context of the caller. For simpler applications where only truly pure functions are needed, this should be adequate. However there are many cases where the C function must be able to differentiate the module instance context from which it is called. This would be typically be the case in a multi-threaded, object oriented C++ testbench modeling environment such as SystemC where it is desired to couple SystemVerilog model instances to multiple concurrent instances of user defined C++ models. In this case, the user may very much care about which specific instances of hardware models are making the function call and for those instances the user may wish to somehow associate a context pointer that refers to a specific C++ model or object instance with the specific SystemVerilog module instance making the call.

This is where the **context** attribute in the C function declaration can help. If the **context** attribute is present in the **extern** declaration, the SystemVerilog infrastructure shall automatically pass a VPI handle as the first argument to the C function, followed by the remaining arguments in the declaration. The VPI handle shall denote the module instance of the SystemVerilog module that is calling the function. This handle then can be used to associate a user defined C model context pointer with the SystemVerilog instances using `vpi_set_user_data()` and `vpi_get_user_data()`.

Author’s Note: One change I made here was the removal of the need to call tf_getinstance() to get the calling instance. I felt that it was cleaner just to have the infrastructure pass the SV module context directly if the context attribute is used. This avoids an extra call and reduces reliance on VPI even more - while still avoiding the need to add new API calls which is the intent of Joao’s original request that we replace dedicated calls with pre-existing VPI calls.

Here is an example of a C++ model, an external, context sensitive C function, and its associated function call from SystemVerilog:

Declare an external, callable, aliased, context sensitive C function

```
extern (* context *) integer MapID( input integer portID ) “MyCFunc”;
```

Define the function and model class on the C++ side:

```
class MyCModel {
    private:
        int locallyMapped( int portID );

    public:
        MyCModel( const char *svInstancePath ){
            vpiHandle svInstance = vpi_handle_by_name( svInstancePath, NULL );
            vpi_put_user_data( svInstance, this ); // Associate “this” with SV instance
        }

    friend int MyCFunc( vpiHandle context, int portID );
};

int MyCFunc( vpiHandle context, int portID ){
    MyCModel *me = (MyCModel *)vpi_get_user_data( context ); // Retrieve local context.
    return me->locallyMapped( portID );
}
```

Example 11-3 Context sensitive C function callable from SystemVerilog

On the SystemVerilog side, the calling syntax would be identical to that for simple binding shown in Example 11-1 on page 2. But now the C side uses VPI API functions, `vpi_get_user_data()`, `vpi_set_user_data()` and `vpi_handle_by_name()` to efficiently associate a SystemVerilog module instance context (denoted by the `vpiHandle context` argument) with a C++ model instance context (denoted by “this” pointer).

Example 11-3 shows how this is done. Notice that the constructor of the model does a one-time association of the C++ model “this” pointer with the SystemVerilog module instance path. It does this by using the VPI functions, `vpi_handle_by_name()` and `vpi_put_user_data()`. By doing this simple association, each time the C function `MyCFunc` is called, the local C++ model context can be efficiently retrieved using `vpi_get_user_data()`.

Author’s Note: Context association as described above seem like a rather trivial, unnecessary feature. But the ramifications of what it buys us is significant. While our simple caller binding use model gives us the simplicity and convenience of the existing CBlend and DirectC use model, what those use models are significantly lacking is a provision for a good interface between HDL models and specific instances of objects that represent C++ models. A simple feature of being able to associate user defined C++ model contexts with specific instances of HDL models allows for extremely flexible interconnect between software and hardware models. It facilitates multiple instances of a given software model type and connections between those software instances and their associated hardware instances.

To understand where context association is useful, consider the following scenario:

