

16. Print, checks, and error handling

The *e* language has many constructs that print an expression, check for errors in the DUT, or add exception handling and diagnostics to an *e* program.

16.1 print

Purpose	Print an expression
Category	Action
Syntax	print <i>exp</i> [,...] [using <i>print-options</i>]
Parameters	<i>exp</i> Valid <i>e</i> expression.
	<i>print-options</i> One or more of the print options, separated by commas (,). Print options are partly implementation-dependent. See also 28.8 .

This prints the value of the given expression. Each type of expression has a default format in which it is printed.

- Scalars and string expressions print as *expression* = *value-of-expression*.
- Structs print in small tables, with a row for each field of a struct.

The predefined **do_print()** method of a struct performs the printing. Changing the **do_print()** method modifies the default format for the struct.

- a) Lists print in small tables with a row for each element in the list, except for scalar lists 16 bits or smaller, which print horizontally.
- b) Integers print leading zeros (0's), except for unbounded integers, 32-bit integers, or decimal integer.

Syntax example:

```
print byte_list using radix = hex
```

16.2 Handling DUT errors

There are several constructs that can be used to perform data or protocol checks on the DUT or to handle any errors that occur.

16.2.1 check that

Purpose	Perform a data comparison, and depending on the results, print a message	
Category	Action	
Syntax	check [that] <i>bool-exp</i> [else dut_error(<i>message</i>: string-exp, ...)]	
Parameters	<i>bool-exp</i>	A Boolean expression that performs a data comparison.
	<i>message</i>	A string or an expression that can be converted to a string. If the <i>bool-exp</i> is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).

This performs a data comparison, and depending on the results, prints a message. Use **check that** to track the number of failed checks with predefined **session** fields. When the **else dut_error** clause is omitted, the *e* program uses the **check that** clause as the error message.

Syntax example:

```

check_count(i:int) is {
    check that i == expected_count else
        dut_error("Bad i: ", i)
}

```

16.2.2 dut_error()

Purpose	Issue a DUT error message	
Category	Action	
Syntax	dut_error (<i>message</i> : string-exp, ...) { <i>action</i> ; ...}	
Parameters	<i>message</i>	A string or an expression that can be converted to a string. The message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).
	<i>action</i>	A series of zero or more actions enclosed in braces ({}) and separated by semicolons (;).

This issues a DUT error message. This action is usually associated with an **if** action, a **check that** action, or an **expect** struct member. Calling **dut_error()** directly is exactly equivalent to:

```

check that FALSE else dut_error()

```

NOTE—When **dut_error()** is called directly or within an **expect**, **session.check_ok** shall always be FALSE.

Syntax example:

```

if 'data_out' != 'data_in' then {
    dut_error("DATA MISMATCH: Expected ", 'data_in')
}

```

16.2.3 dut_errorf()

Purpose	Issue a DUT error message	
Category	Action	
Syntax	dut_errorf (<i>format</i> : string, <i>item</i> : exp ...) { <i>action</i> ; ...}	
Parameters	<i>format</i>	A string expression containing a standard C formatting mask for each item (see 28.6.3)
	<i>item</i>	A legal e expression. String expressions shall be enclosed in double quotes (“”). If the expression is a struct instance, the struct ID is printed. If the expression is a list, an error shall be issued.
	<i>action</i>	A series of zero or more actions enclosed in braces ({ }) and separated by semicolons (;).

This issues a formatted DUT error message. This action is usually associated with an **if** action.

16.2.4 `dut_error_struct`

Purpose	Define DUT error response
Category	Predefined struct
Syntax	<pre> struct dut_error_struct { get_message() : string; source_struct() : any_struct; source_location() : string; source_struct_name() : string; source_method_name() : string; check_effect() : check_effect; set_check_effect(effect:check_effect); write(); pre_error() is empty } </pre>
Struct members	get_message() Returns the message defined by the temporal or data DUT check; this is printed by <code>dut_error_struct.write()</code> .
	source_struct() Returns a reference to the struct where the temporal or data DUT check is defined.
	source_location() Returns a string giving the line number and source module name, e.g., At line 13 in @checker.
	source_struct_name() Returns a string giving the name of the source struct, e.g., <code>packet</code> .
	source_method_name() Returns a string giving the name of the method containing the DUT data check, e.g., <code>done()</code> .
	check_effect() Returns the check effect of that DUT check, e.g., <code>ERROR_AUTOMATIC</code> .
	set_check_effect() Sets the check effect in this instance of the <code>dut_error_struct</code> . Call this method from <code>pre_error()</code> to change the check effect of selected checks.
	pre_error() The first method that is called when a DUT error occurs, unless the check effect is <code>IGNORE</code> . This method is defined as empty, unless extended by the user. Extending this method to modify error handling for a particular instance or set of instances of a DUT error.
write() The method that is called after <code>dut_error_struct.pre_error()</code> is called when a DUT error happens. This method causes the DUT message to be displayed, unless the check effect is <code>IGNORE</code> . To perform additional actions, extend this method.	

This defines the DUT error response. To modify the error response, extend either `write()` or `pre_error()`. Only the `write()` and `pre_error()` methods are called directly by *e* programs, but the other fields and predefined methods of `dut_error_struct` can also be used in extending `write()` or `pre_error()`.

When a `dut_error` is triggered, the runtime engine shall call `pre_error()`, unless the `check_effect` is set to `IGNORE`. Upon return of `pre_error()`, the `write()` method is called, causing a message to be printed. Both these methods can be customized.

The other `dut_error_struct` methods previously listed (as *Struct members*) can also be used to create more meaningful error messages or the response can be conditioned based upon the `check_effect`.

NOTE—Do not use `dut_error_struct.write()` to change the value of the check effect. Use `pre_error()` instead.

