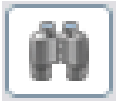
SYNOPSYS®

VMT User's Manual

Included in the VCS Verification Library and the DesignWare Library

To search this manual or the entire set, press this toolbar button. For help, refer to intro.pdf.



Release 3.10a

March 20, 2008

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert Plus, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Contents

Tables

Preface

About This Manual

This manual contains general introductory, usage, and reference material about Vera Modeling Technology (VMT) features. Information for specific DesignWare Verification IP can be found in the documentation for each suite of models.

Related Documents

This manual is part of the VMT document set. The document set also includes:

- *VMT Installation Guide* – Contains system requirements, installation procedures, and setup information.
- *VMT Release Notes* – Contains new features, fixed problems, and known problems and limitations common to all VMT models.

IMPORTANT: This document set does NOT contain information about:

- Using DesignWare VIP in an RVM (Reference Verification Methodology) environment
- Using DesignWare VIP in SystemVerilog testbenches

For documentation about using DesignWare Verification IP in these situations, refer to the specific VIP suite documentation that is installed at:

$DESIGNWARE_HOME/vip/<vip_suite_name>/latest/doc

Manual Overview

This manual contains the following chapters and appendixes:

Chapter 1 "VMT Introduction"	Includes VMT-specific terminology, an overview of model operation, and an explanation of how different VMT commands work.
Chapter 2 "Using VMT Models"	Explains model messaging, how to work with VMT commands and command streams, interrupt handling, and memory patterns.
Chapter 3 "VMT Common Command Reference"	Contains all common commands, macros, and model messages.

Appendix A "Reporting Problems"	Provides procedures for creating model MCD files. These files are used by the Support Center staff as a troubleshooting aid.
Appendix B "Glossary"	Defines terms used throughout this book.

Typographical and Symbol Conventions

The following conventions are used throughout this document:

Table 1: Documentation Conventions

Convention	Description and Example
%	Represents the UNIX prompt.
Bold	User input (text entered by the user). `% cd $DESIGNWARE_HOME/iip`
Monospace	System-generated text (prompts, messages, files, reports). `No Mismatches: 66 Vectors processed: 66 Possible"`
Italic or *Italic*	Variables for which you supply a specific value. As a command line example: `% setenv DESIGNWARE_HOME prod_dir` In body text: In the previous example, *prod_dir* is the directory where your product must be installed.
| (Vertical rule)	Choice among alternatives, as in the following syntax example: `-effort_level low | medium | high`
[ ] (Square brackets)	Enclose optional parameters: `pin1 [pin2 ... pinN]` In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* … *pinN*]).
TopMenu > SubMenu	Pulldown menu paths, such as: **File > Save As …**

Getting Help

If you have a question about using Synopsys products, please consult product documentation that is installed on your network or located at the root level of your Synopsys product CD-ROM (if available). You can also contact the Synopsys Support Center in the following ways:

- Open a call to your local support center using this Web page:

  http://www.synopsys.com/support/support.html

- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center:
  - United States:
    Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.
  - Canada:
    Call 1-650-584-4200 from 7 AM to 5:30 PM Pacific Time, Mon—Fri.
  - All other countries:
    Find other local support center telephone numbers at the following URL:

    http://www.synopsys.com/support/support_ctr

Additional Information

For additional Synopsys documentation, refer to the following page:

http://www.synopsys.com/products/designware/docs/toc/dwlibdocs.php

For up-to-date information about the latest implementation IP and verification models, visit the DesignWare home page:

http://www.synopsys.com/designware

Comments?

To report errors or make suggestions, please send e-mail to:

support_center@synopsys.com.

To report an error that occurs on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your e-mail message. This will provide us with information to identify the source of the problem.

1 VMT Introduction

Chapter Contents

VMT Model Overview

VMT models include bus functional models and monitor models that are written in OpenVera and can be instantiated in OpenVera, Verilog, or VHDL testbenches. All VMT models have a common command interface style that allows you to easily integrate standard bus protocol devices into your system testbenches.

During simulation, VMT models report interesting events and it is up to the testbench to react to those events. VMT models do not take any special action (such as stopping the simulation) when they detect an error condition. To control when the testbench reacts to a simulation event, you can use watchpoints. When the watchpoint triggers, the testbench can perform whatever action is desired. For example, to react to detected errors, you can set a watchpoint to trigger on all messages that have a type of VMT_MSG_ERROR. You can then design your testbench to execute a specific set of instructions each time the watchpoint triggers.

A model testbench consists of one or more threads of execution. Each thread can execute VMT commands which get passed to the VMT model using command streams. VMT commands can be "blocking" or "non-blocking." A blocking command means the testbench thread will not progress past the blocking command until that command is executed by the model. With a non-blocking command, the testbench thread progresses without waiting.

New command streams are created using the new_stream command. When a new command stream is created, a new active command queue is also created. All command streams can accept commands from testbench threads. All non-queued commands will be executed regardless of which command stream they are associated with. Only queued commands placed on the active queue will be executed; the non-active queues simply queue the commands for later execution.

The following figure shows how a VMT model interacts with a testbench and simulator.

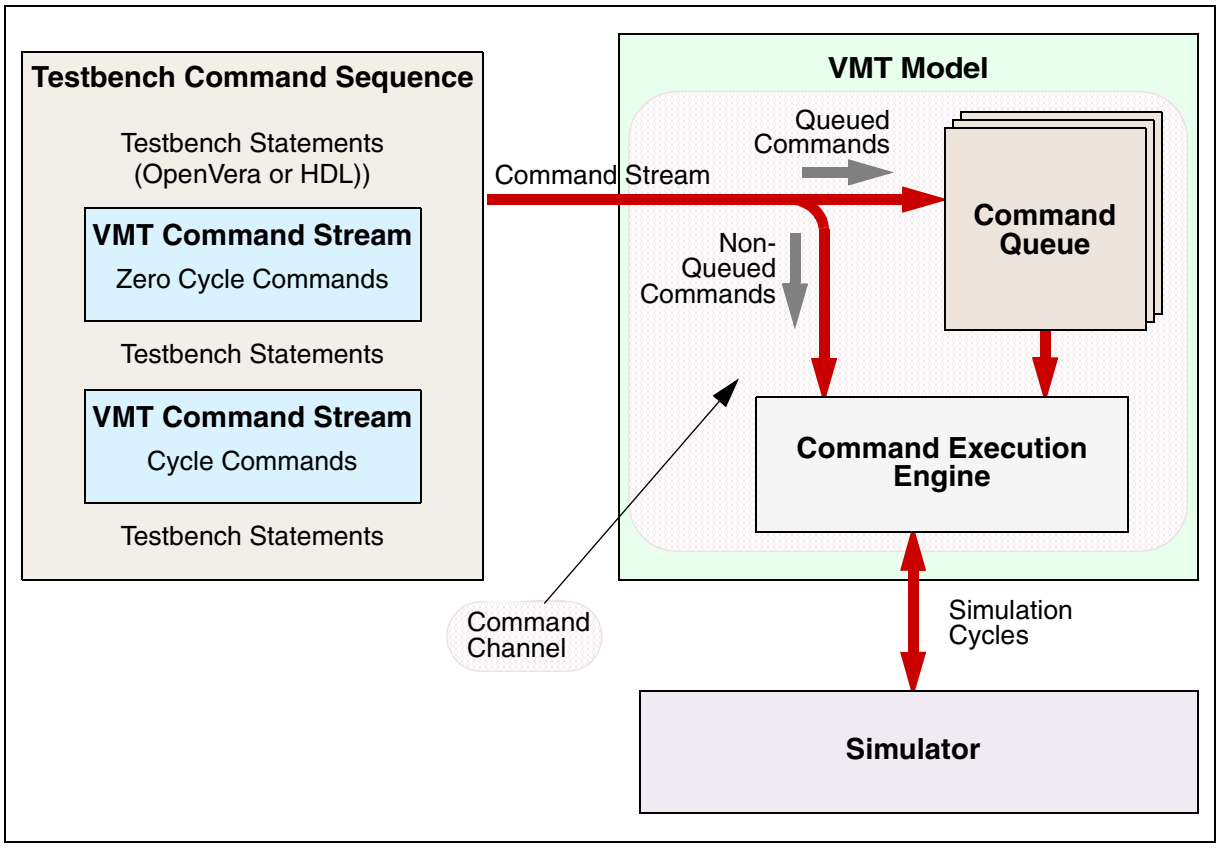


Figure 1: VMT Command Flow

In the figure above, the VMT model accepts commands that are part of a testbench command sequence consisting of VMT commands in one or more command streams. The model executes the commands from the active command queue or directly from the command stream, depending on whether the command is queued or not.

Simulator Control

VMT models are designed to operate with a choice of testbench languages: OpenVera, Verilog, or VHDL. Your choice of testbench language helps determine the simulation environment. The following environments are available:

- Vera
- VCS with NTB
- Pioneer NTB

These environments are described next.

VCS with NTB

The most efficient environment is achieved using VCS with Native TestBench (NTB), in which the OpenVera testbench, VMT model, and design under test (DUT) are compiled and executed natively by VCS. As shown below, this environment yields a tightly integrated and efficient simulation.

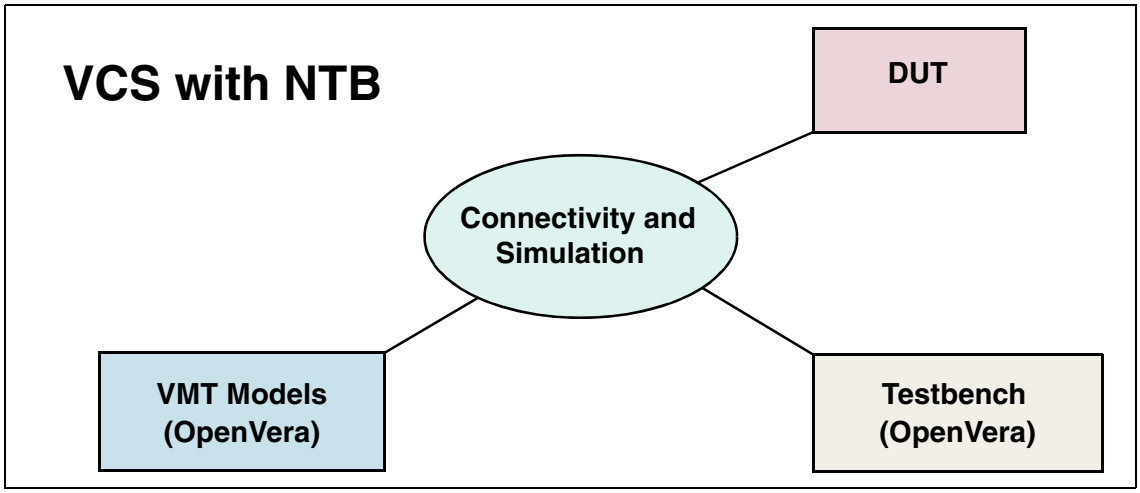


Figure 2: Simulation Control for VCS with NTB

Vera

For Vera environments, there are two domains for a given simulation, as shown below.

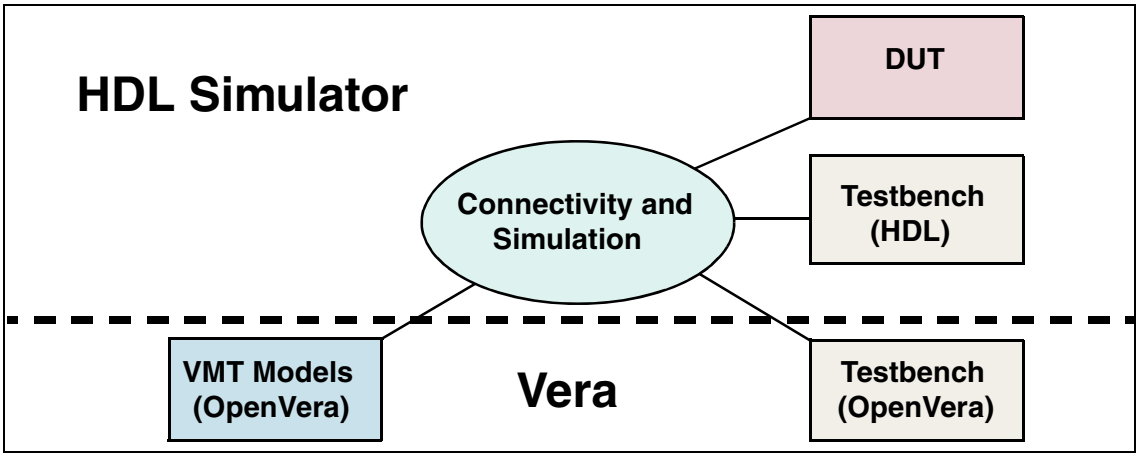


Figure 3: Simulation Control for HDL and Vera

As shown above, the testbench code, which is where the VMT model is instantiated, can be either HDL or OpenVera. The top of the code hierarchy is an HDL layer (either Verilog or VHDL) that always provides connectivity to the simulator and commonly includes clock generation. The HDL simulator is the simulation 'master' and hands control to the Vera simulator to control the VMT model. If the testbench code is in OpenVera, the HDL layer is still present, serving again as a connectivity and simulation control layer.

VMT Commands

There are two types of VMT commands: model-specific and common. Model-specific commands exercise protocol functions that are specific to that model and are applicable to model-specific functionality. Common commands are applicable to most or all models and are useful for testbench flow control, general-purpose configuration tasks and pin-level error injection.

Not all common commands may be functional for every model. Consult specific model documentation to see which common commands are available.

Some commands do not generate bus cycles. For example, all non-queued, non-blocking commands do not generate any bus cycles and are called zero cycle commands. Likewise, commands that check or modify model characteristics are zero cycle commands. An unlimited number of zero cycle commands can execute without advancing simulation time. Cycle commands take at least one clock cycle to complete execution.

Attention

If you are controlling VMT models from an HDL testbench or an OpenVera testbench simulating in VCS NTB, then you must allow at least one clock cycle to elapse in your testbench before issuing any commands to the VMT model; this allows VMT models to initialize. After initialization, VMT models can accept commands from the HDL.

This does not apply when controlling VMT models in a Vera environment.

Queued and Blocking Commands

Within all VMT models, there is a queue that the models use to process commands. This queue accepts commands from the testbench without advancing simulation time. Note that the commands are not necessarily executed when they are sent to the queue; they are just passed from the testbench into the model's queue. The model then processes the commands from the internal model queue as simulation time advances.

Commands that are loaded into the model's queue are guaranteed to execute in the order they were originally queued. So, if commands "A," "B," and "C" were sent from the testbench to the model in that order, the model would process them in "A-B-C" order.

The following figure shows a group of cycle commands (in this case, read and write) and zero cycle commands (set_config_param and enable_msg_type), and the relationship between the time when the testbench sends the commands to the model and when the commands are executed.

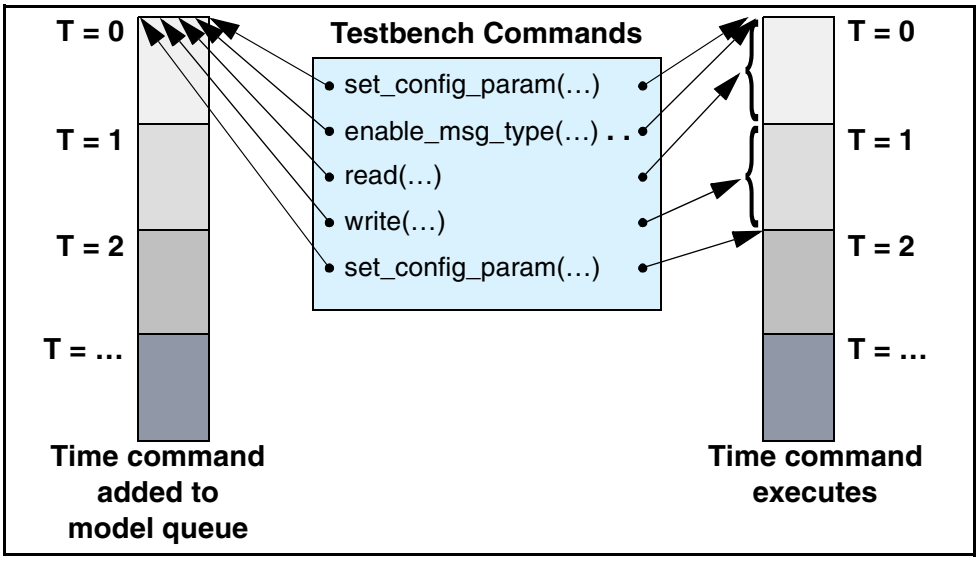


Figure 4: Model Command Queuing

In the figure, you can see that all of the cycle and zero cycle commands are sent to the model at time zero. Since each of the cycle commands takes one clock to execute, the last set_config_param is read by the model at time zero, but it does not take effect until time 2 because of the guaranteed execution order.

Model queuing allows the testbench command stream to be decoupled from model command execution. This enables pipelined operations as well as split transactions.

Command queuing can be turned on or off with the VMT_FORCE_CMD_BLOCKING model configuration parameter. Turning this parameter on causes commands not to be queued, which means pipelined operations are not possible because the model operates on each command individually. When command queuing is off, the command stream is blocked for every command until current command execution is complete.

Command Channels and Command Streams

As previously noted in the VMT Command Flow illustration, a *command channel* is a conceptual structure in a VMT model that manages the following:

- Communication with the testbench through a command stream
- Execution of the model commands in the command execution engine

The number of command channels for a VMT model is hard coded and cannot be changed. Most VMT models have only one command channel, which meets the needs of most protocols. Models that support full-duplex protocols can have two command channels, one for receiving and one for transmitting.

A *command stream* is a communication path between the testbench and the VMT model. When a VMT model is instantiated in a design, it has one initial command stream for each command channel. You can create one or more additional command streams, which is a technique you might use if your testbench needs to perform tasks similar to interrupt handling.

A command channel can execute commands from only one command stream at a time. To execute two or more command streams concurrently, multiple command channels must be used (for example, modeling full duplex send and receive capability in a UART). Each command channel runs independently from other command channels. If one command channel's stream is blocked, streams in other command channels are not affected. Therefore, each command channel can run concurrently with other command channels.

The new_stream command is used to generate new command streams. The first argument in the new_stream command refers to the command channel. Every new *stream ID* generated is unique and corresponds to a particular command channel.

Attention

You must *never* send commands to the same instance/stream ID from different testbench threads. Command execution order, results, and stability are unpredictable if this rule is not followed.

Issue a new_stream command whenever your testbench spawns a new thread that will issue model commands to an instance while another active testbench thread is concurrently issuing model commands to the same instance.

Each command stream has a unique stream ID identifier. A VMT constant, VMT_DEFAULT_STREAM_ID, is used to identify the default command stream. Always use the VMT_DEFAULT_STREAM_ID command stream when sending commands to a model before issuing start or new_stream commands.

Attention

If you use models that use multiple command channels, such as the SIO TxRx model, you must use a different default stream ID to identify the default command stream for each command channel.

All command streams continue to send commands to the model queue until a blocking command is executed or a control mechanism, such as #10 in Verilog, is used to cause simulation time to advance. There are several common VMT commands that are blocking commands (block_stream and any

command that returns a result, such as get_config_param). These commands allow synchronization between the actual simulation time a command is executed and the command stream. These blocking commands halt loading the model queue with more commands.

Using the block_stream command causes commands to be blocked until all of the previously-loaded commands complete execution. For commands that return results, the queue is blocked until the result is available and can be returned to the testbench. When these commands are used, pipeline operations may be starved of new commands in the queue and are essentially filled with idle cycles.

Each model has one default command stream per command channel. VMT models do not allow multiple command streams to run concurrently on one command channel. However, command streams can be used as interrupt handlers. One such example is a command stream that executes when a watchpoint trigger occurs.

A new command stream can be created dynamically from the new_stream command. This must be paired with an end_stream as the last command in the stream.

When any new command stream is created, a new command queue is created in the model. When a command stream is created by the new_stream command, commands execute from this stream until the command stream completes or another stream is created. The model always processes from the newest command stream and works back to the oldest.

The following figure shows an example of how a VMT model would handle two newly-created command streams, occurring at different times. Each stream is written in its own concurrent process/fork block.

