

17. Packing and unpacking

Packing and unpacking operate on scalars, strings, lists, and *structs*. The following sections show how to perform basic packing and unpacking of these data types using two of the *e* basic packing tools, the **pack()** and **unpack()** methods. *See also:* 3.3.

267

17.1 Basic packing

The following sections detail how to pack, unpack, and swap data.

17.1.1 pack()

Purpose	Perform concatenation and type conversion
Category	Pseudo-routine
Syntax	pack (<i>option</i> :pack option, <i>item</i> : exp, ...): list of bit
Parameters	<p><i>option</i></p> <p>For basic packing, this parameter is one of the following choices. See 17.3 for information on other pack options.</p> <p>a) packing.high — Places the least significant bit of the last physical field declared or the highest list item at index [0] in the resulting list of bit. The most significant bit of the first physical field or lowest list item is placed at the highest index in the resulting list of bit.</p> <p>b) packing.low — Places the least significant bit of the first physical field declared or the lowest list item at index [0] in the resulting list of bit. The most significant bit of the last physical field or highest list item is placed at the highest index in the resulting list of bit.</p> <p>c) NULL — If NULL is specified, the global default is used. This global default is set initially to packing.low.</p>
	<p><i>item</i></p> <p>A legal <i>e</i> expression that is a path to a scalar or a compound data item, such as a <i>struct</i>, field, list, or variable.</p>

This performs concatenation of items, including items in a list or fields in a *struct*, in the order specified by the pack options parameter and returns a list of bits. This method also performs type conversion between any of the following:

- scalars
- strings
- lists and list subtypes (derived *struct*)

Packing is commonly used to prepare high-level *e* data into a form that can be applied to a DUT. It operates on scalar or compound (struct or list) data items. Pack expressions are untyped expressions. In many cases, the *e* program can deduce the required type from the context of the pack expression (see 3.3). An unbounded integer cannot be packed.

Syntax example:

```
i_stream = pack(packing.high, opcode, operand1, operand2);
```

266

17.1.2 `unpack()`

Purpose	Unpack a bit stream into one or more expressions	
Category	Pseudo-routine	
Syntax	<code>unpack(option: pack option, value: exp, target1: exp [, target2: exp, ...])</code>	
Parameters	<i>option</i>	For basic packing, this parameter is one of the following choices. See 17.3 for information on other pack options. <ul style="list-style-type: none"> a) packing.high — Places the most significant bit of the list of bits at the most significant bit of the first field or lowest list item. The least significant bit of the list of bits is placed into the least significant bit of the last field or highest list item. b) packing.low — Places the least significant bit of the list of bits at the least significant bit of the first field or lowest list item. The most significant bit of the list of bits is placed into the most significant bit of the last field or highest list item. c) NULL — If NULL is specified, the global default is used. This global default is set initially to packing.low.
	<i>value</i>	A scalar expression or list of scalars that provides a value that is to be unpacked.
	<i>target1, target2</i>	One or more expressions separated by commas (,). Each expression is a path to a scalar or a compound data item, such as a <i>struct</i> , field, list, or variable.

This converts a raw bit stream into high level data by storing the bits of the *value* expression into the *target* expressions. If the value expression is not a list of bit, it is first converted (see 17.5) into a list of bit by calling `pack()` using **packing.low**. Then, the list of bits is unpacked into the target expressions.

The *value* expression is allowed to have more bits than are consumed by the *target* expressions. In that case, if **packing.low** is used, the extra high-order bits are ignored; if **packing.high** is used, the extra low-order bits are ignored.

Unpacking is commonly used to convert raw bit stream output from the DUT into high-level e data. It operates on scalar or compound (struct or list) data items.

