

24. Predefined routines library

Predefined routines are *e* macros that look like methods. The distinguishing characteristics of *predefined routines* are:

- They are not associated with any particular struct
- They share the same name space for user-defined routines and **global** methods
- They cannot be modified or extended with the **is only**, **is also** or **is first** constructs
- They have no debug information ****Keep??**

See also: 15.2.

24.1 Deep copy and compare routines

The following routines perform recursive copies and comparisons of nested structs and lists. See also: 4.10.

24.1.1 deep_copy()

Purpose	Make a recursive copy of a struct and its descendants
Category	Predefined routine
Syntax	deep_copy (<i>struct-inst</i> : exp): struct instance
Parameters	<i>struct-inst</i> An expression that returns a struct instance.

This returns a deep, recursive copy of the struct instance. This routine descends recursively through the fields of a *struct* and its descendants, copying each field by value, copying it by reference, or ignoring it, depending on the **deep_copy** attribute set for that field.

The return type of **deep_copy()** is the same as the declared type of the struct instance.

Table 1 details how the copy is made, depending on the type of the field and the **deep_copy** attribute (**normal**, **reference**, **ignore**) set for that field. See also: 4.10.

Table 1—Copying procedure

Field type/ attribute	normal	reference	ignore
scalar	The new field holds a copy of the original value.	The new field holds a copy of the original value.	The new field holds a copy of the original value.
scalar list	A new list is allocated with the same size and same elements as the original list.	The new list field holds a copy of the original list pointer.*	A new list is allocated with zero size.

294 45

50

55

Table 1—Copying procedure (Continued)

Field type/ attribute	normal	reference	ignore
struct	A new struct instance with the same type as the original <i>struct</i> is allocated. Each field is copied or ignored, depending on its deep_copy attribute.	The new struct field holds a pointer to the original <i>struct</i> .*	No allocation occurs; the field is set to NULL.
list of structs	A new list is allocated with the same number of elements as the original list. New struct instances are also allocated and each field in each <i>struct</i> is copied or ignored, depending on its deep_copy attribute.	The new list field holds a copy of the original list pointer.*	A new list is allocated with zero size.

*If the list or *struct* that is pointed to is duplicated (possibly because another field with a normal attribute is also pointing to it), the pointer in this field is updated to point to the new instance. This duplication applies only to instances duplicated by the **deep_copy()** itself and not to duplications made by the extended/overridden **copy()** method.

The following considerations also apply.

- A deep copy of a scalar field (numeric, Boolean, or enumerated) or a string field is the same as a shallow copy performed by a call to **copy()**.
- A *struct* or list is duplicated no more than once during a single call to **deep_copy()**. If there is more than one reference to a *struct* or list instance and that instance is duplicated by the call to **deep_copy()**, every field that referred to the original instance is updated to point to the new instance.
- The **copy()** method of the *struct* is called by **deep_copy()**. The *struct*'s **copy()** method is called before its descendants are deep copied. If the default **copy()** method is overwritten or extended, this new version of the method is used.
- Add the **reference** attribute to fields that store shared data and to fields that are backpointers (pointers to the parent *struct*). *Shared data* in this context means data shared between objects inside the deep copy graph and objects outside the deep copy graph. A *deep copy graph* is the imaginary directed graph created by traversing the *structs* and lists duplicated, where its nodes are the *structs* or lists and its edges are deep references to other *structs* or lists.

Syntax example:

```
var pmv: packet = deep_copy(sys.pmi);
```

24.1.2 deep_compare()

Purpose	Perform a recursive comparison of two struct instances
Category	Predefined routine
Syntax	deep_compare (<i>struct-inst1</i> : exp, <i>struct-inst2</i> : exp, <i>max-diffs</i> : int): list of string
Parameters	<i>struct-inst1</i> , <i>struct-inst2</i> An expression returning a struct instance.
	<i>max-diffs</i> An integer representing the maximum number of differences to report.

This returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a *struct* and its descendants, comparing each field or ignoring it depending on the **deep_compare** attribute set for that field.

The two struct instances are “deep equal” if the returned list is empty.

Deep equal is defined as:

- Two struct instances are deep equal if they are of the same type and all their fields are deep equal.
- Two scalar fields are deep equal if an equality operation applied to them is TRUE.
- Two list instances are deep equal if they are of the same size and all their items are deep equal.