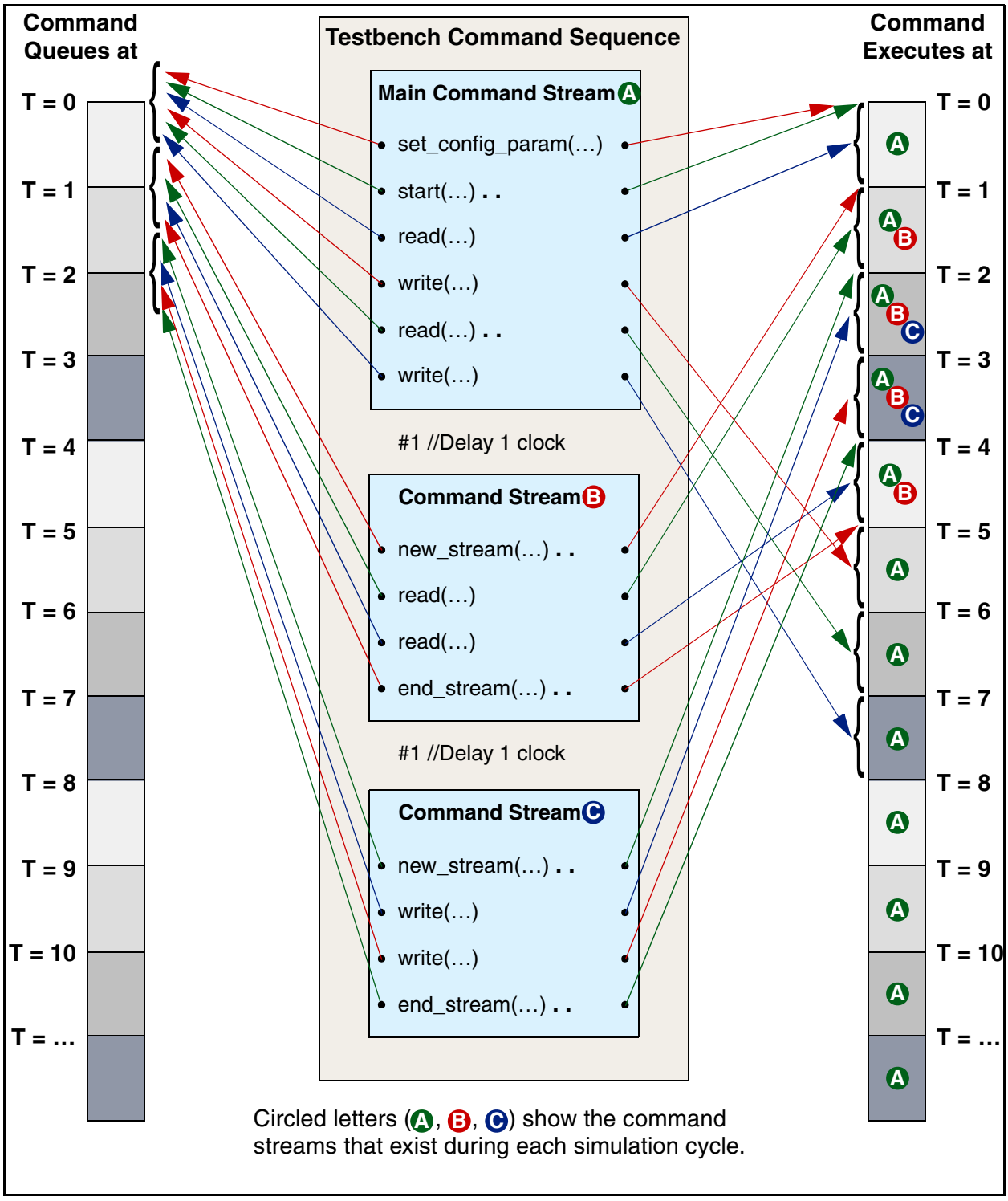


Figure 5: Using Multiple Command Streams

As shown in the preceding figure, all of the reads and writes take just one clock cycle to complete execution. As shown, at time 1, command stream B sends a new_stream command to the model. This creates a new queue of commands from which the commands will execute. It should also be noted that because none of the commands in stream B are blocking, all of them are loaded into stream B's command queue at time 1. However, only the first read is executed.

At time 2, command stream C sends a new_stream command to the model, creating three concurrent command queues. Because stream C is the newest, the model now switches and processes all commands from command stream C's queue. At time 4, command stream C executes its end_stream command and terminates the command stream queue in the model. Now the newest command queue in the model the one belonging to command stream B, so the model resumes processing commands from this queue.

Command stream B executes its end_stream at time 5, and then the main command stream resumes executing the rest of the commands in its queue.

For usage consistency, all models have a command queue. Most commands have a first argument of a streamID. The stream IDs relate a command to a specific command stream, for example, an interrupt handler command stream as opposed to the main command stream. Also, the main command stream must have a start command before any other command streams are spawned.

Summary

In conclusion, here are some things to remember about VMT models and commands:

- All non-queued commands are zero cycle commands.
- Every command stream has one and only one queue associated with it. Likewise a command queue is associated with one and only one command stream.
- Only one command queue per command channel will be active at a time. The active queue is either the most recently created, or it's predecessor once the stream associated with it has been closed.
- The command that "opens" a new command stream between the testbench and the model (new_ stream), also creates a new command queue, which becomes the active command queue.
- All command streams can and will accept commands at all times, not just the command stream that is associated with the active command queue. However, command execution is based on the command stream's blocked status.
- All commands can be blocking commands if a VMT configuration parameter (VMT_FORCE_ CMD_BLOCKING) is ON.
- All command channels operate independently of each other, so all command channels can be executing commands simultaneously.

2
Using VMT Models

Chapter Contents

Controlling Messages

All VMT models use common message services to output all messages. These services allow you to:

- **Specify message destination** – You may route messages to one or more files, a simulator transcript window, or any combination of these.
- **Enable/disable categories of messages or specific messages** – Specific message IDs and most message types can be independently enabled or disabled. The only exceptions are messages that are Fatal, which are always enabled.
- **Format messages** – You can customize message formats to define what fields of information each message contains. Fields of information in messages are called message *features*.

The commands that allow you to enable, route, and configure messages are as follows:

• enable_msg_type disable_msg_type	Control the message types you want reported and whether to output messages to a simulator transcript, log file, or both.
• enable_msg_feature disable_msg_feature	Control the output of message fields. Use these commands to customize the format and data contained in messages reported to a simulator transcript, log file, or both.

• enable_msg_id disable_msg_id	Enable or disable one specific message reporting to either a log file, simulator transcript, or both. Use these commands to "fine-tune" message reporting when you want to see only one or a few messages of a particular type, or when you want to ignore only one or a few messages of a particular type.
• enable_type_ctrl_msg_id	Resets settings of a specific message to the settings for messages of that type. Use this command to return a particular message to its default reporting defaults.
• open_msg_log close_msg_log	Open or close a message log file. Use these commands to manage a message log file.
• enable_msg_log disable_msg_log	Enable or disable message output to a message log file, transcript window, or both. Use these commands to start and stop message reporting in a transcript or log file.

Defaults for Message Types and Features

Messages can be routed to transcript windows and message log files. Each of these destinations have two sets of defaults: one for message types and one for message features. These defaults are listed in the following sections:

- Defaults for Transcript Windows
- Defaults for Log Files

To change which message types are enabled or to restore the default settings, use the enable_msg_type command. To change which message features are enabled or to restore the default settings, use the enable_msg_feature command.

Defaults for Transcript Windows

For simulator transcript windows, the default message types are:

- Fatal
- Error
- Warning

Also, the default message features are:

- Descriptive Text
- Type
- Instance Name
- Simulator Time
- Text
- Arguments

Transcript example:

```
Designware Model Error from top.U1 at 1100:
    During Reset HWDATA invalid value.
```

Defaults for Log Files

For log files, the default message types are:

- Fatal
- Error
- Warning
- Protocol Transaction

Also, the default message features are:

- Message Type
- Simulator Time
- Text
- Arguments

Log file example:

```
Protocol Transaction Started at 2850 Completed at 2950:
WRITE 32BIT SINGLE
HADDR   : 000000FC
HWDATA  : 00000067
RESP    : OK
```

Controlling Message Routing

Message routing is controlled through the enable_msg_log and disable_msg_log commands. These commands allow you to output model messages to any file you choose. Messages can be appended to a file or the file can be overwritten. If you do not specify a file name, the name will be *inst_name*.msg. You should use the msg suffix on all message log files so as not to confuse them with the model replay logging files.

The enable command returns a msg_logID that is used in conjunction with the message filtering commands (enable_msg_type and disable_msg_type) to allow you to selectively control which types of messages are routed to which files. So, for example, all error messages could be routed to an error.msg file.

Different message formatting can be done for each file as well, using the msg_logID and message format configuration commands (enable_msg_feature and disable_msg_feature).

Controlling Message Format and Configuration

To control the look and feel of model messages, each part of the messages are broken up into different *features*. Individual features can be enabled or disabled using the enable_msg_feature and disable_msg_feature message format configuration commands. These commands enable you to turn on and off simulation time, instance name, message ID, and other formatting so you can create message log files that contain only the information you want, and makes the log files much easier to parse or view.

Like the messaging filtering, the msg_logID is used to identify message log files. Predefined constants VMT_MSG_ROUTE_SIM or VMT_MSG_ROUTE_ALL are used for the simulator transcript window or combined simulator transcript and log file filtering, respectively.

Here is an example showing message formatting commands and results:

1. Message output before changes:

```
Designware Model ERROR [AHB_MASTER_ERRMVALID] from top.U1 at 1100:
   All outputs, except HWDATA, must be valid (not 'X') at the rising
   edge of HCLK.
```

2. Turn off the simulation time for all messages:

```
U1.disable_msg_feature(streamID, VMT_MSG_SCOPE_ALL, VMT_MSG_SIM_TIME,
    msg_logID) ;
```

   Message output after change:

```
Designware Model ERROR [AHB_MASTER_ERRMVALID] from top.U1:
   All outputs, except HWDATA, must be valid (not 'X') at the rising
   edge of HCLK.
```

3. Turn off the message ID for messages of type error:

```
U1.disable_msg_feature(streamID, VMT_MSG_SCOPE_ERROR, VMT_MSG_ID,
    msg_logID) ;
```

   Message output after change:

```
Designware Model ERROR from top.U1:
   All outputs, except HWDATA, must be valid (not 'X') at the rising
   edge of HCLK.
```

4. Disable the base format text for the AHB_MASTER_ERRMVALID message:

```
U1.disable_msg_feature(streamID,AHB_MASTER_ERRMVALID, VMT_MSG_DESC, msg_logID) ;
```

   Message output after change:

```
ERROR top.U1 All outputs, except HWDATA, must be valid (not 'X') at
the rising edge of HCLK.
```

5. Make AHB_MASTER_ERRMVALID even shorter - disable the message text:

```
U1.disable_msg_feature(streamID, AHB_MASTER_ERRMVALID, VMT_MSG_TEXT,
    msg_logID) ;
```

   Message output after change:

```
ERROR top.U1 HWDATA HCLK.
```

6. Enable message ID and time for the AHB_MASTER_ERRMVALID message. Note the use of bitwise OR (|) to specify multiple features:

```
U1.enable_msg_feature(streamID, AHB_MASTER_ERRMVALID,
    VMT_MSG_ID|VMT_MSG_SIM_TIME);
```

   Message output after change:

```
ERROR AHB_MASTER_ERRMVALID top.U1 1100 HWDATA HCLK.
```

7. Return everything to its default setting:

```
U1.enable_msg_feature(streamID, VMT_MSG_SCOPE_ALL,
    VMT_MSG_FEATURES_DEFAULT|VMT_MSG_FEATURES_LOG_DEFAULT, msg_logID) ;
```

   Message output reverts to the original format:

```
Designware Model ERROR [AHB_MASTER_ERRMVALID] from top.U1 at 1100:
   All outputs, except HWDATA, must be valid (not 'X') at the rising
   edge of HCLK.
```

Message Filtering

Message types can be turned on or off for a particular routing by using the enable_msg_type and disable_msg_type message filter commands. For enabling and disabling types for message log files, the msg_logID is used. For enabling and disabling types for the simulator transcript window, the predefined constant VMT_MSG_ROUTE_SIM is used. Another predefined constant, VMT_MSG_ROUTE_ALL_LOGS, controls filtering for log files. A third constant, VMT_MSG_ROUTE_ALL, controls combined simulator transcript and log file routing.

Specific messages can also be filtered. To filter a specific message, you use the disable_msg_id command and specify the message ID and a msg_logID. To obtain the ID of a message, enable the message ID feature with the enable_msg_feature command, or search through the message IDs that are in the documentation for your specific model and the list of general VMT messages.

Message Types

This section describes the different message types and their conventions. Display of messages by message type is controlled by the enable_msg_type and disable_msg_type commands.

Consult model-specific documentation for listings and short descriptions of all model messages.

Note

Not all models support all message types.

Fatal, Error and Warning Messages

Fatal messages report problems that halt the simulation. Fatal messages cannot be disabled.

Error messages report problems with the model, its configuration, or its environment, but from which the model can recover and resume simulation. Error messages can be disabled.

Warning messages report conditions that can indicate incorrect settings or data, but are not errors. Warning messages can be disabled.

By default, Error and Warning message output is enabled in simulator transcripts and in log files.

Timing and X-Handling Messages

Timing messages report timing problems; such as setup and hold violations. X-handling messages report unknown values on model input ports, when the model substitutes a default value on the port.

By default, Timing and X-handling message output is disabled in simulator transcripts and in log files.

Note Messages

Note messages are information regarding the model interaction with testbench control. These messages describe what is happening within the model. The following is an example of a note message:

```
Note U1.top 1400:
    WatchPoint "Failed Read Expect" Triggered.
```

By default, Note message output is disabled in simulator transcript windows and in log files.

Report Messages

Report messages contain detailed, model-specific information requested by the testbench. They allow you determine what has happened in a model. By default, Report message output is enabled in simulator transcript windows and in log files.

Testbench Notification Messages

A notify message signals a testbench of an event of interest, but does not generate simulator transcript or log file output. Although these messages never print any output, they do trigger watchpoints. Notify messages support all watchpoint data retrieval commands.

By default, Notify message output is disabled in simulator transcript windows and in log files.

Protocol Cycle Messages

Protocol messages always have some reference to actual protocol that the model represents. For Protocol Cycle messages, the messages are reported on a cycle-by-cycle basis and give relevant protocol information.

Here are four examples of protocol cycle messages:

```
Protocol Cycle U1.top 1100:
    Start Burst Read to address: (00000B00)
Protocol Cycle U1.top 1200:
    Data Beat 2 to address:      (00000B04)
Protocol Cycle U1.top 1300:
    Data Beat 3 to address:      (00000B08)
Protocol Cycle U1.top 1400:
    Data Beat 4 to address:      (00000B0C)
```

By default, Protocol Cycle message output is disabled in simulator transcript windows and in log files.

Protocol Transaction Messages

Protocol Transaction messages report relevant protocol information about a transaction and appear at transaction boundaries. Because Protocol Transaction messages are only output on transaction boundaries, they are usually associated with monitor models. Protocol Transaction message information for a transaction is buffered until the transaction is complete or terminates with some irregularity, then a single, formatted messages is given for that transaction.

Here is an example of a protocol transaction messages:

```
Protocol Transaction U1.top 1400:
         Burst Read:
               Address     : (00000B00)
               Transfer Size: 4 Word Burst
               Transfer Type: INCR
```

By default, Protocol Transaction message output is disabled in simulator transcript windows and enabled in log files.

Protocol Error Messages

Protocol Error messages appear when a protocol error is detected and give relevant protocol information. Here is a Protocol Error message example:

```
Protocol Error U1.top at 1100:
    Burst Read to invalid address: (DEADBEEF)
Protocol Error U1.top at 1200:
    Malformed Packet in burst transfer to address : (00000B04)
Protocol Error U1.top at 1300:
    Invalid Packet Type in Data Beat 3 to address:      (00000B08)
Protocol Error U1.top at 1400:
    Invalid Response to bus request :  FF
```

By default, Protocol Error message output is disabled in simulator transcript windows and enabled in log files.

Command Messages

Command messages display information about the commands the model is executing. These messages are intended to help you debug testbenches and verify command execution. Command messages give useful information about model commands at the time the command is queued, executing, and completed.

As the following example shows, command messages can report:

- Instance name
- Simulation time
- Queued, executing, or completed command status
- Command name and a unique identifying tag
- Command argument data passed to and from the command

When you enable Command messages, the model reports queued, executing, and completed messages. You may choose to report only one or two of these actions. Use the enable_msg_id or disable_msg_id commands to enable or disable the specific Message IDs for queued, executing, or completed commands. Use:

- VMT_MSGID_CMD_QUEUED for queued commands
- VMT_MSGID_CMD_EXECUTE for executing commands
- VMT_MSGID_CMD_DONE for completed commands

Note

The enable_msg_id or disable_msg_id commands *types* argument accepts the VMT_MSG_ALL constant, that allows you to enable or disable all message types *except* Command messages. You must enable or disable Command messages explicitly.

Command messages do not trigger watchpoints. If you attempt to create a watchpoint for a Command message ID, the model generates a VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT Error message.

Here are two Command message result examples for the same command, with explanations of the result differences:

```
DesignWare Model COMMAND [VMT_MSGID_CMD_EXECUTE] from
prog_shell.uhca_ahb_mon at 51235000:
    Command Execute: get_watchpoint_data_string : (Tag = 758)
       In Handle   dataHandle: 757
              In dataPosition: 1010
              In     dataLine: 0
             Out    dataValue:
             Out       status: -1
```

In the previous example, the get_watchpoint_data_string command had started to execute. In the next example, the get_watchpoint_data_string command has completed. There are two important differences between these command messages:

1. The *dataValue* output argument value is a null string in the "Execute" example above and contains "OKAY" in the "Done" example below, because, in the previous example, the value had not yet been computed.
2. The *status* output argument value is preset to -1 in the "Execute" example and contains 4 in the "Done" example. If you need to extract data from a command message, parse output argument from "Done" command messages only.

```
DesignWare Model COMMAND [VMT_MSGID_CMD_DONE] from
prog_shell.uhca_ahb_mon at 51235000:
    Command Done: get_watchpoint_data_string : (Tag = 758)
       In Handle   dataHandle: 757
              In dataPosition: 1010
              In     dataLine: 0
             Out    dataValue: OKAY
             Out       status: 4
```

Note

Not all models implement VMT_MSGID_CMD_EXECUTE messages for all commands. Model-specific Command messages require updates to the model.

By default, Command message output is disabled in simulator transcript windows and in log files.

Watchpoints

Watchpoints give you the ability to control testbench timing based on significant events in simulation. A watchpoint is a construct that triggers on a user-selected event. In turn, a testbench can watch for a watchpoint to be triggered. This gives the testbench the ability to sense trigger events and then gate code until the trigger occurs.

Each VMT model provides an extensive list of possible trigger events for watchpoints, which include:

- Message type (message types are listed in the Predefined Message Types Table that appears in the description of the enable_msg_type command.)
- Message ID
- Notification ID

When creating a watchpoint, you choose which of these will be the trigger event. Some events are generic in that they apply to testing in general, and others are related to specific key events in a protocol.

For example, all warning-type messages that are generated by a model can be used as a watchpoint trigger event. This allows the testbench to detect when a warning condition occurs, after which it can perform some action. Another example is a specific notification message that is issued at the end of a reset sequence. When used as a trigger event, this notification allows the testbench to know, dynamically, when reset is complete and that it is time to start sending protocol traffic. Note that message types and message IDs can be used as trigger events regardless of whether they are enabled (displayed).

In addition to controlling testbench timing, many watchpoints provide a way to query the trigger event or the action/transaction that caused the trigger event. For example, you could set a watchpoint on the end of a particular transaction type. Then, when the watchpoint is triggered, the testbench could extract the data payload that was transferred. Exactly what information is available is a function of the trigger event. The data remains available until the end of the cycle in which the event triggered (to help manage memory resources.)

The next section explains how to set watchpoints and access this information.

Using Messages as Watchpoints

You can create watchpoints for specific messages or message types, and then investigate the associated trigger event. To create a watchpoint, you need to specify the message type or message ID, which you can find in the message tables or lists in the reference portion of the model documentation.

The structure of messages includes fields that contain specific information about the associated event. You can get this information from the model during your simulation and then use it to control your testbench. As previously stated, the data remains available until the end of the cycle in which the event triggered.

All model messages, *except* command messages, generate events that can be captured in a testbench with the watch_for command. When triggered, watch_for returns a handle to a data object that contains message-specific data, including the message type, ID, and the message's data fields (if any).

The following table shows message event data available for messages.

Table 2: Message Event Data

Message Data ID	Data Type	Description
VMT_MSG_EVENT_ARG_MSG_TYPE	Integer	Message type. See the enable_msg_type command reference for a complete list of all message types. Use the get_watchpoint_data_int command to obtain the message type.
VMT_MSG_EVENT_ARG_MSG_ID	Integer	Message ID. See "VMT Messages" or model-specific message tables for lists of all message IDs. Use the get_watchpoint_data_int command to obtain the message ID.

Table 2: Message Event Data (Continued)

Message Data ID	Data Type	Description
<msg_id>_ARG_<data_arg>	Integer String Bit	Message-specific data field values. The data type varies for different data fields. See "VMT Messages" or model-specific message tables for data fields and their respective data types. Use the get_watchpoint_data_int, get_watchpoint_data_string, or get_watchpoint_data_bit commands to capture the message data field value, depending on data type. **Example:** VMT_MSGID_CFG_ILLEGAL_PARAM_ARG_PARAM_NAME allows you to capture the *param_name* data field of the VMT_MSGID_CFG_ILLEGAL_PARAM message event.

The following example shows how to set a watchpoint on the MODEL_MSGID_DATA_MISMATCH message, which is a fictional message that would be generated when a data mismatch event is detected. When setting a watchpoint on a message ID, locate the message ID in the message tables. The message tables contain the information you need to create the watchpoint.

The entry for this message appears next so you can see how to use the information to create a watchpoint and then get the message information when the watchpoint triggers.