Syntax example:

```
unpack(packing.high, lob, s1, s2);
```

17.1.3 swap()

Purpose	Swap small bit chunks within larger chunks	
Category	Pseudo-routine	
Syntax	<i>list-of-bit.swap</i> (<i>small</i> : int, <i>large</i> : int): list of bit	
Parameters	<i>small</i>	An integer that is a factor of <i>large</i> .
	<i>large</i>	An integer that is either UNDEF or a factor of the number of bits in the entire list. If UNDEF, the method reverses the order of small chunks within the entire list, e.g., <code>lob.swap(1, UNDEF)</code> is the same as <code>lob.reverse()</code> .

This predefined list method accepts a list of bits, changes the order of the bits, and then returns the reordered list of bits. This method is often used in conjunction with **pack()** or **unpack()** to reorder the bits in a bit stream going to or coming from the DUT.

- If *large* is not a factor of the number of bits in the entire list, an error message shall result.
- If *small* is not a factor of *large*, an error message shall also result. The only exception is if *large* is UNDEF and *small* is not a factor, the swap is not performed and no error message is issued.

Example

Figure 1 shows two swaps. The first swap reverses the order of nibbles in every byte. The second swap reverses the whole list.

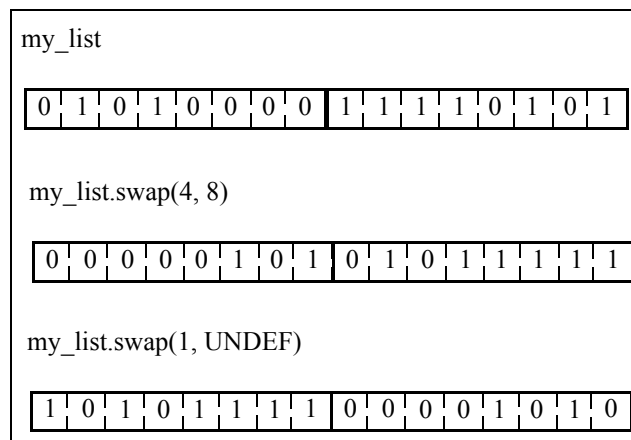


Figure 1—Swapping examples

Syntax example:

```
s2 = s1.swap(2, 4);
```

17.2 Predefined pack options

This section details the predefined pack options: **packing.high**, **packing.low**, and **packing.global_default**.

17.2.1 pack_options struct

The predefined instances are all instances of the `pack_options struct`. The `pack_options` declaration is:

```

struct pack_options {
    reverse_fields: bool;
    reverse_list_items: bool;
    final_reorder: list of int;
    scalar_reorder: list of int;
};

```

The following sections describe each of its fields.

17.2.1.1 reverse_fields

If this flag is set to `FALSE`, the fields in a `struct` are packed in the order they appear in the struct declaration; if `TRUE`, they are packed in reverse order. The default is `FALSE`.

17.2.1.2 reverse_list_items

If this flag is set to `FALSE`, the items in a list are packed in ascending order; if `TRUE`, they are packed in descending order. The default is `FALSE`.

17.2.1.3 scalar_reorder

The `scalar_reorder` field can be used to perform one or more `swap()` operations on each scalar before packing. The list in the `scalar_reorder` field shall include an even number of items. Each pair of items in the list is the parameter list of a `swap()` operation (see 17.1.3). To perform multiple swaps, use multiple pairs of parameters (each pair of parameters corresponds to a pair of swap parameters).

17.2.1.4 final_reorder

The `final_reorder` field can be used after packing each element in the packing expression to perform final swapping on the resulting bit stream. The list in the `final_reorder` field shall include an even number of items. Each pair of items in the list is the parameter list of a `swap()` operation (see 17.1.3). To perform multiple swaps, use multiple pairs of parameters (each pair of parameters corresponds to a pair of swap parameters).

17.2.2 Predefined settings

Table 1 shows the corresponding settings for each of the predefined pack options (`packing.high`, `packing.low`, and `packing.global_default`). To customize a packing option, see 17.3.

