28. Predefined routines library

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

— 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.

See also 17.2.

28.1 Deep copy and compare routines

The following routines perform recursive copies and comparisons of nested structs and lists. See also 6.11.

28.1.1 deep_copy()

Purpose	Make a recursive copy of a struct and its descendants
Category	Predefined routine
Syntax	**deep_copy(***struct-inst*: exp**)**: struct instance
Parameters	*struct-inst* 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 41 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 6.11.

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 back-pointers (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.

Table 41—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]	A new list is allocated with zero size.
struct	A new struct instance with the same type as the original struct is allocated. Each field is copied or ignored, depending on its **deep_copy** attribute.	The new struct field holds a pointer to the original struct.	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 struct is copied or ignored, depending on its **deep_copy** attribute.	The new list field holds a copy of the original list pointer. [a]	A new list is allocated with zero size.

[a]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.

Syntax example:

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

28.1.2 deep_compare()

Purpose	Perform a recursive comparison of two struct instances
Category	Predefined routine
Syntax	**deep_compare(***struct-inst1*: exp**,** *struct-inst2*: exp**,** *max-diffs*: int**)**: list of string
Parameters	*struct-inst1*, *struct-inst2* — An expression returning a struct instance.
	max-diffs — 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 follows:

— 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 42 details the differences that are reported, depending on the type of the field and the **deep_compare** attribute (**normal**, **reference**, or **ignore**) set for that field. See also 6.11.

Table 42—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.
struct	If two structs are not of the same type, their type difference is reported. Also, any differences in common fields are reported. [a, b] If two structs 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 structs 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. [b]	The fields are not compared and no differences for them or their descendants are reported.

[a]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.
[b]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

a) for scalar field differences, *value* is the result of **out(***field***).**

b) for struct field type differences, the type of the field is appended to the path and *value* is the type of the field.

c) for list field size differences, **size()** is appended to the path and *value* is the result of **out(***field***.size()).**

d) for a shallow comparison of struct fields that point outside the deep compare graph, *value* is the struct address.

e) 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)
```

28.1.3 deep_compare_physical()

Purpose	Perform a recursive comparison of the physical fields of two struct instances	
Category	Predefined routine	
Syntax	**deep_compare_physical(***struct-inst1*: exp**,** *struct-inst2*: exp**,** *max-diffs*: int**)**: list of string	
Parameters	*struct-inst1*, *struct-inst2*	An expression returning a struct instance.
	max-diffs	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 28.1.2), 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.

28.2 Integer arithmetic routines

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

28.2.1 min()

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

This returns the smaller of the two numeric values.

Syntax example:

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

28.2.2 max()

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

This returns the larger of the two numeric values.

Syntax example:

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

28.2.3 abs()

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

This returns the absolute value of the expression.

Syntax example:

```
print abs(x)
```

28.2.4 odd()

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

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

Syntax example:

```
print odd(x)
```

28.2.5 even()

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

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

Syntax example:

```
print even(x)
```

28.2.6 ilog2()

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

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

Syntax example:

```
print ilog2(x)
```

28.2.7 ilog10()

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

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

Syntax example:

```
print ilog10(x)
```

28.2.8 ipow()

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

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

Syntax example:

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

28.2.9 isqrt()

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

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

Syntax example:

```
print isqrt(x)
```

28.2.10 div_round_up()

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

This returns the result of *x* / *y* rounded up to the next integer. See also 4.9.2.

Syntax example:

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

28.3 Real arithmetic routines

Table 43 shows the arithmetic routines support of **real** type objects:

Table 43—Arithmetic routines supporting real types

Routine	Description
floor(real): real	Returns the largest integer that is less than or equal to the parameter.
ceil(real): real	Returns the smallest integer that is greater than or equal to the parameter.
round(real): real	Returns the closest integer to the parameter. In the case of a tie then it returns the integer with the higher absolute value.
log(real): real	Returns the natural logarithm of the parameter.
log10(real): real	Returns the base-10 logarithm of parameter.
pow(real, real): real	Returns the value of the first parameter raised to the power of second one.
sqrt(real): real	Returns the square root of the parameter.
exp(real): real	Returns the value of ***e*** raised to the power of the parameter.
sin(real): real	Returns the sine of the parameter given in radians.
cos(real): real	Returns the cosine of the parameter given in radians.
tan(real): real	Returns the tangent of the parameter given in radians.
asin(real): real	Returns the arc sine of the parameter.
acos(real): real	Returns the arc cosine of the parameter.
atan(real): real	Returns the arc tangent of the parameter.
sinh(real): real	Returns the hyperbolic sine of the parameter.
cosh(real): real	Returns the hyperbolic cosine of the parameter.
tanh(real): real	Returns the hyperbolic tangent of the parameter.

Table 43—Arithmetic routines supporting real types *(continued)*

Routine	Description
asinh(real): real	Returns the inverse hyperbolic sine of the parameter.
acosh(real): real	Returns the inverse hyperbolic cosine of the parameter.
atanh(real): real	Returns the inverse hyperbolic tangent of the parameter.
atan2(real, real): real	Returns the arc tangent of the two parameters.
hypot(real, real): real	Returns the distance of the point defined by the two parameters from the origin.
is_nan(real): bool	Returns TRUE if the parameter's value is Not-a-Number (NaN).
is_finite(real): bool	Returns TRUE if the parameter's value is a finite real value (that is, it is not infinity, negative infinity, or NaN).
NOTE—For integer routines like **ilog()**, **ilog10()**, **ilog2()**, **ipow()**, and **isqrt()**, whose return type is based on the expected type, if the expected type is **real**, then the return type is **int** (bits:*).	

28.4 bitwise_op()

Purpose	Perform a Verilog-style unary reduction operation	
Category	Pseudo-routine	
Syntax	**bitwise_***op***(***exp*: numeric-type**)**: bit	
Parameters	*op*	One of **and**, **or**, **xor**, **nand**, **nor**, or **xnor**.
	exp	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 44 shows the predefined pseudo-methods for bitwise operations.

Table 44—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)
```