MODEL_MSGID_DATA_MISMATCH
The received data byte does not match the expected data byte.
Msg Type: VMT_MSG_ERROR
Fields:

```
MODEL_MSGID_DATA_MISMATCH_ARG_RCV_DATA
    Type: bit[7:0]
MODEL_MSGID_DATA_MISMATCH_ARG_EXP_DATA
    Type: bit[7:0]
MODEL_MSGID_DATA_MISMATCH_ARG_BYTE_COUNT
    Type: integer
MODEL_MSGID_DATA_MISMATCH_ARG_TRANS_NAME
    Type: string
MODEL_MSGID_DATA_MISMATCH_ARG_CYCLE_NUMBER
    Type: integer
```

As shown, MODEL_MSGID_DATA_MISMATCH is a VMT_MSG_ERROR type and is generated each time a data mismatch event occurs. This message includes five related pieces of information about the event, which are listed below the Fields label. The information includes the data field name for each piece of information, followed by the data type.

1. Using the Message ID, create the watchpoint:

   ```
   create_watchpoint('VMT_MESSAGE_ID, 'MODEL_MSGID_DATA_MISMATCH, wp_handle);
   ```

   The watchpoint is created and returns the watchpoint identification handle.

2. Watch for the data mismatch event to occur:

   ```
   watch_for(wp_handle, wp_data_handle);
   ```

   When the data mismatch occurs, this command returns the event identification handle. Note that the watch_for command is a blocking command, which you can learn more about in Queued and Blocking Commands.

3. After the watchpoint event occurs you can get the data, and this is when the data Fields and data Type information from the listing is needed. In the following examples, notice how the command names match the data Types, and how the data Fields are specified as the *position* argument in each get command.

```
get_watchpoint_data_int(wp_data_handle, 'VMT_MSG_EVENT_ARG_MSG_ID, msg_id_value,
status);
get_watchpoint_data_vec_8(wp_data_handle, 'MODEL_MSGID_DATA_MISMATCH_ARG_RCV_DATA,
word0, data_value1, status);
get_watchpoint_data_vec_8(wp_data_handle, 'MODEL_MSGID_DATA_MISMATCH_ARG_EXP_DATA,
word0, data_value2, status);
get_watchpoint_data_int(wp_data_handle, 'MODEL_MSGID_DATA_MISMATCH_ARG_BYTE_COUNT,
data_value3, status);
get_watchpoint_data_string(wp_data_handle, 'MODEL_MSGID_DATA_MISMATCH_ARG_TRANS_NAME,
line0, data_value4, status);
get_watchpoint_data_int(wp_data_handle, 'MODEL_MSGID_DATA_MISMATCH_ARG_CYCLE_NUMBER,
data_value5, status);
```

Using Notifications as Watchpoints

Some VMT models provide a set of notifications for obvious protocol and model events, such as the start or end of a transaction. A *notification* is a type of message that is used exclusively for watchpoints and is not displayed in the simulation transcript. You can use notifications in watchpoints exactly how you use messages.

The message type for notification messages is VMT_MSG_NOTIFY, and you can use this type to set a watchpoint on all notification messages. You can also use the message ID of specific notifications as watchpoints. Just like the structure of messages, notifications include fields that contain specific information about the associated event. You can get this information from the model during your simulation and then use it to control your testbench.

You can find notifications in tables or lists in the reference portion of the model documentation. To use a notification as a watchpoint, you can follow the examples in the "Using Messages as Watchpoints" section.

Managing Result Data

Users need to manage the storage of results to reduce model memory requirements. Typically, when the model executes a command that has an associated result, a handle to the result is returned and control is immediately returned to the testbench. It is up to the user to decide what to do with the results.

To prevent memory leakage, you can use one of the following methods:

- Tell the command to ignore the results. Use this technique when you know that you will not need the results.

  a. Set a variable equal to "VMT_IGNORE"

  b. Specify this variable as the return handle when you issue the result-generating command

  By passing in a variable set to the value of VMT_IGNORE as the result handle, the model does not store any results generated by the command. No memory is allocated. The value of the result handle that is returned is the same as the value that was passed in (that is, VMT_IGNORE).

Note

In the following examples, <model_inst> represents the full hierarchical path to the instance of the VMT model.

Example:

```
// Results of the read will not be stored by the model
integer ignore = VMT_IGNORE;
<model_inst>.read(streamID, 32'h12344321, ignore);
```

- Use the corresponding result command to get the result.

  By using the model-specific result command to actually get the result data, the model automatically frees the associated memory. This means that you may use the result handle only once to get the data, and subsequent calls with the same result handle will be considered an error.

  Example:

```
// Results of the read will be returned to the testbench and
// cleaned up in the model
integer rslt_handle;
<model_inst>.read(streamID, 32'h12344321, rslt_handle);
<model_inst>.read_result(streamID, rslt_handle, rslt_data);
```

- Explicitly delete the memory that is associated with a result handle. To do this, you use the delete_handle command.

  This last method explicitly deletes the memory associated with a handle. You can either delete results for a specific handle by passing that handle as the parameter, or delete all handles by passing VMT_ALL as the argument.

  Example:

```
// Three result handles are allocated, one for each read
integer rslt_handle;
<model_inst>.read(streamID, 32'h12344321, rslt_handle);
<model_inst>.read(streamID, 32'hA5A5A5A5, rslt_handle);
<model_inst>.read(streamID, 32'hDEADBEEF, rslt_handle);
// Clean up done for the result handle to read at DEADBEEF
<model_inst>.delete_handle(streamID, rslt_handle);
// Clean up done for all result handles
<model_inst>.delete_handle(streamID, VMT_ALL);
```

Using Stream Blocking Commands

Any model command that actually causes simulation time to advance before it returns from the model is defined as a "blocking" command. Each command description defines whether a command is blocking or non-blocking. Further, any command can be configured to be blocking by setting the VMT_FORCE_CMD_BLOCKING configuration parameter to ON.

Command blocking is used when the command stream needs to be synced up with other testbench control signals, a branching decision needs to be made before more commands are sent to the model, or a result is needed from the model. In both of the examples below, the model commands are put in a for loop along with a testbench task that drives a signal on the slave. This testbench driver task takes no simulation time. For the testbench to operate correctly, the driving of the signal on the slave must happen after each write is complete.

In the first example, all of the tasks in the for loop take no simulation time, so the entire for loop is completed in zero simulation time. When the loop is done, there are 10 write/read pairs in the command queue for the apb_mstr1 model. However, when simulation advances and starts executing these writes and reads, the testbench signal on the slave will not be driven because the for loop is done executing.

```
// Loop is executed in 0 simulation time
for ( i=0; i < 10; i=i+1) begin
    apb_mstr1.write (streamID,  'GPIO_ADDR, 'BASE_DATA + i);
    apb_slave_tb_driver = i; // This drives a test bench signal on slave
    apb_mstr1.read (streamID, 'GPIO_ADDR, rslt_handle);
end
```

In the second example, the block_stream command after the write causes simulation time to advance while the model waits for the write to complete, thus syncing up the driving of the slave testbench signal after the write is complete. This same effect could be achieved by putting any blocking command in the for loop, such as a read_result command after the read.

```
// Loop is executed over many cycles because block_stream
for ( i=0; i < 10; i=i+1) begin
    apb_mstr1.write (streamID,  'GPIO_ADDR, 'BASE_DATA + i);
    apb_mstr1.block_stream (streamID,  0, status);
    apb_slave_tb_driver = i; // This drives a test bench signal on slave
    apb_mstr1.read (streamID, 'GPIO_ADDR, rslt_handle);
end
```

Creating Pipelined Command Streams

VIP-specific commands that return results (such as get_result) are blocking, and need to block the command stream until the result is returned to the testbench. This allows for branching decisions to be made on the results that are returned. For models that have pipelined protocols, these commands will break the pipeline if the result is not immediately available. The following two examples show how a model with a pipeline depth of two can read results, either breaking the pipeline or not. In the first example a read followed immediately by a result command breaks the pipeline. In the second example, the result is delayed until the result is available, thus the pipe is not broken.

First Example:

```
// Breaks pipeline so that each read's data and address phases will not overlap
ahb_mstr1.read (streamID,  addr_A, rslt_handle_A);
ahb_mstr1.get_result (streamID,  rslt_handle_A, rslt_data_A);
ahb_mstr1.read (streamID,  addr_B, rslt_handle_B);
ahb_mstr1.get_result (streamID,  rslt_handle_B, rslt_data_B);
ahb_mstr1.read (streamID,  addr_C, rslt_handle_C);
ahb_mstr1.get_result (streamID,  rslt_handle_C, rslt_data_C);
```

Second Example:

```
// Pipeline is not broken because read result is delayed
ahb_mstr1.read (streamID,  addr_A, rslt_handle_A);
ahb_mstr1.read (streamID,  addr_B, rslt_handle_B);
ahb_mstr1.get_result (streamID,  rslt_handle_A, rslt_data_A);
ahb_mstr1.read (streamID,  addr_C, rslt_handle_C);
ahb_mstr1.get_result (streamID,  rslt_handle_B, rslt_data_B);
ahb_mstr1.read (streamID,  addr_D, rslt_handle_D);
ahb_mstr1.get_result (streamID,  rslt_handle_C, rslt_data_C);
```

Resetting Models

Note

Not all VMT-based models have reset capabilities. Consult the model documentation for availability and details.

You may wish to reset a model during simulation. "Reset" can mean either resetting the model itself through a *model-specific* reset, or resetting the model's environment in the testbench. VMT models that have implemented reset capabilities can perform both environment resets and model-specific resets. Specifically, these models can perform three types of environment resets and a device reset:

Soft Reset A soft reset clears a model's command queue and brings the model to a known state. The soft reset does not change configuration settings or remove watchpoints. You are not allowed to issue a start command after a soft reset.

Firm Reset A firm reset clears command queues and deletes all watchpoints. You use a firm reset when you want to retain the current model configuration, but clear all watchpoints and the command queue. Since all watch_for commands in the testbench are triggered by a firm reset, you must trap the VMT_WP_TERMINATED_BY_RESET handle returned when the watch_for is cleared in all "watch_for" routines to run post-reset routines or to clean up unwanted testbench threads.

Hard Reset A hard reset clears command queues, deletes watchpoints, clears all configuration settings to their defaults, and removes all message log file handles. You use a firm reset when you want to get the model to the same state it was in just after instantiation. New configurations can be set, then a model start command must be re-issued.

Note

Soft, firm, and hard resets do not perform any device reset functions.

Model-specific Reset A model-specific reset or protocol reset typically performs the actions associated with device reset pin assertion. Such model-specific actions could include: clearing or deallocating device memory, clearing buffers or FIFOs, or resetting internal counters and status registers. Model-specific resets are defined by modeled device specifications and uniquely implemented by each model. A model-specific reset may also perform some or all of the actions of a soft, firm, or hard reset.

Memory Patterns

In addition to constant fill commands available to some VMT models, VMT provides the nine pattern fills described in the following table. These patterns are used to specify default fill patterns for memory spaces or buffers.

Table 3: VMT Memory Patterns

Pattern Name	Default Initial Value	Description
VMT_MEM_PATTERN_ZERO		
	—	Sets all bits in the region to 0.
VMT_MEM_PATTERN_ONE		
	—	Sets all bits in the region to 1.
VMT_MEM_PATTERN_A5		
	—	Sets all bytes in the region to 0xA5 (1010 0101).
VMT_MEM_PATTERN_5A		
	—	Sets all bytes in the region to 0x5A (0101 1010).
VMT_MEM_PATTERN_X		
	—	Sets all bits in the region to X.
VMT_MEM_PATTERN_WALK0		
	width`'b0`	Sets all words in the region to a walking 0 pattern. If the initial value is not valid (all ones except one bit zero), the command issues a message. **Example:** For a 16-bit pattern with initial value FFFE: ● The starting pattern is all bits of the first word set to 1 except the LSB. ● The next pattern word has the 0 moved 1 bit toward the MSB (FFFD). ● The third pattern word has the 0 moved another bit toward the MSB (FFFB).
VMT_MEM_PATTERN_WALK1		
	width`'1`	Sets all words in the region to a walking 1 pattern. If the initial value is not valid (all zeros except one bit one), the command issues a message. **Example:** For a 16-bit pattern with initial value 0001: ● The starting pattern is all bits of the first word set to 0 except the LSB. ● The next pattern word has the 1 moved 1 bit toward the MSB (0002). ● The third pattern word has the 1 moved another bit toward the MSB (0004).

Table 3: VMT Memory Patterns (Continued)

Pattern Name		
	Default Initial Value	**Description**
VMT_MEM_PATTERN_INCR		
	0	Sets all words in the region to an incrementing pattern.
VMT_MEM_PATTERN_DECR		
	$2^{width-1}$	Set all words of region to a decrementing pattern.

3 VMT Common Command Reference

Chapter Contents

Command Summary

Table 4 on page 38 contains three columns, labeled "Queued," "Blocking," and "Zero Cycle." VMT commands have different behaviors depending on whether or not they are queued, blocking, and zero cycle:

- **Queued commands** are placed on a command queue and may not be executed immediately when received. However, commands always execute in the order they were sent to the model from a particular command stream. Configuration and "set…" commands are examples of queued commands.
- **Blocking commands** prevent other commands from being passed from the testbench to the command queue until they have finished executing. All commands that return results are blocking commands, except those commands that return results handles.
- **Zero cycle commands** do not advance simulation time. All common commands and all commands that change or read settings are zero cycle commands

For more information on queued and blocking commands, see the "Queued and Blocking Commands" discussion.

Attention

Not all common commands are available for every model. Consult specific model documentation to see which common commands are available.

Except where noted, the commands in Table 4 can be used in OpenVera, Verilog, and VHDL testbenches.

Table 4: VMT Common Command Summary

Command Name	Queued	Blocking	Zero Cycle	Description
Command Streams				
start	N	N	Y	Starts model execution.
block_stream	N	Y	Y	Blocks the current command stream.
new_stream	N	N	Y	Starts a new command stream associated with a specified command channel. Returns the Stream ID of the new command stream.
end_stream	Y	N	Y	Ends execution of the command stream started by the new_stream command.
reset_model	N	N	Y	Resets the model.
Model Configuration Parameters				
set_config_param	Y	N	Y	Changes a specified configuration parameter value.
get_config_param	Y	Y	Y	Reads a specified configuration parameter value.
Ports				
set_port	Y	N	Y	Drives a value onto a specified port.
get_port	Y	Y	Y	Reads a value from a specified port.
Registers				
set_register	Y	N	Y	Changes the value of a specified model register.
get_register	Y	Y	Y	Reads the value of a specified model register.
Memory				
delete_handle	Y	N	Y	Deletes result handle, freeing result data memory.
Model Version				
get_version	N	Y	Y	Returns the current version of the model.

Table 4: VMT Common Command Summary (Continued)

Command Name	Queued	Blocking	Zero Cycle	Description
Message Handling				
enable_msg_type	Y	N	Y	Enables one or more message types from a specified model instance.
disable_msg_type	Y	N	Y	Disables one or more message types from a specified model message routing log file.
open_msg_log	Y	Y	Y	Enables message output to a message log file.
close_msg_log	Y	N	Y	Disables message output to a message log file and closes the file.
enable_msg_log	Y	N	Y	Enables message output to a message log file or simulator transcript window.
disable_msg_log	Y	N	Y	Disables message output to a message log file or simulator transcript window.
enable_msg_id	Y	N	Y	Enables a specific message to a message log file or simulator transcript window.
disable_msg_id	Y	N	Y	Disables a specific message to a message log file or simulator transcript window.
enable_type_ctrl_msg_id	Y	N	Y	Resets settings of a specific message to the settings for messages of that type.
enable_msg_feature	Y	N	Y	Enables a user-defined message format in model messages.
disable_msg_feature	Y	N	Y	Disables a user-defined message format in model messages.
Message Display				
print_msg	Y	N	Y	Prints a text message in a simulation transcript.
Watchpoints				
watch_for	N	Y	Y	Blocks the current command stream until a specific model event occurs.
create_watchpoint	N	N	Y	Defines a new watchpoint for a specific message type or identifier.
create_watchpoint_range	N	N	Y	Defines a new watchpoint for a range of message types or identifiers.
combine_watchpoints	N	N	Y	Defines a new watchpoint that is a Boolean AND or OR of two previously-defined watchpoints.
destroy_watchpoint	Y	N	Y	Removes a previously-created watchpoint.
enable_watchpoint	Y	N	Y	Enables watch_for triggering of a watchpoint.

Table 4: VMT Common Command Summary (Continued)

Command Name	Queued	Blocking	Zero Cycle	Description
disable_watchpoint	Y	N	Y	Disables watch_for triggering of a watchpoint.
set_watchpoint_trigger	Y	N	Y	Defines a watchpoint triggering profile.
get_watchpoint_trigger	Y	Y	Y	Returns a watchpoint triggering profile.
get_watchpoint_data_count	N	N	Y	Returns the number of members in the specified watchpoint data.
get_watchpoint_data_name	N	N	Y	Returns the name of the watchpoint event data at a given position.
get_watchpoint_data_type	N	N	Y	Returns the data type at a specified position in the specified watchpoint data.
get_watchpoint_data_size	N	N	Y	Returns the length of data at a specified position in the specified watchpoint data.
get_watchpoint_data_int	N	N	Y	Returns an integer value at a specified position in the specified watchpoint data.
get_watchpoint_data_string	N	N	Y	Returns a string of text at a specified position in the specified watchpoint data.
get_watchpoint_data_bit	N	N	Y	Returns the bit data at a specified position in the specified watchpoint data.
get_watchpoint_data_vec_ <size>	N	N	Y	Returns a specified word of vector data at a specified position in the specified watchpoint data.

Table 5: VMT Common Command Macro Summary

Macro Name	Description
Watchpoints	
VMT_CREATE_WP_MSG_TYPE	Deprecated command; not recommend for new design. Use create_watchpoint instead. This macro creates watchpoints for different message types.
VMT_CREATE_WP_MSG_ID	Deprecated command; not recommend for new design. Use create_watchpoint instead. This macro creates watchpoints for different message IDs.

Command Reference

This section contains an alphabetical listing of all common VMT commands. Each command reference page or pages contains all or most of the following information, arranged in the order listed:

- *Command name*, followed by a the same short description that appears in the command summary tables.
- Command **Syntax** in a non-language specific form.
- Command **Arguments** described individually.
- A detailed command **Description**.
- Usage **Prototypes** for **OpenVera**, **Verilog**, and **VHDL**.
- A list of **Related Commands** with a short description of each.

VHDL Command Structure

VMT commands used in VHDL testbenches have a different form than commands in OpenVera or Verilog testbenches. VHDL requires that the instance name be an argument, not an extension to the command name. The *shellInstName* is shown in the VHDL prototypes, but is not shown in the generic **Syntax** description or the list of **Arguments**.

For example, the enable_msg_type command would look like the following in Verilog. The instance name *monitor* is an extension to the command name:

```
monitor.enable_msg_type ( `MAIN_STREAM, `VMT_MSG_ALL, `VMT_MSG_ROUTE_ALL );
```

The same command in a VHDL testbench looks like the following, in which the instance name *monitor* appears as an argument to the command:

```
ahb_monitor_vmt_pkg.enable_msg_type ( "monitor", MAIN_STREAM, VMT_MSG_ALL,
                                      VMT_MSG_ROUTE_ALL );
```

block_stream

Blocks the current command stream.

Queued: **No** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

block_stream (*streamId*, *timeout*, *cmd_status***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
timeout	An integer that defines the maximum number of clock cycles to block while waiting for queued commands to complete.
cmd_status	A returned integer. Returns 0 when the command completes successfully. Returns 1 when the command times out.

Description

The block_stream command blocks the current command stream until all commands in the queue associated with the specified *streamId* have completed. The *timeout* argument sets the maximum number of clock cycles to block while waiting for queued commands to complete. If *timeout* is set to 0, the command stream will be blocked until all queued commands complete, regardless of the number of clock cycles the commands may take.

Messages

- VMT_MSGID_INVALID_SID
- VMT_MSGID_INVALID_TIMEOUT_ARG

Prototypes

OpenVera

```
task block_stream (
        integer streamId,
        integer timeout,
        var integer cmd_status );
```

Verilog

```
task block_stream;
        input [31:0] p_streamId;
        input [31:0] p_timeout;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure block_stream (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT timeout : IN integer;
        VARIABLE cmd_status : INOUT integer );
```

Related Commands

• new_stream	Starts a new command stream associated with a specified command channel. Returns the Stream ID of the new command stream.
• end_stream	Ends execution of the command stream started by the new_ stream command.

close_msg_log

Disables message output to a message log file and closes the file.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

close_msg_log (*streamId*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
msg_logID	An integer containing a message log ID that identifies a specific message log file (the *msg_logID* value returned by the open_msg_log command *or* a predefined set of message log IDs.

Description

The close_msg_log command disables the routing of messages to the specified message log file, closes the file, and invalidates the *msg_logID*.

Note

Simulation message transcripts cannot be disabled.