Topology is taken into account. If two non-scalar instances are not in the same location/order in the deep compare graphs, they are not equal. A deep compare graph is the imaginary directed graph created by traversing the *structs* and lists compared, where its nodes are the *structs* or lists and its edges are deep references to other *structs* or lists.

Table 2 details the differences that are reported, depending on the type of the field and the **deep_compare** attribute (**normal**, **reference**, **ignore**) set for that field. *See also*: 4.10.

Table 2—Reporting procedure

Field type/ attribute	normal	reference	ignore
scalar	Their values, if different, are reported.	Their values, if different, are reported.	The fields are not compared.
scalar list	Their sizes, if different, are reported. All items in the smaller list are compared to those in the longer list and their differences are reported.	The fields are equal if their addresses are the same. The items are not compared.	The fields are not compared.

296

297

Table 2—Reporting procedure (Continued)

Field type/ attribute	normal	reference	ignore
struct	If two <i>structs</i> are not of the same type, their type difference is reported. Also, any differences in common fields is reported.* † If two <i>structs</i> are of the same type, every field difference is reported.	The fields are equal if their addresses are the same. The items are not compared.	The fields are not compared and no differences for them or their descendants are reported.
list of structs	Their sizes, if different, are reported. All <i>structs</i> in the smaller list are deep compared to those in the longer list and their differences are reported.	The fields are equal if their addresses are the same and they point to the same struct instance. †	The fields are not compared and no differences for them or their descendants are reported.

*Two fields are considered common only if the two *structs* are the same type, if they are both subtypes of the same base type, or if one is a base type of the other.

†If the reference points inside the deep compare graph, a limited topological equivalence check is performed, not just an address comparison.

The difference string [reported](#) has the following format:

```
Differences between inst1-id and inst2-id
-----
path:    inst1-value  !=  inst2-value
```

where

path is a list of field names separated by periods (.), from (and not including) the struct instances being compared to the field with the difference.

value for scalar field differences, *value* is the result of `out(field)`.
for struct field type differences, `type()` is appended to the path and *value* is the result of `out(field.type())`.
for list field size differences, `size()` is appended to the path and *value* is the result of `out(field.size())`.
for a shallow comparison of struct fields that point outside the deep compare graph, *value* is the struct address.
for a comparison of struct fields that point to different locations in the deep compare graphs (topological difference), *value* is `struct#` appended to an index representing its location in the deep compare graph.

NOTE—The same two struct instances or the same two list instances are not compared more than once during a single call to `deep_compare()`.

Syntax example:

```
var diff: list of string = deep_compare(pmi[0], pmi[1], 100);
```

24.1.3 deep_compare_physical()

Purpose	Perform a recursive comparison of the physical fields of two struct instances
Category	Predefined routine
Syntax	deep_compare_physical (<i>struct-inst1</i> : exp, <i>struct-inst2</i> : exp, <i>max-diffs</i> : int): list of string
Parameters	<i>struct-inst1</i> , <i>struct-inst2</i> An expression returning a struct instance.
	<i>max-diffs</i> An integer representing the maximum number of differences to report.

Syntax example:

```
var diff: list of string = deep_compare_physical(pmi[0],
    pmi[1], 100);
```

This returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a struct and its descendants, ignoring all non-physical fields and comparing each physical field or ignoring it, depending on the **deep_compare_physical** attribute set for that field.

This routine is the same as the **deep_compare()** routine (see 24.1.2), except only physical fields (indicated by the % operator prefixed to the field name) are compared.

NOTE—Adding a field under a **when** construct only causes the parent type and the **when** subtype to be different if the added field is a physical field.

300

24.2 Arithmetic routines

The following sections describe the predefined arithmetic routines in *e*.

24.2.1 min()

Purpose	Get the minimum of two numeric values
Category	Pseudo routine
Syntax	min (<i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type
Parameters	<i>x</i> A numeric expression.
	<i>y</i> A numeric expression.

301

This returns the smaller of the two numeric values.

Syntax example:

```
print min((x + 5), y);
```

24.2.2 max()

Purpose	Get the maximum of two numeric values	
Category	Pseudo routine	
Syntax	max (<i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type	
Parameters	<i>x</i>	A numeric expression.
	<i>y</i>	A numeric expression.

This returns the larger of the two numeric values.