28.5 get_all_units()

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

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.

Syntax example:

```
print get_all_units(XYZ_channel)
```

28.6 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 4.11, 5.1.10, and Table 23.

28.6.1 append()

Purpose	Concatenate expressions into a string	
Category	Pseudo-routine	
Syntax	**append()**: string **append(***item*: exp**, ...)**: string	
Parameters	*item*	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.

This calls **to_string()** (see 27.4.4) 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)
```

28.6.2 appendf()

Purpose	Concatenate expressions into a string according to a given format	
Category	Pseudo-routine	
Syntax	**appendf(***format*: string**,** *item*: exp**,** ...**)**: string	
Parameters	*format*	A string expression containing a standard C formatting mask for each *item* (see 28.7.3).
	item	A legal ***e*** 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. An expression can match either a string format (`%s`) or a numeric format. If it matches a string format, the current radix is used. If it matches a numeric format, that numeric format defines the conversion to a string (see 28.7.3). Once all the expressions are converted, they are concatenated and returned 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)
```

28.6.3 bin()

Purpose	Concatenate expressions into string, using binary representation for numeric types	
Category	Pseudo-routine	
Syntax	**bin(***item*: exp**,** ...**)**: string	
Parameters	*item*	A legal ***e*** 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 27.4.4).

Syntax example:

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

28.6.4 dec()

Purpose	Concatenate expressions into string, using decimal representation for numeric types
Category	Pseudo-routine
Syntax	**dec(***item*: exp**, ...)**: string
Parameters	*item* A legal ***e*** 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 27.4.4).

Syntax example:

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

28.6.5 hex()

Purpose	Concatenate expressions into string, using hexadecimal representation for numeric types
Category	Pseudo-routine
Syntax	**hex(***item*: exp**, ...)**: string
Parameters	*item* A legal ***e*** 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 27.4.4).

Syntax example:

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

28.6.6 quote()

Purpose	Enclose a string in double quotes
Category	Routine
Syntax	**quote(***text*: string**)**: string
Parameters	*text* 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))
```

28.6.7 str_chop()

Purpose	Chop the tail of a string	
Category	Routine	
Syntax	**str_chop(***str*: string**,** *length*: int**)**: string	
Parameters	*str*	An expression of type **string**.
	length	An integer representing the desired length.

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.

Syntax example:

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

28.6.8 str_empty()

Purpose	Check if a string is empty	
Category	Routine	
Syntax	**str_empty(***str*: string**)**: bool	
Parameters	*str*	An expression of type **string**.

This returns `TRUE` if the string is empty.

Syntax example:

```
print str_empty(s1)
```

28.6.9 str_exactly()

Purpose	Get a string with exact length	
Category	Routine	
Syntax	**str_exactly(***str*: string**,** *length*: int**)**: string	
Parameters	*str*	An expression of type **string**.
	length	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.

Syntax example:

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

28.6.10 str_insensitive()

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

This returns an AWK-style regular expression string that is the case-insensitive version of the original regular expression. See also 4.11.2.

Syntax example:

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

28.6.11 str_join()

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

This returns a single string that is the concatenation of the strings in the list of strings, separated by the separator. The strings in the list are not changed.

Syntax example:

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

28.6.12 str_len()

Purpose	Get string length
Category	Routine
Syntax	**str_len(***str*: string**)**: int
Parameters	*str* 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")
```

28.6.13 str_lower()

Purpose	Convert string to lowercase
Category	Routine
Syntax	**str_lower(***str*: string**)**: string
Parameters	*str* — An expression of type **string**.