Table 1—Predefined packing options

Flag	packing.high	packing.low	packing.global_default
reverse_fields	True	False	False
reverse_list_items	True	False	False
final_reorder	0	0	0
scalar_reorder	0	0	0

17.2.3 `packing.high` 1

This `pack_options` instance traverses the source fields or variables in the reverse order from the order in which they appear in code, placing the least significant bit of the last field or list item at index [0] in the resulting list of bit. The most significant bit of the first field or list item is placed at the highest index in the resulting list of bit. 5

17.2.4 `packing.low` 10

This `pack_options` instance traverses the source fields or variables in the order they appear in code, placing the least significant bit of the first field or list item at index [0] in the resulting list of bit. The most significant bit of the last field or list item is placed at the highest index in the resulting list of bit.

17.2.5 `packing.global_default` 15

This `pack_options` instance is used when the first parameter of `pack()`, `unpack()`, `do_pack()`, or `do_unpack()` is `NULL`. It has the same flags as `packing.low`.

17.3 Customizing pack options 20

Each of the predefined instances defined within the `pack_options struct` (see 17.2.1) can also be modified. To customize the packing options:

- a) Create an instance of the `pack_options struct`; 25
- b) Modify one or more of its fields;
- c) Pass the `struct` instance as the first parameter to `pack()`, `unpack()`, `do_pack()`, or `do_unpack()`.

17.4 Packing and unpacking specific types 30

This section defines how to operate on `structs`, lists, scalars, and strings.

17.4.1 Packing and unpacking structs 35

Packing a struct creates an ordered bit stream from all the physical fields (marked with `%`) in the `struct`, starting with the first physical field declared. Other fields (called *virtual fields*) are ignored by the packing process. If a physical field is of a compound type (`struct` or list) the packing process descends recursively into the `struct` or list. 40

Unpacking a bit stream into a `struct` fills the physical fields of the `struct`, starting with the first field declared and proceeding recursively through all the physical fields of the `struct`. Unpacking a bit stream into a field that is a list follows some additional rules (see 17.4.2).

Unpacking a struct that has not yet been allocated (with `new`) causes the `e` program to allocate the `struct` and run the struct's `init()` method. Unlike `new`, the struct's `run()` method is not called. 45

17.4.1.1 Customizing packing for a particular struct

A `struct` is packed or unpacked using its predefined methods `do_pack()` and `do_unpack()`. It is possible to modify these predefined methods for a particular `struct`. These methods are called automatically whenever data is packed from or unpacked into the `struct`. 50

55

17.4.1.1.1 do_pack()

Purpose	Pack the physical fields of the struct
Category	Predefined method of any struct
Syntax	<code>do_pack(options:pack options, l: *list of bit)</code>
Parameters	<i>options</i> This parameter is an instance of the pack_options struct (see 17.3).
	<i>l</i> An empty list of bits that is extended as necessary to hold the data from the struct fields.

The **do_pack()** method of a *struct* is called automatically whenever the *struct* is packed. This method appends data from the physical fields (the fields marked with %) of the *struct* into a list of bits according to flags determined by the pack options parameter. The virtual fields of the *struct* are skipped. The method shall issue a runtime error message if this *struct* has no physical fields.

The **do_pack()** method can be extended to create a unique packing scenario for that *struct* by creating a custom **pack_options** instance (see 17.3).

The following considerations also apply.

- Do not call the **do_pack()** method of any *struct* directly, e.g., `my_struct.do_pack()`. Use **pack()** instead, e.g., `pack(packing.high, my_struct)`.
- Do not call **pack(me)** in the **do_pack()** method. This causes infinite recursion. Call **packing.pack_struct(me)** instead.
- Append the results of any pack operation within **do_pack()** to the empty list of bits referenced in the **do_pack()** parameter list.
- If the **do_pack()** method is modified and then physical fields are added later in an extension to the *struct*, that **do_pack()** might need to be modified again.