Syntax example:

```

extend dut_error_struct {
    write() is also {
        if source_struct() is a XYZ_packet (p) then {
            print p.parity_calc()
        }
    }
}

```

16.2.5 set_check()

Purpose	Set check severity
Category	Predefined routine
Syntax	set_check (<i>static-match</i> : string, <i>check-effect</i> : keyword)
Parameters	<p><i>static-match</i> A regular expression enclosed in double quotes (" "). Only checks whose message string matches this regular expression are modified. The pattern is matched against the constant part of the message string. The match string shall use the native <i>e</i> syntax or an AWK-like syntax (see 4.11).</p> <p>Any AWK-like syntax shall be enclosed in forward slashes, e.g., <code>/Vi0/</code>. Also, the <code>*</code> character in native <i>e</i> syntax matches only non-white characters. Use <code>. . .</code> to match white or non-white characters.</p>
	<p><i>check-effect</i> <i>check-effect</i> is one of the following:</p> <ol style="list-style-type: none"> ERROR—issues an error message, increases <code>num_of_dut_errors</code>, breaks the run immediately and returns to the simulator prompt. ERROR_BREAK_RUN—issues an error message, increases <code>num_of_dut_errors</code>, and breaks the run at the next cycle boundary. ERROR_AUTOMATIC—issues an error message, increases <code>num_of_dut_errors</code>, breaks the run at the next cycle boundary, and performs the end of test checking and finalization of test data that is normally performed when <code>stop_run()</code> is called. ERROR_CONTINUE—issues an error message, increases <code>num_of_dut_errors</code>, and continues execution. WARNING—issues a warning, increases <code>num_of_dut_warnings</code>, and continues execution. IGNORE—issues no messages, does not increase <code>num_of_dut_errors</code> or <code>num_of_dut_warnings</code>, and continues execution.

This sets the severity or the check effect of specific DUT checks, so failing checks produce errors or warnings. This routine affects only checks that are currently loaded.

Syntax example:

```

extend sys {
    setup() is also {
        set_check("...", WARNING)
    }
}

```

16.3 Handling user errors

The *e* language has several constructs for handling user errors, such as file I/O errors or semantic errors. This subclause describes the constructs used for handling these kinds of errors.

- **warning()** issues a warning message when a given error occurs.
- **error()** issues an error message and exits when a given error is detected.
- **fatal()** issues an error message and exits to the OS prompt when a given error is detected.
- **try** defines an alternative response for fixing or bypassing an error.

Errors handled by these constructs do not increase the **session.num_of_dut_errors** and **session.num_of_dut_warnings** fields that are used to track DUT errors. In addition, the error responses defined with these constructs are not influenced by modifications to **dut_error_struct** or by **set_check()** configurations.

See also the **run** option (28.8).

16.3.1 warning()

Purpose	Issue a warning message
Category	Action
Syntax	warning (<i>message</i> : string-exp, ...)
Parameters	<i>message</i> A string or an expression that can be converted to a string. When the warning action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

This issues the specified warning error message. It does not affect execution.

Syntax example:

```
warning("len exceeds 50")
```

16.3.2 error()

Purpose	Issue an error message and halt all method execution
Category	Action
Syntax	error (<i>message</i> : string-exp, ...)
Parameters	<i>message</i> A string or an expression that can be converted to a string. When the error action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

This issues the specified error message and halts all methods being currently run. The only exception to this is if the **error** action appears inside the first action block given in a **try** action. In that case, the *e* program jumps to the **else** action block within the **try** action and continues running. Calling **error()** directly is exactly equivalent to:

```
assert FALSE else error()
```

Unlike the **check that** action, **error()** does not use **dut_error_struct**.

Syntax example:

```
check_size() is {
    if pkt.size != LARGE then {
        error("packet size is ", pkt.size)
    }
}
```

16.3.3 fatal()

Purpose	Issue error message and exit to the OS prompt
Category	Action
Syntax	fatal (<i>message</i> : string, ...)
Parameters	<i>message</i> A string or an expression that can be converted to a string. When the fatal() action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

This issues the specified error message, halts all activity, exits immediately, and returns to the OS prompt. **fatal()** returns a non-zero status to the OS shell.

Syntax example:

```
fatal("Run-time error - exiting")
```

16.3.4 try

Purpose	Define an alternative response for fixing or bypassing an error
Category	Action
Syntax	try { <i>action</i> ; ...} [else { <i>action</i> ; ...}]
Parameters	<i>action</i> A series of zero or more actions enclosed in braces ({}) and separated by semicolons (;).

This executes the action block following **try**. If an error occurs, it executes the action block specified in the **else** clause. If no error occurs, the **else** clause is skipped. When the **else** clause is omitted, execution after an error continues normally from the first action following the **try** block.

Syntax example:

```
try {
    var my_file : file = files.open(file_name, "w", "Log file")
} else {
    warning("Could not open ", file_name,
        "; opening temporary log file sim.log")
}
```

16.4 Handling programming errors: assert

Purpose	Check the <i>e</i> code for correct behavior	
Category	Action	
Syntax	assert <i>bool-exp</i> [else error (<i>message</i> : string-exp, ...)]	
Parameters	<i>bool-exp</i>	A Boolean expression that checks the behavior of the code.
	<i>message</i>	A string or an expression that can be converted to a string. If the <i>bool-exp</i> is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).

The *e* language has a special construct, the **assert** action, for handling certain programming errors, such as internal contradictions or invalid parameters. It checks the *e* code for correct behavior. Use this action to catch coding errors.

When an assert fails, it prints the specified error message, plus the line number and name of the file in which the error occurred. If the **else error** clause is omitted, **assert** prints a global error message.

When an error is encountered, **assert** stops the method being executed.

Syntax example:

```
assert a < 20
```