

24. Messages 1

24.1 Overview 5

The messaging feature is a centralized and flexible mechanism used to write text messages to the screen or log files. It lets a developer easily insert formatted messages into code and provides the user with powerful and flexible controls to selectively enable or disable groups of messages. 10

The three most typical uses for messages are:

- a) Summaries—writing summary information at the beginning or end of significant chunks of activity;
- b) Tracing—writing detailed trace messages during the simulation upon interesting events;
- c) Debugging—writing detailed debug messages during the run to help the user or developer debug unexplained behaviors. 15

Messages are different from plain **out()** and **outf()** calls (see 28.6); they have a standard-format prefix that can be turned on or off. Messages are also different from **dut_error()** calls (see 16.2.2); they do not signify failure, increment error counters, or increment warning counters. 20

24.2 The message model

Upon execution, 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, etc.), and to send them to various destinations (files and the screen). 25

The message loggers are instantiated by the programmer in the `unit` hierarchy and then configured using constraints (see 24.5) or method calls (see 24.6). 30

There is a predefined message logger instantiated in every environment under `sys` called **logger**.

24.3 message and messagef 35

Purpose	Create a (formatted) message and send it to a message logger	
Category	Statement	
Syntax	message (<i>[tag]</i> , <i>verbosity</i> , [<i>exp</i> , ...]) [<i>action_block</i>] messagef (<i>[tag]</i> , <i>verbosity</i> , <i>format_exp</i> , [<i>exp</i> , ...]) [<i>action_block</i>]	40
Parameters	<i>tag</i>	A constant of type <code>message_tag</code> , either NORMAL or a user-defined tag (see 24.3.1).
	<i>verbosity</i>	A constant of type <code>message_verbosity</code> : one of NONE , LOW , MEDIUM , HIGH , or FULL (see 24.3.2). 45
	<i>exp</i>	Value(s) to write.
	<i>action_block</i>	A block of actions to perform, the output of which is taken as part of the message output. 50
	<i>format_exp</i>	For messagef() , an outf() -style format string for the output.

When a **message()** or **messagef()** action is executed, the following happens. 55

- 1
- a) If there are no handling loggers for the action, then the action is skipped (see 24.4.2).
- b) If there are handling loggers for the action, the action creates a message (consisting of a list of strings, plus related information) and sends it to all of the handling loggers. Those loggers format the message and send it to the screen and/or log files.
- 5
- 1) For **message()** first string in the message is created by appending all of the expressions, like **out()** does.
- 2) For **messagef()**, the first string is created using the *format-exp*, similar to **outf()**.
- 10
- i) **messagef()** does not automatically add a newline (`\n`) to the message string. Therefore, if the optional *action-block* requires a newline to be written before it is executed, terminate the *format-exp* using `\n`.
- ii) If the fully composed message string—including that portion written by the optional *action-block*—is not terminated by a newline, a newline is appended. **messagef()** also allows appending of the *action-block* output to the **messagef()** header output.
- 15
- c) If an *action-block* exists, it gets executed. This typically contain further output-producing actions, calls to reporting methods, etc. The output of all of those is added, as a list of string, to the message.

Message code shall not modify the flow of the simulation in any way. Time-consuming operations in message headers or action blocks are strictly disallowed.

20 Syntax examples:

```

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) {
  write the_packet;
};
  -- Output this message and write 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.

```

24.3.1 Tag

Both **message()** and **messagef()** have an optional first parameter of type `message_tag`, which is initially defined as:

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

This can be extended, e.g.,

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

45 If a *tag* is not specified (i.e., the first parameter of **message()** or **messagef()** is a legal value for verbosity), then the value `NORMAL` is prepended. Thus, the following two lines are the same:

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

50 Message tags are used for associating specific message actions with a message logger (see 24.4).

24.3.2 Verbosity

The *verbosity* parameter can be set to **NONE**, **LOW**, **MEDIUM**, **HIGH**, or **FULL** (from lowest to highest). Lower *verbosity* implies a more important message.

