

AlertLog and Report Package

User Guide

User Guide for Release 2025.12

By

Jim Lewis

SynthWorks VHDL Training

Jim@SynthWorks.com

<http://www.SynthWorks.com>

Table of Contents

1	AlertLogPkg and ReportPkg Overview.....	5
2	Simple Mode: Errors and Messaging in a Basic Testbench.....	5
3	Hierarchy Mode: Model Based Error and Message Handling	7
4	Package References	10
5	Setting the Test Name and AlertLogIDs.....	10
5.1	SetTestName: Setting the Test Name.....	10
5.2	GetTestName	10
5.3	GetAlertLogName.....	10
5.4	AlertLogIDType	11
5.5	AlertLogReportModeType	11
5.6	AlertLogPrintParentType.....	11
5.7	AlertLogIDs	11
5.7.1	NewID: Creating a New ID	12
5.7.2	GetID: Find Existing ID or Create it if it does not exist.....	13
5.7.3	FindID: Find an AlertLogID	13
5.7.4	IsInitialized.....	13
5.8	PathTail - Used to Discover Instance Name of a Component.....	13
5.9	GetAlertLogParentID	14
6	Alert Method Reference	14
6.1	AlertType	14
6.2	Alert	14
6.3	AlertIf and AlertIfNot	14
6.4	AlertIfEqual and AlertIfNotEqual	15
6.5	AlertIfFilesNotMatch.....	15
6.6	IncrementAlertCount.....	16
6.7	SetAlertEnable: Alert Enable / Disable	16
6.8	GetAlertEnable.....	16
6.9	SetAlertStopCount: Alert Stop Counts.....	16
6.10	GetAlertStopCount	17
6.11	SetAlertPrintCount	17
6.12	GetAlertPrintCount	17
6.13	ClearAlertStopCounts: Reset Alert Stop Counts	18
6.14	ClearAlerts: Reset Alert Counts	18
6.15	ClearAlertCounts: Reset Alert and Alert Stop Counts	18
6.16	SetGlobalAlertEnable: Alert Global Enable / Disable.....	18
6.17	GetGlobalAlertEnable	18
7	Reporting and Getting Alert Counts	18
7.1	AlertCountType.....	18

7.2	Reporting Alerts: ReportAlerts, ReportNonZeroAlerts.....	19
7.2.1	Report Summary.....	19
7.2.2	Report Details.....	19
7.2.3	EndOfTestReports (in ReportPkg)	20
7.2.4	ReportAlerts: Reporting Alerts	21
7.2.5	ReportNonZeroAlerts.....	21
7.2.6	Overloaded ReportAlerts for Reporting AlertCounts	21
7.3	GetAlertCount.....	22
7.4	GetEnabledAlertCount	22
7.5	GetDisabledAlertCount	22
7.6	Math on AlertCountType.....	22
7.7	SumAlertCount: AlertCountType to Integer Error Count.....	23
8	Log Method Reference	23
8.1	LogType.....	23
8.2	Log	23
8.3	SetLogEnable: Enable / Disable Logging.....	23
8.4	Reading Log Enables from a FILE	24
8.5	IsLogEnabled /GetLogEnable	24
9	Alert and Log Prefix and Suffix to Message	24
10	Affirmation Reference.....	25
10.1	AffirmIf / AffirmIfNot	25
10.2	AffirmIfEqual and AffirmIfNotEqual	26
10.3	AffirmIfFilesMatch.....	26
10.4	AffirmIfTranscriptsMatch	27
10.5	GetAffirmCount.....	27
10.6	IncAffirmCount	27
11	Reporting Test Completion: HTML and Junit Reports.....	27
11.1	EndOfTestReports (in ReportPkg)	28
11.2	WriteAlertSummaryYaml (Called by EndOfTestReports).....	28
11.3	WriteAlertYaml (Called by EndOfTestReports).....	29
12	Requirements Tracking.....	30
12.1	NewReqID	30
12.2	GetReqID	30
12.3	ReadSpecification.....	31
12.4	SetPassedGoal	32
12.5	AffirmIf for Requirements.....	32
12.6	Using Other Forms of AffirmIF with Requirements	32
12.7	WriteRequirementsYaml (Called by EndOfTestReports)	33
12.8	Test Failure Due to Missed Requirements	33

12.9	ReportAlerts and Requirements	33
13	Alert and Log Output Control Options.....	35
13.1	SetAlertLogJustify	35
13.2	OsvvmOptionsType	35
13.3	SetAlertLogOptions: Configuring Report Options	35
13.4	ReportAlertLogOptions: Print Report Options	38
13.5	Getting AlertLog Report Options	38
14	Deallocating and Re-initializing the Data structure.....	39
14.1	DeallocateAlertLogStruct	39
14.2	InitializeAlertLogStruct	39
15	Deprecated: Older CSV Type Reporting	40
15.1	Deprecated Reporting Test Completion: CSV Reports	40
15.1.1	Deprecated: WriteAlerts	40
15.1.2	Deprecated: WriteTestSummary	41
15.1.3	Deprecated: ReadTestSummaries	41
15.1.4	Deprecated: ReportTestSummaries.....	41
15.1.5	Deprecated: WriteTestSummaries.....	42
15.1.6	Deprecated: Use Model	42
15.2	Deprecated: Requirements Reporting: CSV Reports	43
15.2.1	Deprecated: ReportRequirements	43
15.2.2	Deprecated: WriteRequirements	43
15.2.3	Deprecated: ReadRequirements.....	44
16	Deprecated Names and their Replacement name	44
17	About AlertLogPkg	44
18	Compiling and Using OSVVM Utility Library	44
19	About the Author - Jim Lewis.....	45

1 AlertLogPkg and ReportPkg Overview

Error reporting and messaging is an important part of any testbench.

Printing too many messages can significantly slow a test down. So for messaging, we need the ability to turn messages on (such as for debug or the final test report) and turn them off.

For error reporting, we need to be able to report messages as they happen as well as produce a summary report when a test finishes. Hence, we need a mechanism to record the number of errors that have happened.

VHDL assert and TextIO capability can certainly do some of this, however, these take work and lack in capability (until VHDL-2019 where we gave them a simple OSVVM like interface).

When a test completes, we need comprehensive error reporting. This is handled by EndOfTestReports in ReportPkg.

2 Simple Mode: Errors and Messaging in a Basic Testbench

First include OSVVM library in your testbench. You can do this by referencing the context declaration that includes all of OSVVM's packages:

```
Library osvvm;  
use osvvm.OsvvmContext;
```

Name the test by calling SetTestName. This name prints as part of the summary report that is done either at the end of the test or if the test reaches a stop condition (such as a FAILURE – more on this later).

```
SetTestName("Axi Li te_Uart1");
```

Messages in OSVVM are called logs. The following indicates the start of a UART transmit transaction:

```
Log("Uart Sending Data: X" & to_hstring(TxData), ALWAYS);
```

This will print as:

```
%% Log  ALWAYS  Uart Sending Data: X4A  at 15110 ns
```

Logs have levels ALWAYS, DEBUG, FINAL, INFO, and PASSED. Log level ALWAYS always prints. The remaining logs print when they are enabled and are disabled by default. ALWAYS is the default if a level is not specified.

Enable a log with SetLogEnable.

```
SetLogEnable(INFO, TRUE);
```

Now that we have INFO messages enabled, let's rewrite our original log to use INFO as we may wish to turn it off when we are running regressions.

```
Log("Uart Sending Data: X" & to_hstring(TxData), INFO) ;
```

This will print as:

```
%% Log INFO Uart Sending Data: X4A at 15110 ns
```

The same method is used to turn DEBUG or any other message on or off. Generally it is recommended to call SetLogEnable from the top level testbench, however, it can be called from anywhere in the testbench infrastructure.

Alerts are for signaling error conditions. The following is a protocol checker that flags an error if the X86Lite model receives an acknowledge when the interface is idle.

```
X86_Rdy_Check: process
begin
    wait until Rising_Edge(CLK) and nRdy = '1' ;
    AlertIf(Busy, "X86 Lite received ready when it was idle", ERROR) ;
end process ;
```

This will print as:

```
%% Alert ERROR X86 Lite received ready when it was idle at 15110 ns
```

Alerts have levels FAILURE, ERROR, and WARNING. Error is the default if a level is not specified. Alerts are enabled by default. Alerts are disabled using SetAlertEnable.

```
SetAlertEnable(WARNING, FALSE) ; -- Test still fails, see Disabling Alerts
```

Affirmations are for adding checking to tests. When an affirmation passes, it calls log with PASSED and when it fails it calls Alert with ERROR.

```
AffirmIf(RxData = Expected, "RxData, Actual: " & to_hstring(RxData),
          "Expected: " & to_hstring(Expected)) ;
```

Note there are two strings associated with AffirmIf. The first is for the received value from the DUT (actual) and the second is for the expected value. When AffirmIf passes and log level "PASSED" is enabled, the following message will print.

```
%% Log PASSED RxData, Actual: X4A at 15110 ns
```

When AffirmIf is FALSE, the following message will print.

```
%% Alert ERROR RxData, Actual: 52 Expected: 4A at 15110 ns
```

As Alerts (FAILURE, ERROR, WARNING), PASSED, and affirmations occur, a data structure inside the AlertLogPkg tabulates them.