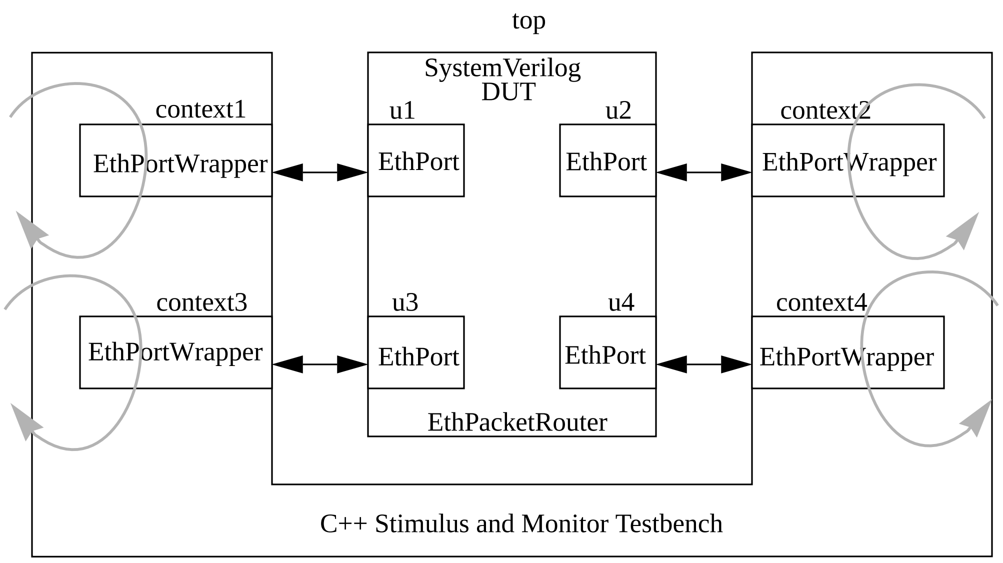

Figure 11-1 Scenario where context specific C functions are needed

A typical way of modeling this in a C++ testbench is to define a class representing the software model, `EthPortWrapper`. The user will then want to instantiate 4 copies of the same model to take maximum advantage of model reuse. Suppose the pointers to these 4 model instances are `context1`, `context2`, `context3`, and `context4`. It may also be the case that 4 independent threads are driving each `EthPortWrapper`.

On the hardware side, the user will have 4 instances of the Verilog module `EthPort`, namely, `top.u1`, `top.u2`, `top.u3`, and `top.u4`.

Now suppose each `EthPort` module declares an external callable C function to receive output packets from the `EthPort` modules on the SystemVerilog side. The user may define a C function called `HandleOutputPacket()`. When this C function is called, how will the user know which `EthPortWrapper` object is to receive the packet? This is where the context association can be used. The C function can associate the "this" pointer of each C model instance with the SystemVerilog module instance in a manner similar to that shown in Example 11-3 on page 4. It can then fetch this pointer using `vpi_get_user_data()` each time the call is made to gain access to the local instance of the `EthPortWrapper` model. In C++, the C function can be declared a `friend` function with private access privileges into the class. For example, `HandleOutputPacket()` could be a friend of `EthPortWrapper` that has full access privileges to all its private data members.

Here is an example showing excerpts of the `EthPortWrapper` C++ models and the `EthPort` SystemVerilog modules.

Define the function and model class on the C++ side:

```
class EthPortWrapper: public CModel {
    private:
        CModel *myParent;
        int dumpPayload( vpiHandle payload );

    public:
        EthPortWrapper( CModel *parent, const char *svInstancePath ) : myParent(parent) {
            vpiHandle svInstance = vpi_handle_by_name( svInstancePath, NULL );
            vpi_put_user_data( svInstance, this );
        }

    friend int HandleOutputPacket( vpiHandle context, int portID, vpiHandle payload );
};

void HandleOutputPacket( vpiHandle context, int portID, vpiHandle payload ){
    EthPortWrapper *me = vpi_get_user_data( context );
    me->myParent->BumpNumOutputs(); // Let top level know another packet received.
    me->dumpPayload( payload );
}
```

Prior to simulation, the user can make the following calls to construct `EthPortWrapper` models and associate them with the appropriate `EthPort` module instances:

```
    void *context1 = (void *)new EthPortWrapper( this, “top.u1” );
    void *context2 = (void *)new EthPortWrapper( this, “top.u2” );
    void *context3 = (void *)new EthPortWrapper( this, “top.u3” );
    void *context4 = (void *)new EthPortWrapper( this, “top.u4” );
```

Example 11-4 Ethernet packet router model association - C side

The code on the SystemVerilog side is listed here:

```
module EthPort(
       MiiOutData,                MiiInData,
       MiiOutEnable,              MiiInEnable,
       MiiOutError,               MiiInError,
       clk, reset );

    input [7:0] MiiOutData;       output [7:0] MiiInData;reg [7:0] MiiInData;
    input MiiOutEnable;           output MiiInEnable;    reg MiiInEnable;
    input MiiOutError;            output MiiInError;     reg MiiInError;
    input clk, reset;

    extern (* context *) void sendPacket(
        input integer portID, input reg [1439:0] payload ) “HandleOutputPacket”;

    reg [1439:0] outputPacketData;

    always @( clk ) begin     // output packet FSM
        if( reset ) begin
            ...
        end
        else begin
        if( outstate == READY ) begin
            if( MiiOutEnable )
                state <= PROCESS_OUTPUT_PACKET;
            end
        end
        else if( outstate == PROCESS_OUTPUT_PACKET ) begin
            // Assemble output packet byte by byte ...
        end
        else if( outstate == OUTPUT_PACKET_COMPLETE ) begin
            sendPacket( myPortID, outputPacketData ); // Make call to C side to handle it.
        end
    end
endmodule
```

Example 11-5 Ethernet packet router model association - SystemVerilog side

Notice that the actual C function name `HandleOutputPacket()` is aliased to the SystemVerilog name `sendPacket()`.