Table 46 shows the recommended usage of verbosity. Each level can assume that all lower levels are also writing (thus, there is no need to repeat them).

Table 46—Verbosity levels

Level	Recommended use	Examples
NONE	Critical messages (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"
HIGH	More detailed per-data-item information, including: <ul style="list-style-type: none"> — the actual value of the packet — sub transaction details. 	"Full details for packet-@36: len=5 kind=small ..."
FULL	Anything else, including writing by using specific methods (just to follow the algorithm of that method).	

24.3.3 Nested message actions

A message action may be designated as a *leader*, i.e., responsible for all message actions inside its lexical scope or in methods called from it. All nested message actions are handled by the leader's loggers (and only by them). The nested messages are formatted sequentially according the leader's rules, as if their output code resides directly inside the leader's lexical scope. This is true even if some of nested message actions are themselves designated as leaders—the outermost leader overrides the rest. To designate a message action as leader, see 24.6.1.7.

For sample usage of nested message actions, see 24.7.2 and 24.7.3.

24.4 Message loggers

24.4.1 Overview

message_logger is a predefined unit type, which is used to manage the output from message actions: filtering it, formatting it, or sending it to one or more destinations. Message loggers are defined programmatically and attached as fields to various units in the unit hierarchy, e.g.,

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

The following can be specified (via methods or constraints) for each message logger:

- which subset of the message actions the logger examines. By default, each logger uses a verbosity of `NONE` and its tag list is empty. This passes the control to the predefined logger `sys.logger` until the (new) logger is explicitly enabled.
- which subset of the unit instances the logger examines. By default, this is the tree starting at the unit to which the logger is attached, e.g., each `vr_xbus_env_u.logger` looks at the unit sub-tree under the corresponding `vr_xbus_env_u` element.
- which destinations (files or screen) to use for receiving output.
- what format to use.

Loggers, like other units, are generated at elaboration time before the simulation run begins.

24.4.2 Handling messages

During execution, messages are processed by loggers in the following manner.

- a) Upon execution of a message action, the set of loggers relevant to that message is determined. The relevant messages are those for which both the following conditions hold.
 - 1) They are associated with the object's origin unit—the unit instance returned by `any_struct.get_unit()` (see 7.5.1).
 - 2) The current setting applies to them according to its verbosity, tag, and other criteria (see 24.6.1).
- b) Then, for each of the relevant loggers:
 - 1) `accept_message()` is called (see 24.6.2); if it returns `FALSE` (due to explicit user refinement), the logger is skipped.
 - 2) `format_message()` is called (see 24.6.2) to retrieve the final message format.
 - 3) The message in its final format is written to each of the destinations assigned to the logger (the screen and/or any log files).

Each message may be sent to a given destination at most once. If two or more loggers handle the same message and are assigned the same destination, the message shall be written only once according to the format defined by one of them ([this is implementation-dependent](#)).

24.5 Configuring message loggers with constraints

Loggers may be configured by using constraints or through a messaging interface (see 24.6). *Constraints* are used during pre-run generation to set the fields of the logger. Then, during `post_generate()` (see 9.4.3), the logger fields are used to configure the logger. At post-generation, the logger becomes attached to the unit (specifically, the unit computed by `logger-instance.get_unit()`). The `post_generate()` method of a logger uses the generated values of the fields to configure the logger via the procedural interface.

Table 47 shows the constrainable fields of the unit `logger`, along with the built-in soft constraints that determine their default values.

Table 47—Logger constrainable fields

Field	Description
<code>tags: list of message_tag;</code> <code>keep soft tags == {};</code>	The message tags for selecting the actions for this logger.
<code>verbosity: message_verbosity;</code> <code>keep soft verbosity == NONE;</code>	The verbosity for selecting the actions for the logger.

Table 47—Logger constrainable fields (Continued)

Field	Description
<code>modules: string;</code> <code>keep soft modules == "*";</code>	The modules wildcard for selecting the actions for the logger.
<code>string_pattern: string;</code> <code>keep soft string_pattern == "...";</code>	The pattern to match against the string in the message action.
<code>to_file: string;</code> <code>keep soft to_file == "";</code>	The file to which the logger writes (or none if the setting is ""). The default extension for the file name is <code>.elog</code> .
<code>to_screen: bool;</code> <code>keep soft to_screen == TRUE;</code>	When set to TRUE, the logger also writes any output to the screen.