When the test is done, a text summary message can be created by calling ReportAlerts.

```
ReportAlerts ;
```

Alternately complete test and build reports (YAML, HTML, and JUnit XML), can be created by using OSVVM scripts build, simulate, and then calling EndOfTestReports:

```
EndOfTestReports ;
```

If the test passed, a message such as the following prints.

```
%% DONE PASSED Axi Lite_Uart1 Passed: 14 Affirmations Checked: 14 at 380810 ns
```

If the test failed, a message such as the following prints. The line starts with "%%" and prints on a single line (here shown with line wrap to fit in the text).

```
%% DONE FAILED Axi Lite_Uart1 Total Error(s) = 6 Failures: 0 Errors: 6
Warnings: 0 Passed: 8 Affirmations Checked: 14 at 380810 ns
```

This is enough to get you started with basic testbenches. Read on when you are ready for more capability. Even for basic testbenches, you will want to learn about the overloading of Alerts and Affirmations as well as alert stop counts.

3 Hierarchy Mode: Model Based Error and Message Handling

As testbench complexity increases, it helps to know which model a particular error came from. This is what hierarchy mode is all about. We also get a better summary report from ReportAlerts and EndOfTestReports.

First include OSVVM library in your testbench. You can do this by referencing the context declaration that includes all of OSVVM's packages:

```
Library osvvm;
use osvvm.OsvvmContext;
```

Name the test by calling SetTestName. This name prints as part of the summary report that is done either at the end of the test or if the test reaches a stop condition (such as a FAILURE – more on this later).

```
SetTestName("Axi Lite_Uart1");
```

In hierarchy mode, each source (generally each verification model) has its own AlertLogID. Create one by declaring it.

```
signal ModelID : AlertLogIDType;
```

Next get the AlertLogID by calling NewID

```
Initialize : process
begin
    ModelID <= NewID("Axi Manager");
    wait;
end process Initialize;
```

It would be nice to be able to do this in one step in a declaration such as the following, and indeed all simulators it has been tested on it works, but formally the VHDL LRM says one is not allowed to access a protected type in a declarative region.

```
signal ModelID : AlertLogIDType := NewID("Axi Manager");
```

Sometimes a model may wish to further classify its Alerts and Logs into subclasses. This is done as follows.

```
signal ModelID, ProtocolID, CheckerID : AlertLogIDType ;
.

Initialize : process
  Variable ID : AlertLogID ;
begin
  ID := NewID("Axi Manager") ;
  ModelID <= ID ;
  ProtocolID := NewID("Protocol", ID) ;
  CheckerID := NewID("Checker", ID) ;
  wait ;
end process Initialize ;
```

Ok. The hard part is done. On all calls to Log, SetLogEnable, AlertIf, AffirmIf, SetAlertEnable, ... the AlertLogID is specified as the first parameter to the call.

Alerts and Logs can be enabled (or disabled) either for the entire testbench or for a single model. However to do this in the testbench, we need to lookup the AlertLogID for the model. No problem. We can use NewID to look up the AlertLogID by using the same string used in its definition in the component.

```
signal TBID, AxiManagerID, UartTxID, UartRxID : AlertLogIDType ;
.

Control : process
begin
  TBID      <= NewID("TB") ;
  AxiManagerID <= NewID("Axi Manager") ; -- These names must match
  UartTxID    <= NewID("UartTx") ;       -- the names used in the component
  UartRxID    <= NewID("UartRx") ;
  .
  SetLogEnable(PASSED, TRUE) ;           -- Turn on for ALL Levels
  SetLogEnable(UartTxID, INFO, TRUE) ;   -- Turn on just UartTxID
  wait ;
end process Control ;
```

Above first we turned on PASSED for all AlertLogIDs, but only turned on INFO for the UartTx model. If you have worked with UARTs you will understand that this is an important capability in tracking what is happening in the UART. If we cannot do this, we have to hunt for the infrequent message from the UART mixed in with the multitude of messages from the AxiManager.

Adding log in the UartTx model.

```
Log(ModelID, "Uart Sending Data: X" & to_hstring(TxData), INFO) ;
```

This will print as the following. Note the context information, "In UartTx".

```
%% Log INFO In UartTx, Uart Sending Data: X4A at 15110 ns
```

Updating the protocol checker in the X86Lite model.

```
X86_Rdy_Check: process
begin
    wait until Rising_Edge(Clk) and nRdy = '1';
    AlertIf(ProtocolID, BusID, "X86 Lite received ready when it was idle");
end process;
```

This will print as:

```
%% Alert ERROR In X86Lite, X86 Lite received ready when it was idle at 15110 ns
```

Updating the affirmation using the equivalent form of AffirmIfEqual and doing it from the testbench.

```
AffirmIfEqual(TBID, RxData, Expected, "RxData, ");
```

When AffirmIfEqual passes and log level "PASSED" is enabled, the following message will print.

```
%% Log PASSED In TB, RxData, Actual: X4A at 15110 ns
```

When AffirmIfEqual fails, the following message will print.

```
%% Alert ERROR In TB, RxData, Actual: 52 Expected: 4A at 15110 ns
```

When the test is done, a summary message is created by calling ReportAlerts.

```
ReportAlerts;
```

If the test passed, we get the same simple message as for Simple Mode.

```
%% DONE PASSED AxiLite_Uart1 Passed: 14 Affirmations Checked: 14 at 380810 ns
```

If the test failed, we get a summary as well as details for each source. The line starts with "%%" and prints on a single line (here shown with line wrap to fit in the text).

```
%% DONE FAILED AxiLite_Uart1 Total Error(s) = 7 Failures: 0 Errors: 7
Warnings: 0 Passed: 8 Affirmations Checked: 14 at 380810 ns
%% Default Failures: 0 Errors: 0 Warnings: 0 Passed: 0
%% OSVVM Failures: 0 Errors: 0 Warnings: 0 Passed: 0
%% TB Failures: 0 Errors: 2 Warnings: 0 Passed: 2
%% Axi Manager Failures: 0 Errors: 3 Warnings: 0 Passed: 0
%% Protocol Failures: 0 Errors: 1 Warnings: 0 Passed: 0
%% Checker Failures: 0 Errors: 2 Warnings: 0 Passed: 2
%% UartTx Failures: 0 Errors: 0 Warnings: 0 Passed: 2
%% UartRx Failures: 0 Errors: 2 Warnings: 0 Passed: 2
```

Note errors in the Default bin would be the result of calling an alert or affirmation without an AlertLogID.

4 Package References

To keep it simple use the OSVVM context clause:

```
library osvvm;
context osvvm.OsvvmContext;
```

Alternately to just use AlertLogPkg use the following package references:

```
library osvvm;
use osvvm.OsvvmGlobalPkg.all; -- required to call SetAlertLogOptions
use osvvm.AlertLogPkg.all;
```

5 Setting the Test Name and AlertLogIDs

Tests always start in simple mode. Hierarchical mode is automatically entered when NewID is called.

All overloading supported for simple mode is also supported in hierarchical mode. The only difference is that in hierarchical mode, the first parameter is the AlertLogID.

Also note that when in hierarchical mode, any alert, log, or affirmation that is called without an AlertLogID will use the "Default" bin.

5.1 SetTestName: Setting the Test Name

SetTestName sets the name of the current test that is printed by ReportAlerts. It is recommended to set the test name at time 0. If it is not set, the default name is AlertLogTop.

```
procedure SetTestName(Name : string);
begin
  SetTestName("Axi Lite_Uart1");
```

5.2 GetTestName

GetTestName returns the string value of name associated with a Test. If no AlertLogID is specified, it will return the name set by SetTestName.

```
impure function GetTestName return string;
```

5.3 GetAlertLogName

GetAlertLogName returns the string value of name associated with an AlertLogID. If no AlertLogID is specified, it will return the name set by SetTestName.

```
impure function GetAlertLogName(AlertLogID : AlertLogIDType := ALERTLOG_BASE_ID)
return string;
```

5.4 AlertLogIDType

AlertLogIDType is defined as a subtype of integer. Using a subtype here works similar to an alias and allows it to be changed as needed.

```
subtype AlertLogIDType is integer ;
```

5.5 AlertLogReportModeType

```
type AlertLogReportModeType is (DISABLED, ENABLED, NONZERO) ;
```

When DISABLED, an AlertLogID does not print in ReportAlerts (text reports) or WriteAlertYaml (Yaml/HTML reports). When NONZERO, an AlertLogID does not print in ReportAlerts, but prints in WriteAlertYaml. Hence, NONZERO shortens the text reports to what is essential. When ENABLED, an AlertLogID prints in ReportAlerts and WriteAlertYaml.

None of these settings impact printing in Alert or Log.

DISABLED is intended to be used for data structures that are not used for self-checking and may produce errors on parameter checks. These errors are not lost as they are still accumulated in the parent ID.

5.6 AlertLogPrintParentType

```
type AlertLogPrintParentType is (PRINT_NAME, PRINT_NAME_AND_PARENT) ;
```

When PRINT_NAME, Alert and Log print the name of the AlertLogID. When PRINT_NAME_AND_PARENT, Alert and Log print the parent ID name, the ID separator, and then the ID name.

5.7 AlertLogIDs

Each level in a hierarchy is referenced with an AlertLogID. There are five functions used to create or find AlertLogIDs: NewID, GetID, FindID, NewReqID, and GetReqID.

Use NewID when the intent is to create the ID. Use GetID when the intent is to look up the ID – but if it does not exist yet then create it. Use FindID to look up an ID that exists and if it does not, then fail. Use NewReqID when the intent is to create a new requirements ID. Use GetReqID when the intent is to look up the requirements ID – but if it does not exist yet then create it.

The main difference between the NewID and GetID is that NewID uses ALERTLOG_BASE_ID as the default parent ID where as GetID uses ALERTLOG_ID_NOT_ASSIGNED. When the parent ID is ALERTLOG_ID_NOT_ASSIGNED,

then the next call to NewID will establish the parent ID. Note though once the parent ID is assigned, it cannot be changed.

NewID is typically called in a verification component. GetID is typically called in the testbench to look up the ID of a verification component. When using GetID it is typical to call it well after time 0, so as to allow NewID to be called first. If it is indeed well after time 0, then calling FindID will produce the same results as GetID.