Prototypes

OpenVera

```
task close_msg_log (
        integer streamId,
        integer msg_logID );
```

Verilog

```
task close_msg_log;
        input [31:0] p_streamId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure close_msg_log (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• open_msg_log	Enables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

combine_watchpoints

Defines a new watchpoint that is a Boolean AND or OR of two previously-defined watchpoints.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

combine_watchpoints (*base1*, *logic*, *base2*, *watchpointHandle*)**;**

Arguments

base1	The integer handle of base watchpoint 1.
logic	An integer that selects the Boolean AND or OR logic of combination as one of the following values (as applied in a single cycle): **VMT_WP_LOGIC_AND** – New watchpoint triggers when both *base1* and *base2* watchpoints trigger. **VMT_WP_LOGIC_OR** – New watchpoint triggers when either *base1* or *base2* watchpoint triggers.
base2	The integer handle of base watchpoint 2.
watchpointHandle	A returned integer that identifies the new watchpoint.

Description

The combine_watchpoints command is used to combine two watchpoints (*base1* and *base2*) to create one combined watchpoint that is based on the result of the Boolean logic condition (*logic*) you choose.

Messages

- INVALID_WATCHPOINT_HANDLE
- INVALID_WATCHPOINT_LOGIC

Prototypes

OpenVera

```
task combine_watchpoints (
        integer base1,
        integer logic,
        integer base2,
        var integer watchpointHandle );
```

Verilog

```
task combine_watchpoints;
        input [31:0] p_base1;
        input [31:0] p_logic;
        input [31:0] p_base2;
        inout [31:0] p_watchpointHandle;
```

VHDL

```
procedure combine_watchpoints (
        CONSTANT shellInstName : IN string;
        CONSTANT base1 : IN integer;
        CONSTANT logic : IN integer;
        CONSTANT base2 : IN integer;
        VARIABLE watchpointHandle : INOUT integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

create_watchpoint

Defines a new watchpoint for a specific message type or identifier.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

create_watchpoint (*wp_type*, *id*, *watchpointHandle*)**;**

Arguments

wp_type An integer that defines the type watchpoint. The following table shows specific *wp_type* settings.

Table 6: Create Watchpoint Types

Type	Description
VMT_MESSAGE_TYPE	Use a VMT message type for the *id* argument. For all VMT message types, see "Predefined Message Types" for all VMT message types.
VMT_MESSAGE_ID	Use a VMT message ID for the *id* argument. See the message table for a specific VMT model or VMT Messages for common VMT messages.

id An integer that defines either the specific message type (when *wp_type* is VMT_MESSAGE_TYPE) or message ID (when *wp_type* is VMT_MESSAGE_ID) of watchpoint.

watchpointHandle A returned integer that identifies the new watchpoint.

Description

The create_watchpoint command creates a new watchpoint for a specified message type, message ID, or constrained random test notification ID.

Note

The create_watchpoint command replaces the VMT_CREATE_WP_MSG_TYPE and VMT_CREATE_WP_MSG_ID macros.

Messages

- VMT_MSGID_UNKNOWN_WATCHPOINT_ID
- VMT_MSGID_UNKNOWN_WATCHPOINT_TYPE
- VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT

Prototypes

OpenVera

```
task create_watchpoint (
        integer wp_type,
        integer id,
        var integer watchpointHandle );
```

Verilog

```
task create_watchpoint;
        input [31:0] p_wp_type;
        input [31:0] p_id;
        inout [31:0] p_watchpointHandle;
```

VHDL

```
procedure create_watchpoint (
        CONSTANT shellInstName : IN string;
        CONSTANT wp_type : IN integer;
        CONSTANT id : IN integer;
        VARIABLE watchpointHandle : INOUT integer );
```

Related Commands

• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

create_watchpoint_range

Defines a new watchpoint for a range of message types or identifiers.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

create_watchpoint_range (*wp_type*, *value_low*, *value_high*, *watchpointHandle*)**;**

Arguments

wp_type	An integer that defines the type watchpoint. See the description of the create_watchpoint command for a table of specific *wp_type* settings.
value_low	The low integer value that selects an event range of interest for a given type of watchpoint. The *wp_type* you have defined will determine the possible values.
value_high	The high integer value that selects an event range of interest for a given type of watchpoint. The *wp_type* you have defined will determine the possible values.
watchpointHandle	A returned integer that identifies the new watchpoint.

Description

The create_watchpoint_range command creates a watchpoint for any model-supported watchpoint type based on the range of values that you select.

Messages

- UNKNOWN_WATCHPOINT_TYPE

Prototypes

OpenVera

```
task create_watchpoint_range (
        integer wp_type,
        integer value_low,
        integer value_high,
        var integer watchpointHandle );
```

Verilog

```
task create_watchpoint_range;
        input [31:0] p_wp_type;
        input [31:0] p_value_low;
        input [31:0] p_value_high;
        inout [31:0] p_watchpointHandle;
```

VHDL

```
procedure create_watchpoint_range (
        CONSTANT shellInstName : IN string;
        CONSTANT wp_type : IN integer;
        CONSTANT value_low : IN integer;
        CONSTANT value_high : IN integer;
        VARIABLE watchpointHandle : INOUT integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

delete_handle

Deletes result handle, freeing result data memory.

Queued: **Yes** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

delete_handle (*streamId*, *handle***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
handle	An integer defining the results handle returned by the command that generated the results.

Description

The delete_handle command deletes the reference to a result handle so that the memory containing the results can be freed. The command can be used to free a specific handle, or all handles, using the VMT_ALL constant.

Messages

- INVALID_SID

Prototypes

OpenVera

```
task delete_handle (
        integer streamId,
        integer handle );
```

Verilog

```
task delete_handle;
        input [31:0] p_streamId;
        input [31:0] p_handle;
```

VHDL

```
procedure delete_handle (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT handle : IN integer );
```

destroy_watchpoint

Removes a previously-created watchpoint.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

destroy_watchpoint (*streamId*, *wpHandle*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
wpHandle	The integer handle of the watchpoint to be destroyed.

Description

The destroy_watchpoint command removes a watchpoint that was created with the create_watchpoint command. When you destroy a watchpoint that is no longer needed, performance is improved more than if the watchpoint was simply disabled or ignored.

Attention

A watch_for that is waiting on a watchpoint that has been destroyed will never unblock.

Messages

- INVALID_WATCHPOINT_HANDLE

Prototypes

OpenVera

```
task destroy_watchpoint (
        integer streamId,
        integer wpHandle );
```

Verilog

```
task destroy_watchpoint;
        input [31:0] p_streamId;
        input [31:0] p_wpHandle;
```

VHDL

```
procedure destroy_watchpoint (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId: IN integer;
        CONSTANT wpHandle : IN integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

disable_msg_feature

Disables a user-defined message format in model messages.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

disable_msg_feature (*streamId*, *scope*, *feature*, *msg_logID*)**;**

Arguments

streamId — An integer that specifies the command stream where the command is sent, returned by the new_stream command.

scope — A 32-bit vector. Reserved.

Note

Message scoping functionality is not implemented in this release; set the *scope* argument to VMT_MSG_SCOPE_ALL.

feature — A 9-bit vector specifying a message feature (such as description, type, or ID) to disable. To specify more than one feature constant, use bitwise OR syntax (see the "Example" section below).

For a list of feature constants that you can specify, see the description of the enable_msg_feature command.

For a list of features that are enabled by default, see Defaults for Message Types and Features.

msg_logID — An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The disable_msg_feature command disables features from being included in message log files and simluation transcript windows. For more information about controlling messages, see Controlling Messages.

Example

The following example disables message descriptions and types in all open message log files and the simulator transcript window. The bitwise OR syntax is used to specify more than one feature.

```
u1.disable_msg_feature(sid, scope, `VMT_MSG_DESC | `VMT_MSG_TYPE, `VMT_MSG_ROUTE_ALL);
```

Prototypes

OpenVera

```
task disable_msg_feature (
        integer streamId,
        bit[(VMT_MESSAGE_MASK_WIDTH-1):0] scope,
        bit[(VMT_MESSAGE_FEATURE_WIDTH-1):0] features,
        integer msg_logID );
```

Verilog

```
task disable_msg_feature;
        input [31:0] p_streamId;
        input [31:0] p_scope;
        input [8:0] p_features;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure disable_msg_feature (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT scope : IN std_logic_vector(31 downto 0);
        CONSTANT features : IN std_logic_vector(8 downto 0);
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• enable_msg_feature	Enables a user-defined message format in model messages.

disable_msg_id

Disables a specific message to a message log file or simulator transcript window.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

disable_msg_id (*streamId*, *msgId*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
msgId	An integer message ID that identifies a specific message. To obtain the ID of a displayed message, enable message IDs using the enable_msg_feature command. Also, message IDs are documented in the message tables for your specific model and the list of general VMT messages.
msg_logID	An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The disable_msg_id command disables a specific message to a message log file or simulator transcript window. This command overrides default messaging behavior, which is controlled by message type.

The specific message is specified by the *msgId* argument. To obtain the ID of a message, enable the message ID feature with the enable_msg_feature command, or search through the message IDs that are listed in the documentation for your specific model.

This command operates on a specified *msg_logID*, a simulator transcript window, all currently-opened log files, or a transcript window *and* all currently-opened log files. For more information about controlling messages, see Controlling Messages.

Prototypes

OpenVera

```
task disable_msg_id (
        integer streamId,
        integer msgId,
        integer msg_logID );
```

Verilog

```
task disable_msg_id;
        input [31:0] p_streamId;
        input [31:0] p_msgId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure disable_msg_id (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msgId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_id	Enables a specific message to a message log file or simulator transcript window.
• enable_type_ctrl_msg_id	Resets settings of a specific message to the settings for messages of that type.

disable_msg_log

Disables message output to a message log file or simulator transcript window.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

disable_msg_log (*streamId*, *msg_logID***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
msg_logID	An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The disable_msg_log command stops the flow of messages to a specified *msg_logID*, a simulator transcript window, all currently-opened log files, or a transcript window *and* all currently-opened log files. Use the enable_msg_log command to start logging messages again. For more information about controlling messages, see Controlling Messages.

Note

This command does not close a log file; use the close_msg_log command.

Prototypes

OpenVera

```
task disable_msg_log (
        integer streamId,
        integer msg_logID );
```

Verilog

```
task disable_msg_log;
        input [31:0] p_streamId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure disable_msg_log (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• enable_msg_log	Enables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

disable_msg_type

Disables one or more message types from a specified model message routing log file.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

disable_msg_type (*streamId*, *types*, *msg_logID***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
types	A 32-bit vector of message types. All VMT models have the predefined message types that are listed in the Predefined Message Types table that appears with the enable_msg_type command. To specify more than one type, use bitwise OR syntax (see the "Example" section below). For a list of types that are enabled by default, see Defaults for Message Types and Features.
msg_logID	An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The disable_msg_type command disables the different types of model messaging. By default, all message types *except* Fatal, Error, and Warning are disabled. Fatal messages are never disabled because they apply to situations from which the simulation cannot recover and usually terminates. For more information about controlling messages, see Controlling Messages.

Example

The following example disables two message types in all open message log files and the simulator transcript window. The bitwise OR syntax is used to specify more than one type.

```
u1.disable_msg_type(sid, `VMT_MSG_WARNING | `VMT_MSG_NOTE, `VMT_MSG_ROUTE_ALL);
```

Prototypes

OpenVera

```
task disable_msg_type (
        integer streamId,
        bit[(VMT_MESSAGE_MASK_WIDTH-1):0] types,
        integer msg_logID );
```

Verilog

```
task disable_msg_type;
        input [31:0] p_streamId;
        input [31:0] p_types;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure disable_msg_type (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT types : IN std_logic_vector(31 downto 0);
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type	Enables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

disable_watchpoint

Disables watch_for triggering of a watchpoint.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

disable_watchpoint (*streamId*, *wpHandle*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
wpHandle	The integer handle of the watchpoint to be destroyed.

Description

The disable_watchpoint command disables a watchpoint to suspend triggering associated watch_for commands.

Messages

- INVALID_WATCHPOINT_HANDLE

Prototypes

OpenVera

```
task disable_watchpoint (
        integer streamId,
        integer wpHandle );
```

Verilog

```
task disable_watchpoint;
        input [31:0] p_streamId;
        input [31:0] p_wpHandle;
```

VHDL

```
procedure disable_watchpoint (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId: IN integer;
        CONSTANT wpHandle : IN integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• enable_watchpoint	Enables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

enable_msg_feature

Enables a user-defined message format in model messages.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_msg_feature (*streamId*, *scope*, *feature*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
scope	A 32-bit vector. Reserved--set *scope* to VMT_MSG_SCOPE_ALL.
feature	A 9-bit vector indicating which message feature to enable. See the following table for a list of feature constants.
msg_logID	An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The enable_msg_feature command enables different message features, which allows custom format configuration on different message types and message IDs.

All message feature constants are listed in the following table, and you can see an example message on the next page. The elements of the example message are numbered to identify the corresponding feature listed in the table.

Table 7: Message Feature Constants

Feature Constant	Description
(1) VMT_MSG_DESC	Controls the descriptive text "Designware Model."
(2) VMT_MSG_TYPE	Controls the message type, such as ERROR or NOTE. You can select from the set of predefined message types.
(3) VMT_MSG_ID	Controls the message identifier label, for example AHB_MASTER_ERRMVALID.
(4) VMT_MSG_INST_NAME	Controls the model instance name, for example "top.U1."
(5) VMT_MSG_SIM_TIME	Controls the simulator time.
(6) VMT_MSG_TEXT	Controls the primary message string, such as "All outputs…"
(7) VMT_MSG_ARGS	Controls primary message arguments separated by spaces (when text is disabled), such as "HWDATA" and "HCLK" in the example below.

Table 7: Message Feature Constants (Continued)

Feature Constant	Description
VMT_MSG_TEXT_EXT	Controls a secondary message string, if one exists. **Note:** Most messages do not have secondary message strings.
VMT_MSG_ARGS_EXT	Controls secondary message arguments, if they exists. **Note:** Most messages do not have secondary message arguments.
VMT_MSG_FEATURES_ALL	Controls all message features.
VMT_MSG_FEATURES_DEFAULT	Controls all of the following features in the transcript, which are the default transcript features: VMT_MSG_DESC VMT_MSG_TYPE VMT_MSG_ INST_NAME VMT_MSG_SIM_TIME VMT_MSG_ TEXT VMT_MSG_ ARGS
VMT_MSG_FEATURES_LOG_DEFAULT	Controls all of the following features in the log file, which are the default log file features: VMT_MSG_TYPE VMT_MSG_SIM_TIME VMT_MSG_ TEXT VMT_MSG_ ARGS

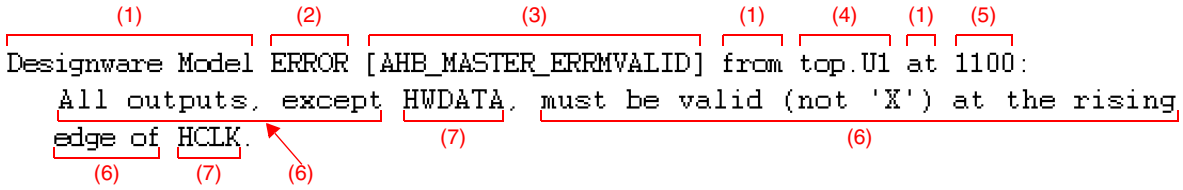


Figure 6: Message Feature Text Example

For more information on controlling VMT message content, see "Controlling Messages."

Prototypes

OpenVera

```
task enable_msg_feature (
        integer streamId,
        bit[(VMT_MESSAGE_MASK_WIDTH-1):0] scope,
        bit[(VMT_MESSAGE_FEATURE_WIDTH-1):0] features,
        integer msg_logID );
```

Verilog

```
task enable_msg_feature;
        input [31:0] p_streamId;
        input [31:0] p_scope;
        input [8:0] p_features;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure enable_msg_feature (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT scope : IN std_logic_vector(31 downto 0);
        CONSTANT features : IN std_logic_vector(8 downto 0);
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• disable_msg_feature	Disables a user-defined message format in model messages.

enable_msg_id

Enables a specific message to a message log file or simulator transcript window.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_msg_id (*streamId*, *msgId*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
msgId	An integer message ID that identifies a specific message. To obtain the ID of a message, enable the message ID feature with the enable_msg_feature command., or search through the message IDs are in the documentation for your specific model and the list of general VMT messages.
msg_logID	An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The enable_msg_id command enables a specific message to a message log file or simulator transcript window. This command overrides default messaging behavior, which is controlled by message type. This command can also re-enable a specific message that was stopped with the disable_msg_id command.

The specific message is specified by the *msgId* argument. To obtain the ID of a message, enable the message ID feature with the enable_msg_feature command, or search through the message IDs that are listed in the documentation for your specific model.

This command operates on a specified *msg_logID*, a simulator transcript window, all currently-opened log files, or a transcript window *and* all currently-opened log files. For more information about controlling messages, see Controlling Messages.

Prototypes

OpenVera

```
task enable_msg_id (
        integer streamId,
        integer msgId,
        integer msg_logID );
```

Verilog

```
task enable_msg_id;
        input [31:0] p_streamId;
        input [31:0] p_msgId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure enable_msg_id (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msgId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• disable_msg_id	Disables a specific message to a message log file or simulator transcript window.
• enable_type_ctrl_msg_id	Resets settings of a specific message to the settings for messages of that type.

enable_msg_log

Enables message output to a message log file or simulator transcript window.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_msg_log (*streamId*, *msg_logID*)**;**

Arguments

streamId — An integer that specifies the command stream where the command is sent, returned by the new_stream command.

msg_logID — An integer containing a message log ID that identifies a specific message log file. The *msg_logID* can be the value returned by the open_msg_log command *or* a predefined message log ID from the following table. Note that other message commands can use the IDs listed in this table.

Table 8: Message Log IDs

Message Log Constant	Description
VMT_MSG_ROUTE_SIM	Applies to messages in the simulator transcript window.
VMT_MSG_ROUTE_ALL_LOGS	Applies to all message log files that are currently opened.
VMT_MSG_ROUTE_ALL	Applies to all message log files that are currently opened and messages in the simulator transcript window.

Description

The enable_msg_log command re-starts the flow of messages that was stopped with the disable_msg_log command. The command operates on a specified *msg_logID*, a simulator transcript window, all currently-opened log files, or a transcript window *and* all currently-opened log files. For more information about controlling messages, see Controlling Messages.

Prototypes

OpenVera

```
task enable_msg_log (
        integer streamId,
        integer msg_logID );
```

Verilog

```
task enable_msg_log;
        input [31:0] p_streamId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure enable_msg_log (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• disable_msg_log	Disables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

enable_msg_type

Enables one or more message types from a specified model instance.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_msg_type (*streamId*, *types*, *msg_logID*)**;**

Arguments

streamId — An integer that specifies the command stream where the command is sent, returned by the new_stream command.

types — A 32-bit vector of message types. All VMT models have the pre-defined message types shown in the Predefined Message Types table below. To specify more than one type, use bitwise OR syntax (see the "Example" section below).

For a list of types that are enabled by default, see Defaults for Message Types and Features.

msg_logID — An integer specifying a message log ID that identifies a message log file. The *msg_logID* can be a value returned by the open_msg_log command *or* a predefined message log ID from the Message Log IDs table that appears with the enable_msg_log command.

Description

The enable_msg_type command enables the different types of model messaging. By default Fatal, Error, and Warning messages are enabled. Fatal messages are always enabled because they apply to situations from which the simulation cannot recover and usually terminates. For more information about controlling messages, see Controlling Messages.

The following table lists the IDs of the message types.

Table 9: Predefined Message Types

Message Constant	Description
VMT_MSG_ERROR	The model has encountered an error, but can recover and resume simulation. **Example:** The model receives a command that would put it into an invalid state.
VMT_MSG_WARNING	The model has encountered a situation that is not an error, but that you should be aware of. **Example:** The model ignores significant bits of an address.
VMT_MSG_TIMING	The model has encountered a timing violation, such as a setup or hold violation.
VMT_MSG_XHANDLING	The model has encountered an X state on an input port during a read transaction. The model substitutes a pre-defined default value.
VMT_MSG_NOTE	The model informs you of normal operation and status.

Table 9: Predefined Message Types (Continued)

Message Constant	Description
VMT_MSG_PROTO_CYCLE	Controls messages about model protocol on cycle boundaries, for example when the model completes beat 3 of a burst read.
VMT_MSG_PROTO_TRANS	Controls messages about model protocol on transaction boundaries. For example, when a model completes a burst read, a protocol transaction message would contain all of the relevant details about that transaction. The transaction information is buffered as the model executes and is output at successful completion or interruption of the transaction.
VMT_MSG_PROTO_ERROR	Controls all model protocol error messages. These messages identify protocol errors such as a malformed packet or an invalid response.
VMT_MSG_CMD	Controls all model command messages, which inform you about the commands the model is executing.
VMT_MSG_REPORT	Controls messages displayed when a command requests model status or information. For example, a print_msg command generates a VMT_MSG_ REPORT message when this message type is enabled.
VMT_MSG_NOTIFY	Controls all testbench notification messages. These messages do not print to a log file or simulator transcript, however they provde full watchpoint suipport, including data arguments.
VMT_MSG_ALL	Controls all model messages *except* command messages.
VMT_MSG_DEFAULT	Controls default model messages: Error and Warning. Fatal messages are always enabled.
VMT_MSG_LOG_DEFAULT	Controls default log file messages (Fatal, Error, Report, Warning, Protocol Cycle and Protocol Error). Fatal messages are always enabled.