The following considerations also apply.

- Only one file may be associated with a logger via constraints. However, multiple files are allowed when using the messaging procedural interface (see 24.6).
- By default, loggers ignore all tags and have a verbosity of NONE. The one exception is **sys.logger**, where the tag list defaults to {NORMAL} and the verbosity defaults to LOW. Thus, all messages are controlled by **sys.logger**.
- By constraining the verbosity to any value above NONE and leaving the tag list empty, the tag list defaults to {NORMAL}.

24.6 Messaging Procedural Interface (PI)

The following methods of struct type *message_logger* can be used to set its configuration before or during execution, refine its filtering or formatting rules, and query for details of the currently processed message action.

What are the various **default settings/action associated with each method/action [[new table](#)]??

24.6.1 Methods for setting configuration

The following methods of struct type *message_logger* are used to set its configuration.

24.6.1.1 set_actions

Purpose	Add, remove, or replace the specified actions for the logger
Category	Method
Syntax	set_actions (<i>verbosity, tags, modules, text, op</i>)
Parameters	<i>verbosity</i> A constant of type <code>message_verbosity</code> : one of NONE , LOW , MEDIUM , HIGH , or FULL (see 24.3.2). This matches the message actions whose verbosity is between LOW and <i>verbosity</i> (when <i>op</i> = add or replace) or between <i>verbosity</i> and FULL (when <i>op</i> = remove).
	<i>tags</i> Matches the message actions whose tags (see 24.3.1) are those specified by the list (of type <code>message_tag</code>).
	<i>modules</i> Matches the message actions residing in the module(s) that match this string expression .
	<i>text</i> Matches the message actions, of which one of the parameters is a literal string that matches the string expression .
	<i>op</i> One of add , replace , or remove . This determines whether the message actions matched by previous parameters are added to the logger, removed from it, or used to replace the currently assigned action.

This adds, removes, or replaces the specified actions for the logger.

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        logger.set_actions(FULL, {NORMAL}, "*", "...", replace);};
};

```

24.6.1.2 set_unit

Purpose	Set the unit tree under <code>root</code> to be on or off for the message logger
Category	Method
Syntax	set_unit (<i>root, status</i>)
Parameters	<i>root</i> The <code>root</code> unit of the unit tree to add to or remove from the set of unit instances associated with the logger.
	<i>status</i> One of on or off ; this determines whether the units are added or removed.

This adds (**on**) or removes (**off**) units from the logger.

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        logger.set_unit(testbench_01, off);};
};

```

24.6.1.3 set_format

Purpose	Set the format for messages associated with the message logger
Category	Method
Syntax	set_format (<i>format</i>)
Parameters	<i>format</i> One of short , long , or none . The format modes are implementation-dependent.

This sets the format mode for any messages associated with the logger.

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {
    logger.set_format(none);
  };
};

```

24.6.1.4 set_flush_frequency

Purpose	Set the message flushing frequency of each message output file associated with the logger
Category	Method
Syntax	set_flush_frequency (<i>num</i>)
Parameters	<i>num</i> The maximum number of message actions executed before their content is flushed out.

This sets the maximum number of message actions allowed before flushing the message queue for the logger.

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {
    logger.set_flush_frequency(15);
  }
};

```

24.6.1.5 set_file

Purpose	Add a file to or remove a file from the specified logger
Category	Method
Syntax	set_file (<i>fname</i> , <i>status</i>)
Parameters	<i>fname</i> The log file to add or remove.
	<i>status</i> One of on or off ; this determines whether the files are added or removed.

This adds a file to the specified logger (or if `off` is specified, removes it from the logger). Each logger can write to any number of files, including writing to the screen (see 24.6.1.6).