5.7.1 NewID: Creating a New ID

Use NewID when the intent is to create the ID, such as in a verification component. If an AlertLogID already exists for the specified name, NewID will return its AlertLogID. It is recommended to use the instance label of a verification component as the Name. When creating an AlertLogID in a verification component, it is recommended to call NewID at time 0.

The interface for NewID is as follows.

```
impure function NewID(
    Name          : string ;
    ParentID      : AlertLogIDType      := ALERTLOG_BASE_ID ;
    ReportMode    : AlertLogReportModeType := ENABLED ;
    PrintParent   : AlertLogPrintParentType := PRINT_NAME_AND_PARENT ;
    CreateHierarchy: boolean           := TRUE
) return AlertLogIDType ;
```

ParentID allows a hierarchy of Alerts to be created with parent and child IDs. Within a verification component it is common to create an ID for the verification component and then create Child IDs that are related to the parent.

```
signal ModelID, ProtocolID, CheckerID : AlertLogIDType ;
.

Initialise : process
    Variable ID : AlertLogID ;
begin
    ID := NewID("Axi Manager") ;
    ModelID <= ID ;
    ProtocolID := NewID("Protocol ", ID) ;
    CheckerID := NewID("Checker", ID) ;
    wait ;
end process Initialise ;
```

The ReportMode and PrintParent parameters are described in AlertLogReportModeType and AlertLogPrintParentType.

Setting the CreateHierarchy parameter to FALSE allows packages to create a unique AlertLogID without triggering ReportAlerts to print the hierarchy of the structure – unless the testbench or a verification component also calls NewID. OSVVM data structures set CreateHierarchy to FALSE.

5.7.2 GetID: Find Existing ID or Create it if it does not exist

Use GetID when the intent is to look up the ID – but if it does not exist yet then create it. Typically GetID is called in the testbench, well after time 0, to look up the ID of a verification component.

The interface for GetID is as follows:

```
impure function GetID(
    Name : string ;
    ParentID : AlertLogIDType := ALERTLOG_ID_NOT_ASSIGNED ;
    CreateHierarchy : Boolean := TRUE
) return AlertLogIDType ;
```

Internally GetID calls NewID:

5.7.3 FindID: Find an AlertLogID

The function, FindID, finds an existing AlertLogID. If the AlertLogID is not found, ALERTLOG_ID_NOT_FOUND is returned. Caution do not use at time 0, as FindID may return ALERTLOG_ID_NOT_FOUND for an ID that has not been initialized by the model yet. Use GetAlertID for this situation.

The interface for FindID is as follows.

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

FindID with a single parameter will return the first AlertLogID whose name matches the Name parameter, hence, only use this if that name is used once in the AlertLog data structure, such as for top-level instance names.

5.7.4 IsInitialized

IsInitialized returns an indication whether an AlertLogID is initialized or not.

```
impure function IsInitialized (ID : AlertLogIDType) return Boolean ;
```

5.8 PathTail - Used to Discover Instance Name of a Component

When used in conjunction with attribute PATH_NAME applied to an entity name, PathTail returns the instance name of component plus any for generate loop indices. For generate loop indices are added as "_#".

```
constant CPU_ALERT_ID : AlertLogIDType :=
    -- Name (string value),           Parent AlertLogID
    NewID(PathTail(CpuModel'PATH_NAME), ALERTLOG_BASE_ID) ;
```

5.9 GetAlertLogParentID

Get the AlertLogID of the parent of a specified AlertLogID.

```
impure function GetAlertLogParentID(AlertLogID : AlertLogIDType) return  
AlertLogIDType ;
```

6 Alert Method Reference

Alert is intended for parameter error checking. For self-checking see affirmations.

6.1 AlertType

Alert levels can be FAILURE, ERROR, or WARNING.

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

6.2 Alert

Alert generates an alert. The following overloading is supported.

```
procedure alert(  
    AlertLogID  : AlertLogIDType ;  
    Message     : string ;  
    Level       : AlertType := ERROR  
) ;  
  
procedure Alert( Message : string ; Level : AlertType := ERROR ) ;
```

Usage of alert:

```
...  
Alert(UartID, "Uart Parity", ERROR) ;  
  
Alert("Uart Parity") ; -- ERROR by default
```

6.3 AlertIf and AlertIfNot

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

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

```
-- with an AlertLogID
AlertIf(UartID, Break='1', "Uart Break", ERROR) ;
AlertIfNot(UartID, ReadVal id, "Read", FAILURE);

-- without an AlertLogID
AlertIf(Break='1', "Uart Break", ERROR) ;
AlertIfNot(ReadVal id, "Read Failed", FAILURE) ;
```

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

```
exit AlertIfNot(UartID, ReadVal id, "in ReadCovDb while reading ...", FAILURE) ;
```

6.4 AlertIfEqual and AlertIfNotEqual

AlertIfEqual and AlertIfNotEqual checks two values and produces an Alert if the check fails. In the following, AType can be std_logic, std_logic_vector, unsigned, signed, integer, integer_vector, real, character, string, or time.

```
-- with an AlertLogID
procedure AlertIfEqual ( AlertLogID : AlertLogIDType ; L, R : AType ;
    Message : string ; Level : AlertType := ERROR ) ;
procedure AlertIfNotEqual ( AlertLogID : AlertLogIDType ; L, R : AType ;
    Message : string ; Level : AlertType := ERROR ) ;

-- without an AlertLogID
procedure AlertIfEqual ( L, R : AType ; Message : string ;
    Level : AlertType := ERROR ) ;    Message : string ;
procedure AlertIfNotEqual ( L, R : AType ;
    Level : AlertType := ERROR ) ;
```

6.5 AlertIfFilesNotMatch

Alert form AlertIfFilesNotMatch is for comparing two files.

```
-- with an AlertLogID
procedure AlertIfFilesNotMatch ( AlertLogID : AlertLogIDType ;
    Name1, Name2 : string;    Message : string := "" ; Level : AlertType := ERROR ;
    IgnoreSpaces : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
procedure AlertIfFilesNotMatch ( AlertLogID : AlertLogIDType ;
    file File1, File2 : text; Message : string := "" ; Level : AlertType := ERROR ;
    IgnoreSpaces : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;

-- without an AlertLogID
procedure AlertIfFilesNotMatch (
    Name1, Name2 : string;    Message : string := "" ; Level : AlertType := ERROR ;
    IgnoreSpaces : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
procedure AlertIfFilesNotMatch (
    file File1, File2 : text; Message : string := "" ; Level : AlertType := ERROR ;
    IgnoreSpaces : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
```

6.6 IncrementAlertCount

Intended as a silent alert. Used by CoveragePkg.

```
-- Hierarchy
procedure IncAlertCount( -- A silent form of alert
    AlertLogID : AlertLogIDType ;
    Level      : AlertType := ERROR
) ;
-- Global Alert Counters
procedure IncAlertCount( Level : AlertType := ERROR ) ;
```

6.7 SetAlertEnable: Alert Enable / Disable

Alerts are enabled by default. SetAlertEnable allows alert levels to be individually enabled or disabled. 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) ;
```

6.8 GetAlertEnable

Get the value of the current alert enable for either a specific AlertLogId or for the global alert counter.

```
-- Hierarchy
impure function GetAlertEnable(AlertLogID : AlertLogIDType ; Level : AlertType)
return boolean ;
-- Global Alert Counter
impure function GetAlertEnable(Level : AlertType) return boolean ;
```

6.9 SetAlertStopCount: Alert Stop Counts

For tests that complete with numerous errors, perhaps it is better to stop after some small number of errors have occurred and start debugging. That is the job of 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'reight, 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.

6.10 GetAlertStopCount

Get the value of the current alert stop count for either a specific AlertLogId or for the global alert counter.

```
-- Hierarchy
impure function GetAlertStopCount(
    AlertLogID : AlertLogIDType ;
    Level : AlertType
) return integer ;
-- Global Alert Stop Count
impure function GetAlertStopCount(Level : AlertType) return integer ;
```

6.11 SetAlertPrintCount

Sets a maximum number of times to print an Alert level for an AlertLogID. The AlertPrintCount is specified either as an integer value for a particular level or as an AlertCountType value that specifies a count for each Alert level. Applies only to that ID, hence, if a child ID does not exceed its count, it prints, independent of the AlertPrintCount of its parent. When an ID is not specified, the ID of the default bin is used (ALERTLOG_DEFAULT_ID).

```
procedure SetAlertPrintCount(
    AlertLogID : AlertLogIDType ; Level : AlertType ; Count : integer) ;
procedure SetAlertPrintCount(Level : AlertType ; Count : integer) ;
procedure SetAlertPrintCount(
    AlertLogID : AlertLogIDType ; Count : AlertCountType) ;
procedure SetAlertPrintCount(Count : AlertCountType) ;
```

6.12 GetAlertPrintCount

Returns the AlertPrintCount for an ID – either as an integer value for a particular level or as an AlertCountType value that contains a count for each Alert level. When an ID is not specified, the ID of the default bin is used (ALERTLOG_DEFAULT_ID).

```
impure function GetAlertPrintCount(AlertLogID : AlertLogIDType ; Level :
AlertType) return integer ;
impure function GetAlertPrintCount(Level : AlertType) return integer ;
impure function GetAlertPrintCount(AlertLogID : AlertLogIDType) return
AlertCountType ;
```

```
impure function GetAlertPrintCount return AlertCountType ;
```

6.13 ClearAlertStopCounts: Reset Alert Stop Counts

ClearAlerts resets alert stop counts back to their default.