Example

For example, the following assignment to `lob`

```
lob = pack(packing.high, i_struct, p_struct);
```

makes the following calls to the **do_pack** method of each struct, where `tmp` is an empty list of bits:

```
i_struct.do_pack(packing.high, *tmp)
p_struct.do_pack(packing.high, *tmp)
```

Syntax example:

```
do_pack(options:pack_options, l: *list of bit) is only {
    var L : list of bit = pack(packing.low, operand2,
        operand1, operand3);
    l.add(L);
};
```

17.4.1.1.2 `do_unpack()`

Purpose	Unpack a packed list of bit into a struct	
Category	Predefined method of any struct	
Syntax	<code>do_unpack(options:pack options, l: list of bit, from: int): int</code>	
Parameters	<i>options</i>	This parameter is an instance of the pack_options struct (see 17.3).
	<i>l</i>	A list of bits containing data to be stored in the struct fields.
	<i>from</i>	An integer that specifies the index of the bit to start unpacking.
	int (return value)	An integer that specifies the index of the last bit in the list of bits that was unpacked.

The `do_unpack()` method is called automatically whenever data is unpacked into the current struct. This method unpacks bits from a list of bits into the physical fields of the struct. It starts at the bit with the specified index, unpacks in the order defined by the pack options, and fills the current struct's physical fields in the order they are defined.

This method returns an integer, which is the index of the last bit unpacked into the list of bits.

The method shall issue a runtime error message if the struct has no physical fields. If there are leftover bits at the end of packing, it is not an error. If more bits are needed than currently exist in the list of bits, a runtime error shall be issued.

The `do_unpack()` method can be extended to create a unique unpacking scenario for that struct by creating a custom **pack_options** instance (see 17.3).

The following considerations also apply.

- Do not call the `do_unpack()` method of any struct directly, e.g., `my_struct.do_unpack()`. Use **unpack()** instead, e.g., `unpack(packing.high, lob, my_struct)`.
- When the `do_unpack()` method is modified, the index of the last bit in the list of bits that was unpacked also needs to be calculated and returned. ****How??**

Example

For example, the following call to `unpack()`

```
unpack(packing.low, lob, c1, c2);
```

makes the following calls to the `do_unpack` method of each struct:

```
c1.do_unpack(packing.low, lob, index);
c2.do_unpack(packing.low, lob, index);
```

Syntax example:

```
do_unpack(options:pack_options, l: list of bit, from: int):int is only {
  var L : list of bit = l[from..];
  unpack(packing.low, L, op2, op1, op3);
```



```

struct instruction {
    %opcode    : uint (bits : 3);
    %operand   : uint (bits : 5);
    %address   : uint (bits : 8);
};

```

The extension to **post_generate()** shown below unpacks a list of bits, `packed_data`, into a variable `inst` of type `instruction` using the **packing.high** option. The results are shown in Figure 3.

```

extend sys {
    post_generate() is also {
        var inst : instruction;
        var packed_data: list of bit;
        packed_data = {1;1;1;1;0;0;0;0;1;0;0;1;1;0;0;1};
        unpack(packing.high, packed_data, inst);
    };
};