Syntax example:

```
print max((x + 5), y);
```

24.2.3 abs()

Purpose	Get the absolute value	
Category	Pseudo routine	
Syntax	abs (<i>x</i> : numeric-type): numeric-type	
Parameters	<i>x</i>	A numeric expression.

This returns the absolute value of the expression.

Syntax example:

```
print abs(x);
```

24.2.4 odd()

Purpose	Check if an integer is odd	
Category	Pseudo routine	
Syntax	odd (<i>x</i> : numeric-type): bool	
Parameters	<i>x</i>	A numeric expression.

This returns TRUE if the expression is odd, FALSE if the expression is even.

Syntax example:

```
print odd(x);
```

24.2.5 even()

1

Purpose	Check if an integer is even
Category	Pseudo routine
Syntax	even (<i>x</i> : numeric-type): bool
Parameters	<i>x</i> A numeric expression.

305

5

306

10

This returns TRUE if the expression passed to it is even, FALSE if the expression is odd.

Syntax example:

15

```
print even(x);
```

24.2.6 ilog2()

20

Purpose	Get the base-2 logarithm
Category	Pseudo routine
Syntax	ilog2 (<i>x</i> : numeric-type): bool
Parameters	<i>x</i> A numeric expression.

308

25

This returns the integer part of the base-2 logarithm of *x*.

30

Syntax example:

```
print ilog2(x);
```

35

24.2.7 ilog10()

Purpose	Get the base-10 logarithm
Category	Pseudo routine
Syntax	ilog10 (<i>x</i> : numeric-type): bool
Parameters	<i>x</i> A numeric expression.

309

40

45

This returns the integer part of the base-10 logarithm of *x*.

Syntax example:

50

```
print ilog10(x);
```

55

24.2.8 ipow()

Purpose	Raise to a power
Category	Pseudo routine
Syntax	ipow (<i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type
Parameters	<i>x</i> A numeric expression.
	<i>y</i> A numeric expression.

This raises *x* to the power of *y* and returns the result.

Syntax example:

```
print ipow(x, y);
```

24.2.9 isqrt()

Purpose	Get the square root
Category	Pseudo routine
Syntax	isqrt (<i>x</i> : numeric-type): int
Parameters	<i>x</i> A numeric expression.

This returns the integer part of the square root of *x*.

Syntax example:

```
print isqrt(x);
```

24.2.10 div_round_up()

Purpose	Division rounded up
Category	Routine
Syntax	div_round_up (<i>x</i> : int, <i>y</i> : int): int
Parameters	<i>x</i> A numeric expression.
	<i>y</i> A numeric expression.

This returns the result of *x* / *y* rounded up to the next integer. *See also*: 2.8.2.

Syntax example:

```
print div_round_up(x, y);
```

24.3 bitwise_op()

Purpose	Perform a Verilog-style unary reduction operation
Category	Pseudo routine
Syntax	bitwise_op (<i>exp</i> : numeric-type): bit
Parameters	<i>op</i> One of and , or , xor , nand , nor , xnor .
	<i>exp</i> A numeric expression.

This performs a Verilog-style unary reduction operation on a single operand to produce a single bit result. There is no reduction operator in *e*, but the **bitwise_op()** routines perform the same functions as reduction operators in Verilog, e.g, **bitwise_xor()** can be used to calculate parity.

For **bitwise_nand()**, **bitwise_nor()**, and **bitwise_xnor()**, the result is computed by inverting the result of the **bitwise_and()**, **bitwise_or()**, and **bitwise_xor()** operations, respectively. Table 3 shows the predefined pseudo-methods for bitwise operations.

Table 3—Bitwise operation pseudo-methods

Pseudo-method	Operation
bitwise_and()	Boolean AND of all bits
bitwise_or()	Boolean OR of all bits
bitwise_xor()	Boolean XOR of all bits
bitwise_nand()	!bitwise_and()
bitwise_nor()	!bitwise_or()
bitwise_xnor()	!bitwise_xor()

Syntax example:

```
print bitwise_and(b);
```

325

24.4 get_all_units()

Purpose	Return a list of instances of a specified unit type
Category	Pseudo routine
Syntax	get_all_units (<i>unit-type</i> : exp): list of unit-type
Parameters	<i>unit-type</i> The name of a unit type, unquoted. The type needs to be defined or an error shall occur.

326

This routine receives a unit type as a parameter and returns a list of instances of this unit type, as well as any unit instances whose type is contained in the specified unit-type (i.e., ****its type "is a"??**).