```
procedure ClearAlertStopCounts ;
```

6.14 ClearAlerts: Reset Alert Counts

ClearAlerts resets alert counts to 0. Recommended as an alternative to SetGlobalAlertEnable – just after the design enters reset, call ClearAlerts.

```
procedure ClearAlerts ;
```

6.15 ClearAlertCounts: Reset Alert and Alert Stop Counts

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

```
procedure ClearAlertCounts ;
```

6.16 SetGlobalAlertEnable: Alert Global Enable / Disable

SetGlobalAlertEnable allows Alerts to be globally enabled and disabled. The Global Alert Enable is enabled by default.

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

Originally intended to be used to disable alerts during startup, as shown below. Instead, for most applications we recommend using ClearAlerts just after reset.

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

6.17 GetGlobalAlertEnable

Get the current value of the global alert enable

```
impure function GetGlobalAlertEnable return boolean ;
```

7 Reporting and Getting Alert Counts

7.1 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.

```
variable ExternalErrors : AlertCountType ;
```

```
External Errors := (FAILURE => 0, ERROR => 6, WARNING => 0) ;
```

7.2 Reporting Alerts: ReportAlerts, ReportNonZeroAlerts

7.2.1 Report Summary

The report summary is the first line printed by ReportAlerts/ReportNonZeroAlerts

```
%% DONE FAILED t17_print_ctrl Total Error(s) = 2 Failures: 0 Errors: 0
Warnings: 0 Total Disabled Error(s) = 2 Failures: 0 Errors: 2 Warnings: 0
Passed: 0 Affirmations Checked: 2 at 20160 ns
```

Note in the above message, two affirmations were done and they both resulted in an error, however, the error was disabled (see SetAlertEnable) so it reported as a Disabled Error.

Options can be enabled or disabled using SetAlertLogOptions. Let's look at the report piece by piece.

If Total Errors /= 0, then the following part of the message prints. In this case, Total Errors is nonzero since FailedOnDisabledErrors is enabled (default) and the disabled errors count as test errors. Hence, Failures, Errors, and Warnings are all 0.

```
Total Error(s) = 2 Failures: 0 Errors: 0 Warnings: 0
```

If FailedOnDisabledErrors is disabled, then the same test will PASS as shown below.

```
%% DONE PASSED t17_print_ctrl Total Disabled Error(s) = 2 Passed: 0
Affirmations Checked: 2 at 20160 ns
```

Next is the Total Disabled Errors summary. It will print if the DisabledAlerts /= 0 or if PrintDisabledAlerts is enabled (default is disabled).

```
Total Disabled Error(s) = 2
```

Next is the details of the Disabled Errors. These will print if FailedOnDisabledErrors is enabled and DisabledAlerts /= 0 or if PrintDisabledAlerts is enabled.

```
Total Disabled Error(s) = 2 Failures: 0 Errors: 2 Warnings: 0
```

Next is the printing of the pass and affirmation counts. If PrintPassed is enabled (default), the passed count will be printed. If PrintAffirmations is enabled (default is disabled) or Affirmation Count /= 0 then the passed and affirmation counts will be printed.

```
Passed: 0 Affirmations Checked: 2
```

7.2.2 Report Details

When in hierarchy mode, a summary for each AlertLogID will be printed. This can include the following details. Each line summary starts with "%%".

```
%% Default Failures: 0 Errors: 2 Warnings: 0 Disabled Failures: 0 Errors:
0 Warnings: 0 Passed: 2 Affirmations: 4
%% OSVVM Failures: 0 Errors: 0 Warnings: 0 Disabled Failures: 0 Errors:
0 Warnings: 0 Passed: 0 Affirmations: 0
```

```

%%      Cpu1      Failures: 0  Errors: 2  Warnings: 0  Disabled Failures: 0  Errors:
0  Warnings: 0  Passed: 2  Affirmations: 4
%%      UartTx     Failures: 0  Errors: 0  Warnings: 0  Disabled Failures: 0  Errors:
2  Warnings: 0  Passed: 2  Affirmations: 4
%%      UartRx     Failures: 0  Errors: 2  Warnings: 0  Disabled Failures: 0  Errors:
0  Warnings: 0  Passed: 2  Affirmations: 4

```

The Failures, Errors, and Warnings always prints.

Disabled Failures, Errors, and Warnings only print if PrintDisabledAlerts is enabled (default is disabled).

Passed only prints if PrintPassed is enabled (default).

Affirmations only prints if PrintAffirmations is enabled (default is disabled).

With PrintDisabledAlerts disabled (default), PrintPassed enabled (default), and PrintAffirmations disabled (default), and DisabledAlerts = 0, the report will be simplified to the following.

```

% Default   Failures: 0  Errors: 2  Warnings: 0  Passed: 2
% OSVVM     Failures: 0  Errors: 0  Warnings: 0  Passed: 0
% Cpu1      Failures: 0  Errors: 2  Warnings: 0  Passed: 2
% UartTx    Failures: 0  Errors: 0  Warnings: 0  Passed: 2
% UartRx    Failures: 0  Errors: 2  Warnings: 0  Passed: 2

```

7.2.3 EndOfTestReports (in ReportPkg)

At test completion, call EndOfTestReports to create

- a test completion message via ReportAlerts,
- a test summary report via WriteAlertSummaryYaml (see section 11),
- a detailed alert report via WriteAlertYaml (see section 11), and
- a detailed coverage report via WriteCovYaml (see CoveragePkg User Guide).
- a detailed requirements (specification tracking) report via WriteRequirementsYaml
- a detailed Scoreboard report via WriteScoreboardYaml (see ScoreboardGenericPkg User Guide)

EndOfTestReports is part of OSVVM's reporting and test tracking capability. For more details on the use model for EndOfTestReports see section 11 [Reporting and Tracking Test Completion](#).

The call interface to EndOfTestReports is:

```

procedure EndOfTestReports (
  ReportAll      : boolean          := FALSE ;
  External Errors : AlertCountType := (0, 0, 0) ;
  Stop           : boolean          := FALSE ;
  Timeout        : boolean          := FALSE
);

```

The ReportAll and ExternalErrors parameters are passed to ReportAlerts. If the Stop parameter is true, the procedure std.env.stop is called with the error count.

7.2.4 ReportAlerts: Reporting Alerts

At test completion call ReportAlerts to generate a simple text based report. If you call EndOfTestReports, it calls ReportAlerts, so you don't have to.

```
procedure ReportAlerts (
    Name          : string := "" ;
    AlertLogID    : AlertLogIDType := ALERTLOG_BASE_ID ;
    External Errors : AlertCountType := (others => 0) ;
    ReportAll     : Boolean        := FALSE
) ;
```

ReportAlerts has 4 optional parameters: Name, AlertLogID, ExternalErrors, and ReportAll. Name overrides the name specified by SetTestName. 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 (FAILURE => 0, ERROR => 5, WARNING => 1) indicates detection logic separate from AlertLogPkg saw 0 Failures, 5 Errors, and 1 Warning. See notes under AlertCountType. When ReportAll is true, ReportAlerts reports a summary as well as details for each source – whether the test failed or passed.

```
--           Name,      AlertLogID,   External Errors
ReportAlerts("Uart1",   Uart1ID,       (FAILURE => 0,  ERROR => 5,  WARNING => 1) ) ;
```

7.2.5 ReportNonZeroAlerts

When in hierarchy mode, a summary for each AlertLogID will be printed. This can include the following details. Each summary starts with "%%".

Within the hierarchy, if a level has no alerts set, then that level will not be printed.

```
procedure ReportNonZeroAlerts (
    Name          : string := OSVVM_STRING_INIT_PARM_DETECT ;
    AlertLogID    : AlertLogIDType := ALERTLOG_BASE_ID ;
    External Errors : AlertCountType := (others => 0)
) ;
```

7.2.6 Overloaded ReportAlerts for Reporting AlertCounts

ReportAlerts can also be used to print a passed/failed message for an AlertCount that is passed into the procedure call. This will not use any of the internal settings or information.

```
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) ;
```

7.3 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 ;
.
.
.
TopTotal Errors := GetAlertCount ;           -- AlertCount for Top of hierarchy
UartTotal Errors := GetAlertCount(UartID) ;   -- AlertCount for UartID
```

7.4 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 ;
.
.
.
TopTotal Errors := GetEnabledAlertCount ;      -- Top of hierarchy
UartTotal Errors := GetEnabledAlertCount(UartID) ; -- UartID
```

7.5 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 $\text{GetAlertCount} = \text{GetEnabledAlertCount} + \text{GetDisabledAlertCount}$.

7.6 Math on AlertCountType

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

7.7 SumAlertCount: AlertCountType to Integer Error Count

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

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

8 Log Method Reference

8.1 LogType

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

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

8.2 Log

If the log level is enabled, then the log message will print. The Enable parameter indicates to print the message even if the log level is not enabled. This is useful to enable printing for a single transaction.

```
-- with an AlertLogID  
procedure Log(  
    AlertLogID : AlertLogIDType;  
    Message    : string;  
    Level      : LogType := ALWAYS; -- Log if LogType is enabled  
    Enable     : boolean := FALSE; -- also log if Enable is TRUE  
);  
  
-- without an AlertLogID  
procedure Log(  
    Message    : string;  
    Level      : LogType := ALWAYS; -- Log if LogType is enabled  
    Enable     : boolean := FALSE; -- also log if Enable is TRUE  
);
```

Usage:

```
Log(UartID, "Uart Parity Received", DEBUG);  
.  
Log("Received UART word", DEBUG);
```

8.3 SetLogEnable: Enable / Disable Logging

Excepting ALWAYS, log enables are disabled by default. 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);  
.  
SetLogEnable(PASSED, TRUE);
```

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, INFO, TRUE, TRUE) ;
```

8.4 Reading Log Enables from a FILE

ReadLogEnables read enables from a file.

