6 Messaging

The messaging feature gives a unified and standardized mechanism for writing and controlling debug printout messages to the screen and to files.

This chapter includes:

- "Introduction" on page 6-1
- "Basic Messaging" on page 6-5
- "The Message Action" on page 6-15
- "Message Loggers" on page 6-20
- "Configuring Loggers via Constraints" on page 6-23
- "The short_name_path() of a Unit" on page 6-25
- "Colors for Message Actions" on page 6-27
- "Nested Messages" on page 6-29
- "Messaging Command Interface" on page 6-31
- "Messaging Procedural Interface" on page 6-41
- "Recommended Methodology for the Message Action in eRM eVCs" on page 6-46

6.1 Introduction

The messaging feature is a centralized and flexible mechanism to print out useful text messages to the screen or to log files. It lets the code developer easily insert formatted and colored messages into the code. It provides the end user with powerful and flexible controls to selectively enable or disable groups of messages.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

The three most typical uses for messages are:

Summaries	Printing summary information at the beginning or end of significant chunks of activity
Tracing	Printing detailed trace messages during the simulation, upon interesting events
Debugging	Printing detailed debug messages during the run, to help the user or developer debug unexplained behaviors

One of the most important aspects of messages is that they give end users a standard and unified interface. For ease of use, those who receive ready-to-use ***e***VCs must be able to debug or trace activity inside the code. This is even more important when the user has multiple ***e***VCs and will need all printouts from all ***e***VCs to have similar formatting and a standard centralized mechanism to control and filter the messages. The messaging feature provides all of these capabilities.

Messages are different from plain **out()** and **outf()** printouts. They have a standard-format prefix that can be turned on or off by the user.

Messages are also different from **dut_error()** printouts. They do not signify failure, and they do not increment error or warning counters.

This section includes:

- "Messaging Requirements" on page 6-2
- "Messaging Solution" on page 6-3
- "About This Chapter" on page 6-4

6.1.1 Messaging Requirements

Uniformity

Many Specman users have invented their own company-wide or project-wide feature for writing message output. This is often a macro on top of the out() action.

However, a uniform way to write message output is needed so that output appears in a standard format (for example, starting with the current time). There must also be a standard way to configure message output at various verbosity levels.

The need for uniformity is especially great in the context of ***e***VCs, because an integrator using several ***e***VCs might want to make changes to all of the output (for example, disable all messages or enable all messages).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Simplicity and Flexibility of Writing Message Actions

Writing message actions should be fairly easy. Moreover, if some output must be created by a printing method, users should be able to call it (that is, it should not be just a glorified **out()** action).

Simplicity and Flexibility of Enabling and Disabling Message Actions

It should be easy to enable and disable specific message actions.

In addition to writing to the screen, users should also be able to divert the output of specific message actions to one or more files. This avoids the need to write a separate file logging facility.

User Control of Output Format

Users should be able to control at least the prefix of the output.

Minimal Performance Overhead When Off

The overhead of message actions should be very low when turned off.

Furthermore, there should be a way to compile a specified subset of the message actions so that they incur no overhead at all.

6.1.2 Messaging Solution

When executed, the message action creates a message and sends it to a message logger. Each message logger can be configured to filter messages in various ways, to format the enabled messages in various ways (adding the time, name of the unit, and so on), and to send them to various destinations (files/screen).

The message loggers are instantiated by the programmer in the unit hierarchy, and then configured by the end user via constraints (though they can also be configured via commands or procedurally).

Note There is a predefined message logger, **sys.logger**. Therefore, defining new loggers is not strictly necessary. We recommend doing so for finer control.

Users can use the interactive **show message** commands to see all loggers and message actions.

Users can use the interactive **set message** commands to change the configuration of loggers (for example, to request more or less output).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Examples

- Add some message actions, and put a logger in the unit:

```
unit my_dsp {

    foo() is {
        ...
        message(LOW, "Starting transmission");
          -- LOW verbosity means this is an important message that
          -- will be shown even when verbosity is set to 'LOW'
        ...
        message(MEDIUM, "Sending packet ", pkt);
          -- MEDIUM verbosity means this is a less important message
        ...
    };

    logger: message_logger is instance;
     -- Instantiate a message logger for this unit. When activated,
     -- it will handle all message actions executing in this unit
     -- or in any unit or struct under it.
};
```

- Configure the logger via constraints:

```
extend sys {
    dsp: my_dsp is instance;
        keep dsp.logger.tags == {NORMAL}
        keep dsp.logger.verbosity == LOW;
          -- This logger will only look at important messages
        keep dsp.logger.to_file == "dsp_results.elog";
        -- Send it also to a file (it goes to the screen by default)
};
```

- Optionally use commands:

  As configured above, sys.dsp.logger will send only important (LOW verbosity) messages to file and screen. If, while debugging, you find that you also want less important messages, you can issue the following command from the Specman prompt:

```
set message -logger=sys.dsp.logger -verbosity=HIGH
```

6.1.3 About This Chapter

The remainder of this chapter consists of two main parts:

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- "Basic Messaging" on page 6-5: This is a QuickStart that might satisfy all of your needs. Even if you plan to use some of the more advanced aspects of the messaging feature, we still recommend reading this section before going on to the rest of the chapter.
- Everything after "Basic Messaging": The rest of the chapter provides detailed explanation of the various aspects of the messaging feature. Even if the "Basic Messaging" section seems to satisfy all of your needs, you might want to glance at "Summary of Messaging Commands" on page 6-31 to see all available commands (everything you can do to message loggers). We also encourage everyone to read "Recommended Methodology for the Message Action in eRM eVCs" on page 6-46.

6.2 Basic Messaging

This section is intended as a QuickStart of the message feature. Some readers might find that this section answers all of their questions. Other readers might require additional information. Either way, we recommend reading this section before reading the rest of this chapter.

This section includes:

- "Basic Message Action" on page 6-5
- "Basic Message Loggers" on page 6-7
- "Basic Messaging Configuration" on page 6-12

6.2.1 Basic Message Action

This section provides basic information on the message action. After reading this section, see "The Message Action" on page 6-15 if you need more details.

This section includes:

- "Example Message Action" on page 6-5
- "Format of Message Output" on page 6-6
- "Message Syntax" on page 6-6

6.2.1.1 Example Message Action

Following is an example of a message action:

```
monitor_bus() @clock is {
    wait @reset_ended;
    message(LOW, "Bus ", bus_num, " is done with reset");
        -- This message will be shown even when the verbosity is set to LOW
    while TRUE {
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
            wait @grant;
            extract_next_burst();
            message(MEDIUM, "Bus ", bus_num, " granted");
            -- Verbosity level is MEDIUM
            -- String to print is "Bus ", bus_num, " granted"
        };
    };
```

6.2.1.2 Format of Message Output

The message output can be in three formats: **short**, **long**, or **none**. The default format is **short**.

You can change the format via constraints or commands.

Examples

- **short**

  [12030] AHB_0 M1: Bus 3 is done with reset

  Notes

  - The color of the short name (“AHB_0” in the example above) can be customized per ***e***VC.
  - The color of the time (“[12030]” in the example above) alternates between BLACK and GRAY as the time changes.

- **long**

  [12030] AHB_0 M3 (HIGH) at line 32 in @ex_bus_watch in ex_bus_unit-@4:
  Bus 3 is done with reset

- **none**

  Bus 3 is done with reset

6.2.1.3 Message Syntax

Syntax

message([*tag*], *verbosity*, *exp, …***)** [*action-block*]

messagef([*tag*], *verbosity*, *format*, *exp, …***)** [*action-block*]

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Parameters

tag	Used to direct the output (for example, to a file). Default is NORMAL
verbosity	Suggested use: NONE: Messages that cannot be disabled LOW: Once per run (reset messages, and so on) MEDIUM: Once per transaction HIGH: More detailed FULL: Even more detailed

action-block	Can include output-producing actions or method calls:
	`message(HIGH, "Master ", me, " received packet:"){`
	`    print the_packet;`
	`};`

6.2.2 Basic Message Loggers

This section provides basic information on message loggers. After reading this section, see "Message Loggers" on page 6-20 for more information.

This section includes:

- "message_logger" on page 6-7
- "sys.logger" on page 6-8
- "Message Tags and Message Destinations" on page 6-8
- "How Loggers Relate to Message Actions" on page 6-9
- "Multiple eVCs with Loggers" on page 6-10
- "Configuring Loggers" on page 6-11

6.2.2.1 message_logger

message_logger is a unit whose job is to manage the output from message actions by:

- Filtering it (for example, ignoring actions above verbosity LOW)
- Formatting it
- Sending the result to one or more destinations (screen, files)

By default, **sys** contains one instance of a logger, namely **sys.logger**. ***e***VC developers can add logger instances to other units within their ***e***VC. Some examples from our golden ***e***VCs are:

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

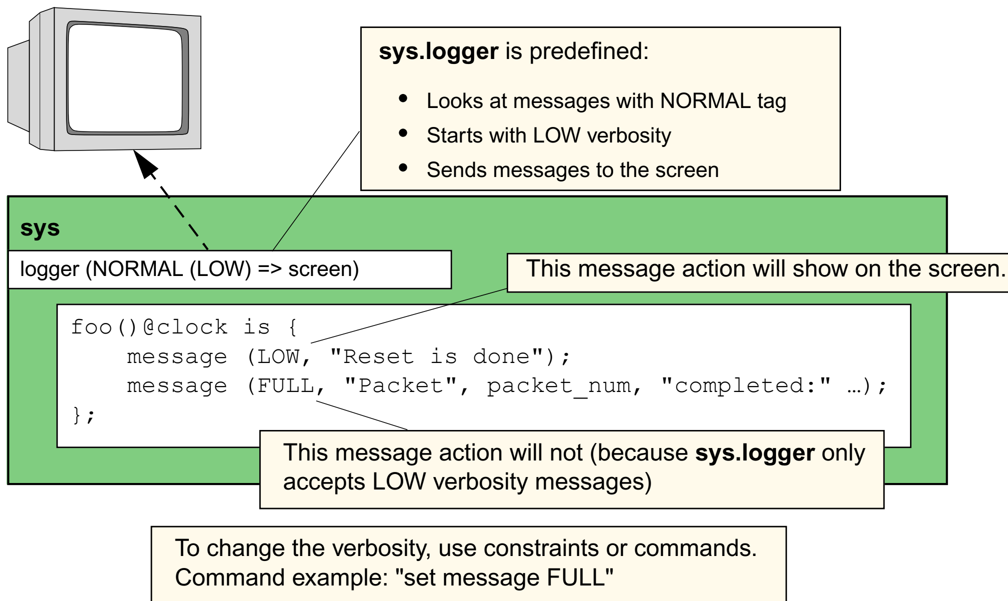
- vr_xbus_env_u.logger, which goes to the screen
- vr_xbus_env_u.file_logger, which sends to a file the output from very specific message actions (marked with a special tag)

6.2.2.2 sys.logger

sys.logger is predefined. It starts with LOW verbosity, NORMAL tag, and sending messages to the screen. You can change these settings using constraints or commands.