This returns a copy of the string with all uppercase characters converted to lowercase.

Syntax example:

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

28.6.14 str_match()

Purpose	Match strings
Category	Routine
Syntax	**str_match(***str*: string**,** *regular-exp*: string**)**: bool
Parameters	*str* — An expression of type **string**.
	regular-exp — An AWK-style or native ***e*** regular expression. If not surrounded by slashes (`/`), the expression is treated as a native style expression (see 4.11).

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 4.10.4.

Syntax example:

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

28.6.15 str_pad()

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

This returns a copy of the original string padded with blanks on the right, up to the 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)
```

28.6.16 str_replace()

Purpose	Replace a substring in a string with another string	
Category	Routine	
Syntax	**str_replace(***str*: string**,** *regular-exp*: string**,** *replacement*: string**)**: string	
Parameters	*str*	An expression of type **string**.
	regular-exp	An AWK-style or native ***e*** regular expression. If not surrounded by slashes (`/`), the expression is treated as a native style expression (see 4.11).
	replacement	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 backslash 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")
```

28.6.17 str_split()

Purpose	Split a string to substrings	
Category	Routine	
Syntax	**str_split(***str*: string**,** *regular-exp*: string**)**: list of string	
Parameters	*str*	An expression of type **string**.
	regular-exp	An AWK-style or native ***e*** regular expression that specifies where to split the string (see 4.11).

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. 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.

The original string is not changed by this operation.

Syntax example:

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

28.6.18 str_split_all()

Purpose	Split a string to substrings, including separators	
Category	Routine	
Syntax	**str_split_all(***str*: string**,** *regular-exp*: string**)**: list of string	
Parameters	*str*	An expression of type **string**.
	regular-exp	An AWK-style or native ***e*** regular expression that specifies where to split the string (see 4.11).

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. The original string is not changed by this operation.

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", "/ +/")
```

28.6.19 str_sub()

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

This returns a copy of 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)
```

28.6.20 str_upper()

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

This returns a copy of the original string, converting all lowercase characters to uppercase characters.

Syntax example:

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

28.7 Output routines

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

28.7.1 out()

Purpose	Print expressions to output, with a newline at the end	
Category	Pseudo-routine	
Syntax	**out()** **out(***item*: exp**, ...)**	
Parameters	*item*	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 **out()**, an empty string is printed, followed by a newline.

This calls **to_string()** (see 27.4.4) 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 newline.

Syntax example:

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

28.7.2 outf()

Purpose	Print formatted expressions to output, with no newline at the end	
Category	Pseudo-routine	
Syntax	**outf(***format*: string**,** *item*: exp**, ...)**	
Parameters	*format*	A string expression containing a standard C formatting mask for each *item* (see 28.7.3).
	item	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.

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 27.4.4) is used for creating the string representation of the expression.

- To add a newline, add the `\n` characters to the format string.
- **outf()** can be used to add the newlines 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])
```

28.7.3 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., `%#x` or `%#010x`.
`min_width`	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 `max_chars` is set large enough), this number is ignored and enough space is used to accommodate the entire expression.
`max_chars`	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 `min_width`, then the `min_width` number is ignored. For real number formats `e`, `f`, and `g`, `max_chars` defines the precision—the number of digits after the decimal point.
`s`	converts the expression to a string. The routine **to_string()** (see 27.4.4) is used to convert a non-string expression to a string.
`d`	prints a numeric expression in decimal format.
`x`	prints a numeric expression in hex format. With the optional `#` character, adds `0x` before the number.
`b`	prints a numeric expression in binary format.
`o`	prints a numeric expression in octal format.
`u`	prints integers (**int** and **uint**) in uint format.
`e`	prints a numeric value in the style `[-]d.ddde?dd` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears.
`f`	prints a numeric value in the style `[-]d.ddde?dd`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears.
`g`	prints a numeric value in the style of either `f` or `e`. The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style `e` is used if the exponent from its conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

Printing real values with integer formatting will cause an automatic conversion to int(bits:*).

28.8 Operating system interface routines

The routines in this subclause enable use of OS commands from within the ***e*** programming language. These routines work on all supported OSs.

28.8.1 spawn()

Purpose	Send commands to the OS
Category	Pseudo-routine
Syntax	**spawn()** **spawn(***command*: string**, ...)**
Parameters	*command* An expression of type **string**.

This takes a variable number of parameters, concatenates them together, and executes the string result as an OS command via **system()** (see 28.8.3).

Syntax example:

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

28.8.2 spawn_check()

Purpose	Send a command to the OS and report error
Category	Routine
Syntax	**spawn_check(***command*: string**)**
Parameters	*command* An expression of type **string**.

This executes a single string as an OS command via **system()** (see 28.8.3), then calls **error()** (see 16.3.2) if the execution of the command returned an error status.

Syntax example:

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

28.8.3 system()

Purpose	Send a command to the OS
Category	Routine
Syntax	**system(***command*: string**)**: int
Parameters	*command* An expression of type **string**.

This executes the string as an OS command and returns the result. On UNIX systems, the command string is passed to the C `system()` call.

Syntax example:

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

28.8.4 output_from()

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

This executes the string as an OS 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")
```

