AlertLog Package
User Guide

User Guide for Release 2015.01

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

http://www.SynthWorks.com

Table of Contents

1 AlertLogPkg Overview

VHDL assert statements are a limited form of an alert and log filtering utility. Through a simulator, you can set an assertion level that will stop a simulation. Through a simulator, you can turn off some assertions from printing. However, none of this capability can be configured in VHDL, and in addition, at the end of a test, there is no way to retrieve a count of the ERROR level assertions that have occurred.

The AlertLogPkg provides Alert and Log procedures that replace VHDL assert statements and gives VHDL direct access to enabling and disabling of features, retrieving alert counts, and set stop counts (limits). All of these features can be used in either a simple global mode or a hierarchy of alerts.

Similar to VHDL assert statements, Alerts have values FAILURE, ERROR, and WARNING. Each is counted and tracked in an internal data structure. Within the data structure, each of these can be enabled or disabled. A test can be stopped if an alert value has been signaled too many times. Stop values for each counter can be set. The default for FAILURE is 0 and ERROR and WARNING are integer'right. If all test errors are reported as an alert, at the end of the test, a report can be generated which provides pass/fail and a count of the different alert values.

What differentiates AlertLogPkg from other alert and verbosity filtering packages is hierarchical alerts. Hierarchical alerts allow reporting of alerts for each model and/or each source of alerts. While the whole testbench will benefit from all models using alerts, a single model can effectively use either global or hierarchical alerts.

Logs provide a mechanism for verbosity control on printing. Through simulator settings, assert has this capability to a limited degree. Verbosity control allows messages (such as debug, DO254 final test reports, or info) that are too detailed for normal testing to be printed when specifically enabled.

AssertLogPkg uses TranscriptPkg to print to either std.textio.OUTPUT, a file, or both. When the TranscriptFile is opened, alert and log print to the file TranscriptFile, otherwise, they use std.textio.OUTPUT. For more details on TranscriptPkg see the TranscriptPkg User Guide (TranscriptPkg_user_guide.pdf).

Already using another package for alerts and verbosity control? AlertLogPkg also provides a mechanism to allow OSVVM alerts and logs to be processed through a separate alert and verbosity control package, such as the BitVis Utility Library, without necessitating recompiling all of the OSVVM library.

2 AlertLogPkg Use Models

Alerts may be used as either a single global alert counter, as a hierarchy of alert counters, or as an alert counter for a particular model. A simple hierarchy could use an alert counter for each model in the testbench. Each model can also have its own separate alert counters for each separate error source (functional, protocol checks, timing checks, ...).

3 Simple Methodology: Global Alert Counters

By default, there is a single global alert counter. All designs that use alert or log need to reference the package AlertLogPkg.

```
use osvvm.AlertLogPkg.all ;
architecture Test1 of tb is
```

Use Alert to flag an error, AlertIf to flag an error when a condition is true, or AlertIfNot to flag an error when a condition is false (similar to assert). Alerts can be of severity WARNING, ERROR, or FAILURE.

```
--                    message,          level
When others =>  Alert("Illegal State", FAILURE) ;
. . .
--         condition,                 message,              level
AlertIf(ActualData /= ExpectedData, "Data Miscompare …", ERROR) ;
. . .
read(Buf, A, ReadValid) ;
--            condition, message,            level
AlertIfNot( ReadValid, "read of A failed", FAILURE) ;
```

The output for an alert is as follows. Alert adds the time at which the log occurred.

```
%% Alert ERROR   Data Miscompare … at time: 20160 ns
```

Similar to assert, by default, when an alert FAILURE is signaled, a test failed message (see ReportAlerts) is produced and the simulation is stopped. This action is controlled by a stop count. The following call to SetAlertStopCount, causes a simulation to stop after 20 ERROR level alerts are received.

```
SetAlertStopCount(ERROR, 20) ;
```

Alerts can be enabled by a general enable, SetGlobalAlertEnable (disables all alert handling) or an enable for each alert level, SetAlertEnable. The following call to SetAlertEnable disables WARNING level alerts.

```
SetGlobalAlertEnable(TRUE) ; -- Default
SetAlertEnable(WARNING, FALSE) ;
```