```

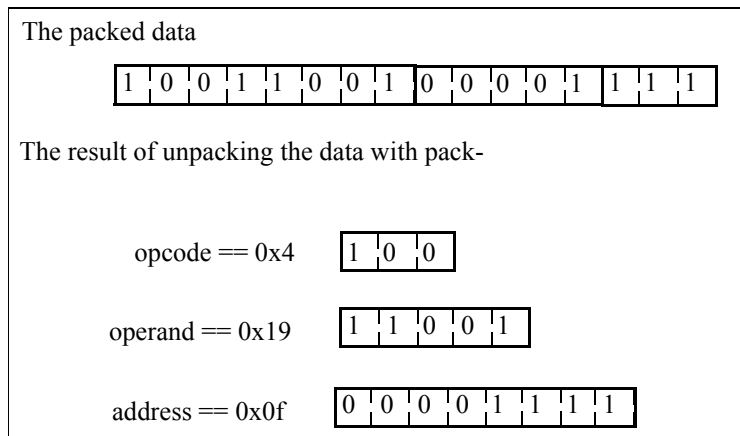


Figure 3—Simple unpacking example showing packing.high

In this case, the expression that provides the value, `packed_data`, is a list of bits. When a value expression is not a list of bits, *e* uses implicit packing to store the data in the target expression (see 17.5).

17.4.2 Packing and unpacking lists

Packing a list creates a bit stream by concatenating the list items together, starting with the item at index [0].

Unpacking a bit stream into a list fills the list item-by-item, starting with the item at index [0]. The size of the unpacked list is determined by whether the list is sized and whether it is empty.

- Unpacking into an empty list expands the list as needed to contain all the available bits.
- Unpacking into a non-empty list unpacks only until the existing list size is reached.
- Unpacking to a *struct* fills the sized lists only to their defined size, regardless of their actual size at the time.
- Unpacking into an unsized, uninitialized list shall cause a runtime error message, because the list is expanded as needed to consume all the given bits.

NOTE—When a *struct* is allocated, the lists within it are empty. If the lists are sized, unpacking is performed until the defined size is reached.

See Clause 2 for more information on sizing lists.

Example

This example shows the recommended way to get a variable number of list items. The specification order is important because the `len1` and `len2` values need to be set before initializing `data1` and `data2`. Declaring `len1` and `len2` before `data1` and `data2` ensures the list length is generated first. Unpacking into a list with a variable number of items [requires packing and passing](#) the number of items in the list before unpacking the list.

```

struct packet{
    %len1: int;
    %len2: int;
    %data1[len1]: list of byte;
    %data2[len1 + len2]: list of byte;
};

```

17.4.3 Packing and unpacking scalar expressions

Packing a scalar expression creates an ordered bit stream by concatenating the bits of the expression together. Unpacking a bit stream into a scalar expression fills the scalar expression by putting the lowest bit of the bit stream into the lowest bit of the scalar expression. If a list is unpacked into one or more scalar expressions and there are not enough bits in the list to put a value into each scalar, a runtime error shall be issued.

Packing and unpacking of a scalar expression is performed using the expression's inherent size, except when the expression contains a bit-slice operator. Missing bits are assumed to be zero (0) and extra bits are allowed (and ignored). *See also:* 3.1.

The bit-slice operator [:] can also be used to select a subrange of an expression to be packed or unpacked. This operator does not change the type of the pack or unpack expression.

17.4.4 Packing and unpacking strings

Packing a string creates an ordered bit stream by concatenating each ASCII byte of the string together from left-to-right, ending with a byte with the value zero (the final NULL byte). Unpacking a string places the bytes of the string into the target expression, starting with the first ASCII byte in the string up to and including the first byte with the value zero (0).

The `as_a()` method, which converts directly between the **string** and **list of byte** types, can also be used to obtain different results (see 3.8.1).

17.5 Implicit packing and unpacking

Implicit packing and unpacking is always performed using the parameters of **packing.low** as follows.

- When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked before it is assigned.

```

var my_list: list of int = {1;2;3};
var int_10: int(bits:10);

```

```
my_list = 'top.foo';
int_10 = pack(NULL, 5);
```

Untyped expressions include HDL signals, pack expressions, and bit concatenations (see 3.3).

- When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned:

```
'top.foo' = {1;2;3};
```

- When the value expression of an unpack action is other than a list of bits, it is implicitly packed before it is unpacked:

```
unpack(packing.low, 5, my_list);
```

Implicit packing and unpacking is not supported for strings, structs, or lists of non-scalar types.

1
5
10
15
20
25
30
35
40
45
50
55