For example, in Figure 6-1, the second message action has a verbosity of FULL, but the logger is set to verbosity LOW. As a result, only the first message action is printed to screen.

Figure 6-1 sys.logger



To enable that second message as well, you must set the verbosity of the logger to FULL via the following command:

```
set message FULL
```

6.2.2.3 Message Tags and Message Destinations

The ***e***VC developer must decide what tags to use in the ***e***VC and how to use them (that is, which tags to send to screen, which tags to send to file(s), and so on).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

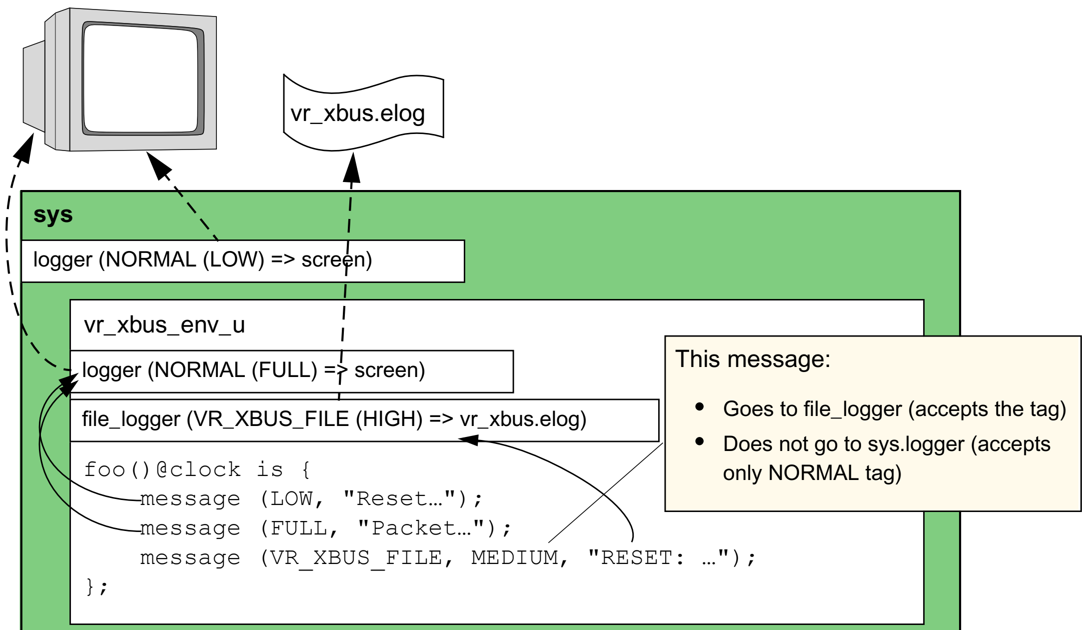
The default tag for a message action (if none was specified) is NORMAL.

By default, **sys.logger** recognizes messages with NORMAL tag. The default for all other loggers is to ignore all tags. They have an empty tag list "{}". Therefore, to activate a logger other than **sys.logger**, you must set its tag list using commands or constraints.

The default destination for all loggers is to send messages to the screen only.

6.2.2.4 How Loggers Relate to Message Actions

Figure 6-2 *e*VC with Screen and File Loggers



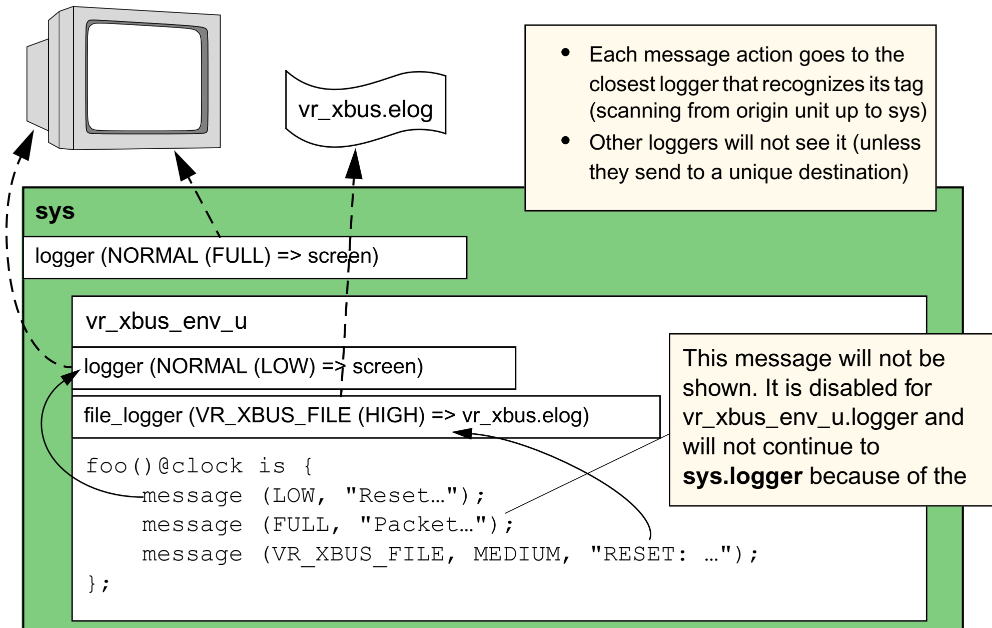
The logger-message logic is as follows:

1. Each logger has a list of destinations, a list of tags it recognizes, and a list of enabled messages.
2. Messages are handled by each logger on the way to **sys** that recognizes the message tag and has a destination that closer loggers did not have.
3. Messages can be sent to multiple destinations but they only have one chance to go to any particular destination.
   - If a message is enabled in a handling logger, it is formatted and sent to all destinations of the logger (except any destinations of a closer handling logger).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- If a message is not enabled in a handling logger, it is not sent to any of the logger's destinations (and it will never be sent to any of the logger's destinations). (See Figure 6-3.)

Figure 6-3 Subunit Logger Blocking a Message



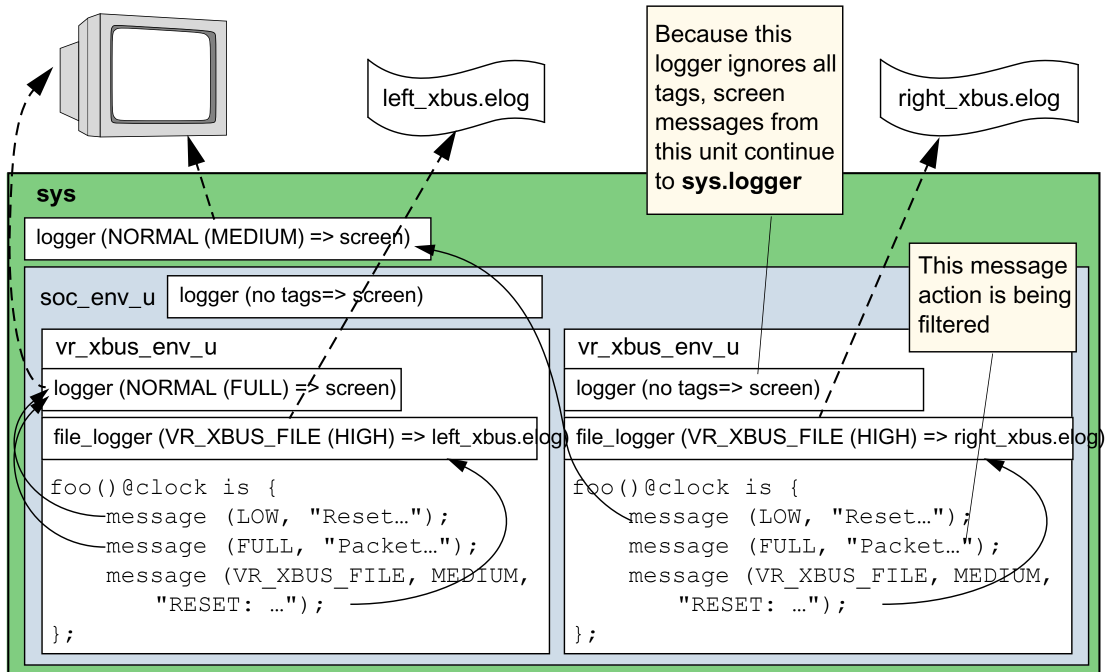
6.2.2.5 Multiple *e*VCs with Loggers

You can have multiple ***e***VCs with loggers. Each ***e***VC should have a screen logger and zero or more file loggers.