Use ReportAlerts to provide a report of errors when a test completes.

```
ReportAlerts ;
```

When a test passes, the following message is generated:

```
%% DONE  PASSED  t1_basic  at 0 ns
```

When a test fails, the following message is generated (on a single line):

```
%% DONE  FAILED  t1_basic  Total Error(s) = 2  Failures: 0  Errors: 1  Warnings: 1
at 0 ns
```

Logs are used for verbosity control. Log level values are ALWAYS, DEBUG, FINAL, and INFO.

```
Log ("A message", DEBUG) ;
```

Log formats the output as follows.

```
%% Log   ALWAYS  A Message  at 0 ns
```

Each log level is independently enabled or disabled. This allows the testbench to support debug or final report messages and only enable them during the appropriate simulation run. The log ALWAYS is always enabled, all other logs are disabled by default. The following call to SetLogEnable enables DEBUG level logs.

```
SetLogEnable(DEBUG, TRUE) ;
```

4 Operation: Hierarchical Task-based Alert Counters

For larger test environments, it is useful to get both a final alert count (FAILURE, ERROR, and WARNING), as well as a report for the interface and/or model (ie: CPU Model or Ethernet Port1) generated the alerts. Hence, the ultimate goal of using hierarchical counters is to get an output from ReportAlerts:

```
%% DONE  FAILED  Testbench  Total Error(s) = 40  Failures: 12  Errors: 20
Warnings: 8  at 170 ns
%%    Default                  Failures: 4  Errors: 4  Warnings: 0
%%    OSVVM                    Failures: 0  Errors: 0  Warnings: 0
%%    CpuModel_1               Failures: 8  Errors: 0  Warnings: 0
%%      CPU Functional Errors  Failures: 2  Errors: 0  Warnings: 0
%%      CPU Protocol Errors    Failures: 2  Errors: 0  Warnings: 0
%%      CPU Timing Errors      Failures: 2  Errors: 0  Warnings: 0
%%    UART_1                   Failures: 0  Errors: 8  Warnings: 0
%%      UART Functional Errors Failures: 0  Errors: 2  Warnings: 0
%%      UART Protocol Errors   Failures: 0  Errors: 2  Warnings: 0
```

Each level in a hierarchy is referenced with an AlertLogID. A new AlertLogID is created by GetAlertLogID. If an AlertLog already exists for the specified name, GetAlertLogID will return its AlertLogID. It is recommended to use the instance label as the Name. The AlertLogID is used as an index into the data structure within the protected type.

```
--                                              Name,     Parent AlertLogID
Constant UartID : AlertLogIDType := GetAlertLogID("UART_1", ALERTLOG_ BASE_ID) ;
```

The AlertLogID is used in the call to Alert, Log, SetAlertEnable, SetAlertStopCount, and SetLogEnable.

```
Alert(UartID, "Uart Parity", ERROR) ;
AlertIf(Break='1', UartID, "Uart Break", ERROR) ;
AlertIfNot(ReadValid, UartID, "Read", FAILURE);
--         AlertLogID,  Level,    Enable,  DescendHierarchy
SetAlertEnable(UartID,  WARNING,  FALSE,   FALSE) ;
--            AlertLogID, Level, Count
SetAlertStopCount(UartID, ERROR,      20) ;

Log(UartID, DEBUG, "Uart Parity Received") ;

--       AlertLogID,  Level,    Enable, DescendHierarchy
SetLogEnable(UartID,  WARNING,  FALSE,  FALSE) ;
```

Printing of Alerts and Logs include the AlertLogID.

```
%% Alert FAILURE in CPU_1, Expect data XA5A5 at 2100 ns
%% Log   ALWAYS  in UART_1,  Parity Error  at 2100 ns
```

To find an existing AlertLogID, use FindAlertLogID. The value -1 is returned if the AlertLogID is not found.

```
Constant UartID : AlertLogIDType := FindAlertLogID(Name => "UART_1") ;
```

5 Method Reference

5.1 Package References

Using AlertLogPkg requires the following package references:

```
library osvvm ;
use osvvm.OsvvmGlobalPkg.all ;
use osvvm.AlertLogPkg.all ;
```