```
procedure ReadLogEnables (FileName : string) ;
procedure ReadLogEnables (file AlertLogInitFile : text) ;
```

The preferred file format is:

```
U_CpuModel DEBUG
U_UART_TX DEBUG INFO
U_UART_RX PASSED INFO DEBUG
```

ReadLogEnables will also read a file of the format:

```
U_CpuModel
DEBUG
U_UART_TX
DEBUG
U_UART_TX
INFO
.
```

8.5 IsLogEnabled /GetLogEnable

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

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

GetLogEnable is a synonym for IsLogEnabled.

```
impure function GetLogEnable(AlertLogID : AlertLogIDType ; Level : LogType)
return boolean ;
impure function GetLogEnable(Level : LogType) return boolean ;
```

9 Alert and Log Prefix and Suffix to Message

A prefix and suffix can be added message that is printed with Alert or Log. These are set on an AlertLogID basis using the following operations.

```
procedure SetAlertLogPrefix(AlertLogID : AlertLogIDType; Name : string) ;
procedure UnSetAlertLogPrefix(AlertLogID : AlertLogIDType) ;
impure function GetAlertLogPrefix(AlertLogID : AlertLogIDType) return string ;
```

```

procedure SetAlertLogSuffix(AlertLogID : AlertLogIDType; Name : string) ;
procedure UnSetAlertLogSuffix(AlertLogID : AlertLogIDType) ;
impure function GetAlertLogSuffix(AlertLogID : AlertLogIDType) return string ;

```

10 Affirmation Reference

Affirmations are a combination of Alerts and Logs. If the Affirmation is true, then a log is generated. If an Affirmation is false, then an alert is generated.

Affirmations are intended to be used for self-checking of a test. Each call to AffirmIf is counted and reported during ReportAlerts. This provides feedback on the amount of self-checking added by a test and is used as a quality metric.

10.1 AffirmIf / AffirmIfNot

AffirmIF has two forms of overloading. The first has a separate ReceivedMessage and ExpectedMessage. When the affirmation passes, log is called with just the Received Message. When an affirmation fails, alert is called with the ExpectedMessage concatenated to the end of the ReceivedMessage.

```

-- with an AlertLogID
procedure AffirmIf(
    AlertLogID      : AlertLogIDType ;
    condition       : boolean ;
    ReceivedMessage : string ;
    ExpectedMessage : string ;
    Enable          : boolean := FALSE
) ;

-- without an AlertLogID
procedure AffirmIf(
    condition       : boolean ;
    ReceivedMessage : string ;
    ExpectedMessage : string ;
    Enable          : boolean := FALSE
) ;

```

The second overloading of AffirmIF has a single Message parameter. Hence, both alert and log print the same message.

```

-- with an AlertLogID
procedure AffirmIf(
    AlertLogID      : AlertLogIDType ;
    condition       : boolean ;
    Message         : string ;
    Enable          : boolean := FALSE
) ;

-- without an AlertLogID
procedure AffirmIf(
    condition       : boolean ;
    Message         : string ;
    Enable          : boolean := FALSE
) ;

```

AffirmIfNot supports the same parameters as AffirmIf.

10.2 AffirmIfEqual and AffirmIfNotEqual

Affirmation form AffirmIfEqual checks if two values are equal. It greatly simplifies the message printing since it differentiates between the Received and Expected values. In the following, AType can be std_logic, std_logic_vector, unsigned, signed, integer, integer_vector, real, character, string or time.

```
-- with an AlertLogID
procedure AffirmIfEqual (
    AlertLogID      : AlertLogIDType ;
    Received, Expected : AType ;
    Message          : string := "";
    Enable           : boolean := FALSE
) ;

-- without an AlertLogID
procedure AffirmIfEqual (
    Received, Expected : AType ;
    Message          : string := "";
    Enable           : boolean := FALSE
) ;
```

AffirmIfNotEqual supports the same parameters as AffirmIfEqual.

10.3 AffirmIfFilesMatch

Affirmation form AffirmIfFilesMatch is for comparing two files.

```
-- with an AlertLogID
procedure AffirmIfFilesMatch (
    AlertLogID      : AlertLogIDType ;
    Name1, Name2    : string;
    Message         : string := "" ;
    Enable          : boolean := FALSE ;
    IgnoreSpaces    : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
procedure AffirmIfFilesMatch (
    AlertLogID      : AlertLogIDType ;
    file File1, File2 : text;
    Message         : string := "" ;
    Enable          : boolean := FALSE ;
    IgnoreSpaces    : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
-- without an AlertLogID
procedure AffirmIfFilesMatch (
    Name1, Name2    : string;
    Message         : string := "" ;
    Enable          : boolean := FALSE ;
    IgnoreSpaces    : boolean := ALERT_LOG_IGNORE_SPACES ;
    IgnoreEmptyLines : boolean := ALERT_LOG_IGNORE_EMPTY_LINES ) ;
procedure AffirmIfFilesMatch (
    file File1, File2 : text;
    Message         : string := "" ;
    Enable          : boolean := FALSE ;
```

```

IgnoreSpaces      : boolean := ALERT_LOG_I IGNORE_SPACES ;
IgnoreEmptyLines : boolean := ALERT_LOG_I IGNORE_EMPTY_LINES ) ;

```

10.4 AffirmIfTranscriptsMatch

AffirmIfTranscriptsMatch compares the current transcript file (defined by GetTranscriptName) with the same named file in the directory specified by the PathToValidatedResults parameter.

```

procedure AffirmIfTranscriptsMatch (
    AlertLogID          : AlertLogIDType ;
    PathToValidatedResults : string;
    Message              : string := "" ;
    Enable               : boolean := FALSE ;
    IgnoreSpaces         : boolean := ALERT_LOG_I IGNORE_SPACES ;
    IgnoreEmptyLines     : boolean := ALERT_LOG_I IGNORE_EMPTY_LINES ) ;

procedure AffirmIfTranscriptsMatch (
    PathToValidatedResults : string;
    Message              : string := "" ;
    Enable               : boolean := FALSE ;
    IgnoreSpaces         : boolean := ALERT_LOG_I IGNORE_SPACES ;
    IgnoreEmptyLines     : boolean := ALERT_LOG_I IGNORE_EMPTY_LINES ) ;

```

10.5 GetAffirmCount

Returns the current affirmation check count.

```
impure function GetAffirmCount return natural ;
```

10.6 IncAffirmCount

Increments the affirmation check count. Intended to be used only in special situations, such as packages that have additional considerations when using affirmations.

```
procedure IncAffirmCount ;
```

11 Reporting Test Completion: HTML and Junit Reports

What happens if a test fails without a completion message? If you have run 100 tests, will you see that one failed to run? AlertLogPkg addresses this with our HTML and JUnit test reports.

The test reports are created in two steps:

- VHDL code creates YAML files
- OSVVM scripting converts the YAML files to HTML and Junit Reports

To create the YAML files, the following three steps must be done in each VHDL test case.

- Set the test name using SetTestName (AlertLogPkg),
- Do Alerts and Affirmations during the test (AlertLogPkg),
- At the end of the test, call EndOfTestReports (instead of ReportAlerts).

- EndOfTestReports writes to several YAML files which are used to generate the TestCase and BuildSummary reports.

Within OSVVM scripting:

- Run each test case with OSVVM script API command simulate or RunTest. When simulate or RunTest completes, HTML based test case reports are created.
- Run the entire test suite with OSVVM script API command build. When build completes HTML and JUnit build summary reports are created.

Details of the reports are shown in the "OSVVM Test Writer's User Guide" and the "Script User Guide" in the Documentation repository.

11.1 EndOfTestReports (in ReportPkg)

At test completion, call EndOfTestReports to creates a collection of reports including:

- a test completion message via ReportAlerts,
- a test summary report via WriteAlertSummaryYaml (later in this section),
- a detailed alert report via WriteAlertYaml (later in this section), and
- a detailed coverage report via WriteCovYaml (see CoveragePkg User Guide).
- a detailed requirements (specification tracking) report via WriteRequirementsYaml
- a detailed Scoreboard report via WriteScoreboardYaml (see ScoreboardGenericPkg User Guide)

Note, EndOfTestReports, writes results to directory "./reports" which is created by OSVVM scripts.

The call interface to EndOfTestReports is:

```
procedure EndOfTestReports (
    ReportAll      : boolean          := FALSE ;
    External Errors : AlertCountType := (0, 0, 0) ;
    Stop           : boolean          := FALSE ;
    TimeOut        : boolean          := FALSE
);
```

11.2 WriteAlertSummaryYaml (Called by EndOfTestReports)

For information only. WriteAlertSummaryYaml is called by EndOfTestReports. As a test is running, OSVVM scripts and WriteAlertSummaryYaml incrementally add to the file OsvvmRun.yml. For a test, WriteAlertSummaryYaml adds the Name, Status, and Results fields to the OsvvmRun.yml file.

When the script build completes, OsvvmRun.yml is renamed to <BuildName>.yml, an HTML report is created <BuildName>.html, and a Junit XML report is created

<BuildName>.xml. Currently the only way to get these files is by running the OSVVM scripts.

The call interface to WriteAlertSummaryYaml is:

```
procedure WriteAlertSummaryYaml (
    File Name      : string := "";
    External Errors : AlertCountType := (0, 0, 0)
);
```

11.3 WriteAlertYaml (Called by EndOfTestReports)

For information only. WriteAlertYaml is called by EndOfTestReports. WriteAlertYaml creates a YAML report that contains all the alert information for a test. This information is recorded in the file, ./reports/<TestSuite>/<TestName>_alerts.yml. When simulate finishes, this information is combined with coverage information and converted to HTML in the test results file, ./reports/<TestSuite>/<TestName>.html. Status for a test is reported as PASSED, FAILED, NOCHECKS (no affirmations done), and TIMEOUT. A test with status NOCHECKS is considered to be failed by default.

The call interface to WriteAlertYaml is:

```
procedure WriteAlertYaml (
    File Name      : string ;
    External Errors : AlertCountType := (0, 0, 0) ;
    Prefix         : string := "" ;
    PrintSettings : boolean := TRUE ;
    PrintChildren  : boolean := TRUE ;
    OpenKind       : File_Open_Kind := WRITE_MODE ;
    Timeout        : boolean
);
```

12 Requirements Tracking

OSVVM can track requirements either as an Alert or as Coverage. As an Alert, a requirement is nothing more than an AlertLogID that has a PASSED goal associated with it. As Coverage, a requirement is a CoverageID that has a goal. Requirements pass if the goal is reached. By default, the goal is 1.

Requirement IDs are created with NewReqID. A group of requirements can be read in with ReadSpecification. Then the ID can be looked up using GetReqID.

Requirements can be checked by using any affirmation and the AlertLogID that is associated with that requirement. Requirements can also be checked using a special form of AffirmIF that takes the requirement name (as a string).

When a test case completes and calls EndOfTestReports, a YAML requirements report is automatically generated. When a test suite completes, OSVVM scripts merge the YAML requirements for that test suite. When the build completes, OSVVM scripts merge the YAML requirements for each test suite that was run during the build and produces an HTML and CSV merged requirements report.

12.1 NewReqID

Use NewReqID when the intent is to create an AlertLogID that is used for requirements tracking. The interface for NewID is as follows.

```
impure function NewReqID(
    Name          : string ;
    Goal          : natural ;
    ParentID      : AlertLogIDType      := REQUIREMENT_ALERTLOG_ID ;
    ReportMode    : AlertLogReportModeType := ENABLED ;
    PrintParent   : AlertLogPrintParentType := PRINT_NAME_AND_PARENT ;
    CreateHierarchy : boolean           := TRUE
) return AlertLogIDType ;
```

Call NewReqID with the name of the requirement and the PassedGoal.

```
signal Uart1ReqID : AlertLogIDType ;
.
.
Uart1ReqID := NewReqID("UART_REQ_1", 2);
```

12.2 GetReqID

GetReqID returns an AlertLogID for a requirement. If an AlertLogID already exists for the requirement, GetReqID will return its AlertLogID. Use GetReqID when the intent is

to look up the ID – but if it does not exist yet then create it. Typically GetReqID is called in the testbench, well after time 0, to look up an ID.

The interface for GetReqID is as follows.

```
impure function GetReqID (
    Name          : string ;
    PassedGoal    : integer := -1 ; -- If -1, use DefaultPassedGoal Var
    ParentID      : AlertLogIDType := ALERTLOG_ID_NOT_ASSIGNED;
    CreateHierarchy : Boolean := TRUE
) return AlertLogIDType ;
```

When the ParentID defaults to ALERTLOG_ID_NOT_ASSIGNED, internally GetReqID changes this to REQUIREMENT_ALERTLOG_ID.

Creating AlertLogIDs for requirements is as follows. Since PassedGoal is not set in the call, the DefaultPassedGoalVar is used (default is 1). Change with SetAlertLogOptions(DefaultPassedGoal => 2).

```
signal Uart1ReqID : AlertLogIDType ;
signal Uart2ReqID : AlertLogIDType ;

.
.
InitialiseProc: process
begin
    Uart1ReqID := GetReqID("UART_REQ_1");
    Uart2ReqID := GetReqID("UART_REQ_2", 2);
```

12.3 ReadSpecification

ReadSpecification reads files in the following format. In the following items enclosed in "[]" are optional and are ignored.

```
<requirement1 ID Name> [, Requirement1 Description] [, Requirement1 PASSED Goal]
<requirement2 ID Name> [, Requirement2 Description] [, Requirement2 PASSED Goal]
.
.
<requirementN ID Name> [, RequirementN Description] [, RequirementN PASSED Goal]
```

The interface for ReadSpecification is as follows.

```
procedure ReadSpecification (FileName : string ; PassedGoal : integer := -1) ;
```

A call to ReadSpecification:

```
ReadSpecification("Uart_Specification.txt") ;
```

If a requirement is not set, by default it will be 1. The default can be changed using SetAlertLogOptions(DefaultPassedGoal => 2). Alternately all requirements can be changed to the PassedGoal specified in the call to ReadSpecification (here 2):

```
ReadSpecification("Uart_Specification.txt", 2) ;
```

12.4 SetPassedGoal

The PassedGoal of an AlertLogID can be replaced by calling SetPassedGoal.

```
procedure SetPassedGoal (AlertLogID : AlertLogIDType ; PassedGoal : integer) ;
  . .
  SetPassedGoal (Req1ID, 2) ;
```

12.5 AffirmIf for Requirements

For requirements, an additional form of AffirmIf that uses the requirements ID name is provided. It has two forms of overloading. The first has a separate ReceivedMessage and ExpectedMessage. When the affirmation passes, log is called with just the Received Message. When an affirmation fails, alert is called with the ExpectedMessage concatenated to the end of the ReceivedMessage.

```
-- with an AlertLogID
procedure AffirmIf(
  RequirementsIDName : string ;
  condition          : boolean ;
  ReceivedMessage    : string ;
  ExpectedMessage    : string ;
  Enable             : boolean := FALSE
) ;
```

The second overloading of AffirmIF has a single Message parameter. Hence, both alert and log print the same message.

```
-- with an AlertLogID
procedure AffirmIf(
  RequirementsIDName : string ;
  condition          : boolean ;
  Message            : string ;
  Enable             : boolean := FALSE
) ;
```

A call to AffirmIF with two parameters is as follows.

```
AffirmIf("T1.1", A > B, "Test 1.1 A (" & to_string(A) & ")",
         "> B (" & to_string(B) & ")" ) ;
```

Note that the RequirementsIDName is case insensitive.

12.6 Using Other Forms of AffirmIF with Requirements

To use other forms of AffirmIF, either the ID needs to be known (NewReqID or GetReqID) or looked up using FindID.

```
AffirmIfEqual (Uart1ReqID, TestValue1, ExpectedValue1, "Message") ;
AffirmIfEqual (FindID("Axi 4Req1"), TestValue2, ExpectedValue2, "Message") ;
```

12.7 WriteRequirementsYaml (Called by EndOfTestReports)

For information only. WriteRequirementsYaml is called by EndOfTestReports. WriteRequirementsYaml creates a YAML report that contains all requirements information for a test case. The file, ./reports/<TestSuite>/<TestName>_req.yml, holds the requirements results for a given test case.

The call interface to WriteRequirementsYaml is:

```
procedure WriteRequirementsYaml (
    File Name : string ;
    Alert Log ID : Alert Log ID Type := REQUIREMENT_ALERTLOG_ID ;
    Open Kind : File_Open_Kind := WRITE_MODE
);
```

12.8 Test Failure Due to Missed Requirements

When a test has requirements, a test will fail if requirements are not met and FailOnRequirementErrors is true (default is TRUE). Disable using SetAlertLogOptions(FailOnRequirementErrors => TRUE).

A test failing due to FailOnRequirementErrors will result in a ReportAlerts such as the following.

```
%% DONE FAILED req_test1 Total Error(s) = 2 Failures: 0 Errors: 0 Warnings: 0
0 Passed: 20 Affirmations Checked: 20 Requirements Passed: 2 of 4 at 50 ns
%% Default Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 0
%% OSVVM Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 0
%% Requirements Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 0
%% T1.1 Failures: 0 Errors: 0 Warnings: 0 Passed: 1 of 1
%% T1.2 Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 2
%% T1.3 Failures: 0 Errors: 0 Warnings: 0 Passed: 2 of 3
%% T1.4 Failures: 0 Errors: 0 Warnings: 0 Passed: 4 of 4
```

12.9 ReportAlerts and Requirements

ReportAlerts prints requirements in both the summary and the detailed report when a test fails.

In the summary report (first line) of ReportAlerts, a requirements summary (# met of # total) will be printed if either PrintRequirements is TRUE (default is FALSE) or both the test has requirements and PrintIfHaveRequirements is TRUE (default is TRUE).

```
%% DONE PASSED req_test2 Total Error(s) = 0 Failures: 0 Errors: 0 Warnings: 0
0 Passed: 20 Affirmations Checked: 20 Requirements Passed: 4 of 4 at 50 ns
```

For a test that fails, detailed printing of requirements will be printed in ReportAlerts if PrintRequirements is TRUE (default is FALSE).