Syntax example:

```
print get_all_units(XYZ_channel);
```

24.5 String routines

None of the string routines in e modify the input parameters. When a parameter is passed to one of these routines, the routine makes a copy of the parameter, manipulates the copy, and returns the copy. *See also:* 2.10, 3.1.10, and Table 5.

327

24.5.1 append()

Purpose	Concatenate expressions into a string
Category	Pseudo routine
Syntax	append() : string append (<i>item</i> : exp, ...): string
Parameters	<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 no items are passed to append() , it returns an empty string.

328

This calls **to_string()** (see 24.5.21) to convert each expression to a string using the current radix setting for any numeric expressions, then it concatenates them and returns the result as a single string.

Syntax example:

```
message = append(list1, " ", list2);
```

24.5.2 appendf()

Purpose	Concatenate expressions into a string according to a given format	
Category	Pseudo routine	
Syntax	appendf (<i>format</i> : string, <i>item</i> : exp, ...): string	
Parameters	<i>format</i>	A string expression containing a standard C formatting mask for each <i>item</i> (see 24.6.3).
	<i>item</i>	A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (""). If the expression is a struct instance, the struct ID is printed.

This converts each expression to a string using the current radix setting for any numeric expressions and the specified format, then it concatenates them and returns the result as a single string. If the number and type of masks in the format string does not match the number and type of expressions, an error shall be issued.

Syntax example:

```
message = appendf("%4d\n %4d\n %4d\n", 255, 54, 1570);
```

24.5.3 bin()

Purpose	Concatenate expressions into string, using binary representation for numeric types	
Category	Pseudo routine	
Syntax	bin (<i>item</i> : exp, ...): string	
Parameters	<i>item</i>	A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using binary representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using **to_string()** (see 24.5.21).

Syntax example:

```
var my_string: string = bin(pi.i, " ", list1, " ", 8);
```

24.5.4 dec()

Purpose	Concatenate expressions into string, using decimal representation for numeric types
Category	Pseudo routine
Syntax	dec (<i>item</i> : exp, ...): string
Parameters	<i>item</i> A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using decimal representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using `to_string()` (see 24.5.21).

Syntax example:

```
var my_string: string = dec(pi.i, " ", list1, " ",8);
```

24.5.5 hex()

Purpose	Concatenate expressions into string, using hexadecimal representation for numeric types
Category	Pseudo routine
Syntax	hex (<i>item</i> : exp, ...): string
Parameters	<i>item</i> A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using hexadecimal representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using `to_string()` (see 24.5.21).

Syntax example:

```
var my_string: string = hex(pi.i, " ", list1, " ",8);
```

24.5.6 quote()

Purpose	Enclose a string in double quotes
Category	Routine
Syntax	quote (<i>text</i> : string): string
Parameters	<i>text</i> An expression of type string .

This returns a copy of the text, enclosed in double quotes (""), with any internal quote or backslash preceded by a backslash (\).

Syntax example:

```
out(quote(message));
```

1

24.5.7 str_chop()

Purpose	Chop the tail of a string	5
Category	Routine	
Syntax	str_chop (<i>str</i> : string, <i>length</i> : int): string	10
Parameters	<i>str</i> An expression of type string .	
	<i>length</i> An integer representing the desired length.	15

This removes characters from the end of a string, returning a string of the desired length. If the original string is already less than or equal to the desired length, this routine returns the original string.

334

Syntax example:

20

```
var test_dir: string = str_chop(tmp_dir, 13);
```

24.5.8 str_empty()

25

Purpose	Check if a string is empty	
Category	Routine	
Syntax	str_empty (<i>str</i> : string): bool	30
Parameters	<i>str</i> An expression of type string .	

This returns `TRUE` if the string is empty.

335

35

Syntax example:

```
print str_empty(s1);
```

40

45

50

55

24.5.9 str_exactly()

Purpose	Get a string with exact length
Category	Routine
Syntax	str_exactly (<i>str</i> : string, <i>length</i> : int): string
Parameters	<i>str</i> An expression of type string .
	<i>length</i> An integer representing the desired length.

This returns a copy of the original string, whose length is the desired length, by adding blanks to the right or by truncating the expression from the right as necessary. If non-blank characters are truncated, the * character appears as the last character in the string returned.