Figure 6-4 below shows an example of multiple ***e***VCs with loggers. In the figure, the verbosity level of each logger is shown in parentheses. The vr_xbus instance on the left is set to FULL verbosity. The vr_xbus instance on the right is set to MEDIUM verbosity. (The logger for that instance is set to disabled (transparent); but because **sys.logger** is set to MEDIUM verbosity, you still get messages from that instance up to MEDIUM verbosity.) In the example, the file loggers of the two vr_xbus instances are directed to two different files (left_xbus.elog and right_xbus.elog).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Figure 6-4 Multiple *e*VCs with Loggers



6.2.2.6 Configuring Loggers

The default configuration for a message logger is:

- Verbosity NONE (that is, ignoring all input). The one exception is **sys.logger**, which defaults to verbosity LOW.
- Writing to the screen
- Not writing to any file
- Empty tag list {}

These parameters (and many others) can be changed using:

- Constraints (most commonly)
- Commands (at any time during the run)
- An API (which parallels the commands)
- The *e*RM Utility (see "Controlling Messaging" on page 8-16)

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.2.3 Basic Messaging Configuration

This section provides basic information on controlling messaging. However, the messaging feature offers much more power and complexity than what is presented here. After reading this section, if you require more details, see:

- "Messaging Command Interface" on page 6-31
- "Messaging Procedural Interface" on page 6-41
- "Configuring Loggers via Constraints" on page 6-23
- "The short_name_path() of a Unit" on page 6-25
- "Colors for Message Actions" on page 6-27
- "Recommended Methodology for the Message Action in eRM eVCs" on page 6-46

Note Even if this basic section seems to satisfy all of your needs, you might want to glance at "Summary of Messaging Commands" on page 6-31 to see all available commands (everything you can do with message loggers). We also encourage everyone to read "Recommended Methodology for the Message Action in eRM eVCs" on page 6-46.

This section includes:

- "How eVC Writers Configure Messaging" on page 6-12
- "How Integrators Configure Messaging" on page 6-14
- "How Users Configure Messaging" on page 6-14

6.2.3.1 How *e*VC Writers Configure Messaging

eVC writers can configure messaging by adding screen and file loggers per ***e***VC and possibly per agent.

Examples

- Add a screen logger.

```
extend vr_xbus_env_u {
    // Use message_logger defaults: verbosity NONE, to_screen,
    // Empty tag list {}
    logger: message_logger is instance;
};
```

  Note By default, each logger uses verbosity NONE, and its tag list is empty. This passes the control to **sys.logger** until the logger is explicitly enabled.

- Add a file logger:

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
extend vr_xbus_env_u {
    file_logger: message_logger is instance;
        keep soft file_logger.verbosity == LOW;
          // Have all actions from all instances of eVC
          // go to vr_xbus.elog
        keep soft file_logger.to_file == "vr_xbus.elog";
        keep soft file_logger.to_screen == FALSE;
        keep soft file_logger.tags == {VR_XBUS_FILE};
          -- Some writers might want to use NORMAL tag
          -- (same as screen)
};
```

Specifying the Short Name and Short Name Color in Message Output

eVC writers can specify the short name and short name color in message output per ***e***VC. For example, they might want the short name output to appear as follows:

[12030] AHB_0 M1: Bus 3 is done with reset

Note The “AHB_0” in the above example is printed to screen as green.

The following code might be used to make this happen:

```
extend vr_xbus_env_u {
    short_name(): string is {
        return append(name);
            -- This assumes there is an instance-id field called
            -- "name", containing, for example, XBUS_0, etc.
            -- The same must also be done for agents (if you want
            -- them to have a short name).
    };
    short_name_style(): vt_style is {
        return GREEN;
            -- This will paint XBUS_0 in GREEN. (It will also be
            -- shown with the eVC's banner).
            -- We recommend leaving agent names BLACK.
    };
};
```

Note Color is removed from text before sending it to files.

Tips

- ***e***VC writers might want more than one file logger. For example, they might want a data_file_logger and a packet_file_logger, each showing information at different abstraction levels. To do this, use separate tags for each file logger.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- ***e***VC writers might want to have some hidden message actions that are enabled only to debug some algorithm within the ***e***VC. To do this, use special tags (for example, VR_XBUS_CRC).
- The ***e***VC documentation should contain the following tables:
  - Loggers (with associated tags)
  - Message action categories (what is shown in verbosities LOW, MEDIUM, HIGH, and FULL) along with a list of special tags, if any

6.2.3.2 How Integrators Configure Messaging

Integrators are those who build a VE (verification environment) for test writers out of multiple ***e***VCs and other components.

Integrators might configure messaging as follows:

- Split file output per ***e***VC instance (if wanted). For example:

```
extend vr_xbus_env_u {
    keep soft file_logger.to_file == append(name,".elog");
};
```

- Change the color of an ***e***VC if not unique in the env.
- Change the verbosity of ***e***VC instances as needed.

6.2.3.3 How Users Configure Messaging

Users can configure messaging for many purposes. Some typical configurations are as follows:

- To get more information on a specific instance. For example:

```
extend XBUS_1 vr_xbus_env_u {
    keep soft logger.verbosity == FULL;
    keep soft logger.tags == {NORMAL};
};
```

- To get more information during the run. This is done via commands. For example:

```
// Get more messages from all units
set message HIGH

// Get more messages from only one logger
set message -logger=sys.left_xbus.logger -verbosity=FULL

// Get all HIGH messages from a particular module
set message -add -verbosity=HIGH @foo
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
// Add all messages with specific text
set message -add "...Arbitration ..."

// Disable all messages to sys.logger
set message NONE

// Make a logger ignore the NORMAL tag
set message -logger=sys.left_xbus.logger -ignore_tags
```

Note You can only issue the above commands after initial generation, because the only logger that exists prior to generation is **sys.logger**. For more information, see "When to Issue Message Commands" on page 6-52.

- To make some messages stand out. For example, you could color all reset messages purple as follows:

```
set message -style=PURPLE "...Reset..."
```

- To change output format. For example, you can set the output format to **long** as follows:

```
set message -format=long
```

- To show the current configuration.

```
show message                   // Shows summary of all loggers
show message -l=all -full      // Shows full details per logger
show message -actions          // Shows all message actions in files
```

6.3 The Message Action

This section provides a detailed explanation of the message action.

This section includes:

- "message Syntax" on page 6-15
- "message Semantics" on page 6-16
- "Examples of the Message Action" on page 6-16
- "Output Appearance" on page 6-17
- "Recommended Verbosity Usage" on page 6-18
- "message_tag" on page 6-19

6.3.1 message Syntax

message([*tag*], *verbosity*, *exp, …***)** [*action-block*]

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

messagef([*tag*], *verbosity*, *format_exp*, *exp, …***)** [*action-block*]

6.3.2 message Semantics

When a **message()** or **messagef()** action is executed, the following happens:

- If there are no handling loggers for the action, then the action is skipped. (For details on how Specman computes the list of handling loggers for a message action, see "How Loggers Handle Messages" on page 6-21.)
- If there are handling loggers for the action, the action creates a message (consisting of a list of string plus related information) and sends it to all of the handling loggers. Those loggers then format the message and send it to the screen or to files.
- For **message()**, the first string in the message is created by appending all of the expressions like **out()** does. For **messagef()**, the first string is created using the *format-exp*, as in **outf()**.

  messagef() does not automatically add a \n (carriage return) between the message and the optional *action-block*. Therefore, if there is an *action-block* and a \n is required before it, end the *format-exp* with a \n.

  If the whole **messagef()** output (including the optional *action-block*) does not end with a \n, then an extra \n is automatically added.

  messagef() also allows appending of the *action-block* output to the **messagef()** header output. For example:

  ```
  messagef(HIGH, "And the winner is: ") {
      if winner != NULL then {
          out(winner.name);
      } else {
          out("Nobody");
      };
  };
  ```

- If an *action-block* exists, it gets executed. It will probably contain further output-producing actions, calls to reporting methods, and so on. The output of all of those is added, as a list of string, to the message.
- Message code should not modify the flow of the simulation in any way. Therefore, time-consuming operations in message headers or action blocks are strictly disallowed.

6.3.3 Examples of the Message Action

```
message(LOW, "Bus ", bus_num, " is done with reset");
    -- Output this message at verbosity LOW.
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
messagef(MEDIUM, "Packet number %d has arrived\n", packet_num);
    -- Output this message using a format string, at verbosity MEDIUM.

message(HIGH, "Master ", me, " has received ", the_packet) {
    print the_packet;
};
    -- Output this message and print the packet, at verbosity HIGH.

message(VR_XBUS_FILE, MEDIUM, "Packet ", num, " sent: ", data);
    -- Output this message at verbosity MEDIUM.
    -- Use VR_XBUS_FILE as the message-tag.
```

6.3.4 Output Appearance

There are three predefined formats for writing messages. These formats are controlled via the **set message -format** command.

Note Users who are not satisfied with the way the three predefined formats look have full programmatic control over formatting (by extending the **format_message()** method of the message logger).

6.3.4.1 Short Format

The **short** format is the default. It looks as follows:

[*time*] *short-name-path*: *message*

Example

```
[12030] AHB_0 M1: Bus 3 is done with reset
```

Typically, *short-name-path* is predefined. For information on setting *short-name-path*, see “The short_name_path() of a Unit” on page 6-25.

Note If *short-name-path* has not been set, then the default output appears as *path-@instance_number.*

6.3.4.2 Long Format

The long format looks as follows:

[*time*] *short-name-path* (*verbosity*) *source* in *struct-instance*:
message

Note The *source* and the *struct-instance* will both be blue hyperlinks in Specview.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Example

```
[12300] AHB_0 M1 (HIGH) at line 12 in @vr_ahb_send in vr_ahb_bfm-@77:
    Bus 3 is done with reset
```

6.3.4.3 No Format (none)

The **none** format does not add anything to the message specified in the message action.

Example