Example

The following example enables two message types in all open message log files and the simulator transcript window. The bitwise OR syntax is used to specify more than one type.

```
u1.enable_msg_type(sid, `VMT_MSG_WARNING | `VMT_MSG_NOTE, `VMT_MSG_ROUTE_ALL);
```

Prototypes

OpenVera

```
task enable_msg_type (
        integer streamId,
        bit[(VMT_MESSAGE_MASK_WIDTH-1):0] types,
        integer msg_logID );
```

Verilog

```
task enable_msg_type;
        input [31:0] p_streamId;
        input [31:0] p_types;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure enable_msg_type (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT types : IN std_logic_vector(31 downto 0);
        CONSTANT msg_logID : IN integer );
```

Related Commands

• disable_msg_type	Disables one or more message types from a specified model instance.
• open_msg_log • close_msg_log	Enables or disables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

enable_type_ctrl_msg_id

Resets settings of a specific message to the settings for messages of that type.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_type_ctrl_msg_id (*streamId*, *msgId*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
msgId	An integer message ID that identifies a specific message.
msg_logID	An integer containing a message log ID that identifies a specific message log file (the *msg_logID* value returned by the open_msg_log command *or* a predefined set of message log IDs.

Description

The enable_type_ctrl_msg_id command changes the message control for the specified message back to the control for that message type.

Message settings for a specific message are modified by the enable_msg_id command and the disable_msg_id command.

Message settings for all messages of a specific type are modified by the enable_msg_type command and the disable_msg_type command. For more information about controlling messages, see Controlling Messages.

Prototypes

OpenVera

```
task enable_type_ctrl_msg_id (
        integer streamId,
        integer msgId,
        integer msg_logID );
```

Verilog

```
task enable_type_ctrl_msg_id;
        input [31:0] p_streamId;
        input [31:0] p_msgId;
        input [31:0] p_msg_logID;
```

VHDL

```
procedure enable_type_ctrl_msg_id (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT msgId : IN integer;
        CONSTANT msg_logID : IN integer );
```

Related Commands

• enable_msg_id	Enables a specific message to a message log file or simulator transcript window.
• disable_msg_id	Disables a specific message to a message log file or simulator transcript window.

enable_watchpoint

Enables watch_for triggering of a watchpoint.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

enable_watchpoint (*streamId*, *wpHandle***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
wpHandle	The integer handle of the watchpoint to be enabled.

Description

The enable_watchpoint command enables a watchpoint so that it can trigger associated watch_for commands when its event occurs.

Messages

- INVALID_WATCHPOINT_HANDLE

Prototypes

OpenVera

```
task enable_watchpoint (
        integer streamId,
        integer wpHandle );
```

Verilog

```
task enable_watchpoint;
        input [31:0] p_streamId;
        input [31:0] p_wpHandle
```

VHDL

```
procedure enable_watchpoint (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId: IN integer;
        CONSTANT wpHandle : IN integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint	Disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

end_stream

Ends execution of the command stream started by the new_stream command.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

end_stream (*streamId***);**

Argument

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.

Description

The end_stream command is used to indicate the end of a command stream. No more commands should be issued to the model using the specified *streamId* after end_stream is sent to the model.

Messages

- INVALID_SID

Prototypes

OpenVera

```
task end_stream (
        var integer streamId );
```

Verilog

```
task end_stream;
        inout [31:0] p_streamId;
```

VHDL

```
procedure end_stream (
        CONSTANT shellInstName : IN string;
        VARIABLE streamId : INOUT integer );
```

Related Commands

• new_stream	Starts a new command stream associated with a specified command channel. Returns the Stream ID of the new command stream.
• block_stream	Blocks the current command stream.

get_config_param

Reads a specified configuration parameter value.

Queued: **Yes** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

get_config_param (*streamId*, *parameter*, *value***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
parameter	An integer that identifies the configuration parameter. Use a predefined macro, either from the table in the description of the set_config_param command, or from the table of configuration parameters for your model.
value	A returned integer that contains the current value of the specified parameter.

Description

The get_config_param command reads the value of a specified configuration parameter of a specified model. For the *parameter* argument, use a predefined macro, either from the table in the description of the set_config_param command, or from the table of configuration parameters for your model.

The command is placed on the command queue and blocked until the command has executed, so that it can return the value of the configuration parameter after all previously-queued commands execute.

Messages

- INVALID_SID
- INVALID_PARAM

Prototypes

OpenVera

```
task get_config_param (
        integer streamId,
        integer parameter,
        var integer value );
```

Verilog

```
task get_config_param;
        input [31:0] p_streamId;
        input [31:0] p_parameter;
        inout [31:0] p_value;
```

VHDL

```
procedure get_config_param (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT parameter : IN integer;
        VARIABLE value : INOUT integer );
```

Related Commands

- set_config_param Changes a specified configuration parameter value.

get_port

Reads a value from a specified port.

Queued: **Yes** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

get_port (*streamId*, *portId*, *value***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
portId	An integer input port identifier. Use a model-specific define for the port ID.

Attention

The get_port command can only be used on input ports. You cannot read the value of an output port.

value	A returned bit vector containing the value.

Description

The get_port command retrieves the value of port. The command is placed on the command queue and blocked until the command has executed, so that it can return the value of the port after all previously-queued commands execute.

Messages

- INVALID_SID
- INVALID_PORTID

Prototypes

OpenVera

```
task get_port (
        integer streamId,
        integer portId,
        var bit [(maxPortSize-1):0] value );
```

Verilog

```
task get_port;
        input [31:0] p_streamId;
        input [31:0] p_portId;
        inout [1023:0] p_value;
```

VHDL

```
procedure get_port (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT portId : IN integer;
        VARIABLE value : INOUT std_logic_vector(1023 downto 0) );
```

Related Commands

- set_port Drives a value onto a specified port.

get_register

Reads the value of a specified model register.

Queued: **Yes** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

get_register (*streamId*, *registerId*, *value***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
registerId	An integer register identifier.
value	A returned integer register value.

Description

The get_register command retrieves the value of an internal register. The command is placed on the queue and blocked until the command has executed, so that it can return the value of the register after all previously-queued commands execute.

Messages

- VMT_MSGID_INVALID_REGISTER_ID

Prototypes

OpenVera

```
task get_register(
        integer streamId,
        integer registerId,
        var integer value);
```

Verilog

```
task get_register;
        input [31:0] p_streamId;
        input [31:0] p_registerId;
        inout [31:0] p_value;
```

VHDL

```
procedure get_register (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT registerId : IN integer;
        VARIABLE value : INOUT integer);
```

Related Commands

- set_register Changes the value of a specified model register.

get_version

Returns the version of the model.

Queued: **No** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

get_version (*version***);**

Argument

version	A returned string containing the model version.

Description

The get_version command returns string that lists the model version.

Prototypes

OpenVera

```
task get_version (
        var string version );
```

Verilog

```
task get_version;
        inout [127:0] p_version;
```

VHDL

```
procedure get_version (
        CONSTANT shellInstName : IN string;
        VARIABLE version : INOUT string );
```

get_watchpoint_data_bit

Returns the bit data at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_bit (*dataHandle, position, value, cmd_status***);**

Arguments

dataHandle	The integer handle of the watchpoint data to query, returned by the watch_for command.
position	An integer position of watchpoint data to check.
value	The returned bit data member.
cmd_status	Returned integer status of 1 if *value* is valid, or returns a -1 if there was an error.

Description

The get_watchpoint_data_bit command returns the bit data of the watchpoint event data at a given position.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION
- BAD_WATCHPOINT_DATA_TYPE

Prototypes

OpenVera

```
task get_watchpoint_data_bit(
        integer dataHandle,
        integer position,
        var bit value,
        var integer cmd_status);
```

Verilog

```
task get_watchpoint_data_bit;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        inout p_value;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure get_watchpoint_data_bit (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        VARIABLE value : INOUT std_logic;
        VARIABLE cmd_status : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_count

Returns the number of members in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_count (*dataHandle, dataCount*)**;**

Arguments

dataHandle	The integer handle of the watchpoint data to be queried returned by the watch_for command.
dataCount	The returned integer of the number of data members.

Description

The get_watchpoint_data_count command returns the number of data members in a watchpoint event data object. The following table shows a list of predefined data members.

Table 10: Predefined Watchpoint Event Data Members

Predefined Constants	Description
VMT_WP_DATA_EVENT_CAUSE	The handle of the watchpoint that triggered on this event
VMT_WP_DATA_EVENT_TYPE	The type of the watchpoint that triggered on the event
VMT_WP_DATA_EVENT_ID	The value identifier of the watchpoint that triggered on the event
VMT_WP_DATA_TRIGGER_CYCLE	The number of times this event has triggered this watchpoint in the current cycle
VMT_WP_DATA_TRIGGER_TOTAL	The total number of times this event has triggered this watchpoint since its creation
VMT_WP_DATA_NEXT_EVENT	The handle of the next event data set associated with this event or -1 if there are no more events that triggered this watchpoint. There will be only one handle for basic watchpoint triggers, but combined watchpoints may trigger due to multiple events and thus have multiple data handles.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE

Prototypes

OpenVera

```
task get_watchpoint_data_count (
        integer dataHandle,
        var integer dataCount);
```

Verilog

```
task get_watchpoint_data_count;
        input [31:0] p_dataHandle;
        inout [31:0] p_dataCount;
```

VHDL

```
procedure get_watchpoint_data_count (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        VARIABLE dataCount : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_int

Returns an integer value at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_int (*dataHandle, position, value, cmd_status***);**

Arguments

dataHandle	The integer handle of the watchpoint data to query.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command. You can also specify VMT_MSG_EVENT_ARG_MSG_TYPE or VMT_MSG_EVENT_ARG_MSG_ID to obtain the message type or ID, respectively.
value	The returned integer of the data member.
cmd_status	Returned integer status of 1 if the returned value is valid, -1 if there was an error.

Description

The get_watchpoint_data_int command returns the integer value of the watchpoint event data at a given position. For more information, see Using Messages as Watchpoints.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION
- BAD_WATCHPOINT_DATA_TYPE

Prototypes

OpenVera

```
task get_watchpoint_data_int(
        integer dataHandle,
        integer position,
        var integer value,
        var integer cmd_status);
```

Verilog

```
task get_watchpoint_data_int;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        inout [31:0] p_value;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure get_watchpoint_data_int (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        VARIABLE value : INOUT integer;
        VARIABLE cmd_status : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_name

Returns the name of the watchpoint event data at a given position.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_name (*dataHandle, position, name, cmd_status*)**;**

Arguments

dataHandle	The integer handle of the watchpoint data to query, returned by the watch_for command.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command.
name	The returned string that identifies the name of the data (80 characters maximum).
cmd_status	Returned integer number of characters in *name*, or -1 if there was an error detected.

Description

The get_watchpoint_data_name command returns the name of the watchpoint event data at a given position.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION

Prototypes

OpenVera

```
task get_watchpoint_data_name (
        integer dataHandle,
        integer position,
        var string name,
        var integer cmd_status);
```

Verilog

```
task get_watchpoint_data_name;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        inout [sizeofstring:0] p_name;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure get_watchpoint_data_name (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle: IN integer;
        CONSTANT position : IN integer;
        VARIABLE name : INOUT string;
        VARIABLE cmd_status : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_size

Returns the length of data at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_size (*dataHandle, position, size***);**

Arguments

dataHandle	The integer handle of the watchpoint data to query.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command.
dataSize	The returned integer indicating the size of the data member. A -1 is returned if there was an error. The meaning of the size depends on the type of the data as follows: **Integer** – no unit - the return value is always one **String** – the number of characters in the string **Vector** – the number of bits in the vector

Description

The get_watchpoint_data_size command returns the size of the watchpoint event data member at a given position.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION

Prototypes

OpenVera

```
task get_watchpoint_data_size (
        integer dataHandle,
        integer position,
        var integer dataSize);
```

Verilog

```
task get_watchpoint_data_size;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        inout [31:0] p_dataSize;
```

VHDL

```
procedure get_watchpoint_data_size (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        VARIABLE dataSize : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_string

Returns a string of text at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_string (*dataHandle, position, line_index, value, cmd_status***);**

Arguments

dataHandle	The integer handle of the watchpoint data to query.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command.
line_index	An integer number that indicates the desired line number from which to get the string, beginning at 0.
value	The returned string data line (80 characters maximum).
cmd_status	Returned integer number of characters set in *value*, or returns a -1 if there was an error.

Description

The get_watchpoint_data_string command returns the selected line of string data of the watchpoint event data at a given position.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION
- BAD_WATCHPOINT_DATA_TYPE

Prototypes

OpenVera

```
task get_watchpoint_data_string(
        integer dataHandle,
        integer position,
        integer line_index,
        var string value,
        var integer cmd_status);
```

Verilog

```
task get_watchpoint_data_string;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        input [31:0] p_line_index;
        inout [sizeofstring:0] p_value;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure get_watchpoint_data_string (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        VARIABLE line_index : IN integer;
        VARIABLE value : INOUT string;
        VARIABLE cmd_status : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_type

Returns the data type at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_type (*dataHandle, position, dataType*)**;**

Arguments

dataHandle	The integer handle of the watchpoint data to query.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command.
dataType	The returned integer indicating the type of the data member. See the following table for possible return values. If there is an error, a -1 is returned.

Description

The get_watchpoint_data_type command returns the type of the watchpoint event data at a given position. The following table shows possible return data types.

Table 11: Predefined Watchpoint Data Types

Predefined Constants	Description
VMT_WP_DATA_INT_TYPE	Integer data. Can hold a handle value represented by an integer as well.
VMT_WP_DATA_STRING_TYPE	String data
VMT_WP_DATA_BIT_TYPE	Bit data
VMT_WP_DATA_VEC_TYPE	Bit Vector Data

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION

Prototypes

OpenVera

```
task get_watchpoint_data_type (
        integer dataHandle,
        integer position,
        var integer dataType);
```

Verilog

```
task get_watchpoint_data_type;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        inout [31:0] p_dataType;
```

VHDL

```
procedure get_watchpoint_data_type (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        VARIABLE dataType : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_data_vec_<size>

Returns a specified word of vector data at a specified position in the specified watchpoint data.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_data_vec_<*size*> (*dataHandle, position, word, value, cmd_status*)**;**

Note

<*size*> is the number of bits in the *value* vector. Substitute one of the following values for <*size*>: 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024. For example: get_watchpoint_data_vec_2.

Arguments

dataHandle	The integer handle of the watchpoint data to query, returned by the watch_for command.
position	An integer position of watchpoint data to check. To specify the position, use one of the data field constants of the message that was specified in the create_watchpoint command.
word	An integer number of the word to get from the vector, beginning at 0.
value	The returned bit vector data word.
cmd_status	Returned integer of bits set in *value*, or returns a -1 if there was an error.

Description

These get_watchpoint_data_vec_<*size*> commands return the selected word of vector data of the watchpoint event data at a given position. The word <*size*> (number of bits in the value vector) is encoded into the command name.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION
- BAD_WATCHPOINT_DATA_TYPE
- INVALID_WATCHPOINT_DATA_VEC_WORD

Prototypes

Note

In the following prototypes, substitute the appropriate value for <*size*>, which is one of the following: 2, 4, 8, 16, 32, 64, 128, 256, 512, or 1024.

OpenVera

```
task get_watchpoint_data_vec_<size>(
        integer dataHandle,
        integer position,
        integer word,
        var bit[<size>-1:0] value,
        var integer cmd_status);
```

Verilog

```
task get_watchpoint_data_vec_<size>;
        input [31:0] p_dataHandle;
        input [31:0] p_position;
        input [31:0] p_word;
        inout [<size>-1:0] p_value;
        inout [31:0] p_cmd_status;
```

VHDL

```
procedure get_watchpoint_data_vec_<size> (
        CONSTANT shellInstName : IN string;
        CONSTANT dataHandle : IN integer;
        CONSTANT position : IN integer;
        CONSTANT word : IN integer;
        VARIABLE value : INOUT std_vector(<size>-1 downto 0);
        VARIABLE cmd_status : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

get_watchpoint_trigger

Returns a watchpoint triggering profile.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

get_watchpoint_trigger (*streamId*, *wpHandle, profile, value*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
wpHandle	The integer handle of the watchpoint for which to retrieve the configuration.
profile	An integer that defines the type of profile configuration to retrieve. For a table of supported watchpoint trigger profiles, see the set_watchpoint_trigger command description.
value	The returned integer value of the profile configuration.

Description

The get_watchpoint_trigger command gets a current configuration value of a watchpoint's triggering profile.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION

Prototypes

OpenVera

```
task get_watchpoint_trigger (
        integer streamId,
        integer wpHandle,
        integer profile,
        var integer value);
```

Verilog

```
task get_watchpoint_trigger;
        input [31:0] p_streamId;
        input [31:0] p_wpHandle;
        input [31:0] p_profile;
        inout [31:0] p_value;
```

VHDL

```
procedure get_watchpoint_trigger (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId: IN integer;
        CONSTANT wpHandle : IN integer;
        CONSTANT profile : IN integer;
        VARIABLE value : INOUT integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger	Returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

new_stream

Starts a new command stream associated with a specified command channel. Returns the Stream ID of the new command stream.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

new_stream (*channelId*, *streamId*)**;**

Arguments

channelId	An integer command channel that the new command stream is associated with.
streamId	A returned integer that specifies the command stream that was created.

Description

The new_stream command creates a new command stream and returns the *streamId* value. The default *channelId* for models with a single command channel is VMT_DEFAULT_CMD_CHANNEL. For models with multiple command channels, the default *streamId* and *channelId* values are model specific. For more information on command streams, see "Command Channels and Command Streams."

Note

Prior to issuing this command, the *streamId* to use for all commands is the default *streamId*. For single command channel models, that value is VMT_DEFAULT_STREAM_ID. For multiple command channel models, consult the model databook for that specific model.

Every new *streamId* generated is unique and corresponds to a particular command channel.

Attention

If the new_stream command is issued before the start command, the returned *streamId* is -1. If you attempt to use the -1 returned *streamId*, those commands will fail and will not be queued.

Messages

- INVALID_CHANNEL_ID

Prototypes

OpenVera

```
task new_stream (
        integer channelId,
        var integer newStreamId );
```

Verilog

```
task new_stream;
        input [31:0] p_channelId;
        inout [31:0] p_newStreamId;
```

VHDL

```
procedure new_stream (
        CONSTANT shellInstName : IN string;
        CONSTANT channelId: IN integer;
        VARIABLE newStreamId : INOUT integer );
```

Related Commands

• end_stream	Ends execution of the command stream started by the new_ stream command.
• block_stream	Blocks the current command stream.

open_msg_log

Enables message output to a message log file.

Queued: **Yes** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

open_msg_log (*streamId*, "*msg_log_filename*", *mode*, *msg_logID*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
"*msg_log_filename*"	A string containing the relative or full pathname to the message log output file. If *msg_log_filename* is not specified, the default filename is **./***inst_name***.msg**.

Note

Log files are intended to be used by a single model instance. Although you can assign a log file to more than one instance by using append mode, message order in the log file will be unpredictable.

mode	An integer that specifies if the messages are appended (VMT_MSG_LOG_MODE_APPEND) to the file or overwritten (VMT_MSG_LOG_MODE_OVR).
msg_logID	A returned integer containing the message log ID that identifies a message log file. This handle controls message filtering and configuration for all messages that are output to the log file.

Description

The open_msg_log command enables the routing of messages to different output files. Log files can be opened in append or overwrite mode. For more information about controlling messages, see Controlling Messages.

Prototypes

OpenVera

```
task open_msg_log (
        integer streamId,
        bit[VMT_MESSAGE_LOG_STRING_WIDTH-1:0] filename,
        integer mode,
        var integer msg_logID );
```

Verilog

```
task open_msg_log;
        input [31:0] p_streamId;
        input [2047:0] p_filename;
        input [31:0] p_mode;
        inout [31:0] p_msg_logID;
```

VHDL

```
procedure open_msg_log (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT filename : IN string;
        CONSTANT mode : IN integer;
        VARIABLE msg_logID : INOUT integer );
```

Related Commands

• enable_msg_type • disable_msg_type	Enables or disables one or more message types from a specified model instance.
• close_msg_log	Disables message output to a message log file.
• enable_msg_log • disable_msg_log	Enables or disables message output to a message log file or simulator transcript window.
• enable_msg_feature • disable_msg_feature	Enables or disables a user-defined message format in model messages.

print_msg

Prints a text message in a simulation transcript.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

print_msg (*streamId*, *message***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
message	A string text message to print.
format	A character or characters to print.

Description

The print_msg command causes the model to issue a message of type Report. The string in the *message* argument displays as the message text.

All standard VMT messaging functionality applies to this command, such as enable/disable, message destination, and so on.