```
%% DONE FAILED req_test1 Total Error(s) = 6 Failures: 0 Errors: 4 Warnings: 0
0 Passed: 20 Affirmations Checked: 20 Requirements Passed: 2 of 4 at 50 ns
%% Default Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 0
%% OSVVM Failures: 0 Errors: 0 Warnings: 0 Passed: 0 of 0
%% Requirements Failures: 0 Errors: 4 Warnings: 0 Passed: 0 of 0
```

```
%%      T1. 1      Failures: 0  Errors: 0  Warnings: 0  Passed: 1 of 1
%%      T1. 2      Failures: 0  Errors: 2  Warnings: 0  Passed: 0 of 2
%%      T1. 3      Failures: 0  Errors: 2  Warnings: 0  Passed: 1 of 3
%%      T1. 4      Failures: 0  Errors: 0  Warnings: 0  Passed: 4 of 4
```

Both `PrintRequirements` and `PrintIfHaveRequirements` can be changed using `SetAlertLogOptions`.

13 Alert and Log Output Control Options

13.1 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(Enable : boolean := TRUE);
```

13.2 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);
```

13.3 SetAlertLogOptions: Configuring Report Options

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

```
procedure SetAlertLogOptions (
    FailOnWarning      : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    FailOnEnabledErrors : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    FailOnRequirementErrors : OsvvmOptionsType := OPT_INIT_PARM_DETECT; --2020.08
    ReportHierarchy     : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteAlertErrorCount : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.05
    WriteAlertLevel      : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteAlertName       : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteAlertTime        : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteLogErrorCount   : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.05
    WriteLogLevel         : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteLogName          : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    WriteLogTime          : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    PrintPassed          : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.05
    PrintAffirmations    : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.05
    PrintEnabledAlerts    : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.05
    PrintRequirements     : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.08
    PrintIfHaveRequirements : AlertLogOptionsType := OPT_INIT_PARM_DETECT; --2020.08
    DefaultPassedGoal     : integer := integer'left; --2020.08
    AlertPrefix           : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    LogPrefix             : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    ReportPrefix          : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    DoneName              : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    PassName              : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    FailName              : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    IdSeparator            : string := OSVVM_STRING_INIT_PARM_DETECT; -- Deprecated
    WriteTimeLast         : OsvvmOptionsType := OPT_INIT_PARM_DETECT;
    TimeJustifyAmount     : integer := integer'left
);
```

With OSVVM release 2024.03, the default settings come from OsvvmSettingsPkg_default. If you want to change that default setting, copy OsvvmSettingsPkg_default.vhd to

OsvvmSettingsPkg_local.vhd and make the changes you desire. This provides a simple way to make these values available to all test cases.

The following options are for ReportAlerts.

Parameter / Constant Name	Description	Value
PrintPassed ALERT_LOG_PRINT_PASSED	If PrintPassed is enabled, print number of passed logs in the report summary and for each AlertLogID	TRUE
PrintAffirmations ALERT_LOG_PRINT_AFFIRMATIONS	If PrintAffirmations is enabled, print the number of affirmations in the report summary and for each AlertLogID	FALSE
FailOnWarning ALERT_LOG_FAIL_ON_WARNING	Count warnings as test errors.	TRUE
FailOnDisabledErrors ALERT_LOG_FAIL_ON_DISABLED_ERRORS	If FailOnDisabledErrors is enabled and DisabledAlerts /= 0, then 1) Test Fails 2) DisabledAlerts are printed in the report summary. 3) DisabledAlerts print for each AlertLogID	TRUE
FailOnRequirementErrors ALERT_LOG_FAIL_ON_REQUIREMENT_ERRORS	If FailOnRequirementErrors is enabled, a Test fails if it does not reach its requirements (PASSED goals).	TRUE
PrintDisabledAlerts ALERT_LOG_PRINT_DISABLED_ALERTS	If PrintDisabledAlerts is Enabled, print DisabledAlerts in the report summary and for each AlertLogID	FALSE
PrintRequirements ALERT_LOG_PRINT_REQUIREMENTS	If PrintRequirements is enabled, then 1) In the summary of ReportAlerts, print a requirements summary (# met of # total). 2) In the detailed report, print a requirements ReportAlerts, print a requirements summary (# met of # total).	FALSE
PrintIfHaveRequirements ALERT_LOG_PRINT_IF_HAVE_REQUIREMENTS	If PrintIfHaveRequirements is enabled and the test has requirements, then in the summary of ReportAlerts, print a requirements summary (# met of # total).	TRUE
ReportHierarchy ALERT_LOG_REPORT_HIERARCHY	When multiple AlertLogIDs exist, print an error summary for each level.	TRUE

The following options are for both alert and log:

Parameter / Constant Name	Description	Value
WriteTimeLast ALERT_LOG_WRITE_TIME_FIRST	Write time last when TRUE or first when FALSE.	FALSE
TimeJustifyAmount ALERT_LOG_TIME_JUSTIFY_AMOUNT	Print level.	9

The following options are for alert:

Parameter / Constant Name	Description	Value
WriteAlertErrorCount ALERT_LOG_WRITE_ERRORCOUNT	Print Error Count immediately after %% Alert	FALSE
WriteAlertLevel ALERT_LOG_WRITE_LEVEL	Print level.	TRUE
WriteAlertName ALERT_LOG_WRITE_NAME	Print AlertLogID name.	TRUE
WriteAlertTime ALERT_LOG_WRITE_TIME	Alerts print time.	TRUE

The following options are for Log:

Parameter / Constant Name	Description	Value
WriteLogErrorCount ALERT_LOG_WRITE_ERRORCOUNT	Print Error Count immediately after %% Log	FALSE
WriteLogLevel ALERT_LOG_WRITE_LEVEL	Print level.	TRUE
WriteLogName ALERT_LOG_WRITE_NAME	Print AlertLogID name.	TRUE
WriteLogTime ALERT_LOG_WRITE_TIME	Logs print time.	TRUE

The following options are PassedGoals – default for GetReqID and ReadSpecification

Parameter / Constant Name	Description	Value
DefaultPassedGoal ALERT_LOG_DEFAULT_PASSED_GOAL	Set the Default Passed Goal for ReadSpecification and GetReqID	1

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

```
SetAlertLogOptions (
    FailOnWarning      => FALSE,
    FailOnDiagnostics => 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.

The following are set in OsvvmSettingsPkg_default.vhd and can be overridden in OsvvmSettingsPkg_local.vhd. The settings have been deprecated in SetAlertLogOptions and produce alert WARNING if used.

Constant Name / Deprecated Parameter	Description	Value
OSVVM_PRINT_PREFIX ALERT_LOG_PRINT_PREFIX	Prefix for each line of ReportAlerts.	"%% "
<i>ReportPrefix - Deprecated</i>		

OSVVM_DONE_NAME <i>DoneName - Deprecated</i>	Name printed after ReportPrefix on first line of ReportAlerts.	"DONE"
OSVVM_PASS_NAME ALERT_LOG_PASS_NAME <i>PassName - Deprecated</i>	Name printed when a test passes.	"PASSED"
OSVVM_FAIL_NAME ALERT_LOG_FAIL_NAME <i>FailName - Deprecated</i>	Name printed when a test fails.	"FAILED"
ALERT_LOG_ALERT_NAME <i>AlertPrefix Deprecated</i>	Name printed at beginning of alert.	"Alert"
ALERT_LOG_LOG_NAME <i>LogPrefix - Deprecated</i>	Name printed at beginning of log.	"Log"
ALERT_LOG_ID_SEPARATOR <i>IdSeparator - Deprecated</i>	Name printed as a separator.	:

13.4 ReportAlertLogOptions: Print Report Options

Prints out AlertLogPkg Report Options.

13.5 Getting AlertLog Report Options

Report options can be retrieved with one of the Get functions below.

```
function GetAlertLogFailureWarning      return AlertLogOptionsType;
function GetAlertLogFailureDiagnostics return AlertLogOptionsType;
function GetAlertLogReportHierarchy     return AlertLogOptionsType;
function GetAlertLogHierarchyInUse      return AlertLogOptionsType;
function GetAlertLogWriteAlertLevel    return AlertLogOptionsType;
function GetAlertLogWriteAlertName     return AlertLogOptionsType;
function GetAlertLogWriteAlertTime     return AlertLogOptionsType;
function GetAlertLogWriteLogLevel      return AlertLogOptionsType;
function GetAlertLogWriteLogName       return AlertLogOptionsType;
function GetAlertLogWriteLogTime      return AlertLogOptionsType;
```

Note while the following are still supported, they just return the constant from OsvvmSettingsPkg_default.vhd identified in SetAlertLogOptions

```
function GetAlertLogAlertPrefix        return string;
function GetAlertLogLogPrefix          return string;
function GetAlertLogReportPrefix       return string;
function GetAlertLogDoneName          return string;
function GetAlertLogPassName          return string;
function GetAlertLogFailureName       return string;
```

14 Deallocating and Re-initializing the Data structure

14.1 DeallocateAlertLogStruct

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

14.2 InitializeAlertLogStruct

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

15 Deprecated: Older CSV Type Reporting

OSVVM can also create Test Completion and Requirements reports as CSV files. The approach using CSV files requires considerable effort on your part.

These have been deprecated in favor of the VHDL (YAML) + Scripting (HTML, JUnit, CSV) approach that happen mostly autonomously with OSVVM VHDL + OSVVM Scripting.

15.1 Deprecated Reporting Test Completion: CSV Reports

As an alternative to creating YAML reports (from VHDL) and HTML and Junit reports (from OSVVM Scripting), OSVVM has an older CSV based approach.

This approach requires a significant amount of work that is specific to your test suite. As a result, it is deprecated.

The process for CSV based test tracking involves the following steps:

- WriteTestSummary – write out a one-line test summary. Each test in a test suite writes to the same results file.
- ReadTestSummaries – read in a set of test summaries written out by WriteTestSummary.
- ReportTestSummaries – Print a summary for each test that was read by ReadTestSummaries

In addition, WriteAlerts produces a CSV based detailed alert report.

CSV was explored as an initial methodology to write out reports for consumption by other tools. Unfortunately, CSV does not do an adequate job in conveying hierarchical relationships and it has been set aside in preference of YAML reports.