Author’s Note: With this simple solution we have completely generalized the SV-to-C function call interface to handle multi-model instance bindings. This feature buys a lot in object oriented, multi-threaded C++ testbench modeling environments like SystemC, TestBuilder, CynLibs, etc. It even applies to home-brewed single threaded pure C or C++ test environments which happen to use multiple instances of classes associated with the hardware.

11.3.4 Queueable function calls

The **queueable** attribute can be used to give the compiler a hint that a function call can be queued rather than having to wait for a full round trip confirmation that the callee has been called and has returned. Queueable functions are an easy way to implement the equivalent of SystemC channels, Unix sockets, message queues, name pipes, etc. They are useful for creating efficient 1-way channels between SystemVerilog processes and C threads. They are also well suited to implement optimized streaming channels between models.

Author’s note: Ironically, the OpenVera LRM has a similar facility called the VSV interface. So they apparently recognized the need to have something above and beyond DirectC for this type of capability. But what this proposal does is it gives you the entire capability without adding a whole new API to support it. With an ultra simple modification of

DirectC, you can cleanly support a socket type of channel like VSV did. And, in an implementation, if you've paid the price to implement DirectC function calls, with slightly more work you can get this feature almost for free.

The only requirement of queueable functions is that they can only have input arguments, no outputs. They can be thought of as one way transaction channels that the infrastructure can queue to arbitrary depth before blocking the caller. From the point of view of the SystemVerilog caller, a queueable function call will return immediately if there is room in the queue. At this point the caller will resume execution at the point after the call. But in fact the function may not called on the C side until some time later. The depth of a queueable function queue is implementation specific. Only when it fills will the caller be blocked.

Implementations that do not provide special optimizations for queueable functions will continue to work correctly although possibly not as efficiently.

Here is an example of a queueable, aliased, context sensitive C function declaration:

```
extern (* queueable, context *) void MyCFunc( input integer portID ) “AliasedCFunc”;
```

Example 11-6 Queueable, aliased, context sensitive C function declaration

11.3.5 Scoping

An external C function declaration can occur within module scope or root scope. No matter where external function declaration exists, the C function that it denotes is always considered to be of global scope.

External function declarations can exist in more than one place. Locally scoped declarations shall always override globally scope definitions. If two globally scoped external declarations have the same function profile (*named_function_proto*), but differing C names or attributes, it shall be considered an error.

11.4 C-to-SV function calls

exported_function_declaration ::= // from Annex ???
    export [attribute_instance] [scopename{::scopename}::]fname[**“**cname**”**] **;**

attribute_instance ::= **(*** attr_spec {**,** attr_spec } ***)** // from Annex A.9.1

attr_spec ::=
        attr_name **=** constant_expression
    | attr_name

attr_name ::= identifier

Pre-defined values for identifier: **pure** | **context** | **queueable**

Syntax 11-2 Exported SystemVerilog function declaration syntax (excerpt from Annex A)

SystemVerilog functions can be designated as callable from the C side by providing an export declaration of the form shown above. If the compiler detects an exported SystemVerilog function, it generates a special C wrapper function that is directly callable from the C side. This wrapper function then performs the internal operations required to actually call the SystemVerilog function. One of these operations is to map the C function input argument data types to SystemVerilog data types and to map the output and/or return arguments back to C data types.

The quoted *cname* is an optional C alias for the function. If present, the generated C wrapper function would actually have this name but internally it would call the SystemVerilog function by its real *fname*. If the *cname* is absent, the generated C wrapper function name will match exactly with the *fname*. In this case, the *fname* would be restricted to legal C identifier syntax.

Exported SystemVerilog function declarations can be decorated with a number of pre-defined attributes. The attributes use the standard SystemVerilog attribute syntax that is listed in Annex A.9.1. For the specific case of external C function declarations the allowable values for the attribute *identifier* are **pure**, **context**, and **queueable**. These attributes are discussed in the following sub-sections.

11.4.1 Pure function calls

Pure function calls are attributed with the **pure** identifier. Calling pure function calls must have no side effects manifested in the SystemVerilog space of the design. The function's outputs must be purely a function of its inputs.

11.4.2 Simple static call bindings

The simplest type of C-to-SV function call name binding is, static name match. In this case the SystemVerilog function is a free standing function that resides in root scope. It is associated with no instance specific module context.

In this simple usage the user is required to do only the following:

- Define a function in root scope of the SystemVerilog code.
- Declare that function as exported *export_function_declaration* syntax described above.
- Call the function from anywhere in the C code using normal C function call semantics.

