Using Protected Types in VHDL Designs

By Jerry Kaczynski

Introduction

Protected types appeared for the first time in an amendment to VHDL standard (IEEE Std 1076a-1999) and were finally approved in 2002 release of the language standard (IEEE Std 1076-2002).

Protected types encapsulate data structures and operations that can be performed on them into one container, quite similar to classes in C++.

Although protected types were introduced to the language to address problems with simultaneous access to shared variables from multiple processes, their applications are much wider, i.e. better structured testbenches, convenient wrappers for complicated data structures, etc.

Since protected types are not backward compatible with some features of the 1993 version of the language, code using protected type must be compiled with ***2002*** option enabled via compilation command switch or checkbox selection in ***Compile Options | VHDL*** window.

Defining Protected Types

One of the purposes of protected types is to provide safe, exclusive access to encapsulated data structures. It implies that those structures should not be directly visible to the end user. The designer of the protected type can grant the desired level of access to internal data structures by providing ***methods***: public (visible to the end user) subprograms manipulating those structures.

To implement this functionality, complete definition of protected type consists of two parts:

- Protected type declaration
- Protected type body

Protected type declaration describes interfaces of methods (procedures and functions operating on data structures belonging to the protected type). We can call this declaration ***public*** part of the protected type description. The only restriction imposed on methods is that their formal parameters cannot be of access or file type.

Protected type body fully defines internal data structures and subprograms operating on them. All elements of protected type body that were not mentioned in the protected type declaration are ***private***, i.e. not visible to the end user of the protected type. It means that all internal data structures and non-method subprograms are private by definition.

If the protected type is defined in the *declarative part of the architecture*, then both protected type declaration and its body should be placed in this region.

If the protected type is defined in a *package*, then protected type declaration should be placed in the package declaration and protected type body in matching package body.

All subprograms defined in the protected type body have full access to internal data structures, so they should not specify them as formal parameters. It implies that all protected type functions accessing internal data must be defined as ***impure functions***.

Sample definition of protected type implementing safe flags that will not lock in case of simultaneous access from multiple processes is shown below:

```
type FLAG_TYPE is protected        -- protected type declaration
   procedure Set;
   procedure Reset;
   impure function Inactive return Boolean;
   impure function Active   return Boolean;
end protected FLAG_TYPE;

type FLAG_TYPE is protected body   -- protected type body
   variable flag : Boolean;        -- local variable declaration

   procedure Set is                -- procedure definition
      begin  flag := True;  end procedure Set;

   procedure Reset is              -- procedure definition
      begin  flag := False;  end procedure Reset;

   impure function Inactive return Boolean is -- function definition
      begin  return not flag;  end function Inactive;

   impure function Active return Boolean is   -- function definition
      begin  return flag;  end function Active;
end protected body FLAG_TYPE;
```

Using Protected Types

The typical objects of protected type are ***shared variables***, declared in declarative parts of design entities and packages, but not processes and subprograms. Shared variable must be declared with ***shared*** keyword in front of the ***variable*** keyword and protected type in subtype indication.

Regular variables declared in processes and subprograms can also be of protected type. In that case shared access to protected type data is not possible, but benefits of encapsulation are still available. Typical example of this situation is a protected type for handling variable length arrays implemented using linked list structures. Since the purpose of this type is to provide easy access to variable length array, not to allow sharing of such array, declaring regular variable of this protected type seems more appropriate than creating shared variable.

Names of protected type variables cannot be used in expressions and cannot be a target of the assignment. The proper use of such variable name is in a selected name, where prefix is the variable name and suffix is method name with optional actual parameters. For example, if the variable is declared like this:

```
shared variable my_flag : flag_type;
```

then it can be used like this:

```
if my_flag.active then . . .
```

In the statement above we are calling function method ***active*** of the ***my_flag*** object of the ***flag_type*** protected type to retrieve status of the flag (value of the internal variable of the protected type).

Because we have full set of access methods in the ***flag_type*** protected type, user of ***my_flag*** does not have to know what data type is used to store flag status. Thanks to encapsulation, flag can be set, reset and checked for either state using methods only.

Please note that the use of shared variables of protected type guarantees exclusive access to shared data structures for only one process at a time, but does not guarantee order of execution. It means that race conditions are still possible if update and reading of shared data structures is scheduled at the same simulation time and delta cycle.

In the sample code below we have an example of such race condition at simulation time 20 ns, delta cycle zero. Simulator has the right to decide whether to set the flag in process PRC2 first or read the flag status in PRC1 before it. If PRC1 gets hold of the flag first, then the loop condition will be true and 21st iteration will complete, making the process suspend until simulation time 21 ns is reached. If PRC2 updates flag first, then the loop in PRC1 will complete in the same cycle.

```
-- architecture declarative part:
shared variable my_flag : flag_type;
 . . .
-- architecture statement part:
prc1: process
begin
   report "PRC1: Process started...";
   while my_flag.inactive loop    -- reference point 1 (loop iteration 21
      wait for 1 ns;              --  at simulation time 20 ns)
   end loop;
   report "PRC1: Flag activation detected!";
   while my_flag.active loop
      wait for 1 ns;
   end loop;
   report "PRC1: Flag deactivation detected! Suspending...";
   wait;
end process;

prc2: process
begin
   report "PRC2: Process started...";
   wait for 20 ns;
   report "PRC2: Setting the flag...";    -- reference point 1
   my_flag.set;                           --  (simulation time 20 ns)
   wait for 20 ns;
   wait for 0 ns;      -- additional delta delay @ simulation time 40 ns
   report "PRC2: Resetting the flag...";
   my_flag.reset;      -- (simulation time 40 ns, delta cycle 1)
   report "PRC2: Suspending...";
   wait;
end process;
```

Please note that the addition of one delta cycle delay in the process PRC2 at the simulation time 40 ns removes race condition, since flag resetting and flag status check happen in different delta cycles.

Conclusion

The use of protected types opens new possibilities for advanced VHDL users, especially system-level modelers and testbench creators. Learning protected types will also make easier the transition to object oriented VHDL extensions planned in the future releases of the standard.