5.2 AlertType

Alert levels can be FAILURE, ERROR, or WARNING.

```
type AlertType is (FAILURE, ERROR, WARNING) ;
```

5.3 Simple Alerts

Simple alerts accumulate alerts in the default AlertLogID (ALERTLOG_DEFAULT_ID). It supports the basic overloading and usage:

```
procedure Alert( Message : string ; Level : AlertType := ERROR ) ;
. . .
Alert("Uart Parity") ;  -- ERROR by default
```

Alert has two conditional forms, AlertIf and AlertIfNot. The following is their overloading.

```
-- without an AlertLogID
procedure AlertIf( condition : boolean ;
    Message : string ; Level : AlertType := ERROR ) ;
impure function  AlertIf( condition : boolean ;
    Message : string ; Level : AlertType := ERROR ) return boolean ;
procedure AlertIfNot( condition : boolean ;
    Message : string ; Level : AlertType := ERROR ) ;
impure function  AlertIfNot( condition : boolean ;
    Message : string ; Level : AlertType := ERROR ) return boolean ;
```

Usage of conditional alerts:

```
AlertIf(Break='1', "Uart Break", ERROR) ;
AlertIfNot(ReadValid, "Read Failed", FAILURE) ;
```

The function form is convenient for use for conditional exit of a loop.

```
exit AlertIfNot(ReadValid, "in ReadCovDb while reading …", FAILURE) ;
```

5.4 Creating Hierarchy: GetAlertLogID

Each level in a hierarchy is referenced with an AlertLogID. The function, GetAlertLogID, creates a new AlertLogID. If an AlertLogID already exists for the specified name, GetAlertLogID will return its AlertLogID. It is recommended to use the instance label as the Name. The interface for GetAlertLogID is as follows.

```
impure function GetAlertLogID(Name : string ; ParentID : AlertLogIDType)
         return AlertLogIDType ;
```

As a function, GetAlertLogID can be called while elaborating the design by using it to initialize a constant or signal:

```
Constant UartID : AlertLogIDType :=
 --                             Name,        Parent AlertLogID
       GetAlertLogID("UART_1", ALERTLOG_BASE_ID);
```

5.5 FindAlertLogID: Find an AlertLogID

The function, FindAlertLogID, finds an existing AlertLogID. If the AlertLogID is not found, ALERTLOG_ID_NOT_FOUND is returned The interface for FindAlertLogID is as follows.

```
impure function FindAlertLogID(Name : string ) return AlertLogIDType ;
```

As a function, FindAlertLogID can be called while elaborating the design by using it to initialize a constant or signal:

```
constant UartID : AlertLogIDType := FindAlertLogID(Name => "UART_1") ;
```

5.6 Hierarchical Alerts

Hierarchical alerts require the AlertLogID to be specified in the call to alert. It supports the basic overloading and usage:

```
procedure alert(
  AlertLogID   : AlertLogIDType ;
  Message   : string ;
  Level     : AlertType := ERROR
) ;
. . .
Alert(UartID, "Uart Parity", ERROR) ;
```

Alert has two conditional forms, AlertIf and AlertIfNot. The following is their overloading. The function form is convenient for use for conditional exit of a loop.

```
procedure AlertIf( condition : boolean ; AlertLogID : AlertLogIDType ;
    Message : string ; Level : AlertType := ERROR ) ;
impure function  AlertIf( condition : boolean ; AlertLogID : AlertLogIDType ;
    Message : string ; Level : AlertType := ERROR ) return boolean ;
procedure AlertIfNot( condition : boolean ; AlertLogID : AlertLogIDType ;
    Message : string ; Level : AlertType := ERROR )  ;
impure function  AlertIfNot( condition : boolean ; AlertLogID : AlertLogIDType ;
    Message : string ; Level : AlertType := ERROR ) return boolean ;
```

Usage of conditional alerts:

```
AlertIf(Break='1', UartID, "Uart Break", ERROR) ;
AlertIfNot(ReadValid, UartID, "Read", FAILURE);
```

When an alert is signaled in a lower level of the hierarchy (such as UartID above), it increments all parent levels until it finds a level whose alert for that level is disabled or the top of the hierarchy.