```
Bus 3 is done with reset
```

6.3.5 Recommended Verbosity Usage

The *verbosity* parameter can be set to NONE, LOW, MEDIUM, HIGH, or FULL.

Lower *verbosity* implies a more important message.

The **set message** action refers to verbosity. For example, "**set message -verbosity**=MEDIUM" enables all message actions whose verbosity is MEDIUM or lower.

Table 6-1 below shows the recommended usage of verbosity. Keep in mind that each level can assume that all lower levels are also printing (and hence there is no need to repeat them).

Table 6-1 Verbosity Levels

Level	Recommended Use	Examples
NONE	Critical messages that users will always want to see. (This level cannot be disabled.)	"WARNING: Running in reduced mode"
LOW	Messages that happen once per run or once per reset.	"Master M3 was instantiated" "Device D6 got out of reset"
MEDIUM	Short messages that happen once per data item or sequence.	"Packet-@36 was sent to port 7" "A write request to pci bus 2 with address=0xf2223, data=0x48883"

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Table 6-1 Verbosity Levels (continued)

Level	Recommended Use	Examples
HIGH	More detailed per-data-item information, including: • Printing the actual value of the packet • Printing subtransaction details	"Full details for packet-@36: len=5 kind=small ..."
FULL	Anything else, including message prints in specific methods (just to follow the algorithm of that method).	

6.3.6 message_tag

Both **message()** and **messagef()** have an optional first parameter of type message_tag.

The type message_tag is initially defined as follows:

```
type message_tag: [NORMAL];
```

It can be extended by the user. For example:

```
extend message_tag: [VR_XBUS_PACKET];
```

If you do not specify a tag, (that is, if the first parameter of **message()** is a legal value for verbosity), then the value NORMAL is prepended. Hence:

```
message(MEDIUM, "Packet done: ", packet);
```

is the same as:

```
message(NORMAL, MEDIUM, "Packet done: ", packet);
```

Example

Following is an example of specifying a message tag in the message action:

```
message(VR_XBUS_PACKET, MEDIUM, "Packet ", num, " sent: ", data);
    -- This message action only goes to loggers that look for this
    -- particular tag "VR_XBUS_PACKET".
```

Message tags are used for associating specific message actions with a message logger (see "Message Loggers" on page 6-20).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

We expect that most message actions will not specify a tag at all. (In other words, they will use the default NORMAL tag.) Other tags will be used only in special cases such as:

- Message actions that are specifically targeted for going to a file
- Temporary message actions to be enabled only in a very specific debugging mode (for example, to see how some algorithm is working)

6.4 Message Loggers

A message logger is a predefined unit, whose job is to manage the output from message actions (filtering it, formatting it, and sending it to one or more destinations).

Message loggers are defined programmatically and are attached as fields to various units in the unit hierarchy. For example:

```
extend vr_xbus_env_u {
    logger: message_logger is instance;
};
```

For each message logger you can specify the following things (via methods, commands, or constraints):

- Which subset of the message actions the logger will look at.

  By default, each logger uses verbosity NONE, and its tag list is empty. This passes the control to **sys.logger** until the logger is explicitly enabled.

- Which subset of the unit instances the logger will look at.

  By default, this is the tree starting at the unit the logger is attached to. For example, each vr_xbus_env_u.logger looks at the unit subtree under the corresponding **vr_xbus_env_u**.

- Which destinations (files or screen) to send output to.
- What format to use.

Typically, loggers are generated at elaboration time before the simulation run begins. You can create a logger instance at run time, but it must be created as a field and not as a variable.

By default (before any file is loaded) there is just one message logger, called **sys.logger**. However, we recommend that each ***e***VC (and, optionally, each agent within the ***e***VC) define one message logger for writing information to the screen and zero or more message loggers for writing information to files.

Note We recommend not referencing or accessing HDL signals and external ports from message loggers.

This section includes:

- “How Loggers Handle Messages” on page 6-21

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- "Configuring Message Loggers" on page 6-22

6.4.1 How Loggers Handle Messages

Once configured, loggers process message actions in the following manner.

1. Whenever a message action is executed, Specman gathers the list of handling loggers. To compute that list, Specman:

   a. Determines the origin unit where the message action is executed, via **get_unit()**.

      Note **get_unit()** is a method of any_struct. **message()** and **messagef()** can be used anywhere in your code.

   b. Gathers all loggers that are looking at that unit.

      These are normally all loggers instantiated in units between the origin unit and **sys** that recognize the tag of the message except those loggers explicitly programmed to ignore the origin unit (for example, by using the **set message -units** command).

   c. Sorts the list of handling loggers on the basis of closest to the message action first and selects the closest one for each destination.

      For example, if **get_unit()** returned:

      ```
      sys.soc.xbus_1.agents[3].BFM
      ```

      then the list might be:

      ```
      sys.soc.xbus_1.agents[3].logger
      sys.soc.xbus_1.logger
      sys.soc.xbus_1.file_logger
      sys.logger
      ```

   d. Removes from the list of handling loggers all loggers that ignore the message tag.

2. For each handling logger, Specman:

   a. Calls **logger.accept_message()**.

      If that method (which is extensible by the user) returns FALSE, then the message is ignored and will not be processed by this logger any more.

   b. Calls **logger.format_message()** to take the raw message and create a final list of string from it.

   c. Sends the created list of string to each of the destinations (file or screen) associated with the logger.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Example

Suppose only the NORMAL tag exists, and we have:

- A file logger and a screen logger in the XBus env
- A screen logger in the XBus agent

We can set verbosity and destination as follows:

```
set message -logger=sys.xbus_1.logger HIGH
set message -logger=sys.xbus_1.file_logger HIGH
set message -logger=sys.xbus_1.file_logger -file=xbus_1.elog
```

And then:

```
set message -logger=sys.xbus_1.agent_5.logger LOW
-- Assume this logger is configured to send to screen (the default)
```

If you have a HIGH verbosity message coming from agent_5, then:

- sys.xbus_1.agent_5.logger handles this message for the screen. However, as that logger has only LOW verbosity messages enabled, the message is not sent to screen.
- sys.xbus_1.file_logger handles this message for the xbus_1.elog file. As that logger has HIGH messages enabled, this message is sent to the file.

6.4.2 Configuring Message Loggers

There are three ways (covered in detail in the next three sections) to configure loggers. You can use:

- Commands
- Methods of the logger
- Constraints

Note Do not try to configure message loggers from the Specman Elite Data Browser.

Commands and methods have almost equivalent power. (In fact, the commands are directly implemented via the methods.) Configuring via constraints is less flexible, but it should be flexible enough for most users. If more flexibility is required, you can use the procedural interface to fine-tune the configuration after generation.

Constraints are used during pre-run generation to set the fields of the logger. Then, during **post_generate()**, the logger fields are used to configure the logger. For more information, see "Configuring Loggers via Constraints" on page 6-23.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.5 Configuring Loggers via Constraints

Loggers should always be generated, usually directly under units.

At post-generation, the logger becomes attached to the unit (specifically, the unit computed by **logger.get_unit()**).

In addition, each logger has several constrainable fields (see "Constrainable Fields and Their Default Values" on page 6-23).

The **post_generate()** method of a logger uses the generated values of the fields to configure the logger via the procedural interface described in "Messaging Procedural Interface" on page 6-41.

This section includes:

- "Constrainable Fields and Their Default Values" on page 6-23
- "Using the Constrainable Fields to Configure Loggers" on page 6-24
- "Constraining Verbosity" on page 6-25

6.5.1 Constrainable Fields and Their Default Values

The various constrainable fields are used in configuring the message logger during **post_generate()** of the logger. The following code shows the constrainable fields and their default values.

```
extend message_logger {

    // The message tags for selecting the actions for this logger
    tags: list of message_tag;
        keep soft tags == {};

    // The verbosity for selecting the actions for the logger
    verbosity: message_verbosity;
        keep soft verbosity == NONE;

    // The modules wildcard for selecting the actions for the logger
    modules: string;
        keep soft modules == "*";

    // The pattern to match against the string in the message action
    string_pattern: string;
        keep soft string_pattern == "...";

    // File name the logger should write to (or none if "")
    // Default extension for the file name is ".elog".
    to_file: string;
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
            keep soft to_file == "";

        // True if we want the message_logger to write to screen
        to_screen: bool;
            keep soft to_screen == TRUE;
    };
```

Note You can associate only one file with a logger via constraints. Using the procedural/command interface, you can specify as many files as you like.

6.5.2 Using the Constrainable Fields to Configure Loggers

The various constrainable fields are used in configuring the message logger during **post_generate()** of the logger. The following code shows how message loggers are configured.

```
    extend message_logger {

        // On post_generate(), configure the message logger according to fields
        post_generate() is also {
            configure_according_to_fields();
        };

        -- Configure this message logger according to the generatable fields
        configure_according_to_fields() is {

            set_base_unit(get_unit());

            -- Set filtering
            set_actions(verbosity,tags,modules,string_pattern,replace);

            -- Set destinations
            if to_file != "" then {
                set_file(to_file, on);
            };

            if to_screen {
                set_screen(on);
            };

            -- Set misc options
            set_format(format);
            set_flush_frequency(flush_frequency);
        };
    };
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.5.3 Constraining Verbosity

You can use constraints in a way similar to commands with an interpretation of **-replace**.

By default, all loggers have all tags ignored and a verbosity of NONE:

```
extend message_logger {
    keep soft tags == {};
    keep soft verbosity == NONE;
};
```

The one exception is **sys.logger**, which is defined as:

```
extend sys {
    keep soft logger.verbosity == LOW;
    keep soft logger.tags == {NORMAL};
};
```

Tags are empty by default, because by default we want all messages to be controlled by **sys.logger**.

This scheme is the same as issuing at the start of the test:

```
set message LOW
```

If you want one specific logger to be HIGH (and not ignored), use:

```
extend sys {
    keep soft xbus_1.logger.verbosity == HIGH;
    keep soft xbus_1.logger.tags == {MY_TAG};
};
```

If you constrain verbosity to any value above NONE and yet leave the tag list empty, Specman automatically constrains the tag list to {NORMAL}.

6.6 The short_name_path() of a Unit

short_name_path() is a method of **any_unit** that returns a shorthand notation for a unit. You must never modify the **short_name_path()** directly. Instead, you should extend the **short_name()** method, which **short_name_path()** calls.