The user will be required to compile the C caller code into an object file that is linked with the compiled SystemVerilog code that includes the generated C wrapper functions. For static call binding, linking must be done based on symbol name match and argument profiles. If aliasing is used the linked function name will match the *cname*. Otherwise it must match exactly with the *fname*. Aliasing can be used in cases where the SystemVerilog function name is not a legal C identifier.

Here is an example of an exported SystemVerilog function that is called from C:

Declare an exported pure C callable SystemVerilog function:

```
export (* pure *) MySvFunc;

function MySvFunc;
       input integer portID;
       output integer mappedID;

    mappedID = map[portID];
endfunction
```

It can be called from the C side as follows:

```
void MyCModel::RunTest( int portID ){
    MySvFunc( portID, &mappedID );
    ...
}
```

Example 11-7 Pure SystemVerilog function callable from C

11.4.3 Context sensitive static call bindings

In most cases where C calls SystemVerilog functions, those functions will be defined in module scope and therefore will be instance specific. In these cases C caller must know the module instance context of the SystemVerilog function that it calls.

This is where the **context** attribute in the exported function declaration is used. If **context** is present in the **export** declaration, the C wrapper function generated by SystemVerilog compiler will expect a VPI handle as its first argument. This will be followed by the actual function call arguments. The VPI handle shall denote the module instance of the SystemVerilog module containing the function.

Here is an example of C++ code calling an instance specific SystemVerilog function:

Declare an exported, context sensitive, aliased, void C callable SystemVerilog function:

```
export (* context *) MySvModule::MySvFunc “MySvModule_MySvFunc”;

module MySvModule( ... );
    ...
    function MySvFunc;
        input integer portID;
        output integer mappedID;

        mappedID = map[portID];
    endfunction
    ...
endmodule
```

Initialize context handle one time in constructor, pass it each time the SystemVerilog function is called from C:

```
class MyCModel {
    private:
        vpiHandle svContext;

    public:
        MyCModel( const char *svInstancePath ){
            svContext = vpi_handle_by_name( svInstancePath, NULL ); }

        void RunTest( int portID ){
            int mappedID;
            MySvModule_MySvFunc( dSvContext, portID, &mappedID );
            ...
        }
};
```

Example 11-8 Context sensitive SystemVerilog function callable from SystemVerilog

On the C side, the user code does a one time initialization of the SystemVerilog function instance handle by calling `vpi_handle_by_name()`. It then passes the handle as the first argument each time the SystemVerilog function is called from C, followed by the remaining function arguments.

In the example, the function `MySvModule_MySvFunc()` is a C wrapper function that is automatically generated by the SystemVerilog compiler. Some implementations may also wish to generate a header file with the function prototypes of all exported SystemVerilog functions as a convenience to the user. This header file can be included with user C code to declare exported functions on the C side. This naturally lets the C compiler automatically perform the proper function argument type checking.

11.4.4 Queueable function calls

As for the SV-to-C direction the **queueable** attribute can be used to give the compiler a hint that a SystemVerilog function call can be queued when called from the C side, rather than having to wait for a full round trip confirmation that the callee has been called and has returned. Queueable C-to-SV function calls have all the advantages described for queueable SV-to-C calls as described in Section 11.3.4 on page 7.

The only requirement of queueable functions is that they can only have input arguments, no outputs. They can be thought of as one way transaction channels that the infrastructure can queue to arbitrary depth before blocking the caller. From the point of view of the C caller, a queueable function call will return immediately if there is room in the queue. At this point the caller will resume execution at the point after the call. But in fact the function may not called on the SystemVerilog side until some time later. The depth of a queueable function queue is implementation specific. Only when it fills will the caller be blocked.

Implementations that do not provide special optimizations for queueable functions will continue to work correctly although possibly not as efficiently.

Here is an example of a queueable, aliased, context sensitive SystemVerilog function export declaration:

```
export (* queueable, context *) MySVFunc “MySvModule_MySvFunc”;
```

Example 11-9 Queueable, aliased, context sensitive SystemVerilog function export declaration

11.4.5 Scoping

Export declarations for SystemVerilog functions can be defined anywhere. If the function is defined in module scope but the export declaration is in root scope, the function name must be qualified with the module name using the *[scopename{::scopename}::]* syntax to prefix the function name shown in Syntax 11-2 on page 8. The scope name can either refer to a module scope or an interface scope for the case where functions are defined in an interface.

Example 11-8 on page 10 shows an example of this. In this case the export declaration for `MySvFunc` is shown in root scope and uses the `MySvModule::` qualifier to indicate that the function is defined in a module scope. If the export declaration had been placed inside the `MySvModule`, the extra qualifier would not have been necessary.

Export declarations can exist in multiple places. However, if multiple export declarations exist in root scope for the same SystemVerilog function, it is an error if they are not identical. If export declarations for a given function exist in both root and module scope, the local module scope declaration will override the root declaration if the declarations are not identical.