5.7 ReportAlerts: Reporting Alerts

At test completion alerts are reported with ReportAlerts.

```
procedure ReportAlerts ( Name : string := "" ; AlertLogID : AlertLogIDType :=
ALERTLOG_BASE_ID ; ExternalErrors : AlertCountType := (others => 0) ) ;
. . .
ReportAlerts ;
```

ReportAlerts has 3 optional parameters: Name, AlertLogID, and ExternalErrors. Name specifies the test name (and can also be set with SetAlertLogName). AlertLogID allows reporting alerts for a specific AlertLogID and its children (if any). ExternalErrors allows separately detected errors to be reported. ExternalErrors is type AlertCountType and the value (WARNING => 1, ERROR => 5, FAILURE => 0) indicates detection logic separate from AlertLogPkg saw 1 Warning, 5 Errors, and 0 Failures. See notes under AlertCountType.

```
--            Name,   AlertLogID,  ExternalErrors
ReportAlerts("Uart1", UartID,      (WARNING => 1, ERROR => 5, FAILURE => 0) ) ;
```

ReportAlerts can also be used to print a passed/failed message for an AlertCount that is passed into the procedure call.

```
procedure ReportAlerts ( Name : String ; AlertCount : AlertCountType) ;
```

This is useful to accumulate values returned by different phases of a test that need to be reported separately.

```
ReportAlerts("Test1: Final", Phase1AlertCount + Phase2AlertCount) ;
```

Also see SetAlertLogOptions.

5.8 SetAlertLogName: Setting the Test Name

SetAlertLogName sets the name ReportAlerts prints when called without parameters - such as when an internal stop count reached.

```
procedure SetAlertLogName(Name : string ) ;
. . .
SetAlertLogName("Uart1") ;
```

5.9 SetGlobalAlertEnable: Alert Global Enable / Disable

SetGlobalAlertEnable allows Alerts to be globally enabled and disabled. The intent is to be able to disable all alerts until the system goes into reset. Alerts are enabled by default.

```
procedure SetGlobalAlertEnable (A : EnableType := TRUE) ;
impure function SetGlobalAlertEnable (A : EnableType := TRUE) return EnableType ;
```

Suppress all alerts before reset by turning alerts off during elaboration with a constant declaration and then turning them back on later.

```
InitAlerts : Process
  constant DisableAlerts : boolean := SetGlobalAlertEnable(FALSE);
begin
  wait until nReset = '1' ; --  Deassertion of reset
  SetGlobalAlertEnable(TRUE)  ;  -- enable alerts
```

5.10 SetAlertEnable: Alert Enable / Disable

SetAlertEnable allows alert levels to be individually enabled. When used without AlertLogID, SetAlertEnable sets a value for all AlertLogIDs.

```
procedure SetAlertEnable(Level : AlertType ;  Enable : boolean) ;
. . .
--                        Level,      Enable
SetAlertEnable(WARNING, FALSE) ;
```

When an AlertLogID is used, SetAlertEnable sets a value for that AlertLogID, and if DescendHierarchy is TRUE, it’s the AlertLogID's of its children.

```
procedure SetAlertEnable(AlertLogID : AlertLogIDType ;  Level : AlertType ;
Enable : boolean ; DescendHierarchy : boolean := TRUE) ;
```

5.11 SetAlertStopCount: Alert Stop Counts

When an alert stop count is reached, the simulation stops. When used without AlertLogID, SetAlertStopCount sets the alert stop count for the top level to the specified value if the current count is integer'right, otherwise, it sets it to the specified value plus the current count.

```
procedure SetAlertStopCount(Level : AlertType ;  Count : integer) ;
. . .
--                Level,     Count
SetAlertStopCount(ERROR,      20) ;  -- Stop if 20 errors occur
```

When used with an AlertLogID, SetAlertStopCount sets the value for the specified AlertLogID and all of its parents. At each level, the current alert stop count is set to the specified value when the current count is integer'right, otherwise, the value is set to the specified value plus the current count.

```
procedure SetAlertStopCount(AlertLogID : AlertLogIDType ;
    Level : AlertType ;  Count : integer) ;
. . .
--                AlertLogID,  Level,      Count
SetAlertStopCount(UartID,      ERROR,      20) ;
```

By default, the AlertStopCount for WARNING and ERROR are integer'right, and FAILURE is 0.