This section includes:

- “Motivation” on page 6-26
- “How short_name_path() Works” on page 6-26
- “How short_name_path() Is Computed” on page 6-27

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.6.1 Motivation

The **short_name_path()** feature provides units with a short name-path, to be used, for example, in the short format of messages.

For example, the **short** format for message output is:

[*time*] *short-name-path*: *message*

Example:

```
[123000] AHB_0 MASTER_1: Received first part of transaction-@33
```

6.6.2 Syntax

short_name_path(): *string*

In the example below, “AHB_0 Master_1” is the result of the message action calling **short_name_path()**.

```
[123000] AHB_0 MASTER_1: Received first part of transaction-@33
```

6.6.3 How short_name_path() Works

short_name() is defined in any_unit as:

```
short_name(): string is empty;
```

In other words, by default **short_name()** returns an empty string.

You can change this method to return the desired short-name for the unit. For example:

```
extend vr_ahb_env {
    evc_name: vr_ahb_name;              // e.g. AHB_0 etc.

    short_name(): string is only {
        return append(evc_name);
    };
};

extend vr_usb_agent {
    index: int;         // The index in some big list

    short_name(): string is only {
        return dec("USB agent[", index, "]");
    };
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.6.4 How short_name_path() Is Computed

The method **short_name_path()**:

1. Collects all of the non-empty **short_name()** strings along the path from **sys** to the unit.
2. Appends all the strings with a blank separator.
3. If the result is not empty, returns it. Otherwise, returns something like "vr_xbus_env_u-@55".

Notes

- If the ***e***VC and the agent have **short_name()** defined as non-empty, but the BFM unit within the agent does not define **short_name()**, then the **short_name_path()** for the agent and the BFM will be the same.

  This is acceptable. You do not have to invent short names for all of the leaves of the unit tree. It is okay for messages coming from the agent and the BFM to all start with something like "(20000) AHB_0 M1:". In that case, presumably it either does not matter where exactly the message is coming from, or it will become clear by the text of the message anyway.

- Suppose you have an SoC that contains AHB_0, AHB_1, and AHB_2. If you then construct a new DUT that contains two instances of that SoC, then it is advisable to define a non-empty **short_name()** for the whole SoC to maintain uniqueness of short-name paths.

  In this case, the message will start with something like
  "[123000] NORTH_RTR AHB_0 MASTER_1: …".

- Beyond pre-generation, **short_name_path()** caches the result for performance.

6.7 Colors for Message Actions

This section includes:

- "Basic Color Handling in Specman" on page 6-27
- "any_unit.short_name_style()" on page 6-28
- "Coloring Time" on page 6-28

6.7.1 Basic Color Handling in Specman

In Specman, we now have the official method for coloring Specview text:

vt.text_style(color: *vt_style*, **text**: *string***)**: *string*

For example, if you have Specman, you could get some red text by writing:

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
out("I see red: ", vt.text_style(RED, "Some red text"));
```

You can use **vt.text_style()** to colorize any part of your message. For example:

```
message(LOW, vt.text_style(ORANGE, "Reset"), " is done");
```

Colors are all uppercase (for example, BLACK).

To see the existing styles (colors), type:

```
vt.show_styles()
```

You can use any style you want from the list. We recommend avoiding BLUE (used for hyperlinks) and both RED and DARK_ORANGE (used for errors).

Note These colors apply only to text going to screen. Text going to file is scrubbed of color information.

6.7.2 any_unit.short_name_style()

In addition to **short_name()**, each unit has a method called **short_name_style()**.

short_name_style() is defined as follows:

short_name_style(): *vt_style*

The default is BLACK. If you change *vt_style* to another color, then the message will contain the short name in that color.

For example, if you write:

```
extend MASTER vr_ahb_agent {
    short_name_style(): vt_style is only {return GREEN};
};
```

then the word “MASTER_1” in the message output below will also appear green (along with the short-name path):

```
[123000] AHB_0 MASTER_1: Received first part of transaction-@33
```

Note The same style will also appear in the ***e***VC banner that gets written after initial generation.

6.7.3 Coloring Time

The color of the *time* portion of the message alternates between GRAY and BLACK as the time changes.

This is done automatically by the logger.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.8 Nested Messages

You can designate a message action as responsible for a series of nested message actions. The initial message action is designated as a **leader** action. The designation can be made by either a command or a method API.

See Also

- "Nested Message Semantics" on page 6-29
- **set message -leader [filter]** on page 6-39

6.8.1 Nested Message Semantics

Once a message action is designated as a leader action, all of its nested message actions are handled by the leader's loggers (and only by them). If more than one message is a potential leader, only the first potential leader in the calling sequence becomes the leader.

After the leader is processed at runtime, the coupling between the calling sequence's loggers and their destinations is final. It applies to all messages of the calling sequence.

All nested messages are formatted when the leader action is ended and sent to the leader message loggers.

Example 1 Print Order

Assume that you have the following code:

```
message(NONE,"I'm A0") {
    out("A1");
    message(NONE, "I'm B0") {
        out("B1");
        message(NONE, "I'm C0") {
            out("C1");
        };
        out("B2");
    };
    out("A2");
};
```

If there is no leader message, then the output is printed in the order that message blocks finish:

```
I'm C0
C1
I'm B0
B1
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
B2
I'm A0
A1
A2
```

With “I’m AO” as the leader message, the order would be:

```
I'm A0
A1
I'm B0
B1
I'm C0
C1
B2
A2
```

Example 2 Logger Handling

Assume that you have the following code:

```
extend sys {
    u1: U1 is instance;
    u2: U2 is instance;

    run() is also {
        u2.foo();
    };
};

unit U1 {
    l1: message_logger is instance;
    keep l1.verbosity == FULL;
    a: A;
};

unit U2 {
    l2: message_logger is instance;
    keep l2.verbosity == FULL;

    foo() is {
        message(NONE, "foo: I'm U2") {
            out("foo: before");
            sys.u1.a.goo();
            out("foo: after");
        };
    };
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
struct A {
    goo() is {
        message(NONE, "goo: I'm message from A");
    };
};
```

If there is no leader message, then the output would be:

```
goo: I'm message from A    -- handled by sys.u1.l1
foo: I'm U2                -- handled by sys.u2.l2
foo: before                -- handled by sys.u2.l2
foo: after                 -- handled by sys.u2.l2
```

With "foo: I'm U2" as the leader message, the order would be:

```
foo: I'm U2                -- handled by sys.u2.l2
foo: before                -- handled by sys.u2.l2
goo: I'm message from A    -- handled by sys.u2.l2
foo: after                 -- handled by sys.u2.l2
```

6.9 Messaging Command Interface

In this section, the command interface receives the most extensive description. The procedural and constraint interfaces (see the following two sections) are explained by analogy to the commands.

This section first gives a brief look at all of the messaging commands and then goes into detail on each of them. This section includes:

- "Summary of Messaging Commands" on page 6-31
- "Detailed Explanation of Commands" on page 6-33

6.9.1 Summary of Messaging Commands

This section contains a summary of all messaging commands. It includes:

- "set message Summary" on page 6-31
- "show message Summary" on page 6-33

6.9.1.1 set message Summary

All of the following commands refer to **sys.logger** unless you specify the **-logger**=*exp* option. You can replace **-logger** with **-l**. Also, you can use **-logger**=**all** to do something for all loggers.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Notes

- The *filter* parameter in some of the commands in Table 6-2 and Table 6-3 specifies a subset of the message actions to operate on. For example, a common filter is **-verbosity** as in **set message -verbosity=HIGH**, which enables all message actions with a verbosity level of NONE to HIGH.
- For **-add** and **-replace**, the meaning of **-verbosity** is from NONE to the verbosity specified. For **-remove**, the meaning of **-verbosity** is from the verbosity specified to FULL.
- If no **-verbosity** is specified, the verbosity of **-add** and **-replace** is FULL. If no **-verbosity** is specified, the verbosity of **-remove** is NONE.

Table 6-2 set message Commands

Command	Effect
set message [**-logger**=*exp*|**all**] [**-replace**|**-add**|**-remove**] [*filter*]	Specify which message actions are watched by the logger. For details on *filter*, see “Syntax for filter” on page 6-35.
set message [**-logger**=*exp*|**all**] **-units**=*exp* [**on**|**off**]	Specify which unit instances are watched by the logger.
set message [**-logger**=*exp*|**all**] **-format**=*name*	Specify the format for the logger.
set message [**-logger**=*exp*|**all**] **-flush_frequency**=*num*	Specify the file flush_frequency for the logger.
set message [**-logger**=*exp*|**all**] **-file**=*file* [**on**|**off**]	Add or remove files to be written to by the logger.
set message [**-logger**=*exp*|**all**] **-screen** [**on**|**off**]	Turn on and off writing to screen by the logger.
set message -style=*style-name* [*filter*]	Change all the actions matching the filter to show the message in specified style. (By default, all messages are printed as BLACK.) For details on *filter*, see “Syntax for filter” on page 6-35.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Table 6-2 set message Commands (continued)

Command	Effect
set message -leader [*filter*]	Define all message actions matching the filter as leaders. (By default, there are no leaders.) For details on leaders, see "Nested Messages" on page 6-29. For details on *filter*, see "Syntax for filter" on page 6-35.
set message [**-logger**=*exp*|**all**] **-ignore**[**_tags**] [=*tags*|=**all**]	Force the logger to ignore the specified tags.

6.9.1.2 show message Summary

Table 6-3 show message Commands

Command	Effect
show message	Show a short summary of all loggers, including each logger's list of tags, highest verbosity, and number of message actions.
show message -logger=*exp*|**all** [**-full**]	Show medium information for the specified logger or all loggers (if **-logger**=**all** is specified). Show complete details if **-full** is specified.
show message -actions [*filter*]	Show all message actions, each with its associated loggers.
show message -units [=*exp*]	Show all units, each with its associated loggers.
show message -actions -ignore [*filter*]	Show the list of actions (matching *filter*) whose tag is ignored by all loggers.