336

Syntax example:

```
var long: string = str_exactly("123", 6);
```

24.5.10 str_insensitive()

Purpose	Get a case-insensitive AWK-style regular-expression
Category	Routine
Syntax	str_insensitive (<i>regular_exp</i> : string): string
Parameters	<i>regular_exp</i> An AWK-style regular expression.

This returns an AWK-style regular expression string which is the case-insensitive version of the original regular expression. *See also*: 2.10.2.

Syntax example:

```
var insensitive: string = str_insensitive("/hello.*");
```

24.5.11 str_join()

Purpose	Concatenate a list of strings
Category	Routine
Syntax	str_join (<i>list</i> : list of string, <i>separator</i> : string): string
Parameters	<i>list</i> An list of type string .
	<i>separator</i> The string used to separate the list elements.

This returns a single string which is the concatenation of the strings in the list of strings, separated by the separator.

Syntax example:

```
var s := str_join(slist, " - ");
```

24.5.12 str_len()

Purpose	Get string length
Category	Routine
Syntax	str_len (<i>str</i> : string): int
Parameters	<i>str</i> An expression of type string .

This returns the number of characters in the original string, not counting the terminating NULL character \0.

Syntax example:

```
var length: int = str_len("hello");
```

24.5.13 str_lower()

Purpose	Convert string to lowercase
Category	Routine
Syntax	str_lower (<i>str</i> : string): string
Parameters	<i>str</i> An expression of type string .

This converts all upper case characters in the original string to lower case and returns the string.

Syntax example:

```
var lower: string = str_lower("UPPER");
```

24.5.14 str_match()

Purpose	Match strings
Category	Routine
Syntax	str_match (<i>str</i> : string, <i>regular-exp</i> : string): bool
Parameters	<i>str</i> An expression of type string .
	<i>regular-exp</i> An AWK-style or native <i>e</i> regular expression. If not surrounded by slashes (/), the expression is treated as a native style expression (see 2.10).

This returns TRUE if the strings match or FALSE if the strings do not match. The routine **str_match()** is fully equivalent to the operator ~. After doing a match, the local pseudo-variables \$1, \$2, ..., \$27 can be

used, which correspond to the parenthesized pieces of the match. \$0 stores the entire matched piece of the string. *See also:* 2.9.4.

Syntax example:

```
print str_match("ace", "/c(e)?$/");
```

24.5.15 str_pad()

Purpose	Pad string with blanks
Category	Routine
Syntax	str_pad (<i>str</i> : string, <i>length</i> : int): string
Parameters	<i>str</i> An expression of type string .
	<i>length</i> An integer representing the desired length.

This returns a copy of the original string padded with blanks on the right, up to desired length. If the length of the original string is greater than or equal to the desired length, then the original string (not a copy) is returned with no padding.

Syntax example:

```
var s: string = str_pad("hello world",14);
```

24.5.16 str_replace()

Purpose	Replace a substring in a string with another string
Category	Routine
Syntax	str_replace (<i>str</i> : string, <i>regular-exp</i> : string, <i>replacement</i> : string): string
Parameters	<i>str</i> An expression of type string .
	<i>regular-exp</i> An AWK-style or native e regular expression. If not surrounded by slashes (/), the expression is treated as a native style expression (see 2.10).
	<i>replacement</i> The string used to replace all occurrences of the regular expression.

A new copy of the original string is created and then all the matches of the regular expression are replaced by the replacement string. If no match is found, a copy of the source string is returned.

- To incorporate the matched substrings in the *replacement* string, use the back-slash escaped numbers: \1, \2, . . .
- In native e regular expressions, the portion of the original string that matches the * or the . . . characters is replaced by the replacement string.
- In AWK-style regular expressions, to replace portions of the regular expressions, mark them with parentheses ().

Syntax example:

```
var s: string = str_replace("crc32", "/(. *32)/", "32_flip");
```

1

24.5.17 str_split()

Purpose	Split a string to substrings		5
Category	Routine		
Syntax	str_split (<i>str</i> : string, <i>regular-exp</i> : string): list of string		10
Parameters	<i>str</i>	An expression of type string .	
	<i>regular-exp</i>	An AWK-style or native e regular expression that specifies where to split the string (see 2.10).	15

This splits the original string on each occurrence of the regular expression and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned as the first or last item, respectively.

339

20

If the regular expression is an empty string, it has the effect of removing all blanks in the original string and the splitting is done on blanks.