5.12 AlertCountType

Alerts are stored as a value of AlertCountType.

```
subtype AlertIndexType is AlertType range FAILURE to WARNING ;
type    AlertCountType is array (AlertIndexType) of integer ;
```

CAUTION: When working with values of AlertCountType, be sure to use named association as the type ordering may change in the future.

5.13 GetAlertCount

GetAlertCount returns the AlertCount value at AlertLogID. GetAlertCount is overloaded to return either AlertCountType or integer.

```
impure function GetAlertCount(AlertLogID : AlertLogIDType := ALERTLOG_BASE_ID)
    return AlertCountType ;
impure function GetAlertCount(AlertLogID : AlertLogIDType := ALERTLOG_BASE_ID)
    return integer ;
. . .
TopTotalErrors  := GetAlertCount ;          -- AlertCount for Top of hierarchy
UartTotalErrors := GetAlertCount(UartID) ;  -- AlertCount for UartID
```

5.14 GetEnabledAlertCount

GetEnabledAlertCount is similar to GetAlertCount except it returns 0 for disabled alert levels. GetEnabledAlertCount is overloaded to return either AlertCountType or integer.

```
impure function GetEnabledAlertCount(AlertLogID : AlertLogIDType :=
    ALERTLOG_BASE_ID) return AlertCountType ;
impure function GetEnabledAlertCount (AlertLogID : AlertLogIDType :=
    ALERTLOG_BASE_ID) return integer ;
. . .
TopTotalErrors  := GetEnabledAlertCount ;           -- Top of hierarchy
UartTotalErrors := GetEnabledAlertCount(UartID) ;   -- UartID
```

5.15 GetDisabledAlertCount

GetDisabledAlertCount returns the count of disabled errors for either the entire design hierarchy or a particular AlertLogID. GetDisabledAlertCount is relevant since a "clean" passing design will not have any disabled alert counts.

```
impure function GetDisabledAlertCount return AlertCountType ;
impure function GetDisabledAlertCount return integer ;
impure function GetDisabledAlertCount(AlertLogID: AlertLogIDType)
    return AlertCountType ;
impure function GetDisabledAlertCount(AlertLogID: AlertLogIDType) return integer ;
```

Note that disabled errors are not added to higher levels in the hierarchy. Hence, often GetAlertCount /= GetEnabledAlertCount + GetDisabledAlertCount.

5.16 ClearAlerts: Reset Alert and Stop Counts

ClearAlerts resets all alert counts to 0 and stop counts back to their default.

```
procedure ClearAlerts ;
```

5.17 Math on AlertCountType

```
function "+" (L, R : AlertCountType) return AlertCountType ;
function "-" (L, R : AlertCountType) return AlertCountType ;
function "-" (R : AlertCountType) return AlertCountType ;
. . .
TotalAlertCount := Phase1Count + Phase2Count ;
TotalErrors := GetAlertCount - ExpectedErrors ;
NegateErrors := -ExpectedErrors ;
```

5.18 SumAlertCount: AlertCountType to Integer Error Count

SumAlertCount sums up the WARNING, ERROR, and FAILURE values into a single integer value.

```
impure function SumAlertCount(AlertCount: AlertCountType) return integer ;
. . .
ErrorCountInt := SumAlertCount(AlertCount) ;
```

5.19 SetAlertLogJustify

SetAlertLogJustify justifies name fields of Alerts and Logs. Call after setting up the entire hierarchy if you want Alerts and Logs justified (hence optional).

```
SetAlertLogJustify ;
```

5.20 LogType

Log levels can be ALWAYS, DEBUG, FINAL, or INFO.

```
type LogType is (ALWAYS, DEBUG, FINAL, INFO) ;
```

5.21 Simple Logs

Simple logs use default AlertLogID. If the log level is enabled, then the log message will print.

```
procedure log( Message : string ; Level : LogType := ALWAYS) ;
. . .
Log("Received UART word", DEBUG) ;
```

5.22 Hierarchical Logs

Hierarchical logs use the specified AlertLogID. If the log level is enabled, then the log message will print.

```
procedure log(
  AlertLogID   : AlertLogIDType ;
  Message   : string ;
  Level     : LogType := ALWAYS
) ;
. . .
Log(UartID, "Uart Parity Received", DEBUG) ;
```

