==
Proposal
==

At any scope where a function declaration can occur, permit the following declarations:

extern "DPI" [**pure** | **context**] [<*cname*>=] <named_function_proto>;

export "DPI" [<*cname*>=] **function** <*fname*>;

This syntax does not require any new keywords: "DPI" is a string qualifying the type of extern/export. For SV3.1 only the string "DPI" is valid.

Note that the <named_function_proto>, as defined in SV3.1 draft3 spec (page 233) must be modified slightly so that the argument list permits open arrays. Only extern DPI functions can use this argument type. Additionally we also want to relax the prototype syntax to permit unnamed arguments, again only for extern DPI prototypes.
If the extern declaration uses unnamed arguments, this function cannot be invoked with passing arguments by name syntax.

Semantics:

----

in both **extern** and **export**, *cname* is the name of the foreign function (extern/export), *fname* is the SV name for the same function. If *cname* not explicitly given, it will be the same as the SV function *fname*. An error will be generated if and only if the *cname* has characters that are not valid in a C function identifier.

This syntax permits several SV functions to be mapped to the same foreign function by supplying the same *cname* for several *fnames*. Note that all these SV functions would have identical argument types (as per rules below).

For any given *cname*, all declarations, regardless of scope, must have exactly the same type signature. The type signature includes the return type, the number, order, direction and types of each and every argument. Type includes dimensions and bounds of any arrays/array dimensions. Signature also includes the pure/context qualifiers that may be associated with an extern definition.

Only one extern declaration or export declaration of a given *fname* is permitted in any given scope. More specifically, for an **extern**, the **extern** must be the sole declaration of *fname* in the given scope. For an **export**, the function must be declared in the scope where the **export** occurs and there must be only one **export** of that *fname* in that scope.

NOTE: the restriction of a single extern definition for a given *fname* in a given scope may be relaxed (as per deferred issue 2) if and only if the extern declaration is identical (same type signature and exactly matching formal names for each and every argument) and exactly matching default values for all such externs of *fname* in the same scope. Argument names and default values are permitted to differ between **extern**s declared at different scopes as long as the type compatibility constraints are met. Extern "DPI" functions are never considered to be a "constant_function_call" as described in the SV 3.1 draft 3 LRM, section A.8.2 (page 247). Specifically, extern function calls cannot be used in any place requiring that the call be a constant_function_call. [Therefore extern calls cannot be made during compilation/elaboration] Exported functions are not affected by this restriction (i.e. exported functions can still qualify as constant functions if they meet the SV criteria for such).

For exported functions, the exported function must be declared in the same scope that contains the **export** "DPI" declaration. Only SV functions may be exported (specifically, this excludes exporting a class method)

Note that extern “DPI” functions declared this way can be invoked by hierarchical reference the same as any normal SV function. Declaring a SV function to be exported does not change the semantics or behavior of this function from the SV perspective (i.e. no effect in SV usage other than making this exported function also accessible to C callers)

The **pure** qualifier for an extern function specifies that this function has absolutely no side effects and uses no data other than its arguments. The function output is predicated only on the values of its input arguments. Side effects include reading/writing to any files or stdin/stdout, changing static or global variables, invoking any other APIs etc. Pure functions cannot invoke exported SV functions.

NOTE: still have to update above paragraph to use appropriate text from sv-cc LRM.

An unqualified extern function can have side effects but may not read or modify any SV signals other than those provided through its arguments. Unqualified externs are not permitted to invoke exported SV functions.

Extern functions with the **context** qualifier may invoke exported SV functions, may read or write to SV signals other than those passed through their arguments, either through the use of other interfaces or as a side-effect of invoking exported SV functions. Context functions are always implicitly supplied a scope representing the fully qualified instance name within which the extern declaration was present. (I.e. an extern function always runs in the instance in which the extern declaration occurred. This is the same semantics as SV functions, which also run in the scope they were defined, rather than in the scope of the caller) Extern context functions are permitted to have side effects and to use other SV interfaces (including but not limited to VPI). However note that declaring an extern context function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that may be needed by those other interfaces. It is the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces. (The svGetScopeName and related functions exist to provide a name based linkage from DPI to other interfaces) Exported functions can only be invoked if the current context refers to an instance from which the named function could be called in SV by an unqualified function call (i.e. a call to the function with no hierarchical path qualifier). In general this implies that an exported SV function is visible from its declared scope to any scope lower down in the hierarchy. Note that this implies that $root functions are visible to all callers. Attempting to invoke an exported SV function from a scope in which it is not directly visible will result in a runtime error; how such errors are handled is implementation dependent. If an extern function needs to invoke an exported function that is not visible from the current scope, it needs to change, via svSetScope, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SV. The current SV context will be preserved across a call to an exported function, even if current context has been modified by an application. Note that context is not defined for non-context externs and attempting to use any functionality depending on context from non-context externs can lead to unpredictable behavior.