Syntax example:

25

```
var s: list of string = str_split("first-second-third", "-");
```

24.5.18 str_split_all()

Purpose	Split a string to substrings, including separators		30
Category	Routine		
Syntax	str_split_all (<i>str</i> : string, <i>regular-exp</i> : string): list of string		35
Parameters	<i>str</i>	An expression of type string .	
	<i>regular-exp</i>	An AWK-style or native e regular expression that specifies where to split the string (see 2.10).	40

This splits the original string on each occurrence of the regular expression and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned as the first or last item, respectively.

340

45

This routine is similar to **str_split()**, except it includes the separators in the resulting list of strings.

Syntax example:

```
var s: list of string = str_split_all(" A B C", "/ +/");
```

50

55

24.5.19 str_sub()

Purpose	Extract a substring from a string
Category	Routine
Syntax	str_sub (<i>str</i> : string, <i>from</i> : int, <i>length</i> : int): string
Parameters	<i>str</i> An expression of type string .
	<i>from</i> The index position from which to start extracting. The first character in the string is at index 0.
	<i>length</i> An integer representing the number of characters to extract.

341

This returns a substring of the specified length from the original string, starting from the specified index position. *from* shall be between 0 and *length*+1 of *str*. If *str* is shorter than *from* + *length*, only the available part is returned.

Syntax example:

```
var dir: string = str_sub("/rtests/test32/tmp", 8, 6);
```

24.5.20 str_upper()

Purpose	Convert a string to uppercase
Category	Routine
Syntax	str_upper (<i>str</i> : string): string
Parameters	<i>str</i> An expression of type string .

This returns a copy of the original string, converting all lower case characters to upper case characters.

Syntax example:

```
var upper: string = str_upper("lower");
```

24.5.21 to_string()

Purpose	Convert any expression to a string
Category	Pseudo-method
Syntax	<i>exp.to_string</i> (): string
Parameters	<i>exp</i> A legal <i>e</i> expression.

****Move this to section 23.3??**

342

This method can be used to convert any type to a string; it can be extended for *structs* and units. 1

- If the expression is a *struct* expression, the **to_string()** method returns a unique identification string for each struct instance., which can be used to reference the *struct*. By default, the identification string is of the form *type-@ num*, where *num* is a unique struct number over all instances of all *structs* in the current run. 5
- If the expression is a list of strings, the **to_string()** method is called for each element in the list. The string returned contains all the elements, with a newline between each element. 10
- If the expression is a list of any type except **string**, the **to_string()** method returns a string containing all the elements, with a space between each element. 10
- If the expression is a numeric type, the expression is converted using the current radix with the radix prefix. 15
- If the expression is a string, the **to_string()** method returns the string. 15
- If the expression is an enumerated or a Boolean type, the **to_string()** method returns the value. 15

Syntax example:

```
print pkts[0].to_string();
```

24.6 Output routines

The predefined output routines print formatted and unformatted information to the screen and to open log files. 25

24.6.1 out()

Purpose	Print expressions to output, with a new line at the end	343 30
Category	Pseudo routine	
Syntax	out() out(item: exp, ...)	
Parameters	<i>item</i> A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (""). If the expression is a struct instance, the struct ID is printed. If no items are passed to out() , an empty string is printed, followed by a new line. 35	

This calls **to_string()** (see 24.5.21) to convert each expression to a string and prints them to the screen (and to the log file if it is open), followed by a new line. 40

Syntax example:

```
out("pkts[1].data is                      ", pkts[1].data);
```

24.6.2 `outf()`

Purpose	Print formatted expressions to output, with no new line at the end
Category	Pseudo routine
Syntax	<code>outf(format: string, item: exp, ...)</code>
Parameters	<i>format</i> A string expression containing a standard C formatting mask for each <i>item</i> (see 24.6.3).
	<i>item</i> A legal <i>e</i> 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.

This converts each expression to a string using the corresponding format string and then prints them to the screen (and to the log file if it is open). For the `%s` mask, `to_string()` (see 24.5.21) is used for creating the string representation of the expression.

- To add a new line, add the `\n` characters to the format string.
- `outf()` can be used to add the new lines where needed.
- Printing of lists is not supported with `outf()`.
- If the number and type of masks in the format string does not match the number and type of expressions, an error shall be issued.