Syntax example:

```
if to_file != "" then {
    set_file(to_file, on);
};
```

24.6.1.6 set_screen

Purpose	Write a logger to the screen or stop writing to the screen
Category	Method
Syntax	set_screen (<i>status</i>)
Parameters	<i>status</i> One of on or off ; this determines whether start or stop writing to the screen.

This writes (`on`) or stops writing (`off`) a logger to the display screen.

Syntax example:

```
if to_screen {
    set_screen(on);
};
```

24.6.1.7 set_leader

Purpose	Set the specified messages' actions as leaders or non-leaders.	
Category	Method	
Syntax	set_leader (<i>verbosity, tags, modules, text, status</i>)	
Parameters	<i>verbosity</i>	A constant of type <code>message_verbosity</code> : one of NONE , LOW , MEDIUM , HIGH , or FULL (see 24.3.2). This matches the message actions (see 24.6.1.1) whose verbosity is between LOW and <i>verbosity</i> (when <i>op</i> = add or replace) or between <i>verbosity</i> and FULL (when <i>op</i> = remove).
	<i>tags</i>	Matches the message actions whose tags (see 24.3.1) are those specified by the list (of type <code>message_tag</code>).
	<i>modules</i>	Matches the message actions residing in the module(s) that match this string expression .
	<i>text</i>	Matches the message actions, of which one of the parameters is a literal string that matches the string expression .
	<i>status</i>	One of on or off ; this determines whether to set the specified messages' actions as leaders (on) or non-leaders (off).

This sets the specified actions to be leaders (or if `off` is specified, to cease from being leaders); see also 24.3.3.

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {
    logger.set_leader(LOW, {NORMAL}, "*", "...", on);
  };
};

```

24.6.1.8 ignore_tags

Purpose	Ignore the specified tags	
Category	Method	
Syntax	ignore_tags (<i>tags</i>)	
Parameters	<i>tags</i>	Ignores the message actions whose tags are those specified by the list (of type <code>message_tag</code>).

This causes the logger to ignore messages of this tag-type.

Syntax example:

```

extend message_tag: [VR_XBUS_PACKET];
extend vr_xbus_env_u {
  run() is also {
    logger.ignore_tags({VR_XBUS_PACKET});
  };
};

```

24.6.2 Hook methods for refining message handling

The predefined methods shown in this subclause can also be modified ([extended](#)).

****Are these the *only methods* in this clause which *can be extended*??**

24.6.2.1 accept_message

Purpose	Check if the current message is enabled
Category	Method
Syntax	<code>accept_message():bool</code>
Parameters	None
Return value	A Boolean value

This returns `TRUE` (the default) if the current message is enabled. When set to `FALSE` (see the example below), all logger messages are ignored and all logger destinations are blocked.

Syntax example:

```
accept_message(): bool is {
    return FALSE;
};
```

24.6.2.2 format_message

Purpose	Format a message
Category	Method
Syntax	<code>format_message():list of string</code>
Parameters	None
Return value	The message

This returns the *list of string*, which will be sent as-is to the file or screen.

Syntax example:

```
extend message_logger {
    format_message(): list of string is only{
        var msg_list := get_message();
        ...}
};
```

24.6.3 Query methods for getting message information

The methods shown in shown in this subclause are available for use within the **accept_message()** (see 24.6.2.1) and **format_message()** methods (see 24.6.2.2). Query methods return information about the message action that was just executed.

24.6.3.1 get_format

Purpose	Get a message's format
Category	Method
Syntax	get_format(): message_format
Parameters	None
Return value	The message's format-type

This returns the message format of the current logger.

Syntax example:

```

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;
    };
  };
};

```

24.6.3.2 get_message

Purpose	Get the current raw message
Category	Method
Syntax	get_message(): list of string
Parameters	None
Return value	The message

This returns the current raw message as produced by the message action.

Syntax example:

```

extend message_logger {
  format_message(): list of string is first {
    if get_format() == short then {
      result = get_message();

```