This:

a) uses consistent syntax with other similar SV 3.1 constructs
b) permits straightforward extension later to extern/exported tasks
c) permits all information needed by DPI coder to be put in the scope where such usage occurs
d) by permitting extern/export declarations to be inside module scope, this simplifies (permits) use of DPI inside library modules using just normal library resolution mechanisms
e) allows for later extension of DPI to handle tasks (if such is ever necessary) without syntax conflict with existing usage
f) permits later extension to address other interfaces (eg "extern VPI function $foobar;") without syntax conflict with existing usage
g) use of extern/export is more symmetrical and was recommended by sv-ec and by Friday's all-committees meeting

NOTES:

1. do not need to put function prototype along with the export because export has to occur in same scope as the function definition being exported
2. **extern**/**export**s can occur anywhere where function declarations can occur. However note that exports must be in same scope as the function being exported.
3. clarified some of the rules regarding the prohibition of multiple extern/exports in a single scope (prohibition is only with respect to a given *fname*)

The following C side functions will exist in DPI to support contexts:

svScope **svFetchScope**()

returns the current scope. Returns NULL if invoked from a call chain starting with a non-context SV extern function call, otherwise an opaque handle to the SV context corresponding to the instance where the extern definition is present.

void svPutScope(**const** svScope)

sets the current context to the given scope handle. If this function is supplied an invalid scope handle results may be unpredictable. Note that this affects all following calls to exported SV functions, as their invocation relies on current scope. Note also it is not necessary to reset the context prior to returning from the extern call; Note that NULL is not a valid argument to this function. Behavior of SV calls if NULL passed to this function may be unpredictable.

void* **svGetUserData**(**const** svScope)

gets the current user data associated with the given scope. See notes for svGetUserData. Note: always returns NULL if called from a non-context DPI function.

svSetUserData(**const** svScope, **void***)

sets the user data (a void*) associated with the current context. Note that a single (*ONE*) user data pointer is associated with each context, regardless of how many extern functions may be defined in that instance. This implies that if two extern functions both set the user data for the same context, then only the last value will be obtained by getContextUserData.

svGetScopeByName(**const char***)

gets a scope handle when given a fully qualified instance name. Returns NULL if no such name available or if "byName" queries not enabled in this simulation run.

svGetScopeName(**const** svHandle)

gets the fully qualified name of a scope handle

int svGetCallerInfo(**char** ***fileName*, **int** **lineNumber*)

returns the file and line number in the SV code from which the extern call was made. If this information is available, returns TRUE and updates fileName and lineNumber to the appropriate values. Behavior is unpredictable if fileName or lineNumber are not appropriate pointers. If this information is not available return FALSE and contents of fileName and lineNumber not modified. Whether this information is available or not is implementation specific. Note that the string provided (if any) is owned by the SV implementation and is valid only until the next call to any SV function. Applications must not modify this string or free it.

Example usage:

------------------- sv code --------------------

```
    // NOTE: distance is SV name, Distance is C name
    extern "DPI" pure Distance=
      function integer distance(input integer a, input integer b);

    module top;
      TB tb;
    endmodule

    module TB
      // NOTE: MyDistance is SV name, Distance is C name. Both this
      // function and the function at $root map to the same C function and
      //therefore have to have identical type signatures
      extern "DPI" pure Distance=
             function integer MyDistance(input integer t,input integer v=10);

      DUT dut;
      ...
      $root.distance(10, 20); // invokes the C function Distance(10,
      MyDistance(2,); // invokes the C function Distance(2, 10)
    endmodule

    module DUT
      ...
      UNIT unit1;
      UNIT unit2;
      ...
    endmodule

    module UNIT
      extern "DPI" context genPacket = function void genIt();
      export "DPI" drivePacket = function driveIt;

      function void driveIt(input l, m, n)
             ...
      endfunction

      ...
      genPacket; // invokes C function genPacket(), which invokes this
      instances driveIt
      // via the C function drivePacket()'
      ...
    endmodule
```