15.1.1 Deprecated: WriteAlerts

WriteAlerts prints all the information in ReportAlerts in a CSV format.

```
procedure WriteAlerts (
    FileName      : string ;
    AlertLogID   : AlertLogIDType := ALERTLOG_BASE_ID ;
    OpenKind      : File_Open_Kind := WRITE_MODE
);
```

The first line of WriteAlerts prints the report summary in a CSV format (see WriteTestSummary). The format of WriteAlerts is as follows:

```
<Name>, <PassedGoal>, <PassedCount>, <Total Errors>, <Failures>, <Errors>, <Warnings>, <AffirmCount>
```

15.1.2 Deprecated: WriteTestSummary

WriteTestSummary creates a one line print of the information that is in the report summary (the first line of ReportAlerts) as a CSV. WriteTestSummary is intended to be used by each separate test in the test suite.

```
procedure WriteTestSummary(  
    FileName : string ;  
    OpenKind : File_Open_Kind := APPEND_MODE ) ;
```

The following WriteTestSummary adds a test summary for the current test to the OsvvmTestSummary.txt file. The default "APPEND_MODE" is used by all tests except maybe the first test. If the regression manager does not delete the OsvvmTestSummary.txt, then the first test will use "WRITE_MODE".

```
WriteTestSummary("./resul ts/OsvvmTestSummary.txt") ;
```

WriteTestSummary prints in the following format. ReqGoal is the count of the number of requirements that have PassedGoals. ReqCount is the count of the number of requirements whose PassedCount is greater or equal to the PassedGoal.

```
<Name>, <ReqGoal>, <ReqCount>, <Total Errors>, <Failures>, <Errors>, <Warnings>, <AffirmCount>, <PassedCount>
```

15.1.3 Deprecated: ReadTestSummaries

ReadTestSummaries reads in the file created by WriteTestSummary and records each separate test results as a requirement.

```
procedure ReadTestSummaries (FileName : string) ;
```

Calling ReadTestSummaries:

```
ReadTestSummaries("./resul ts/OsvvmTestSummary.txt") ;
```

15.1.4 Deprecated: ReportTestSummaries

ReportTestSummaries produces a report of the results from the test summary file. ReportTestSummaries has no input parameters and is called as follows.

```
ReportTestSummaries ;
```

ReportTestSummaries produces a report such as the following. Note each line starts with %. The "Affirmations ..." printed on a separate line is a line wrap in the text below.

```
%% PASSED test_1 Total Error(s) = 0 Failures: 0 Errors: 0 Warnings: 0  
Affirmations Passed: 11 of 11 Requirements Passed: 2 of 2  
%% FAILED test_2 Total Error(s) = 2 Failures: 0 Errors: 0 Warnings: 0  
Affirmations Passed: 22 of 22 Requirements Passed: 2 of 4  
%% PASSED test_3 Total Error(s) = 0 Failures: 0 Errors: 0 Warnings: 0  
Affirmations Passed: 33 of 33 Requirements Passed: 6 of 6  
%% FAILED test_4 Total Error(s) = 2 Failures: 0 Errors: 2 Warnings: 0  
Affirmations Passed: 44 of 44 Requirements Passed: 6 of 8  
%% PASSED test_5 Total Error(s) = 0 Failures: 0 Errors: 0 Warnings: 0  
Affirmations Passed: 55 of 55 Requirements Passed: 10 of 10  
%% PASSED test_6 Total Error(s) = 0 Failures: 0 Errors: 0 Warnings: 0  
Affirmations Passed: 66 of 66 Requirements Passed: 10 of 12
```

If a test does not run, it will fail if (AffirmCount <= 0) and (RequirementsGoal > RequirementsPassed) and (FailOnRequirementsVar = TRUE). FailOnRequirements is set by SetAlertLogOptions and TRUE is the default. These will print as follows:

```
%% FAILED test_7    Total Error(s) = 1  Failures: 0  Errors: 0  Warnings: 0
Affirmations Passed: 0 of 0 Requirements Passed: 0 of 1
%% FAILED test_8    Total Error(s) = 1  Failures: 0  Errors: 0  Warnings: 0
Affirmations Passed: 0 of 0 Requirements Passed: 0 of 1
```

If (AffirmCount = 0) and either (RequirementsGoal = 0) or FailOnRequirements is FALSE, the test results are indeterminant. These will print as follows:

```
%% ?????? test_7    Total Error(s) = 1  Failures: 0  Errors: 0  Warnings: 0
Affirmations Passed: 0 of 0 Requirements Passed: 0 of 1
%% ?????? test_8    Total Error(s) = 1  Failures: 0  Errors: 0  Warnings: 0
Affirmations Passed: 0 of 0 Requirements Passed: 0 of 1
```

15.1.5 Deprecated: WriteTestSummaries

WriteTestSummaries produces a csv file of the same information that is printed ReportTestSummaries.

```
procedure WriteTestSummaries (FileName : string ; OpenKind : File_Open_Kind) ;
```

15.1.6 Deprecated: Use Model

One potential use model is to just get a report of what passed and failed. This would involve the following.

```
-- In Test1.vhd
WriteTestSummary("./resul ts/OsvvmTestSummary.txt", WRITE_MODE) ;
.
.
-- In Test2.vhd
WriteTestSummary("./resul ts/OsvvmTestSummary.txt") ;
.
.
-- In TestN.vhd
WriteTestSummary("./resul ts/OsvvmTestSummary.txt") ;
.
.
-- In OsvvmTestSummary.vhd
ReadTestSummaries("./resul ts/OsvvmTestSummary.txt") ;
ReportTestSummaries ;
-- WriteTestSummaries("PointLess_ IsNotSameAs_OsvvmTestSummary.txt") ;
```

However, the above approach fails to find tests that for some reason did not run and did not produce any output. Note also that using, WriteTestSummaries in the above example would produce the same information that is already in OsvvmTestSummary.txt after all of the tests have finished. So there is no reason to do WriteTestSummaries above.

First, remember that test summaries are recorded as requirements. To find what tests did not run and to make the order of printing match a predefined order, lets define the tests that we wanted to run as a list of test names. Then read the name of tests in as a specification. Hence, we will modify OsvvmTestSummary.vhd as follows:

```
-- In 0svvmTestSummary.vhd
ReadSpecification("./0svvmTestsToRun.txt");
ReadTestSummaries("./results/0svvmTestSummary.txt");
ReportTestSummaries;
WriteTestSummaries("0svvmAllTestResults.txt");
```

In this case, tests that did not run will show up as having an AffirmCount of 0 and a RequirementGoal of 1, hence, by defaults of ReportTestSummaries, the test will fail.

15.2 Deprecated: Requirements Reporting: CSV Reports

ReportRequirements produces a test summary as well as a tabulation of each requirement. WriteRequirements prints the same information in a CSV format. ReadRequirements reads multiple files produced by WriteRequirements and merges them.

Caution: ReportRequirements and WriteRequirements only report/write requirements that are in the hierarchy under the REQUIREMENT_ALERTLOG_ID. The current implementation allows requirements to be anywhere in the AlertLog hierarchy.

15.2.1 Deprecated: ReportRequirements

ReportRequirements produces a report that has the test summary of the ReportAlerts and then just the IDs that are under the Requirements bin (via GetReqID or ReadSpecification). A ReportRequirements for the previous ReportAlerts is as follows.

```
ReportRequirements;
.
.
%> DONE FAILED req_test1 Total Error(s) = 6 Failures: 0 Errors: 4 Warnings: 0
0 Passed: 20 Affirmations Checked: 20 Requirements Passed: 2 of 4 at 50 ns
%> T1.1 Failures: 0 Errors: 0 Warnings: 0 Passed: 1 of 1
%> T1.2 Failures: 0 Errors: 2 Warnings: 0 Passed: 0 of 2
%> T1.3 Failures: 0 Errors: 2 Warnings: 0 Passed: 1 of 3
%> T1.4 Failures: 0 Errors: 0 Warnings: 0 Passed: 4 of 4
```

15.2.2 Deprecated: WriteRequirements

WriteRequirements creates a CSV file that has the same information as ReportRequirements. The format is identical to WriteAlerts.

```
procedure WriteRequirements (
    FileName      : string;
    AlertLogID    : AlertLogIDType := REQUIREMENT_ALERTLOG_ID;
    OpenKind      : File_Open_Kind := WRITE_MODE
);
```

15.2.3 Deprecated: ReadRequirements

ReadRequirements reads a file that is in the format of WriteRequirements. The intent of ReadRequirements is to read and merge the WriteRequirements information from multiple separate tests.

```
procedure ReadRequirements (
    FileName      : string ;
    ThresholdPassed : boolean := FALSE
);
```

If the ThresholdPassed parameter is true and if the read PassedCount exceeds the read PassedGoal, the PassedCount is replaced by the PassedGoal.

16 Deprecated Names and their Replacement name

From time to time, names must change to keep consistency across a large set of packages. Where possible OSVVM uses a soft deprecation methodology where the subprogram takes on the new name and the original name is supported via an alias. For example SetTestName was originally named SetTestName and FindID was originally named FindAlertLogID. As a result, you will find aliases in AlertLogPkg as follows:

```
alias SetAlertLogName is SetTestName [string] ;
alias FindAlertLogID is FindID [string return AlertLogIDType] ;
```

17 About AlertLogPkg

AlertLogPkg is part of the OSVVM Utility library.

AlertLogPkg was created by Jim Lewis of SynthWorks. Please support our work by buying your VHDL Training from SynthWorks.

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.

18 Compiling and Using OSVVM Utility Library

Reference all packages in the OSVVM Utility library by using the context declaration:

```
library OSVVM ;
context osvvm.OsvvmContext ;
```

Compilation order for OSVVM Utility Library is in OSVVM_release_notes.pdf. Rather than learning this, we recommend using the OSVVM compilation scripts.

OSVVM Utility library is released under the Apache open source license. It is free (both to download and use - there are no license fees). You can download it from osvvm.org or from our development area on GitHub (<https://github.com/OSVVM/OSVVM>).

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.

We also support the OSVVM user community and blogs through <http://www.osvvm.org>. Interested in sharing about your experience using OSVVM? Let us know, you can blog about it at osvvm.org.

19 About the Author - Jim Lewis

Jim Lewis, the founder of SynthWorks, has thirty plus 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.