5.23 SetLogEnable: Enable / Disable Logging

SetLogEnable allows alert levels to be individually enabled. When used without AlertLogID, SetLogEnable sets a value for all AlertLogIDs.

```
procedure SetLogEnable(Level : LogType ;  Enable : boolean) ;
. . .
Log(UartID, "Uart Parity Received", DEBUG) ;
```

When an AlertLogID is used, SetLogEnable sets a value for that AlertLogID, and if Hierarchy is true, the AlertLogIDs of its children.

```
procedure SetLogEnable(AlertLogID : AlertLogIDType ;
    Level : LogType ;  Enable : boolean ; DescendHierarchy : boolean := TRUE) ;
. . .
--       AlertLogID,  Level,   Enable,  DescendHierarchy
SetLogEnable(UartID,  WARNING, FALSE,   FALSE) ;
```

5.24 IsLoggingEnabled

IsLoggingEnabled returns true when logging is enabled for a particular AlertLogID.

```
impure function IsLoggingEnabled(Level : LogType) return boolean ;
impure function IsLoggingEnabled(AlertLogID : AlertLogIDType ; Level : LogType)
    return boolean ;
. . .
If IsLoggingEnabled(UartID, DEBUG) then
. . .
```

5.25 OsvvmOptionsType

OsvvmOptionsType defines the values for options. User values are: OPT_DEFAULT, DISABLED, FALSE, ENABLED, TRUE. The values DISABLED and FALSE are handled the same. The values ENABLED and TRUE are treated the same. The value OPT_USE_DEFAULT causes the variable to use its default value. OsvvmOptionsType is defined in OsvvmGlobalPkg.

```
type OsvvmOptionsType is (OPT_INIT_PARM_DETECT, OPT_USE_DEFAULT, DISABLED, FALSE,
ENABLED, TRUE) ;
```

5.26 SetAlertLogOptions: Configuring Report Options

The output from Alert, Log, and ReportAlerts is configurable using SetAlertLogOptions.

```
procedure SetAlertLogOptions (
  FailOnWarning         : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  FailOnDisabledErrors  : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  ReportHierarchy       : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteAlertLevel       : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteAlertName        : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteAlertTime        : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteLogLevel         : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteLogName          : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  WriteLogTime          : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
  AlertPrefix           : string := OSVVM_STRING_INIT_PARM_DETECT;
  LogPrefix             : string := OSVVM_STRING_INIT_PARM_DETECT;
  ReportPrefix          : string := OSVVM_STRING_INIT_PARM_DETECT;
  DoneName              : string := OSVVM_STRING_INIT_PARM_DETECT;
  PassName              : string := OSVVM_STRING_INIT_PARM_DETECT;
  FailName              : string := OSVVM_STRING_INIT_PARM_DETECT
) ;
```

The following options are for ReportAlerts.

FailOnWarning	Count warnings as test errors.	Enabled
FailOnDisabledErrors	Disabled errors are test errors.	Enabled
ReportHierarchy	When multiple AlertLogIDs exist, print an error summary for each level.	Enabled
ReportPrefix	Prefix for each line of ReportAlerts.	"%% "
DoneName	Value printed after ReportPrefix on first line of ReportAlerts.	"DONE"
PassName	Value printed when a test passes.	"PASSED".
FailName	Value printed when a test fails.	"FAILED"

The following options are for alert:

WriteAlertLevel	Print level.	Enabled

WriteAlertName	Print AlertLogID name.	Enabled
WriteAlertTime	Alerts print time.	Enabled
AlertPrefix	Value printed at beginning of alert.	"%% Alert"

The following options are for Log:

WriteLogLevel	Print level.	Enabled
WriteLogName	Print AlertLogID name.	Enabled
WriteLogTime	Logs print time.	Enabled
LogPrefix	Value printed at beginning of log.	"%% Alert"

SetAlertOptions will change as AlertLogPkg evolves. Use of named association is required to ensure future compatibility.

```
SetAlertLogOptions (
    FailOnWarning            => FALSE,
    FailOnDisabledErrors     => FALSE
) ;
```

After setting a value, a string value can be reset using OSVVM_STRING_USE_DEFAULT and an OsvvmOptionsType value can be reset using OPT_USE_DEFAULT.