28.8.5 output_from_check()

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

This executes the string as an OS command, returns the output as a list of string, and then calls **error()** (see 16.3.2) 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")
```

28.8.6 get_symbol()

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

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

Syntax example:

```
current_display = get_symbol("DISPLAY")
```

28.8.7 date_time()

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

This returns the current date and time as a string. The date/time is represented in the standard format supplied by the C library routine `ctime`.

Syntax example:

```
print date_time()
```

28.8.8 getpid()

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

This returns the current process ID as an integer.

Syntax example:

```
print getpid()
```

28.9 set_config()

Purpose	Set values of global configuration parameters	
Category	Predefined routine	
Syntax	**set_config(***category*: keyword**,** *option*: keyword**,** *value*: exp [**,** *option*: keyword**,** *value*: exp...]**)**	
Parameters	*category*	Is one of the following: **cover**, **gen**, **memory**, **print**, or **run**, or any additional implementation-dependent category.
	option	The valid **cover** options are: — **mode** (either **normal** or **count_only**) — **absolute_max_buckets** The valid **generate** options are: — **absolute_max_list_size** — **max_depth** — **max_structs** The valid **memory** options are: — **gc_threshold** — **gc_increment** — **max_size** — **absolute_max_size** The valid **print** option is: **radix**. The valid **run** option is: **tick_max**. The implementation can also introduce additional options.
	value	The valid values for each option are implementation specific.

This routine sets the configuration options to the specified values.

Syntax example:

```
set_config(memory, gc_threshold, 100M)
```

28.10 Random routines

The ***e*** language supports the routines shown in Table 45 to generate random **real** numbers:

Table 45—Random routines

Routine	Description
rdist_uniform(from: real, to:real): real	Returns a random **real** number using uniform distribution in the range **from** to **to**.
NOTE—The behavior of rdist_uniform() in ***e*** is equivalent to Verilog's $rdist_uniform() defined in IEEE Std 1364 (17.9.2).	

28.11 Simulation-related routines

The routines in this subclause relate to interactions with the simulator. See also Clause 23.

28.11.1 simulator_command()

Purpose	Issue a simulator command	
Category	Predefined routine	
Syntax	**simulator_command(***command*: string**)**	
Parameters	*command*	A valid simulator command, enclosed in double quotes (`" "`). **simulator_command()** cannot be used to pass commands that change the state of simulation, such as `run`, `restart`, `restore`, or `exit` (to the simulator).

This passes a command to the HDL simulator from ***e***. The command shall not return a value. The output of the command is sent to the standard output and log file.

Syntax example:

```
simulator_command("force -deposit memA(31:0)")
```

28.11.2 stop_run()

Purpose	Stop a simulation run cleanly
Category	Predefined routine
Syntax	**stop_run()**

This stops the simulator and initiates post-simulation phases. This method needs to be called by a user-defined method or TCM to stop the simulation run cleanly. The following things occur when **stop_run()** is invoked:

a) The **quit()** method of each struct under **sys** is called. Each **quit()** method emits a “quit” event for that struct instance at the end of the current tick.

b) All executing threads shall continue until the end of the current tick.

c) At the end of the current tick, the extract, check, and finalize test phases are performed.

d) If a simulator is linked here, ***e*** terminates the simulation cleanly after the test is finalized.

Plus, the following restrictions also apply:

— Executing a tick after calling **stop_run()** shall be considered an error.

— If the simulator `exit` command is called before **stop_run()**, the global methods for extracting, checking, and finalizing the test are called.

NOTE—Use **sys.extract()** and extend that to make something happen right after stopping a run [rather then extending or modifying the **stop_run()** method].

See also 27.2.2.5 and the **run** option of 28.9.

Syntax example:

```
stop_run()
```

28.11.3 get_timescale()

Purpose	Return the current timescale
Category	Predefined routine
Syntax	**get_timescale()** : string

This returns the current timescale, used to represent the time in sys.time (see also 4.3.1.4).

Syntax example:

```
print get_timescale()
```