Messages

- VMT_MSGID_PRINT_MSG_TEXT

Prototypes

OpenVera

```
task print_msg (
        integer streamId,
        string message );
```

Verilog

```
task print_msg;
        input [31:0] p_streamId;
        input [639:0] p_message;
```

VHDL

```
procedure print_msg (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT message: IN string );
```

reset_model

Resets the model.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

reset_model (*rst_type*)**;**

Argument

rst_type	An integer reset type. Can be set to VMT_SOFT, VMT_FIRM, VMT_HARD or model-specific reset types. See the command description for an explanation of each reset type.

Description

The reset_model command clears model command queues, watchpoints, configuration, and/or ports, depending on the *rst_type* chosen. Valid *rst_type* settings are summarized in the following table.

Table 12: Reset Types

Reset Type	Reset Actions			Issue start after reset
	Command Queue	Watchpoints	Configuration Parameters	
VMT_SOFT	Yes	No	No	No. Not allowed.
VMT_FIRM	Yes	Yes	No	No. Not allowed.
VMT_HARD	Yes	Yes	Yes	Yes. Must be issued.
model-specific[a]	Varies by model. See model documentation.			

a. Consult model-specific documentation for further details on model-specific reset capabilities.

A VMT_SOFT reset, or *soft reset*, is used to clear the command queue and bring the model back to a known state. The soft reset preserves all configuration settings. As a result, the start command cannot be re-issued after a soft reset.

A VMT_FIRM reset, or *firm reset*, does everything that a soft reset does, plus it deletes all watchpoints. You use a firm reset to retain the configuration of the model, but clear all watchpoints and command queue. All watch_for commands in the testbench are triggered by a firm reset, then watch_for returns a handle of value VMT_WP_TERMINATED_BY_RESET. As a result, all of your watch_for routines must trap VMT_WP_TERMINATED_BY_RESET to implement any reset activities or to clean up unwanted testbench threads.

A VMT_HARD, or *hard reset* does everything that a firm reset does, plus the following:

- Clears all configuration settings to their defaults.
- Removes all message log file handles.

You use a hard reset when you want to get the model to the same state it was in just after instantiation. New configurations can be set, and then a model start command re-issued.

Note

VMT_SOFT, VMT_FIRM, and VMT_HARD do not perform any device reset functions. A model-specific reset *may* perform a device reset. Consult the model documentation for all information on model-specific reset.

Messages

- VMT_MSGID_INVALID_RESET_TYPE

Prototypes

OpenVera

```
task reset_model (
        integer rst_type );
```

Verilog

```
task reset_model;
        input [31:0] p_rst_type;
```

VHDL

```
procedure reset_model (
        CONSTANT shellInstName : IN string;
        CONSTANT rst_type: IN integer );
```

Related Command

- start — Starts model execution.

set_config_param

Changes a specified configuration parameter value.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

set_config_param (*streamId*, *parameter*, *value*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
parameter	An integer that identifies the configuration parameter. Use a predefined macro, either from the table below or from the table of configuration parameters for your model.
value	An integer that contains the new value of the parameter.

Description

The set_config_param command sets the value of a specified configuration parameter for a specified model.

The only configuration parameter that is common to all VMT models is listed in the table below. For all other configuration parameters, refer to the table of configuration parameters for your model.

Table 13: Common Configuration Parameters

[Default] and Legal Values	Description
VMT_FORCE_CMD_BLOCKING	
ON [OFF]	If ON, causes commands not to be queued and blocks the command stream flow until command execution is complete.

Messages

- INVALID_SID
- INVALID_PARAM
- INVALID_FORCE_BLOCK_PARAM

Prototypes

OpenVera

```
task set_config_param (
        integer streamId,
        integer parameter,
        integer value );
```

Verilog

```
task set_config_param;
        input [31:0] p_streamId;
        input [31:0] p_parameter;
        input [31:0] p_value;
```

VHDL

```
procedure set_config_param (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT parameter : IN integer;
        CONSTANT value : IN integer );
```

Related Commands

- get_config_param Reads a specified configuration parameter value.

set_port

Drives a value onto a specified port.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

set_port (*streamId*, *portId*, *value*, *numDelayClocks*, *numDriveClocks*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
portId	An integer output port identifier. Use a model-specific define for the port ID.

Attention

The set_port command can only be used on output ports. You cannot set the value of an input port.

value	A bit vector containing the value.
numDelayClocks	An integer number of clock cycles to delay before the new value is forced on the port.
numDriveClocks	An integer number of clock cycles the new value is forced on the port.

Description

The set_port command drives a value onto a specified port. This command overwrites any value the model would normally drive. The *numDelayClocks* argument sets the delay, in clock cycles, before the new value is forced. The *numDriveClocks* argument sets the duration, in clock cycles, that the new *value* is forced on the port.

Messages

- INVALID_SID
- INVALID_PORTID
- INVALID_DLYCLKS_ARG
- INVALID_DRVCLKS_ARG

Prototypes

OpenVera

```
task set_port (
        integer streamId,
        integer portId,
        bit [(maxPortSize-1):0] value,
        integer numDelayClocks,
        integer numDriveClocks );
```

Verilog

```
task set_port;
        input [31:0] p_streamId;
        input [31:0] p_portId;
        input [1023:0] p_value;
        input [31:0] p_numDelayClocks;
        input [31:0] p_numDriveClocks;
```

VHDL

```
procedure set_port (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT portId : IN integer;
        CONSTANT value : IN std_logic_vector(1023 downto 0);
        CONSTANT numDelayClocks : IN integer;
        CONSTANT numDriveClocks : IN integer );
```

Related Commands

- get_port — Reads a value from a specified port.

set_register

Changes the value of a specified model register.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

set_register (*streamId*, *registerId*, *value***);**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
registerId	An integer register identifier.
value	An integer register value.

Description

The set_register command changes the value of the specified internal model register.

Messages

- VMT_MSGID_INVALID_REGISTER_ID

Prototypes

OpenVera

```
task set_register(
        integer streamId,
        integer registerId,
        integer value);
```

Verilog

```
task set_register;
        input [31:0] p_streamId;
        input [31:0] p_registerId;
        input [31:0] p_value;
```

VHDL

```
procedure set_register (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId : IN integer;
        CONSTANT registerId : IN integer;
        CONSTANT value : IN integer);
```

Related Commands

- get_register Reads the value of a specified model register.

set_watchpoint_trigger

Defines a watchpoint triggering profile.

Queued: **Yes** Blocking: **No** Zero Cycle: **Yes**

Syntax

set_watchpoint_trigger (*streamId*, *wpHandle, profile, value*)**;**

Arguments

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.
wpHandle	The integer handle of the watchpoint to be configured.
profile	An integer that defines the type of profile configuration to set. See the following table for supported profile types.
value	An integer of the new value of the profile configuration.

Description

The set_watchpoint_trigger command configures a watchpoint's triggering profile. Configuring the profile automates the enabling and disabling of the watchpoint and, therefore, how watch_for commands unblock.

The following table shows the supported *profile* types. Throughout this table, a *cycle* equates to the clock cycle for the model. .

Table 14: Watchpoint Triggering Profile Configuration Types

Predefined Constants	Description
VMT_WP_TRIGGER_PARAM	Controls the event triggering protocol within a single cycle. Valid values are: ● VMT_WP_TRIGGER_HANDSHAKE - watch_for commands can be triggered multiple times per cycle. ● VMT_WP_TRIGGER_ONE_SHOT - watch_for commands can be triggered once per cycle. This is the default setting.
VMT_WP_START_PARAM	Delays the start of the watchpoint for a specified number of cycles. Effectively suppresses triggering of the watchpoint until the given number of cycles has passed.
VMT_WP_STOP_PARAM	Disables the watchpoint after a specified number of cycles. This parameter is a cycle timer that decrements to zero, then literally disables the watchpoint.
VMT_WP_IGNORE_PARAM	Suppresses triggering for the specified number of events. This parameter is an event counter that suppresses triggering while it decrements to zero.
VMT_WP_LIMIT_PARAM	Disables the watchpoint after the specified number of events. This parameter is an event timer that decrements to zero, then disables the watchpoint.
VMT_WP_EXACT_PARAM	Triggers once when exactly the specified number of events has occurred. Just like ignoring (n-1) and limiting (1) in one profile.

Messages

- INVALID_WATCHPOINT_DATA_HANDLE
- INVALID_WATCHPOINT_DATA_POSITION
- INVALID_WATCHPOINT_PROFILE_VALUE

Prototypes

OpenVera

```
task set_watchpoint_trigger (
        integer streamId,
        integer wpHandle,
        integer profile,
        integer value);
```

Verilog

```
task set_watchpoint_trigger;
        input [31:0] p_streamId;
        input [31:0] p_wpHandle;
        input [31:0] p_profile;
        input [31:0] p_value;
```

VHDL

```
procedure set_watchpoint_trigger (
        CONSTANT shellInstName : IN string;
        CONSTANT streamId: IN integer;
        CONSTANT wpHandle : IN integer;
        CONSTANT profile : IN integer;
        CONSTANT value : IN integer);
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• watch_for	Blocks the current command stream until a specific model event occurs.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• get_watchpoint_trigger	Defines a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

start

Starts model execution.

Queued: **No** Blocking: **No** Zero Cycle: **Yes**

Syntax

start

Description

The start command is issued in the initialization sequence for the model. All model parameters, such as bus width, must be set before start is issued.

Prototypes

OpenVera

```
task start;
```

Verilog

```
task start;
```

VHDL

```
procedure start (
        CONSTANT shellInstName : IN string );
```

Related Command

- reset_model Resets the model.

start_stream

Attention

Deprecated command; not recommend for new design. Use new_stream instead.

Starts a new command stream and returns the Stream ID of the new command stream.

Queued: **No** Blocking: **No**

Syntax

start_stream (*streamId***);**

Argument

streamId	An integer that specifies the command stream where the command is sent, returned by the new_stream command.

Description

The start_stream command returns a new *streamId* associated with a new command stream.

Attention

If the start_stream command is issued before the start command, the returned *streamId* is -1. If you attempt to use the -1 returned *streamId*, those commands will fail and will not be queued.

Note

Prior to issuing this command, the *streamId* to use for all commands is the default *streamId*, VMT_DEFAULT_STREAM_ID.

Prototypes

OpenVera

```
task start_stream (
        var integer newStreamId );
```

Verilog

```
task start_stream;
        inout [31:0] p_newStreamId;
```

VHDL

```
procedure start_stream (
        CONSTANT shellInstName : IN string;
        VARIABLE newStreamId : INOUT integer );
```

Related Commands

• end_stream	Ends execution of the command stream started by the new_ stream command.

watch_for

Blocks the current command stream until a specific model event occurs.

Queued: **No** Blocking: **Yes** Zero Cycle: **Yes**

Syntax

watch_for (*wpHandle*, *objHandle*)**;**

Argument

watchPoint	An integer event identifier.
handle	A returned integer handle to the event that triggered the watchpoint.

Note

When a watchpoint terminates because of the reset_model command, *handle* returns VMT_WP_TERMINATED_BY_RESET. For more information, see the description of the reset_model command.

Description

The watch_for command waits for an event from the model. The command blocks until the event happens.

Messages

- VMT_MSGID_INVALID_WATCHPOINT

Prototypes

OpenVera

```
task watch_for (
        integer wpHandle,
        var integer objHandle );
```

Verilog

```
task watch_for;
        input [31:0] p_wpHandle;
        inout [31:0] p_objHandle;
```

VHDL