----------------------- end sv code -----------------

---- c code ----

```
    int Distance(int a, int b)
    {
      return (int)(sqrt(a*a + b*b));
    }

    extern void drivePacket(int a, int b, int c);
```

```
/* NOTE: genPacket is context-aware function */
void genPacket()
{
   ...
   // call the drivePacket function in the instance from which
   // genPacket invoked
   drivePacket(x,y,z);
   ...
}
```

---- end c code -----

EXAMPLE CLARIFYING THE NOTION OF CONTEXT/SCOPES

This example is specifically to clarify the notion of the context of a function vs. the location where a function is called. In Verilog, functions run in the context where they are defined not where they are called from. Specifically, the only variables a function can "see" without qualification are the variables that occur in its context, not those visible in the location where it was called.

For example:

```
module foo(input clk);
   reg a, b; // top level decls

   function dummy(input bar);
         a = bar;
   endfunction

   always @(posedge clk) begin
      ...
      dummy(b); // call 1
      ...
      begin: myblock
         reg a; // local decls
         ...
         dummy(b); // call 2
      end
      ...
   end
endmodule
```

(please forgive any syntax *errors*). In the above example, both call 1 and call 2 cause the same variable, the top-level a, to be modified, even though in call2's local scope there is a different version of "a" available.

In short: for a SV function, context is always where the function was declared, not where it was called from. I hope this explanation clarifies the definition of "function context": it is the context where the function was declared (i.e. where the extern appears or where the function definition of an exported function appears) and is conceptually, just the fully qualified name of that function (minus the function name part itself).

SHORT TERM DEFERALL

(i.e. to be completed once extern/export syntax SV side semantics completed) - reusable mechanism for user data associated with a scope. Current suggestion is to modify sv(Get | Set)ScopeUserData to take an additional string argument. Internally tool maintains a table per instance mapping strings to associated user data. This permits several independent applications to manage user data in a consistent way without clashing with each other (i.e. permits interoperable DPI applications requiring context)

DEFERED ISSUES (TO SV3.2+) + WORKAROUNDS

1. syntax to export a specific instance of a given SV function for example, export only function "foo" from instance "top.here"
   Deferred proposal:

   ```
   export “DPI” [cname=] function path.fname;
   ```

   Workaround: At $root,

   ```
   export "DPI" c_foo=function globalFoo;

   function global_foo(...)
         top.here.foo(...)
   endfunction
   ```

   NOTE that the context of the exported c_foo function is $root and *not* top.here, which would be the case if foo was exported directly in its definition.

2. how to permit multiple IPs to all use the same shared externs (present in a shared “include” file)

   Deferred proposal:

   Permit multiple externs to coexist in the same scope as long as all the externs define exactly the same function. Note that this is a stronger requirement than that the externs be required to have the same type signature; this requirement in addition forces that all the externs use exactly the same names (if any) for the formals.

   Workarounds:

   Typically, shared library functions will typically also required that a set of complex types (structs) also be shared. Neither C nor SV permit for the multiple definition of types in a given scope. In C this issue is addressed by inclusion guards to prevent against the multiple inclusion of the same header file.

   This same approach may also be used with SV, as follows:

   ```
   `ifndef SV_HEADER_NAME_SV
   `define SV_HEADER_NAME_SV
   … shared datatype definitions and extern functions etc
   ```

```
`endif
```

SystemVerilog also permits an additional mechanism to be used, namely having a shared module contain all the required types and functions. This approach is commonly used with existing Verilog IP and would remain equally valid when DPI is used (in many respects, modules are the SV equivalent of C++ namespaces)

```
module MyBusLibrary;
… datatype definitions and extern functions etc
endmodule
```

3. syntax to permit a function definition to be exported from a scope other than where the function is declared.

   Deferred proposal:

   ```
   export “DPI” [cname=] function DefinitionScope::fname
   ```

   (note DefinitionScope may itself have DefinitionScope:: qualifiers, for the case of nested definition scopes, such as nested modules)

   Possible workaround:

   suggestion to investigate the “extends” mechanism currently under discussion in sv-ac, which permits for out-of-module code to “extend” the definition of a module. Using an extend, out-of-module code could contain the required export and still be considered local to the module, thus accomplishing the requirements with no new syntax.