6.9.2 Detailed Explanation of Commands

This section contains a detailed explanation of all messaging commands. It includes:

- "set message Commands" on page 6-34
- "show message Commands" on page 6-40

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.9.2.1 set message Commands

This section includes details on the following commands:

- "set message [-logger=exp|all] [-replace|-add|-remove] [filter]" on page 6-34
- "set message [-logger=l_exp] -units=u_exp [on|off]" on page 6-36
- "set message [-logger=exp|all] -file=file [on|off]" on page 6-37
- "set message [-logger=exp|all] -screen [on|off]" on page 6-38
- "set message [-logger=exp|all] -format=format-name" on page 6-38
- "set message [-logger=exp|all] -flush[_frequency]=num" on page 6-38
- "set message -style=style-name [filter]" on page 6-39
- "set message -leader [filter]" on page 6-39
- "set message [-logger=exp|all] -ignore[_tags] [=tags|=all]" on page 6-39

set message [-logger=*exp*|all] [-replace|-add|-remove] [*filter*]

This command selects a set of message actions specified by the filter, and adds/removes/replaces (default: **-replace**) the list of message actions watched by the specified logger (default: **sys.logger**).

All options add the tag specified in the *filter* to the list of tags recognized by the specified logger. To ignore a tag, use **set message -ignore_tags** (see below).

Following is an explanation of the three options:

-replace Enables the actions specified in *filter*, and disables all others.

In most cases you want to specify directly (and not incrementally) what each logger will do. Therefore, **-replace** is the default. For example:

```
set message LOW
```

is the same as:

```
set message -remove    // Remove all
set message -add LOW   // Add just what the user specified
```

-add Enables the actions specified in *filter* for this logger (in addition to the actions already enabled). The default for *filter* is FULL (that is, enable all actions).

-remove Disables the actions specified in *filter* from this logger. The default for *filter* is NONE (that is, disable all actions).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

For example, you can say:

```
set message -replace @vr_xbus*
    -- Look at only the 20 message actions in the vr_xbus* modules
set message -add -verbosity=MEDIUM @vr_xsoc*
    -- Add on the 10 message actions in the vr_xsoc* modules that have
    -- MEDIUM verbosity
set message -remove "...arbitration..."
    -- But remove all message actions mentioning "arbitration"
```

Syntax for *filter*

The **set message** command has an optional *filter* parameter, specifying which of the currently loaded or compiled message actions will be affected.

The syntax for *filter* is:

[**[-verbosity**=**]***verbosity*] [**-tags**=*tags-list*] [*@module-wildcard*] ["*string*"]

Note The order should be exactly as specified above.

[-verbosity=]*verbosity*	Matches the message actions whose verbosity is between LOW and *verbosity* (when using **set message -add** or **set message -replace**) or between *verbosity* and FULL (when using **set message -remove**). **Note** Instead of **-verbosity** you can omit the switch name and just specify the value. For example, “set message FULL” is the same as “set message -verbosity=FULL”.
-tags=*tags-list*	Matches the message actions whose tags are those specified by the list. **Note** If the **-tags** option is not used, then only the message actions whose tag is NORMAL are affected.
@module-wildcard	Matches the message actions residing in the specified module(s).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

"*string*"	Matches the message actions whose concatenated string is *string*. This is the same string that would be shown using **show message**.
	
	Consider the following example of a message action:
	
	`message(LOW, "Bus ", bus_num, " is done with reset");`
	
	The string shown would be: "Bus … is done with reset". And, for example, you could match it with:
	
	`show message "... reset"`
	
	Note In the following example, the string would also be "Bus … is done with reset":
	
	`messagef(LOW, "Bus %d is done with reset\n", bus_num);`

Note The "*string*" and *@module* parameters are the same as in the **set check** command. For more information, see "set checks" on page 12-1 of the *Specman Command Reference*.

Example

```
set message -add @vr_ahb*
    -- Add all actions in modules matching vr_ahb* to system.logger
set message -remove -verbosity=HIGH @vr_ahb*
    -- Remove all message actions with HIGH verbosity in specified modules
set message -add -logger=sys.dsp.logger @vr_dsp*
    -- Add to the specified logger all message actions in specified modules
set message -add -logger=sys.packet_logger -tags={VR_XBUS_PACKET}
    -- Add to the specified logger all message actions whose message tag is
    -- VR_XBUS_PACKET
```

set message [-logger=*l_exp*] -units=*u_exp* [on|off]

This command adds or removes the unit subtree under *u_exp* to the set of units watched by the specified logger (default: **sys.logger**).

For example, the following command removes all units under (and including) the audio_subsystem from the set of units watched by sys.logger.

```
set message -units=sys.audio_subsystem off
```

As detailed in the description of loggers, a message action gets processed by a logger only if that logger looks at both the executing message action and the unit in which the message action executed (as determined by **get_unit()**).

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Initially, each logger looks at its native unit tree, that is, the unit tree under the unit to which the logger is attached. It is an error to specify **-unit**=*some_unit* if *some_unit* is not included in the native unit tree for the logger.

Note *u_exp* can be either a unit or a list of units (in which case, the subtrees under each of the units are turned on or off).

The **set message -units** command is useful if you want to see less output from one of the many instances of, say, an ***e***VC. For example, suppose you have:

```
uarts[10]: list of uart
```

You can then issue:

```
set message -units=sys.uarts[3] off
```

Note Usually you want to have lower verbosity for almost everything, and higher verbosity for just one or more units. This is best achieved by using lower verbosity in **sys.logger** and turning the logger(s) of the desired unit(s) to higher verbosity (without using the **-units** option at all).

set message [-logger=*exp*|all] -file=*file* [on|off]

This command adds *file* to the specified logger, or (if **off** is specified) removes the *file* from the specified logger.

The default extension for *file* is ".elog". Thus the following command will write to foo.elog:

```
set message -logger=sys.main_file_logger -file=foo
```

Each logger can write to any number of files, in addition to optionally writing to the screen (see "set message [-logger=exp|all] -screen [on|off]" on page 6-38).

Users often want to log information into separate files (for example, one file per ***e***VC type or one file per ***e***VC instance, one file altogether, and so on). We recommend using the message action for that purpose as well. Some points to keep in mind are:

- Several loggers can write to the same file.

  We recommend setting the writing to files via constraints (see "Configuring Loggers via Constraints" on page 6-23).

- We recommend using a message_tag to mark message actions meant for going to loggers that will write to files. For example, if you would normally want to write to two files (one for packet information and one for byte information), we suggest defining two tags as in the following example:

  ```
  extend message_tag: [VR_XBUS_PACKET, VR_XBUS_BYTE];
  ```

  Then (using a command, a constraint, or a method) you can have only message actions using VR_XBUS_PACKET go to the packet logger, and so on. With commands, this will look as follows:

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
set message -logger=sys.packet_logger -tags={VR_XBUS_PACKET}
set message -logger=sys.packet_logger -file=packet_log.elog
```

- When you use the **set message -file**=*file* command, Specman first checks if that file is already open (via some other **set message -file** command). If the file is not open, Specman opens it (and writes a message to the screen indicating that). If the file is already open, Specman simply marks that logger as also going to that file.
- When you use **set message -file**=*file* **off**, Specman will stop writing to that file. When there are no more loggers writing to a file, then Specman closes the file (and writes a message to the screen indicating that).
- Specman flushes the buffer of each file after every *n* writes to that file, where *n* starts out as 10 but can be modified using the **set message -flush_frequency** command (see "set message [-logger=exp|all] -flush[_frequency]=num" on page 6-38).
- Every file gets flushed whenever you get to the Specman prompt.
- All files are flushed and closed just before Specman exits. There is no auto-closing at end of test.

set message [-logger=*exp*|all] -screen [on|off]

This command makes a logger write to the screen or stop writing to the screen (if **off** is specified).

set message [-logger=*exp*|all] -format=*format-name*

This command specifies the format of the logger. The format must be a legal value of the type message_format.

Initially, message_format is defined as:

```
type message_format: [short, long, none];
```

Loggers start out with the short format. For details on how these formats influence the output, see "Output Appearance" on page 6-17.

Note Users can extend message_format and can also format the output programmatically by extending the method **logger.format_message()**. For more information, see "Messaging Procedural Interface" on page 6-41.

set message [-logger=*exp*|all] -flush[_frequency]=*num*

This command causes flushing to the disk of each message output file associated with the specified logger every *num* writes. The default is every 10 writes.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

The message output files are flushed out in any case when quitting Specman and any time you reach the Specman prompt. However, if you are looking at the output files using, for example, the "tail -f" UNIX command, you might want to flush out files frequently.

Using **set message -flush**=1 will cause flushing after each write (but will, of course, have some performance implications).

Note Using the **set message -flush_frequency** command also flushes the files immediately (that is, when the command is issued).

set message -style=*style-name* [*filter*]

This command changes the style associated with all specified actions to the requested style. For example, to color all reset messages purple so that they will stand out, you can use:

```
set message -style=PURPLE "...Reset..."
```

Notes

- Initially all messages are printed as BLACK.
- This command is global (to all loggers).
- This command only influences the actual message (that is, the information after the colon). You must use **unit.short_name_style()** to influence the style of the short_name.

set message -leader [*filter*]

This command sets the leader for a series of subsequent nested messages (see "Nested Messages" on page 6-29). For example, to set all messages with the NORMAL tag as leaders, you can use:

```
set message -leader on
```

Note Setting leaders changes the execution order of nested messages. For more information, see "Nested Message Semantics" on page 6-29.

set message [-logger=*exp*|all] -ignore[_tags] [=*tags*|=all]

This command forces the logger to ignore the specified tags.

Notes

- If no tags are specified, then NORMAL tag is assumed.
- If "=all" is specified, all tags are ignored, which effectively disables the logger.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- The difference between **-ignore_tags** and **-remove** is that **-remove** eliminates the unique destinations of a logger for messages with the specified tags, whereas **-ignore_tags** eliminates the impact of the logger's destinations on messages with the specified tags. When **-ignore_tags** is used, the end result depends on the remaining loggers on the way to **sys**. Verbosity might be increased or decreased.

6.9.2.2 show message Commands

This section includes details on the following commands:

- "show message" on page 6-40
- "show message -logger=exp|all [-full]" on page 6-40
- "show message -actions [filter]" on page 6-41
- "show message -units [=exp]" on page 6-41
- "show message -win[dow]" on page 6-41

show message

This command shows a short summary of all loggers. The output looks like the following:

```
0. sys.logger - NORMAL (HIGH) ,TRACE_SEQUENCE (HIGH) ,TRACE_OBJECTION
(MEDIUM)  - 35 actions
1. sys.ex_soc_env.atms[0].logger  - 0 actions
2. sys.ex_soc_env.atms[0].file_logger - NORMAL (HIGH)  - 24 actions
3. sys.ex_soc_env.atms[1].logger  - 0 actions
4. sys.ex_soc_env.atms[1].file_logger - NORMAL (HIGH)  - 24 actions
5. sys.ex_soc_env.atms[2].logger  - 0 actions
6. sys.ex_soc_env.atms[2].file_logger - NORMAL (HIGH)  - 24 actions
7. sys.ex_soc_env.c_bus_env.logger  - 0 actions
8. sys.ex_soc_env.c_bus_env.file_logger  - 0 actions
9. sys.ex_soc_env.logger  - 0 actions
10. sys.ex_soc_env.file_logger  - 0 actions
```

show message -logger=*exp*|all [-full]

This command shows information for the specified logger(s).

To see information for all loggers, use:

```
show message -logger=all
```

If **-full** is specified, then all information for the specified loggers is displayed. Otherwise, MEDIUM information is displayed.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

The output of this command looks like the following:

```
0. sys.logger - NORMAL (HIGH) ,TRACE_SEQUENCE (HIGH) ,TRACE_OBJECTION
(MEDIUM)  - 35 actions:
 Units: 14   Format: short
 Destinations: screen
```

Note The actual verbosity (HIGH in the above example) represents the current maximum verbosity (as changed, for example, via a **set message** command). This can be different from the initial verbosity as set by constraints (LOW in the above example).

show message -actions [*filter*]

This command shows all message actions matching *filter* (default: all message actions).

For each action, it shows the list of loggers looking at it.

show message -units [=*exp*]

This command shows as a tree all unit instances under *exp* (default: all unit instances).

For each unit, it shows which loggers are looking at it (closest to the unit first).

show message -win[dow]

This command opens the ***e***RM Utility with various options for loggers, messages, and destinations. (See "Controlling Messaging" on page 8-16.)

6.10 Messaging Procedural Interface

A message logger has an API, corresponding (more or less) to the commands explained in "Messaging Command Interface" on page 6-31. This section explains the messaging procedural interface.

This section includes:

- "Methods for Setting Configuration" on page 6-42
- "Methods for Showing Configuration" on page 6-43
- "Methods Called While Handling Messages" on page 6-43
- "Query Methods for Getting Message Information" on page 6-44

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.10.1 Methods for Setting Configuration

The following methods of message_logger set the configuration. For details on the command parameters, see "set message Commands" on page 6-34.

set_actions(*verbosity*: message_verbosity, *tags*: list of message_tag, *modules*: string, *text*: string, *op*: message_operation**)** • Adds, removes, or replaces the specified actions for the logger. • Corresponding command: "set message [-logger=exp|all] [-replace|-add|-remove] [filter]"
set_units(*root*: any_unit, *to*: message_on_off**)** • Sets the unit tree under *root* to be either on or off for the message logger. • Corresponding command: "set message [-logger=l_exp] -units=u_exp [on|off]"
set_format(*format*: message_format**)** • Corresponding command: "set message [-logger=exp|all] -format=format-name"
set_flush_frequency(*flush_frequency*: int**)** • Corresponding command: "set message [-logger=exp|all] -flush[_frequency]=num"
set_file(*fname*: string, *to*: message_on_off**)** • Corresponding command: "set message [-logger=exp|all] -file=file [on|off]"
set_screen(*to*: message_on_off**)** • Corresponding command: "set message [-logger=exp|all] -screen [on|off]"
set_style(*verbosity*: message_verbosity, *tags*: list of message_tag, *modules*: string, *text*: string, *style*: vt_style) • Sets the specified actions to the requested style (for example, GREEN) • Corresponding command: "set message -style=style-name [filter]"
set_leader(*verbosity*: message_verbosity, *tags*: list of message_tag, *modules*: string, *text*: string, *status*: message_on_off**)** • Sets the specified actions to be leaders • Corresponding command: "set message -leader [filter]"

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

ignore_tags(_tags:_list of message_tag**)**
• Forces the logger to ignore the specified tags

6.10.2 Methods for Showing Configuration

The following methods of message_logger show configuration. For details on the command parameters, see "show message Commands" on page 6-40.

show_message(_all_: bool, _full_: bool**)**
• Corresponding command: "show message -logger=exp|all [-full]"
show_actions(_verbosity_: message_verbosity, _tags_: list of message_tag, _modules_: string, _text_: string**)**
• Corresponding command: "show message -actions [filter]" **Note** This command shows information for all loggers.
show_units(_root_: any_unit**)**
• Corresponding command: "show message -units [=exp]" **Note** This command shows information for all loggers.
get_tags(): list of message_tag
• Returns the tags currently recognized by the logger

6.10.3 Methods Called While Handling Messages

The following methods are predefined, but they can also be modified by the user.

Note These methods can use the various query methods (see "Query Methods for Getting Message Information" on page 6-44) to get information about the current message.

```
extend message_logger {

    // Return TRUE if the current message should be enabled.
    // By default returns TRUE, but you can set it to FALSE when you
    // want to ignore the logger messages and block all of the logger
    // destinations.
    accept_message(): bool is {
        return TRUE;
    };
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        // Returns the list of string, which will be sent as-is to the
        // file or screen (Specman will add a "\n" at the end of
        // each string when printing them to the file/sceen). If you want an
        // extra empty line at the end, add a "\n" to the last string.
        // By default, this method obeys the current format as described
        // in the manual, but you can add more formats or change the meaning
        // of the existing formats.
        format_message(): list of string is {
            ...    // Compute result according to current format
        };
    };
```

For example, if you want the short format to be:

>>> 100: *message-text*

then you could write:

```
    extend message_logger {
        format_message(): list of string is first {
            if get_format() == short then {
                result = get_message();
                result[0] = dec(">> ", sys.time, " ", result[0]);
                return result;
            };
        };
    };
```

6.10.4 Query Methods for Getting Message Information

The following methods are available for use within **accept_message()** and **format_message()**. They return information about the message action that was just executed.

get_action_style(): vt_style Return the vt_style for the current message action (for example, GREEN)
get_format(): message_format Return the message format of the current logger
get_message(): list of string Return the current raw message as produced by the message action

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

get_message_action_id(): int Return a unique number identifying the message action
get_tag(): message_tag Return the tag of the message action
get_time(): string Return the value and color of "[sys.time]", for example, "[5]"
get_verbosity(): message_verbosity Return the verbosity of the message action
source_location(): string Return the source location where the message occurred, for example, "At line 12 in @foo"
source_method_name(): string Return the name of the method where the message occurred, for example, "foo()"
source_struct(): any_struct Return the actual source struct where the message occurred
source_struct_name(): string Return the name of the struct type where the message occurred, for example, "packet"

The following example shows the use of query methods for formatting messages:

```
<'

extend message_logger {

    format_message(): list of string is only{
        var msg_list := get_message();
        var msg := msg_list[0];
        var s: string;

        -- This code colorizes the msg (-style= <color>)
        var style := get_action_style();
        if (style != BLACK) {
            msg = vt.text_style(style,msg);
        };
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

```
            var current_struct := source_struct();
            var short_name_path :=
                         current_struct.get_unit().short_name_path();
            var time := get_time();
            var verbosity_str := get_verbosity();

            case format {
                long: {
                    s = append(time, short_name_path," (",verbosity_str,") " ,
                            source_location(), " in ",current_struct,
                                                                   ":\n",msg);
                };
                short: {
                    s = append(time, short_name_path,
                            (short_name_path == NULL ? " " : ": "), msg);
                };
                none: {
                    s = append(msg);
                };
            };

            if (msg_list.size() > 1) {
                result = {s;msg_list[1..]};
            } else {
                result = {s};
            };
        };

    };

    '>
```

6.11 Recommended Methodology for the Message Action in eRM eVCs

This section includes:

- “Screen Logger” on page 6-47
- “File Logger(s)” on page 6-47
- “Handling Short Names and Styles” on page 6-49
- “Coloring of Whole Message Actions” on page 6-50
- “Special Tags” on page 6-51

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- "Tracing Sequences" on page 6-51
- "at_message_verbosity()" on page 6-51
- "Verb Tenses" on page 6-52
- "Soft Constraints" on page 6-52
- "sys.logger" on page 6-52
- "When to Issue Message Commands" on page 6-52
- "Extending message_logger" on page 6-54
- "show message -actions" on page 6-54
- "Scaling the Time Portion of Message Output" on page 6-54

6.11.1 Screen Logger

Each ***e***VC and agent should have a screen logger. For example:

```
extend vr_xbus_env_u {
    logger: message_logger is instance;
};
```

We recommend that developers not constrain the tags (accepting all defaults) so that **sys.logger** can provide centralized control. To get more information about a particular instance, users can write something like:

```
extend XBUS_1 vr_xbus_env_u {
    keep soft logger.verbosity == HIGH;
};
```

Note Always use soft constraints to allow additional changes later.

6.11.2 File Logger(s)

File loggers are not required, but you should probably have a file logger at the ***e***VC env level. If desired, you can also have a file logger (and even, in special cases, multiple file loggers) for each agent.

As configured by the ***e***VC developer, we recommend that all of these loggers go to the file *evc_name*.elog and start out with LOW verbosity.