```
procedure watch_for (
        CONSTANT shellInstName : IN string;
        CONSTANT wpHandle : IN integer;
        VARIABLE objHandle : INOUT integer );
```

Related Commands

• create_watchpoint	Defines a new watchpoint for one message type or identifier.
• create_watchpoint_range	Defines a new watchpoint for multiple message types or identifiers.
• combine_watchpoints	Defines a new watchpoint that is a boolean AND or OR of two previously-defined watchpoints.
• destroy_watchpoint	Removes a previously-created watchpoint.
• disable_watchpoint • enable_watchpoint	Enables or disables watch_for triggering of a watchpoint.
• set_watchpoint_trigger • get_watchpoint_trigger	Defines or returns a watchpoint triggering profile.
• get_watchpoint_data_count • get_watchpoint_data_name • get_watchpoint_data_type • get_watchpoint_data_size • get_watchpoint_data_int • get_watchpoint_data_string • get_watchpoint_data_bit • get_watchpoint_data_vec_<size>	Returns watchpoint data.

Command Macro Reference

VMT_CREATE_WP_MSG_TYPE

Attention

Deprecated macro; not recommend for new design. Use the command create_watchpoint (page 48) instead.

VMT macros are only supported under OpenVera control.

Creates watchpoints for different message types.

Syntax

VMT_CREATE_WP_MSG_TYPE (*types*, *handle***);**

Arguments

types	A 32-bit vector that defines the message types for which to create a watchpoint.
handle	A returned integer handle to the new watchpoint.

Description

The VMT_CREATE_WP_MSG_TYPE macro creates a watchpoint for a specified message type. See Table 9 on page 73 for a list of all message types.

VMT_CREATE_WP_MSG_ID

Attention

Deprecated macro; not recommend for new design. Use the command create_watchpoint (page 48) instead.

VMT macros are only supported under OpenVera control.

Creates watchpoints for different message IDs.

Syntax

VMT_CREATE_WP_MSG_ID (*msgID*, *handle***);**

Arguments

msgID	A returned integer set of predefined message IDs that are defined for each message that comes from a VMT model.
handle	A returned integer handle to the new watchpoint.

Description

The VMT_CREATE_WP_MSG_ID macro creates a watchpoint for a specified message IDs. Each message ID is documented within each command reference page in this chapter and also in all verification IP databooks for VMT models.

Some message IDs, such as VMT_MSGID_SID_INVALID, are common to most commands.

Fields: **Field Type:**

VMT Messages

VMT_MSGID_CMD_DONE
Command Done: <*cmd*> : (Tag = <*tag*>)
<*arglist*>
Msg Type: VMT_MSG_CMD

Fields:	**Field Type:**
VMT_MSGID_CMD_DONE_ARG_CMD	String
VMT_MSGID_CMD_DONE_ARG_TAG	Integer
VMT_MSGID_CMD_DONE_ARG_ARGLIST	String

VMT_MSGID_CMD_EXECUTE
Command Execute: <*cmd*> : (Tag = <*tag*>)
<*arglist*>
Msg Type: VMT_MSG_CMD

Fields:	**Field Type:**
VMT_MSGID_CMD_EXECUTE_ARG_CMD	String
VMT_MSGID_CMD_EXECUTE_ARG_TAG	Integer
VMT_MSGID_CMD_EXECUTE_ARG_ARGLIST	String

VMT_MSGID_CMD_QUEUED
Command Queued: <*cmd*> : (Tag = <*tag*>)
<*arglist*>
Msg Type: VMT_MSG_CMD

Fields:	**Field Type:**
VMT_MSGID_CMD_QUEUED_ARG_CMD	String
VMT_MSGID_CMD_QUEUED_ARG_TAG	Integer
VMT_MSGID_CMD_QUEUED_ARG_ARGLIST	String

VMT_MSGID_ADDR_EXIST_AS_FIFO
The address <*address*> already exists as a FIFO
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_ADDR_EXIST_AS_FIFO_ARG_ADDRESS	Integer

VMT_MSGID_ALREADY_STARTED
'start' has already been executed
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_BAD_WATCHPOINT_DATA_TYPE
'<*command*>' can not evaluate data type : <*type*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_BAD_WATCHPOINT_DATA_TYPE_ARG_COMMAND	String
VMT_MSGID_BAD_WATCHPOINT_DATA_TYPE_ARG_TYPE	Integer

VMT_MSGID_BLOCKING_CALLBACK
Model detected a blocking implementation for non-blocking call to callback method: <*callback*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_BLOCKING_CALLBACK_ARG_CALLBACK	String

VMT_MSGID_CCONTROL_INVALID_CBK_FLAG
Internal Error While Running C Testbench, Invalid Flag Passed to The CallBack Function.
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_CFG_ILLEGAL_PARAM
Cannot set <*param_name*> to <*value*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CFG_ILLEGAL_PARAM_ARG_PARAM_NAME	String
VMT_MSGID_CFG_ILLEGAL_PARAM_ARG_VALUE	Integer

VMT_MSGID_CFG_ILLEGAL_PARAM_COMBO
Cannot set <*param_name*> with value <*value*> because it conflicts with <*param_name2*> with value <*value2*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CFG_ILLEGAL_PARAM_COMBO_ARG_PARAM_NAME	String
VMT_MSGID_CFG_ILLEGAL_PARAM_COMBO_ARG_VALUE	Integer
VMT_MSGID_CFG_ILLEGAL_PARAM_COMBO_ARG_PARAM_NAME2	String
VMT_MSGID_CFG_ILLEGAL_PARAM_COMBO_ARG_VALUE2	Integer

VMT_MSGID_CFG_POST_START
Configuration error; '<*param_name*>' can not be changed after start
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CFG_POST_START_ARG_PARAM_NAME	String

VMT_MSGID_CMD_BEFORE_START
Non configure command '<*command*>' is not valid before 'start'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CMD_BEFORE_START_ARG_COMMAND	String

VMT_MSGID_CMD_INVALID_PARAM
'<*name*>' : Invalid parameter <*id*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CMD_INVALID_PARAM_ARG_NAME	String
VMT_MSGID_CMD_INVALID_PARAM_ARG_ID	Integer

VMT_MSGID_CMD_NOT_SUPPORTED
The model does not support the '<*name*>' command
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CMD_NOT_SUPPORTED_ARG_NAME	String

VMT_MSGID_CONFIG_PARAM_ERR
<*command*> : Invalid Parameter <*parameter*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_CONFIG_PARAM_ERR_ARG_COMMAND	String
VMT_MSGID_CONFIG_PARAM_ERR_ARG_PARAMETER	Integer

VMT_MSGID_COV_DATA_READ_FORMAT_ERROR
Cannot recognize '<*element*>' as coverage data element line while loading coverage data. Skipping this line
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_COV_DATA_READ_FORMAT_ERROR_ARG_ELEMENT String

VMT_MSGID_CP_EXISTS
Coverage point '<*coverage_point*>' already exists. Did not create new coverage point
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_CP_EXISTS_ARG_COVERAGE_POINT String

VMT_MSGID_DISABLE_FATAL_MSG
Can not DISABLE fatal message with id <*message_id*>.
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_DISABLE_FATAL_MSG_ARG_MESSAGE_ID Integer

VMT_MSGID_ENABLE_MSG_LOG_FAILURE
Unable to open message log file "<*filename*>".
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_ENABLE_MSG_LOG_FAILURE_ARG_FILENAME String

VMT_MSGID_EXCEEDED_INST_REG_LIMIT
Registration limit for this coverage instance has been exceeded, unable to do registerInstance
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_FAILED_AT_DISABLEFIFO_ADDRESS
Disable for fifo location <*location*> failed
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_FAILED_AT_DISABLEFIFO_ADDRESS_ARG_LOCATION Integer

VMT_MSGID_FIFOALLMEM_ALREADY_ENABLED
All the memory addresses are already enabled as FIFOs
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_FIFO_NOT_ENABLED
The address <*address*> of memory is not enabled as a FIFO location
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_FIFO_NOT_ENABLED_ARG_ADDRESS Integer

VMT_MSGID_FILE_WRITE_ERROR
Cannot open file '<*filename*>' to write coverage data to
Msg Type: VMT_MSG_ERROR
Fields: **Field Type:**
VMT_MSGID_FILE_WRITE_ERROR_ARG_FILENAME String

VMT_MSGID_ILLEGAL_DISABLE_FIFOALLMEM_ENABLED
Cannot disable individual address when all the memory addresses are enabled as FIFO
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_ILLEGAL_ENABLE_FIFOALLMEM_ENABLED
All the memory addresses are already enabled as a FIFO
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_ILLEGAL_OPERATION_IN_NON_RVM_MODE
Illegal notify operation while not in RVM mode
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_INTEGER_PARAMETER_CONTAINS_X
<*command*> : Integer parameter '<*parameter_name*>' contains X (<*value*>)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INTEGER_PARAMETER_CONTAINS_X_ARG_COMMAND	String
VMT_MSGID_INTEGER_PARAMETER_CONTAINS_X_ARG_PARAMETER_NAME	String
VMT_MSGID_INTEGER_PARAMETER_CONTAINS_X_ARG_VALUE	Integer

VMT_MSGID_INVALID_ADDR
Invalid address: <*address*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_ADDR_ARG_ADDRESS	Integer

VMT_MSGID_INVALID_CHANNEL_ID
<*command*> : Invalid argument : channelId (<*channel_id*>) does not exist
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_CHANNEL_ID_ARG_COMMAND	String
VMT_MSGID_INVALID_CHANNEL_ID_ARG_CHANNEL_ID	Integer

VMT_MSGID_INVALID_CMD_HANDLE
Invalid command handle : <*handle*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_CMD_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_INVALID_CROSS_STREAM_HANDLE
Handle : <*handle*> was not found in current stream <*current_stream*>, but as a non started command in stream <*found_stream*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_CROSS_STREAM_HANDLE_ARG_HANDLE	Integer
VMT_MSGID_INVALID_CROSS_STREAM_HANDLE_ARG_CURRENT_STREAM	Integer
VMT_MSGID_INVALID_CROSS_STREAM_HANDLE_ARG_FOUND_STREAM	Integer

VMT_MSGID_INVALID_DATAWIDTH
Invalid data width: <*width*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_DATAWIDTH_ARG_WIDTH	Integer

VMT_MSGID_INVALID_DLYCLKS_ARG
<*command*> : Invalid argument : numDelayClocks (<*delay_clock*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_DLYCLKS_ARG_ARG_COMMAND	String
VMT_MSGID_INVALID_DLYCLKS_ARG_ARG_DELAY_CLOCK	Integer

VMT_MSGID_INVALID_DRVCLKS_ARG
<*command*> : Invalid argument : numDriveClocks (<*drive_clocks*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_DRVCLKS_ARG_ARG_COMMAND	String
VMT_MSGID_INVALID_DRVCLKS_ARG_ARG_DRIVE_CLOCKS	Integer

VMT_MSGID_INVALID_FIFO_INDEX
The index <*index*> is not a valid index in the FIFO at address <*address*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_FIFO_INDEX_ARG_INDEX	Integer
VMT_MSGID_INVALID_FIFO_INDEX_ARG_ADDRESS	Integer

VMT_MSGID_INVALID_FORCE_BLOCK_PARAM
<*command*> : Invalid VMT_FORCE_CMD_BLOCKING value <*value*> use (ON/OFF)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_FORCE_BLOCK_PARAM_ARG_COMMAND	String
VMT_MSGID_INVALID_FORCE_BLOCK_PARAM_ARG_VALUE	Integer

VMT_MSGID_INVALID_MSGID
'<*command*>' got an invalid message id: <*id*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_MSGID_ARG_COMMAND	String
VMT_MSGID_INVALID_MSGID_ARG_ID	Integer

VMT_MSGID_INVALID_MSG_LOG_ID
<*command*> : Invalid message log id <*log_id*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_MSG_LOG_ID_ARG_COMMAND	String
VMT_MSGID_INVALID_MSG_LOG_ID_ARG_LOG_ID	Integer

VMT_MSGID_INVALID_NUM_WORDS
Invalid number of words: <*words*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_NUM_WORDS_ARG_WORDS	Integer

VMT_MSGID_INVALID_PARAM
<*command*> : Invalid argument : parameter (<*parameter*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_PARAM_ARG_COMMAND	String
VMT_MSGID_INVALID_PARAM_ARG_PARAMETER	Integer

VMT_MSGID_INVALID_PATTERN_BASE
Invalid memory pattern base: 'b<*pattern_base*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_PATTERN_BASE_ARG_PATTERN_BASE	Integer

VMT_MSGID_INVALID_PORTID
<*command*> : Invalid argument : portId (<*port*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_PORTID_ARG_COMMAND	String
VMT_MSGID_INVALID_PORTID_ARG_PORT	Integer

VMT_MSGID_INVALID_PORT_WIDTH
Model port '<*port*>' width '<*width*>' does not match the configured width '<*config_width*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_PORT_WIDTH_ARG_PORT	String
VMT_MSGID_INVALID_PORT_WIDTH_ARG_WIDTH	Integer
VMT_MSGID_INVALID_PORT_WIDTH_ARG_CONFIG_WIDTH	Integer

VMT_MSGID_INVALID_REGISTER_ID
<*command*> : Invalid argument : registerId (<*register_id*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_REGISTER_ID_ARG_COMMAND	String
VMT_MSGID_INVALID_REGISTER_ID_ARG_REGISTER_ID	Integer

VMT_MSGID_INVALID_RESET_TYPE
<*command*> : Invalid argument : resetType (<*reset_type*>) must be either <*supported_types*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_RESET_TYPE_ARG_COMMAND	String
VMT_MSGID_INVALID_RESET_TYPE_ARG_RESET_TYPE	Integer
VMT_MSGID_INVALID_RESET_TYPE_ARG_SUPPORTED_TYPES	String

VMT_MSGID_INVALID_RVM_XACT
Model got an invalid transaction on an input channel
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_INVALID_SID
<*command*> : Invalid streamId (<*stream*>)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_SID_ARG_COMMAND	String
VMT_MSGID_INVALID_SID_ARG_STREAM	Integer

VMT_MSGID_INVALID_TIMEOUT_ARG
<*command*> : Invalid argument : timeout (<*timeout*>) must be >= 0
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_TIMEOUT_ARG_ARG_COMMAND	String
VMT_MSGID_INVALID_TIMEOUT_ARG_ARG_TIMEOUT	Integer

VMT_MSGID_INVALID_WATCHPOINT
'watch_for' got an invalid handle <*handle*>; stream will suspend indefinitely
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_ARG_HANDLE	Integer

VMT_MSGID_INVALID_WATCHPOINT_DATA_HANDLE
'<*command*>' got an invalid watchpoint data handle <*handle*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_DATA_HANDLE_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_DATA_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_INVALID_WATCHPOINT_DATA_POSITION
'<*command*>' got an invalid watchpoint data position <*position*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_DATA_POSITION_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_DATA_POSITION_ARG_POSITION	Integer

VMT_MSGID_INVALID_WATCHPOINT_DATA_STRING_LINE
'<*command*>' got an invalid line (out of bounds): <*line*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_DATA_STRING_LINE_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_DATA_STRING_LINE_ARG_LINE	Integer

VMT_MSGID_INVALID_WATCHPOINT_DATA_VEC_WORD
'<*command*>' got an invalid word (out of bounds): <*word*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_DATA_VEC_WORD_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_DATA_VEC_WORD_ARG_WORD	Integer

VMT_MSGID_INVALID_WATCHPOINT_HANDLE
'<*command*>' got an invalid watchpoint handle <*handle*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_HANDLE_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_INVALID_WATCHPOINT_LOGIC
'<*command*>' got an invalid logic operator value: <*operator*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_LOGIC_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_LOGIC_ARG_OPERATOR	Integer

VMT_MSGID_INVALID_WATCHPOINT_PROFILE
'<*command*>' got an invalid profile: <*profile*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_PROFILE_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_PROFILE_ARG_PROFILE	Integer

VMT_MSGID_INVALID_WATCHPOINT_PROFILE_VALUE
'<*command*>' got an invalid profile value: <*value*>.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_PROFILE_VALUE_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_PROFILE_VALUE_ARG_VALUE	Integer

VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT
'<*command*>' can not create a watchpoint by message id <*label*>(<*id*>) because its notification events are suppressed.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT_ARG_COMMAND	String
VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT_ARG_LABEL	String
VMT_MSGID_INVALID_WATCHPOINT_SUPPRESSED_EVENT_ARG_ID	Integer

VMT_MSGID_LIC_SLI_ERROR
Encountered SLI error '<*error*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_LIC_SLI_ERROR_ARG_ERROR	String

VMT_MSGID_LIST_INDEX_ERROR
The location <*location*> for the FIFO at address <*address*> is not filled
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_LIST_INDEX_ERROR_ARG_LOCATION	Integer
VMT_MSGID_LIST_INDEX_ERROR_ARG_ADDRESS	Integer

VMT_MSGID_MATCHING_COV_DATA_ELEM_ERROR
Matching coverage data element '<*element*>' exists. Skipping this line
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_MATCHING_COV_DATA_ELEM_ERROR_ARG_ELEMENT	String

VMT_MSGID_OPEN_LOG_FAILURE
Unable to open Output mcd File (<*filename*>), model change dump file will not be created.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_OPEN_LOG_FAILURE_ARG_FILENAME	String

VMT_MSGID_OPEN_MSG_LOG_FAILURE
Unable to open message log file "<*filename*>".
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_OPEN_MSG_LOG_FAILURE_ARG_FILENAME	String

VMT_MSGID_PORT_CONFLICT
set_port : Port <*id*> drive conflict; move set_port ahead of command you want to affect
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_PORT_CONFLICT_ARG_ID	Integer

VMT_MSGID_PORT_NOT_CONNECTED
Model port '<*port*>' not connected
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_PORT_NOT_CONNECTED_ARG_PORT	String

VMT_MSGID_PROGRAMMABLE_COV_UNSUPPORTED
Programmable coverage is not supported, <*method*> request ignored.
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_PROGRAMMABLE_COV_UNSUPPORTED_ARG_METHOD	String

VMT_MSGID_SS_BEFORE_START
command <*command*> is not valid before 'start'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_SS_BEFORE_START_ARG_COMMAND	String

VMT_MSGID_TRIGGER_ALREADY_ADDED
The trigger at location <*location*> for FIFO at address <*address*> is already present
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_TRIGGER_ALREADY_ADDED_ARG_LOCATION	Integer
VMT_MSGID_TRIGGER_ALREADY_ADDED_ARG_ADDRESS	Integer

VMT_MSGID_TRIGGER_NOT_PRESENT
Trigger is not present at index <*index*> in Fifo at address <*address*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_TRIGGER_NOT_PRESENT_ARG_INDEX	Integer
VMT_MSGID_TRIGGER_NOT_PRESENT_ARG_ADDRESS	Integer

VMT_MSGID_UNABLE_TO_FIND_REG_INST
Coverage instance not registered, unable to do unregisterInstance
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_UNALIGNED_ACCESS
Unaligned access: addr: <*address*>, data width: <*width*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_UNALIGNED_ACCESS_ARG_ADDRESS	Integer
VMT_MSGID_UNALIGNED_ACCESS_ARG_WIDTH	Integer

VMT_MSGID_UNKNOWN_FLUSH_TYPE
Unknown flush type (<*type*>)
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_UNKNOWN_FLUSH_TYPE_ARG_TYPE	Integer

VMT_MSGID_UNKNOWN_PATTERN
Unknown memory pattern
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_UNKNOWN_PORT
<*operation*> Port : Bad Port <*id*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_UNKNOWN_PORT_ARG_OPERATION	String
VMT_MSGID_UNKNOWN_PORT_ARG_ID	Integer

VMT_MSGID_UNKNOWN_WATCHPOINT_ID
'create_watchpoint' got an unknown watchpoint id : <*id*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_UNKNOWN_WATCHPOINT_ID_ARG_ID	Integer

VMT_MSGID_UNKNOWN_WATCHPOINT_TYPE
'create_watchpoint' got an unknown watchpoint type : <*type*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_UNKNOWN_WATCHPOINT_TYPE_ARG_TYPE	Integer

VMT_MSGID_XACT_ALL_ITEMS_ILLEGAL_IN_DISALLOW
VMT_XACT_ALL_ITEMS is an illegal item identifier in disallow_enum_item
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_XACT_ALL_ZERO_WEIGHTS
At least one weight value for attribute '<*attribute*>' has to be greater than zero
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_ALL_ZERO_WEIGHTS_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_BAD_ATTR_USAGE
Attribute '<*attribute*>' can not be used in an expression
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_ATTR_USAGE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_BAD_EXPR_ATTR
Attribute '<*attribute*>' does not exsist on protocol transaction item with index <*index*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_EXPR_ATTR_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_BAD_EXPR_ATTR_ARG_INDEX	Integer

VMT_MSGID_XACT_BAD_REPEAT_RANGE_WEIGHT
Invalid weight (<*weight*>) for repeat range, can not be negative
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_REPEAT_RANGE_WEIGHT_ARG_WEIGHT	Integer

VMT_MSGID_XACT_BAD_THROTTLE_LIMIT
Bad throttle limits (low = <*low_limit*> high = <*high_limit*>); low limit must be less than high limit and greater than '0'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_THROTTLE_LIMIT_ARG_LOW_LIMIT	Integer
VMT_MSGID_XACT_BAD_THROTTLE_LIMIT_ARG_HIGH_LIMIT	Integer

VMT_MSGID_XACT_BITVEC_RANGE_ERROR
Low value (<*low_value*>) is greater than high value (<*high_value*>) for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_BITVEC_RANGE_ERROR_ARG_LOW_VALUE	Integer
VMT_MSGID_XACT_BITVEC_RANGE_ERROR_ARG_HIGH_VALUE	Integer
VMT_MSGID_XACT_BITVEC_RANGE_ERROR_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_CANNOT_GET_HIDDEN_CP
<*coverage_point*> is a hidden coverage point. Cannot get its reference.
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CANNOT_GET_HIDDEN_CP_ARG_COVERAGE_POINT	String

VMT_MSGID_XACT_CANNOT_MODIFY_READONLY_CP
<*coverage_point*> is a read-only coverage point. Cannot <*operation*>.
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CANNOT_MODIFY_READONLY_CP_ARG_COVERAGE_POINT	String
VMT_MSGID_XACT_CANNOT_MODIFY_READONLY_CP_ARG_OPERATION	String

VMT_MSGID_XACT_CHOICE_LIMIT
Total number of items in a choice are more than max allowed : <*max_items*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CHOICE_LIMIT_ARG_MAX_ITEMS	Integer

VMT_MSGID_XACT_CHOICE_NEG_WEIGHT
Weights for choice items can not be less than zero
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_XACT_CI_CLEARED
Item in the <*xact*> to be evaluated is missing. Finishing evaluation with a failure
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CI_CLEARED_ARG_XACT	String

Extended Msg Text and Args:
The item was probably removed by a clear_item command
Fields: None

VMT_MSGID_XACT_CONSTRAINTS_LIMIT
Total number of constraints for attribute '<*attribute*>' are more than max allowed : <*max_weights*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CONSTRAINTS_LIMIT_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_CONSTRAINTS_LIMIT_ARG_MAX_WEIGHTS	Integer

VMT_MSGID_XACT_CONSTRAINT_ERROR
Invalid string '<*constraint*>' as part of constraint for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_CONSTRAINT_ERROR_ARG_CONSTRAINT	String
VMT_MSGID_XACT_CONSTRAINT_ERROR_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_DIFF_ASSIGN_TYPES
Assignment types for attribute '<*attribute*>' are different; use only weights or only percentages
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_DIFF_ASSIGN_TYPES_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_EMPTY_CHOICE
Can not execute a choice with no transactions, or all weights set to '0'
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_XACT_EMPTY_SEQUENCE
Can not execute a sequence with no transactions
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_XACT_ENUM_REUSE
Enumerated identifier '<*value*>' for attribute '<*attribute*>' can only be assigned weight once
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_ENUM_REUSE_ARG_VALUE	String
VMT_MSGID_XACT_ENUM_REUSE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_EXPR_C_ATTR_ERR
<message>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_EXPR_C_ATTR_ERR_ARG_MESSAGE	String

VMT_MSGID_XACT_EXPR_C_EVAL_ERR
<message>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_EXPR_C_EVAL_ERR_ARG_MESSAGE	String

VMT_MSGID_XACT_EXPR_C_EXPR_ERR
<message>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_EXPR_C_EXPR_ERR_ARG_MESSAGE	String

VMT_MSGID_XACT_EXPR_C_PARSE_ERR
Attribute relations parse error : <*message*>
expression : '<*expression*>'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_EXPR_C_PARSE_ERR_ARG_MESSAGE	String
VMT_MSGID_XACT_EXPR_C_PARSE_ERR_ARG_EXPRESSION	String

VMT_MSGID_XACT_EXPR_TYPE_MISMATCH
Attribute type mismatch for sub-expression <*eid*> in '<*expression*>'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_EXPR_TYPE_MISMATCH_ARG_EID	Integer
VMT_MSGID_XACT_EXPR_TYPE_MISMATCH_ARG_EXPRESSION	String

VMT_MSGID_XACT_ILLEGAL_HIT_LIMIT_VAL
Illegal hit count limit (<*limit*>) for coverage point '<*coverage_point*>'. Hit count limit must be a positive integer.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_ILLEGAL_HIT_LIMIT_VAL_ARG_LIMIT	Integer
VMT_MSGID_XACT_ILLEGAL_HIT_LIMIT_VAL_ARG_COVERAGE_POINT	String

VMT_MSGID_XACT_ILLEGAL_NOTIFY_ID_VAL
Illegal notification ID (<*id*>) for coverage point '<*coverage_point*>'. Notification ID must be a positive integer.
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_ILLEGAL_NOTIFY_ID_VAL_ARG_ID	Integer
VMT_MSGID_XACT_ILLEGAL_NOTIFY_ID_VAL_ARG_COVERAGE_POINT	String

VMT_MSGID_XACT_INFO_NO_GENERATOR
The info object with handle <*handle*>, does not contain a valid generator
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INFO_NO_GENERATOR_ARG_HANDLE	Integer

VMT_MSGID_XACT_INFO_NO_HANDLE
The info object with handle <*handle*>, does not contain a valid data handle
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INFO_NO_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_XACT_INT_RANGE_ERROR
Low value (<*low_value*>) is greater than high value (<*high_value*>) for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INT_RANGE_ERROR_ARG_LOW_VALUE	Integer
VMT_MSGID_XACT_INT_RANGE_ERROR_ARG_HIGH_VALUE	Integer
VMT_MSGID_XACT_INT_RANGE_ERROR_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_INVALID_ATTR_NAME
Invalid attribute name : <*name*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_ATTR_NAME_ARG_NAME	String

VMT_MSGID_XACT_INVALID_ATTR_TYPE_COMB
The attributte '<*attribute*>' is not of type <*xact_type*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_ATTR_TYPE_COMB_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_INVALID_ATTR_TYPE_COMB_ARG_XACT_TYPE	String

VMT_MSGID_XACT_INVALID_ATTR_VALUE_SET
After relationship applied, attribute (<*attribute*>) gets invalid value (d<*dec_value*>, 0x<*hex_value*>)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_ATTR_VALUE_SET_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_INVALID_ATTR_VALUE_SET_ARG_DEC_VALUE	Integer
VMT_MSGID_XACT_INVALID_ATTR_VALUE_SET_ARG_HEX_VALUE	Integer

VMT_MSGID_XACT_INVALID_CHOICE_IDX
Invalid item index (<*index*>) for choice item
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_CHOICE_IDX_ARG_INDEX	Integer

VMT_MSGID_XACT_INVALID_CONSTRAINT_ATTR
Invalid attribute name : <*attribute*> in constraint '<*constraint*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_CONSTRAINT_ATTR_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_INVALID_CONSTRAINT_ATTR_ARG_CONSTRAINT	String

VMT_MSGID_XACT_INVALID_ENUM_ID
Invalid enumerated identifier '<*value*>' for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_ENUM_ID_ARG_VALUE	String
VMT_MSGID_XACT_INVALID_ENUM_ID_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_INVALID_ENUM_VALUE
Invalid enumerated value <*value*> for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_ENUM_VALUE_ARG_VALUE	Integer
VMT_MSGID_XACT_INVALID_ENUM_VALUE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_INVALID_HANDLE
Invalid <*type*> : <*handle*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_HANDLE_ARG_TYPE	String
VMT_MSGID_XACT_INVALID_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_XACT_INVALID_INFO_HANDLE
Invalid info object handle <*handle*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_INFO_HANDLE_ARG_HANDLE	Integer

VMT_MSGID_XACT_INVALID_PAYLOAD_TYPE
Invalid payload type, method only valid on payload of type <*payload_type*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_PAYLOAD_TYPE_ARG_PAYLOAD_TYPE	String

VMT_MSGID_XACT_INVALID_SEQUENCE_IDX
Invalid item index (<*index*>) for sequence item
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_SEQUENCE_IDX_ARG_INDEX	Integer

VMT_MSGID_XACT_INVALID_THROTTLE_LIMIT
Invalid throttle limit min/max (<*low_limit*>/<*high_limit*>); min must be atleast <*min_limit*> and max > min
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_THROTTLE_LIMIT_ARG_LOW_LIMIT	Integer
VMT_MSGID_XACT_INVALID_THROTTLE_LIMIT_ARG_HIGH_LIMIT	Integer
VMT_MSGID_XACT_INVALID_THROTTLE_LIMIT_ARG_MIN_LIMIT	Integer

VMT_MSGID_XACT_INVALID_WEIGHT_VALUE
Invalid weight (<*weight*>), can not be negative, for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_INVALID_WEIGHT_VALUE_ARG_WEIGHT	Integer
VMT_MSGID_XACT_INVALID_WEIGHT_VALUE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_LONG_CONSTRAINT_ERR
Constraints string '<*constraint*>' to long; max 256 char
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_LONG_CONSTRAINT_ERR_ARG_CONSTRAINT	String

VMT_MSGID_XACT_MULTIPLE_STARS
Only one star is valid in constraints for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_MULTIPLE_STARS_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_NEGATIVE_WEIGHT
Negative weight (<*weight*>) on Attribute (<*attribute*>)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NEGATIVE_WEIGHT_ARG_WEIGHT	Integer
VMT_MSGID_XACT_NEGATIVE_WEIGHT_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_NOT_A_PTG
Specified reference transaction (<*index*>) is not a protocol transaction
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NOT_A_PTG_ARG_INDEX	Integer

VMT_MSGID_XACT_NO_ATTR_NAME
No attribute name found in string '<*expression*>'
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NO_ATTR_NAME_ARG_EXPRESSION	String

VMT_MSGID_XACT_NO_CP_BY_NAME
<*coverage_point*>: Coverage point by the name '<*name*>' could not be found in this model
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NO_CP_BY_NAME_ARG_COVERAGE_POINT	String
VMT_MSGID_XACT_NO_CP_BY_NAME_ARG_NAME	String

VMT_MSGID_XACT_NO_PAYLOAD_FILE
Could not open payload file : <*filename*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NO_PAYLOAD_FILE_ARG_FILENAME	String

VMT_MSGID_XACT_NO_XACT_ON_STREAM
No start_transactions command is associated with streamId <*stream_id*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_NO_XACT_ON_STREAM_ARG_STREAM_ID	Integer

VMT_MSGID_XACT_OPEN_PLAYBACK_FAILURE
Unable to open random playback input file (<*filename*>)
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_OPEN_PLAYBACK_FAILURE_ARG_FILENAME	String

VMT_MSGID_XACT_PAYLOAD_READ_ERR
<message>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_PAYLOAD_READ_ERR_ARG_MESSAGE	String

VMT_MSGID_XACT_PAYLOAD_WIDTH_ERR
Invalid width specification : <*width*>; should be between 1 and <*max_width*>
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_PAYLOAD_WIDTH_ERR_ARG_WIDTH	Integer
VMT_MSGID_XACT_PAYLOAD_WIDTH_ERR_ARG_MAX_WIDTH	Integer

VMT_MSGID_XACT_PERCENT_TOO_LARGE
Total weight values for attribute '<*attribute*>' add up to more that 100 percent
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_PERCENT_TOO_LARGE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_PERCENT_TOO_SMALL
Total weight values for attribute '<*attribute*>' does not add up to 100 percent
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_PERCENT_TOO_SMALL_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_REFERENCE_INDEX
Invalid reference item index (<*index*>) for attribute relationship
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_REFERENCE_INDEX_ARG_INDEX	Integer

VMT_MSGID_XACT_RELATION_INDEX
Invalid item index (<*index*>) for attribute relationship
Msg Type: VMT_MSG_ERROR

Fields:	**Field Type:**
VMT_MSGID_XACT_RELATION_INDEX_ARG_INDEX	Integer

VMT_MSGID_XACT_REPEAT_RANGE_ERROR
Low value (<*low*>) is greater than high value (<*high*>) for repeat range
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_REPEAT_RANGE_ERROR_ARG_LOW	Integer
VMT_MSGID_XACT_REPEAT_RANGE_ERROR_ARG_HIGH	Integer

VMT_MSGID_XACT_REPEAT_RANGE_INDEX
Invalid item index (<*index*>) for repeat range
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_REPEAT_RANGE_INDEX_ARG_INDEX	Integer

VMT_MSGID_XACT_REPEAT_RANGE_ITEM
Could not add repeat range to last item
Msg Type: VMT_MSG_ERROR
Fields: None

VMT_MSGID_XACT_REPEAT_RANGE_LIMIT
Total number of repeat ranges are more than max allowed : <*max_ranges*>
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_REPEAT_RANGE_LIMIT_ARG_MAX_RANGES	Integer

VMT_MSGID_XACT_SC_MISSING_CONSTRAINTS
No constraints provided for attribute '<*attribute*>'
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_SC_MISSING_CONSTRAINTS_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_SC_TYPE_MISMATCH
Constraint value mismatch. Attribute '<*attribute*>' expects '<*type*>' type values
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_SC_TYPE_MISMATCH_ARG_ATTRIBUTE	String
VMT_MSGID_XACT_SC_TYPE_MISMATCH_ARG_TYPE	String

VMT_MSGID_XACT_STREAM_HAS_XACT
The streamId <*stream_id*> already has an active transaction generator
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_STREAM_HAS_XACT_ARG_STREAM_ID	Integer

VMT_MSGID_XACT_WRONG_GET_METHOD_ON_ATTR
Incorrect get method used to access attribute(<*attribute*>).
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_WRONG_GET_METHOD_ON_ATTR_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_WRONG_MODEL_TYPE
Transaction generator (<*xact_type*>) gets wrong model type input (<*model*>)
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_WRONG_MODEL_TYPE_ARG_XACT_TYPE	String
VMT_MSGID_XACT_WRONG_MODEL_TYPE_ARG_MODEL	String

VMT_MSGID_XACT_WRONG_SET_METHOD_ON_ATTR
Incorrect set method used to change attribute(<*attribute*>) weight.
Msg Type: VMT_MSG_ERROR
Fields:

	Field Type:
VMT_MSGID_XACT_WRONG_SET_METHOD_ON_ATTR_ARG_ATTRIBUTE	String

VMT_MSGID_CCONTROL_AIX_IS_UNSUP
Unable To Run C Testench, C Control Is Not Supported On AIX.
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_CCONTROL_CINSTANCEID_NOT_SET
Unable To Run C Testbench, CInstanceId Attribute Is Not Set. Please Set This Attribute And ReRun The Simulation.
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_CCONTROL_CMDHANDLE_FETCH_FAIL
Error While Running C Testbench, Unable To Fetch Cmd Handles From VmtModelManager.
<*error*>
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_CCONTROL_CMDHANDLE_FETCH_FAIL_ARG_ERROR	String

VMT_MSGID_CCONTROL_MTIVHDL_LINUX_IS_UNSUP
Unable To Run C Testench, In Order To Run C Control On Linux With MTI-Vhdl You Need To Use Vera Version 6 Or Higher.
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_CCONTROL_NOT_SUPPORTED
Model Instance(<*name*>) Does Not Support C Control.
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_CCONTROL_NOT_SUPPORTED_ARG_NAME	String

VMT_MSGID_CCONTROL_PENDING_REG_FAILURE
<error>
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_CCONTROL_PENDING_REG_FAILURE_ARG_ERROR	String

VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_DIR
Internal Error, Illegal Direction(<*direction*>) Encountered While Connecting To Port(<*port*>) With Width(<*width*>).
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_DIR_ARG_DIRECTION	Integer
VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_DIR_ARG_PORT	String
VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_DIR_ARG_WIDTH	Integer

VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_IDX
Internal Error, Illegal Index(<*index*>) Passed To(<*command*>).
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_IDX_ARG_INDEX	Integer
VMT_MSGID_CCONTROL_PORT_ACC_ILLEGAL_IDX_ARG_COMMAND	String

VMT_MSGID_CCONTROL_PORT_ACC_SIG_CONN_FAIL
Port Access Failure. Unable To Connect To Node(<*node*>) With Width(<*width*>) And Direction(<*direction*>).
Possible Causes Of Failure Are:
- Your Simulator Was Not Compiled With The Correct Options, Refer To The Users Guide For More Details.
- Path To The Port Is Specified Incorrectly , Refer To The Users Guide For More Details.
- Width or Direction Attributes Are Incorrectly Specified.

Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_CCONTROL_PORT_ACC_SIG_CONN_FAIL_ARG_NODE	String
VMT_MSGID_CCONTROL_PORT_ACC_SIG_CONN_FAIL_ARG_WIDTH	Integer
VMT_MSGID_CCONTROL_PORT_ACC_SIG_CONN_FAIL_ARG_DIRECTION	String

VMT_MSGID_CCONTROL_RUN_C_TESTBENCH_FAIL
Unable To Run C Testbench.
<*error*>
Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_CCONTROL_RUN_C_TESTBENCH_FAIL_ARG_ERROR	String

VMT_MSGID_CCONTROL_UNABLE_TO_RUN_C
Unable To Run C Testench, Registration For This Instance Failed. Turn On Warnings To See The Registration Failure.
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_CCONTROL_UNSUPP_MODEL_VERSION
Model(<*name*>) Is Not C Control Compatible.
Please Update The Model In Order To Use C Control.
Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_CCONTROL_UNSUPP_MODEL_VERSION_ARG_NAME	String

VMT_MSGID_CCONTROL_VERA_CALLBACK_FAIL
<error>
Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_CCONTROL_VERA_CALLBACK_FAIL_ARG_ERROR	String

VMT_MSGID_EMPTY_INSTANCE_NAME
Invalid Instance Name, Instance Name cannot be an empty string.
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_LIC_ACTCHECKOUT_FAIL
ACT license checkout failure. ACT authorization not granted for model '<*name*>'
Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_LIC_ACTCHECKOUT_FAIL_ARG_NAME	String

VMT_MSGID_LIC_CHECKOUT_FAIL
License checkout failure. Authorization not granted for model '<*name*>'
Msg Type: VMT_MSG_FATAL

Fields:	**Field Type:**
VMT_MSGID_LIC_CHECKOUT_FAIL_ARG_NAME	String

VMT_MSGID_RESCOURE_ALLOCATION
Unable to allocate Semaphore/Region resources
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_XACT_BAD_ATTR_TYPE
Bad attribute type : <*type*> for attribute '<*attribute*>'
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_ATTR_TYPE_ARG_TYPE	Integer
VMT_MSGID_XACT_BAD_ATTR_TYPE_ARG_ATTRIBUTE	String

VMT_MSGID_XACT_BAD_PLAYBACK_DATA
Invalid random playback data : '<*data*>'
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_XACT_BAD_PLAYBACK_DATA_ARG_DATA	String

VMT_MSGID_XACT_BAD_XACT
Bad transaction handle
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_XACT_FATAL_FUNCTION
This method '<*method*>' is not supported by this transaction type
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_XACT_FATAL_FUNCTION_ARG_METHOD	String

VMT_MSGID_XACT_INVALID_ATTRID
Invalid attribute id <*attr_id*>
Msg Type: VMT_MSG_FATAL
Fields:

	Field Type:
VMT_MSGID_XACT_INVALID_ATTRID_ARG_ATTR_ID	Integer

VMT_MSGID_XACT_PLAYBACK_SEQ_FAILURE
Random sequence in playback file does not match expected model sequence
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_XACT_RESERVED_ATTR_NAME
'ALL_ATTRIBUTES' is a reserved attribute name
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_XACT_RESERVED_ENUM_VALUE
'ALL_ITEMS' is a reserved enumerated value
Msg Type: VMT_MSG_FATAL
Fields: None

VMT_MSGID_ALL_MEM_FULLEMPTY_ON
The FIFO feature is set for all addresses. So the full and empty messages have been enabled for the entire address range
Msg Type: VMT_MSG_NOTE
Fields: None

VMT_MSGID_CCONTROL_C_TESTBENCH_END
End C/C++ Testbench(<*name*>).
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_CCONTROL_C_TESTBENCH_END_ARG_NAME	String

VMT_MSGID_CCONTROL_C_TESTBENCH_START
Start C/C++ Testbench(<*name*>).
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_CCONTROL_C_TESTBENCH_START_ARG_NAME	String

VMT_MSGID_ENABLE_MSG_LOG_SUCCESS
Enabled message log file "<*filename*>" with log id <*log_id*>.
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_ENABLE_MSG_LOG_SUCCESS_ARG_FILENAME	String
VMT_MSGID_ENABLE_MSG_LOG_SUCCESS_ARG_LOG_ID	Integer

VMT_MSGID_ENABLE_RVM_MODE
Enable RVM Mode
Msg Type: VMT_MSG_NOTE
Fields: None

VMT_MSGID_FIFO_EMPTY_AT_ADDRESS
The FIFO at address <*address*> is empty
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_FIFO_EMPTY_AT_ADDRESS_ARG_ADDRESS	Integer

VMT_MSGID_FIFO_FULL_AT_ADDRESS
The FIFO at address <*address*> is full
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_FIFO_FULL_AT_ADDRESS_ARG_ADDRESS	Integer

VMT_MSGID_INCR_COV_ON
Incremental coverage enabled
Msg Type: VMT_MSG_NOTE
Fields: None

VMT_MSGID_OPEN_MSG_LOG_SUCCESS
Opened message log file "<*filename*>" with log id <*log_id*>.
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_OPEN_MSG_LOG_SUCCESS_ARG_FILENAME	String
VMT_MSGID_OPEN_MSG_LOG_SUCCESS_ARG_LOG_ID	Handle

VMT_MSGID_READ_TRIGGER_AT_FIFO_INDEX_ADDRESS
Read Trigger at FIFO at index <*index*> at address <*address*> reached
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_READ_TRIGGER_AT_FIFO_INDEX_ADDRESS_ARG_INDEX	Integer
VMT_MSGID_READ_TRIGGER_AT_FIFO_INDEX_ADDRESS_ARG_ADDRESS	Integer

VMT_MSGID_SUMMARY
Summary of messages:
<*summary*>
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_SUMMARY_ARG_SUMMARY	String

VMT_MSGID_WRITE_TRIGGER_AT_FIFO_INDEX_ADDRESS
Write Trigger at FIFO at index <*index*> at address <*address*> reached
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_WRITE_TRIGGER_AT_FIFO_INDEX_ADDRESS_ARG_INDEX	Integer
VMT_MSGID_WRITE_TRIGGER_AT_FIFO_INDEX_ADDRESS_ARG_ADDRESS	Integer

VMT_MSGID_XACT_COVERAGE_POINT_MATCHED
Coverage point '<*coverage_point*>' matched. Lasted over <*clocks*> clocks [<*start*> -> <*stop*>]
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_XACT_COVERAGE_POINT_MATCHED_ARG_COVERAGE_POINT	String
VMT_MSGID_XACT_COVERAGE_POINT_MATCHED_ARG_CLOCKS	Integer
VMT_MSGID_XACT_COVERAGE_POINT_MATCHED_ARG_START	Integer
VMT_MSGID_XACT_COVERAGE_POINT_MATCHED_ARG_STOP	Integer

VMT_MSGID_XACT_PAYLOAD_EXHAUSTED
Payload exhausted
Msg Type: VMT_MSG_NOTE
Fields: None

VMT_MSGID_XACT_REPORT
<report>
Msg Type: VMT_MSG_NOTE
Fields:

	Field Type:
VMT_MSGID_XACT_REPORT_ARG_REPORT	String

VMT_MSGID_PRINT_MSG_TEXT
<text>
Msg Type: VMT_MSG_REPORT
Fields:

	Field Type:
VMT_MSGID_PRINT_MSG_TEXT_ARG_TEXT	String

VMT_MSGID_TR_REPORT
<report>
Msg Type: VMT_MSG_REPORT
Fields:

	Field Type:
VMT_MSGID_TR_REPORT_ARG_REPORT	String

VMT_MSGID_XACT_DUMP
<xact>
Msg Type: VMT_MSG_REPORT
Fields:

	Field Type:
VMT_MSGID_XACT_DUMP_ARG_XACT	String

VMT_MSGID_BLOCKED_SID
<*command*> : Command sent to blocked streamId (<*stream*>)
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_BLOCKED_SID_ARG_COMMAND	String
VMT_MSGID_BLOCKED_SID_ARG_STREAM	Integer

VMT_MSGID_CCONTROL_REGISTER_FAIL
Model Registration Failed.
<*error*>
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_CCONTROL_REGISTER_FAIL_ARG_ERROR	String

VMT_MSGID_FILE_READ_ERROR
Incremental coverage file '<*filename*>' not found with incremental coverage enabled. Ignore if this is the first test in the testsuite
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_FILE_READ_ERROR_ARG_FILENAME	String

VMT_MSGID_REPLAY_LOGGING_ON
Model Change Dump turned on, mcd file is '<*filename*>', simulation performance will be degraded.
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_REPLAY_LOGGING_ON_ARG_FILENAME	String

VMT_MSGID_RVM_CH_TERMINATED
RVM channel input terminated due to <*cause*>
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_RVM_CH_TERMINATED_ARG_CAUSE	String

VMT_MSGID_XACT_RANDOM_PLAYBACK_ON
This test is running with random playback input file (<*filename*>)
Msg Type: VMT_MSG_WARNING

Fields:	**Field Type:**
VMT_MSGID_XACT_RANDOM_PLAYBACK_ON_ARG_FILENAME	String

A
Reporting Problems

If you think a VMT model is not working properly, you will need to contact a Synopsys Support Center. Before you contact technical support, you need to create an "MCD" (Model Change Dump) file for the model. The MCD file captures all of a model's activity during simulation (that is, stimulus and response) in ASCII text format. Transmitting a MCD file to technical support will help ensure accurate diagnosis of the problem.

Creating MCD Files

In order to create an MCD file for a model instance, create a file called vmt_mcd.cfg in the directory where you run the simulation. All VMT models look for this file and, if it exists, read its contents to determine which instance to dump. You can select a model instance in any of the following ways:

1. **Create an empty vmt_mcd.cfg file.** If an empty vmt_mcd.cfg file exists, all instances in the simulation will create MCD files. Each instance will dump debug activity into a file called *instance_name***.mcd**.
2. **Specify the instance or instances** for which you want to create MCD files in the vmt_mcd.cfg file.

   Example:

   ```
   top.u1
   top.u2.u1
   ```

   This causes instance **top.u1** and **top.u2.u1** to create MCD files.

Note

How to identify VMT model instances is explained in the following section.

Identifying an Instance

Each model instance has a unique name, this instance name is different depending on if the model is being driven from a HDL testbench or from a OpenVera testbench.

HDL Testbench Users

In this case the instance name is the full HDL path to the instance.

Verilog Example