Syntax example:

```
outf("%s %#08x", "pkts[1].data[0] is      ", pkts[1].data[0]);
```

24.6.3 Format string

The format string for the `outf()` and for the `appendf()` routine uses the following syntax:

```
"%[0|-][#][min_width][.[max_chars]](s|d|x|b|o|u)"
```

where:

0	pads with 0 instead of blanks. Padding is only done when right alignment is used, on the left end of the expression.
-	aligns left. The default is to align right.
#	adds 0x before the number. Can be used only with the x (hexadecimal) format specifier, e.g., <code>%#x</code> or <code>%#010x</code> .
<code>min_width</code>	is a number that specifies the minimum number of characters. This number determines the minimum width of the field. If there are not enough characters in the expression to fill the field, the expression is padded to make it this many characters wide. If there are more characters in the expression than this number (and if <code>max_chars</code> is set large enough), this number is ignored and enough space is used to accommodate the entire expression.
<code>max_chars</code>	is a number that specifies the maximum number of characters to use from the expression. Characters in excess of this number are truncated. If this number is larger than <code>min_width</code> , then the <code>min_width</code> number is ignored.

- s converts the expression to a string. The routine **to_string()** (see 24.5.21) is used to convert a non-string expression to a string. 1
- d prints a numeric expression in decimal format. 5
- x prints a numeric expression in hex format. With the optional # character, adds 0x before the number. 10
- b prints a numeric expression in binary format. 10
- o prints a numeric expression in octal format. 10
- u prints integers (**int** and **uint**) in **uint** format. 10

24.7 OS interface routines 345

The routines in this section enable use of operating system commands from within the *e* programming language. These routines work on all supported operating systems. 15

24.7.1 spawn() 20

Purpose	Send commands to the operating system	
Category	Pseudo routine	25
Syntax	spawn() spawn(command: string, ...)	
Parameters	<i>command</i> An expression of type string .	346 30

This takes a variable number of parameters, concatenates them together, and executes the string result as an operating system command via **system()** (see 24.7.3). 35

Syntax example:

```
spawn("touch error.log;", "grep Error my.elog > error.log");
```

24.7.2 spawn_check() 40

Purpose	Send a command to the operating system and report error	
Category	Routine	45
Syntax	spawn_check(command: string)	
Parameters	<i>command</i> An expression of type string .	347 50

This executes a single string as an operating system command via **system()** (see 24.7.3), then calls **error()** (see 14.3.2) if the execution of the command returned an error status. 55

Syntax example:

```
spawn_check("grep Error my.elog >& error.log");
```

24.7.3 system()

Purpose	Send a command to the operating system
Category	Routine
Syntax	system (<i>command</i> : string): int
Parameters	<i>command</i> An expression of type string .

This executes the string as an operating system command using the C **system()** (**where is this ref??) call and returns the result.

Syntax example:

```
stub = system("cat my.v");
```

24.7.4 output_from()

Purpose	Collect the results of a system call
Category	Routine
Syntax	output_from (<i>command</i> : string): list of string
Parameters	<i>command</i> An expression of type string .

This executes the string as an operating system command and returns the output as a list of string. Under UNIX, **stdout** and **stderr** go to the string list.

Syntax example:

```
log_list = output_from("ls *log");
```

24.7.5 output_from_check()

Purpose	Collect the results of a system call and check for errors
Category	Routine
Syntax	output_from_check (<i>command</i> : string): list of string
Parameters	<i>command</i> An expression of type string .

This executes the string as an operating system command, returns the output as a list of string, and then calls **error()** (see 14.3.2) if the execution of the command returns an error status. Under UNIX, **stdout** and **stderr** go to the string list.

Syntax example:

```
log_list = output_from_check("ls *.log");
```

1

24.7.6 get_symbol()

5

Purpose	Get UNIX environment variable
Category	Routine
Syntax	get_symbol (<i>env-variable</i> : string): string
Parameters	<i>env-variable</i> An expression of type string .

10

351
15

This returns the environment variable as a string or an empty string if the symbol is not found.

Syntax example:

```
current_dir = get_symbol("PWD");
```

20

24.7.7 date_time()

Purpose	Retrieve current date and time
Category	Routine
Syntax	date_time (): string

25

30

This returns the current date and time as a string.

Syntax example:

```
print date_time();
```

35

24.7.8 getpid()

Purpose	Retrieve process ID
Category	Routine
Syntax	getpid (): int

40

45

This returns the current process ID as an integer.

352
353

Syntax example:

```
print getpid();
```

50

55