Following are two typical cases.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

One File Logger Using NORMAL Tag

In this case, the ***e***VC developer decides not to create special message actions for writing to files. Hence, the normal message actions (NORMAL tag) are also used for sending to a file (depending on verbosity).

The developer defines something like the following:

```
extend vr_xbus_env_u {
    file_logger: message_logger is instance;
    keep soft file_logger.verbosity == LOW;
    keep soft file_logger.tags == {NORMAL};
    keep soft file_logger.to_file == "vr_xbus.elog";
    keep soft file_logger.to_screen == FALSE;
};
```

End users might choose to have HIGH verbosity for the file logger and LOW verbosity for the screen, or vice versa. In that case, they could write:

```
extend vr_xbus_env_u {
    keep soft file_logger.verbosity == HIGH;
    keep soft logger.verbosity == LOW;
};
```

Several Per-Topic File Loggers

In this case, the ***e***VC developer chooses to provide several per-topic file loggers. For example, one logger might show the raw data (in some special tabular format) while another would show packet-level traffic (in another special format). In the corresponding message actions, we suggest using tags like the following:

```
messagef(VR_XBUS_DATA, MEDIUM, "%5d. %2x %8x\n", index, control, data);
```

The developer would then define something like the following:

```
extend vr_xbus_env_u {
    data_file_logger: message_logger is instance;
        keep soft data_file_logger.verbosity == LOW;
        keep soft data_file_logger.to_file == "vr_xbus.elog";
        keep soft data_file_logger.tags == {VR_XBUS_DATA};
        keep soft data_file_logger.format == none;
    packet_file_logger: message_logger is instance;
        keep soft packet_file_logger.verbosity == LOW;
        keep soft packet_file_logger.to_file == "vr_xbus.elog";
        keep soft packet_file_logger.tags == {VR_XBUS_PACKET};
        keep soft packet_file_logger.format == none;
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

As defined above, the two streams go into the same file. However, end users (or integrators) can decide to split them any way they like. For example, if there are three instances of the vr_xbus_env_u, distinguished by their name (*instance_id*) field (which would be XBUS_1, XBUS_2, and XBUS_3), then the end user might want six files, one for each (name, stream) combination. This can be done as follows:

```
extend vr_xbus_env_u {
    keep soft data_file_logger.to_file == append(name, "_data.elog");
    keep soft packet_file_logger.to_file == append(name, "_packet.elog");
};
```

Other combinations are easily achieved as well. For example:

- No file logging needed.
- Log everything to one file.
- Log to *n* separate files, one for each log type.
- Log to *m* separate files, one for each instance.
- Log to *n* * *m* files.
- Anything in between.

Note Each specified file gets created even if it is empty.

6.11.3 Handling Short Names and Styles

Short Names

We recommend that each ***e***VC and each agent have a non-empty **short_name()**. Typically, this would be the name (*instance_id*) field of the ***e***VC or agent. For example:

```
extend vr_xbus_env_u {
    short_name(): string is {
        return append(name);
    };
};
```

This ensures that the full short-name path printed with each message will look something like:

```
[10340] AHB_1 M2: Reset is done
```

Notes

- Going lower than the agent level (for example, defining a non-empty short name for the BFM) is generally not needed. The context is usually clear from the message.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

- Users usually will not need to change the short names.

Styles (Colors)

For each ***e***VC, providers should select a color that will be used when writing the short name of that ***e***VC. To see all available styles, use **vt.show_styles()**.

For example:

```
extend vr_ahb_env {
    short_name_style(): vt_style is {
        return GREEN;
    };
};
```

This will cause two things:

- The word “AHB_1” will be green in the message below:

  [10340] AHB_1 M2: Reset is done

- The name of AHB_1 in the **show_banner()** output (after initial generation) will also appear green.

Usually, users will not need to modify this. However:

- They might decide to change some of the styles to make each ***e***VC unique.
- They might also decide to have the styles go by ***e***VC instance, rather than by ***e***VC type, using something like:

  ```
  return (name == AHB_1 ? GREEN : name == AHB_2 ? CYAN : PURPLE);
  ```

- They might decide to color each agent separately. That is, they could add a (non-BLACK) style to M1, M2, and so on. Generally we discourage this, because so many colors might cause a rainbow effect.

6.11.4 Coloring of Whole Message Actions

By default, the actual text of the message (after the colon) appears black. We recommend that ***e***VC providers not change that.

However, occasionally, end users or integrators might want to change the message text color. For example, they might want to color some rare messages so that they will be easier to spot. They can do that as follows:

```
set message -style=CYAN "Reset ..."
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.11.5 Special Tags

Sometimes ***e***VC developers might want to insert special message actions to help a user (or the developers themselves, remotely) debug a particular algorithm.

For example, the logic for matching scoreboard items might be very complex and unsure. So the developer could insert message actions like the following:

```
message(VR_XBUS_SCOREBOARD, HIGH, "Looking for match for ", the_packet) {
    print the_packet;
};
```

This tag would be disabled by default, but it could be enabled when needed.

6.11.6 Tracing Sequences

The **trace sequence** command issues its messages using **message()**. Therefore, they come out in the familiar message format.

These messages use the TRACE_SEQUENCE tag, which is enabled by default in **sys.logger**.

Using the **set message** commands, you can write these messages to files, filter them by verbosity, and so on.

See Also

- "Tracing Sequences" on page 5-71

6.11.7 at_message_verbosity()

There is a global method **at_message_verbosity()**. Its syntax is:

at_message_verbosity(verbosity: *message_verbosity***)**: bool;

The method returns TRUE if you are currently at the specified verbosity (or higher). It might be used as follows:

```
extend packet {
    do_print() is also {
        if at_message_verbosity(HIGH) then {
            out("Some more details about this packet:");
            out("...");
        };
    };
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

If you are currently executing a message, **at_message_verbosity()** uses the verbosity of the current message logger. Otherwise, it uses the verbosity of **sys.logger**.

6.11.8 Verb Tenses

We recommend phrasing message in the past tense. For example:

```
Packet-@3 sent to BFM

Received packet-88
```

The exception to this is when you want to insert a message before some lengthy operation begins. In that case you should use the present continuous tense. For example:

```
message(MEDIUM, "Sending ", packet, " to BFM");
send_to_bfm(packet);
```

While you can configure loggers in various ways; some standard recommendations follow.

6.11.9 Soft Constraints

We recommend configuring loggers via constraints. Use the procedural or command interface only when needed. In general, it is better to use soft constraints because you never know if your decisions might need to be overridden.

6.11.10 sys.logger

We recommend that end users normally use **sys.logger**.

By default, only LOW verbosity messages go to screen. If end users just want messages at some verbosity, they can use **sys.logger**. For example, they might issue the following command:

```
set message -verbosity=MEDIUM
```

Of course, they could achieve the same thing via a constraint as follows:

```
extend sys {keep soft logger.verbosity == MEDIUM};
```

End users should use the per-***e***VC and agent loggers only when they need more flexibility (for example, higher verbosity for one ***e***VC out of many).

6.11.11 When to Issue Message Commands

Normally, we recommend configuring loggers via constraints. However, sometimes you may want to change the verbosity of some loggers dynamically during runtime.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

Note Initially only **sys.logger** exists. The other loggers get created during the **gen** phase. Therefore, until **sys** is fully generated, only **sys.logger** should be used.

Following are some typical places to issue **set message** commands:

- If you are running with a simulator, you can issue **set message** commands after the **test** command. For example:

```
test
set message -logger=sys.soc1.logger -verbosity=FULL
...        // Issue simulator command to run the test here
```

- Often, you might want to enable messaging only after a specific method has been reached. For example:

```
break foo.bar
test
set message -verbosity=FULL
continue
```

- When running in standalone mode (without a simulator), there is no natural stop before the actual run. Hence, you can set a breakpoint on **sys.run**. For example:

```
break sys.run
test
set message -verbosity=FULL
continue
```

- Commands that relate only to **sys.logger** can be issued even before initial generation. These commands will also carry over after generation. For example, issuing the following command causes the entire test to run in HIGH verbosity:

```
set message -verbosity=HIGH
test
```

- You can configure messaging from within ***e*** code, using either:

```
specman("set message ...")
```

or via the corresponding API, for example:

```
extend vr_xbus_env_u {
    run() is also {
        logger.set_actions(FULL, {NORMAL}, "*", "...", replace);
            -- Sets vr_xbus_env_u.logger to FULL verbosity
    };
};
```


© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.

6.11.12 Extending message_logger

We discourage subtyping message_logger.

While it is possible to define "**unit** *my_logger* **like** message_logger {…}" and instantiate *my_logger*, in general we recommend using the original message_logger.

It is acceptable to extend message_logger and change, for example, the **format_message()** method.

6.11.13 show message -actions

We recommend using **show message -actions** to find the source of message actions.

The **show message -actions** command can be used to see what message actions exist in the current verification environment. This lets you inspect the verification environment and see what it can do.

The **show message -actions** command also lets you find a specific message action and put a breakpoint on it. For example, if you want to find the message action that printed "Channel 3 sent a packet" and you want to stop in the debugger just before it is emitted, you can do the following:

1. Issue the command:

   ```
   show message -actions "... sent a packet"
   ```

2. Click the hyperlink to go to the source.
3. Put a breakpoint there.

6.11.14 Scaling the Time Portion of Message Output

To scale the time portion of your message output to a specified time unit, for example **ns**, use the **set_config()** method. For example:

```
set_config(print, scale, ns);
```

or, to use the current Specman timescale:

```
set_config(print, scale, simdef);
```

This might cause the printing to look as follows:

```
[123000 ns] AHB_0 MASTER_1: Received first part of transaction-@33
```

Note The NULL simulator (the Specman standalone simulator) uses **fs** as the default time unit.

For more details, see Chapter 12 "Specman Timescale" in the *Specman Elite Integrator's Guide*.

© 2006 Cadence Design Systems, Inc. All rights reserved worldwide.