```
top.u1
```

Here the name of the Verilog module is "top" and it has an instance of the VMT model called "u1."

VHDL Example

```
/top/u1
```

Again, the name of the VHDL entity is "top" and this it has an instance of the VMT model called "u1."

Note

The path separator for most VHDL simulator is "/", but the separator may be different for you simulator. Refer to your simulator documentation for the correct path separator.

OpenVera Testbench Users

In this case the instance name is the full HDL path to the instance plus the name passed to the constructor of the models instance.

Example

```
top.u1.u1
```

Here the name of the Verilog module is 'top' and this module has an instance of the VMT model and this instance is called "u1."

Also, when the instance was created, the string "u1" was passed to the new() constructor of the model.

Checking if MCD has been Enabled

When MCD has been turned on for a model instance, a message with Warning severity will be printed from the model instance. For example:

```
DesignWare Model Warning from test_top.u1.u1 at 0:
    Model Change Dump turned on, mcd file is 'test_top.u1.u1.mcd',
simulation performance will be degraded.
```

Impact of Turning MCD On

When MCD has been turned on for a model, the model prints debug information to a file. As a result, simulation performance is degraded.

B
Glossary

active command queue	Command queue from which a model is currently taking commands; *see also* command queue.
BFM	Bus-Functional Model – A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. *See also* Full-Functional Model.
big-endian	Data format in which most significant bit comes first; normal order of bytes in a word.
blocked command stream	A command stream that is blocked due to a blocking command issued to that stream The next command available to the model begins executing after the blocking command completes execution. *See also* command stream, blocking command, and non-blocking command.
blocking command	A command that prevents a testbench from advancing to next testbench statement until the command completes. Blocking commands typically return data to the testbench from the model.
command channel	Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.
command queue	First-in-first-out queue from which a model command execution engine retrieves commands; see also active command queue. VMT models support multiple command queues.
command stream	The communication channel between the testbench and the model.
configuration parameter	A setting that defines a characteristic or behavior of a model. Configuration parameters are used to define such things as bus width, operating modes, and error injection.

constraint	A limit applied to a constrained random test attribute, such as a weighting factor, address range, data range, or control range.
cycle command	A command that executes and causes HDL simulation time to advance.
design_dir	A directory structure containing all models required for a design, all required HDL libraries, design testbenches, and optionally, Synopsys developed testbench examples.
DesignWare Library	A collection of synthesizable IP and verification IP components that is authorized by a single DesignWare license. Products include SmartModels, VMT model suites, DesignWare Memory Models, Building Block IP, and the AMBA OCB synthesizable components.
endian	Ordering of bytes in a multi-byte word; see also little-endian and big-endian.
Full-Functional Model	A simulation model that describes the complete range of device behavior, including code execution. See also BFM.
HDL	Hardware Description Language – examples include Verilog and VHDL.
initiator	Any model that generates bus or protocol transactions.
instantiate	The act of placing a model into a design.
interface	Set of ports and parameters that defines a connection point to a component.
IP	Intellectual property – A term that encompasses simulation models and synthesizable blocks of HDL code.
little-endian	Data format in which the least-significant bit comes first.
model	A verification IP component.
monitor	A model capable of logging model and bus activity, and constrained random test transactions and responses.
non-blocking command	A command that immediately allows the testbench thread to advance to the next testbench statement.
non-queued command	A command that is not placed in the command queue for execution. Non-queued commands execute immediately.
payload	A file or random generator that can supply data to transactions to be sent from the initiator to a responder. Lack of payload causes a transaction generator to terminate.
protocol transaction	Model-specific data transfers, such as read, write, burst read, or burst write.

queued command	A command that is placed in the command queue for execution. Queued commands execute in the order they were placed in the queue.
responder	Any model that reads or writes data in response to commands from the initiator.
seed value	A value inserted into a random number generator to assure a unique starting value each time the generator is run.
transaction	A protocol transaction, transaction sequence, or transaction choice.
transaction sequence	A set of protocol transactions or included transaction sequences and/or transaction choices executed serially.
transaction choice	A set of protocol transactions or included transaction sequences and/or transaction choices. Each time the transaction choice is executed, one of the transaction choice branches is selected for execution.
VIP	Verification Intellectual Property – A generic term for a simulation model in any form.
wrap, wrapper	Code, usually VHDL or Verilog, that surrounds a design or model, allowing easier interfacing. Usually requires an extra, sometimes automated, step to create the wrapper.
zero-cycle command	A command that executes without HDL simulation time advancing.

Index

Q

R

S

T

V

W

Z