```

1         result[0] = dec(">> ", sys.time, " ", result[0]);
           return result;
           };
5     };
};

```

24.6.3.3 get_message_action_id

Purpose	Get the message ID
Category	Method
Syntax	<code>get_message_action_id():int</code>
Parameters	None
Return value	The message ID

This returns a unique number identifying the message action.

Syntax example:

```

25     extend message_logger {
           format_message(): list of string is first {
               if get_format() == long then {
                   result = get_message_action_id();
                   return result;
           };
30     };
};
};

```

24.6.3.4 get_tag

Purpose	Get a message's tag
Category	Method
Syntax	<code>get_tag():message_tag</code>
Parameters	None
Return value	A message-tag

This returns the tag of the message action.

Syntax example:

```

50     extend message_logger {
           format_message(): list of string is first {
               if get_format() == long then {
                   result = get_tag();
                   return result;
55     };
};
};

```

```

    };
  };
};

```

1

24.6.3.5 get_verbosity

5

Purpose	Get a message's verbosity	
Category	Method	10
Syntax	get_verbosity() :message_verbosity	
Parameters	None	15
Return value	The message's verbosity	

This returns the verbosity of the message action.

20

Syntax example:

```

extend message_logger {
  format_message(): list of string is first {
    if get_format() == none then {
      result = get_verbosity();
      return result;
    };
  };
};

```

25

30

24.6.3.6 source_location

Purpose	Get the message's location	
Category	Method	35
Syntax	source_location() :string	
Parameters	None	40
Return value	The source location	

This returns the source location where the message occurred, e.g., "At line 12 in @foo".

45

Syntax example:

```

var current_location := source_location();

```

50

55

24.6.3.7 source_method_name


Purpose	Get the method name for the current message
Category	Method
Syntax	<code>source_method_name():string</code>
Parameters	None
Return value	The method's name

This returns the name of the method where the message occurred, e.g., "foo".

Syntax example:

```
var current_method_name := source_method_name();
```

24.6.3.8 source_struct

Purpose	Get the message's source <i>struct</i> 
Category	Method
Syntax	<code>source_struct():any_struct</code>
Parameters	None
Return value	A <i>struct</i>


This returns the *struct* where the message occurred.

Syntax example:

```
var current_struct := source_struct();
```

24.6.3.9 source_struct_name

Purpose	Get a message's struct type-name
Category	Method
Syntax	<code>source_struct_name():string</code>
Parameters	None
Return value	The <i>struct</i> 's type-name

This returns the name of the struct type where the message occurred, e.g., "packet" .

Syntax example:

```
var current_struct_name := source_struct_name();
```

24.7 Examples

24.7.1 Example 1

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)
};
```

24.7.2 Example 2

Confirm the writing order, assuming 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 written in the order that message blocks finish:

1 I'm C0
 C1
 I'm B0
 B1
 5 B2
 I'm A0
 A1
 A2

10 With I'm A0 as the leader message, the order would be:

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

20 24.7.3 Example 3

Verify a logger handler, assuming the following code.

```

25 extend sys {
    u1: U1 is instance;
    u2: U2 is instance;
    run() is also {
        u2.foo();
    };
30 };

unit U1 {
    l1: message_logger is instance;
    keep l1.verbosity == FULL;
    a: A;
35 };

unit U2 {
    l2: message_logger is instance;
    keep l2.verbosity == FULL;
40

    foo() is {
        message(NONE, "foo: I'm U2") {
            out("foo: before");
            sys.u1.a.goo();
            out("foo: after");
45        };
    };
};

struct A {
    goo() is {
        message(NONE, "goo: I'm message from A");
50    };
};

```

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

```
goo: I'm message from A -- handled by sys.u1.11
foo: I'm U2              -- handled by sys.u2.12
foo: before              -- handled by sys.u2.12
foo: after               -- handled by sys.u2.12
```

1

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

5

```
foo: I'm U2              -- handled by sys.u2.12
foo: before              -- handled by sys.u2.12
goo: I'm message from A -- handled by sys.u2.12
foo: after               -- handled by sys.u2.12
```

10

15

20

25

30

35

40

45

50

55

1

5

10

15

20

25

30

35

40

45

50

55