5.27 DeallocateAlertLogStruct

DeallocateAlertLogStruct deallocates all temporary storage allocated by AlertLogPkg. Also see ClearAlerts.

5.28 InitializeAlertLogStruct

InitializeAlertLogStruct is used after DeallocateAlertLogStruct to create and initialize internal storage.

6 Compiling AlertLogPkg and Friends

Use of AlertLogPkg requires use NamePkg and OsvvmGlobalPkg. The compile order is: NamePkg.vhd, OsvvmGlobalPkg.vhd, TranscriptPkg.vhd, and AlertLogPkg.vhd. Compiling the packages requires VHDL-2008.

7 Interfacing OSVVM to another Alert and Log Package

The only required subprograms of AlertLogPkg are Alert, Log, and IsLoggingEnabled. The following section describes how to edit the package body.

7.1 Alert

In the procedure body for AlertLogPkg, the implementation of Alert shown below needs to be replaced with code that calls the desired package.

```
procedure alert(
  AlertLogID   : AlertLogIDType ;
  Message      : string  ;
  Level        : AlertType := ERROR
) is
begin
  AlertLogStruct.Alert(AlertLogID, Message, Level) ;
end procedure alert ;
```

Change all other calls to AlertLogStruct.Alert to a call to Alert above.

7.2 Log

In the procedure body for AlertLogPkg, the implementation of Log shown below needs to be replaced with code that calls the desired package.

```
procedure log(
  AlertLogID   : AlertLogIDType ;
  Message      : string ;
  Level        : LogType := ALWAYS
) is
begin
  AlertLogStruct.Log(AlertLogID, Message, Level) ;
end procedure log ;
```

Change all other calls to AlertLogStruct.Log to a call to Log above.

7.3 IsLoggingEnabled

In the procedure body for AlertLogPkg, the implementation of IsLoggingEnabled shown below needs to be replaced with code that calls the desired package.

```
impure function IsLoggingEnabled(
    AlertLogID : AlertLogIDType ; Level : LogType
) return boolean is
begin
  return AlertLogStruct.IsLoggingEnabled(AlertLogID, Level) ;
end function IsLoggingEnabled ;
```

Change all other calls to AlertLogStruct.IsLoggingEnabled to a call to IsLoggingEnabled above.

7.4 Remove Protected Type and Shared Variable

Remove the protected type declaration and body for AlertLogStructPType. Remove the shared variable declaration AlertLogStruct.

8 About AlertLogPkg

AlertLogPkg was developed and is maintained by Jim Lewis of SynthWorks VHDL Training. It originated as an interface layer to the BitVis Utility Library (BVUL). However, it required a default implementation and that default implementation grew into its own project.

Please support our effort in supporting AlertLogPkg and OSVVM by purchasing your VHDL training from SynthWorks.

AlertLogPkg is released under the Perl Artistic open source license. It is free (both to download and use - there are no license fees). You can download it from http://www.synthworks.com/downloads. It will be updated from time to time. Currently there are numerous planned revisions.

If you add features to the package, please donate them back under the same license as candidates to be added to the standard version of the package. If you need features, be sure to contact us. I blog about the packages at http://www.synthworks.com/blog. We also support the OSVVM user community and blogs through http://www.osvvm.org.

Find any innovative usage for the package? Let us know, you can blog about it at osvvm.org.

9 Future Work

AlertLogPkg.vhd is a work in progress and will be updated from time to time.

Caution, undocumented items are experimental and may be removed in a future version.

10 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has twenty-eight years of design, teaching, and problem solving experience. In addition to working as a Principal Trainer for SynthWorks, Mr Lewis has done ASIC and FPGA design, custom model development, and consulting.

Mr. Lewis is chair of the IEEE 1076 VHDL Working Group (VASG) and is the primary developer of the Open Source VHDL Verification Methodology (OSVVM.org) packages. Neither of these activities generate revenue. Please support our volunteer efforts by buying your VHDL training from SynthWorks.

If you find bugs these packages or would like to request enhancements, you can reach me at jim@synthworks.com.