

## 6. Structs, subtypes, and fields 1

The basic organization of an *e* program is a tree of structs. A struct is a compound type that contains data fields, procedural methods, and other members. It is the *e* equivalent of a class in other OO languages. A base struct type can be extended by adding members. Subtypes can be created from a base struct type, which inherit the base type's members and contain additional members. A *field* is a feature of a struct that can hold data. Fields can be scalars or references to structs or lists. 5

An extension can reside in a module outside of where it was originally defined, in which case, the extending module shall be loaded after the original module. For more details about load ordering, see Annex B. 10

### 6.1 Structs overview 15

Structs are used to define data elements and behavior of components of a test environment. 15

- A struct can hold all types of data and methods.
- All user-defined structs inherit from the predefined base struct type, **any\_struct**.
- For reusability of *e* code, use **extend** to add struct members or change the behavior of a previously defined struct. 20

Inheritance is implemented in *e* by using either of the following mechanisms:

- a) “when” inheritance is specified by defining subtypes with **when** struct members. 25
- b) “like” inheritance is specified with the **like** clause in new struct definitions.

The best inheritance methodology for most applications is “when” inheritance. See also Annex C.

### 6.2 Defining structs: struct 30

<b>Purpose</b>	Define a data struct	
<b>Category</b>	Statement	<span style="float: right;">35</span>
<b>Syntax</b>	<b>struct</b> <i>struct-type</i> [ <b>like</b> <i>base-struct-type</i> ] { <i>struct-member</i> ; ...}	
<b>Parameters</b>	<i>struct-type</i>	The name of the new struct type. <span style="float: right;">40</span>
	<i>base-struct-type</i>	The type of the struct from which the new struct inherits its members.
	<i>struct-member</i> ; ...	The contents of the struct. The following are types of struct members: <span style="float: right;">45</span> <ul style="list-style-type: none"> <li>— data fields for storing data</li> <li>— methods for procedures</li> <li>— events for defining temporal triggers</li> <li>— coverage groups for defining coverage points</li> <li>— <b>when</b>, for specifying inheritance subtypes</li> <li>— declarative constraints for describing relations between data fields</li> <li>— <b>on</b>, for specifying actions to perform upon event occurrences</li> <li>— <b>expect</b>, for specifying temporal behavior rules</li> </ul> The definition of a struct can be empty, containing no members. <span style="float: right;">50</span>

Structs are used to define the data elements and behavior of components and the test environment. Structs contain struct members of the types listed in the preceding *Parameters* description. Struct members can be conditionally defined (see 6.6). 55

The optional **like** clause is an inheritance directive. All struct members defined in *base-struct-type* are implicitly defined in the struct subtype, *struct-type*. New struct members can also be added to the inheriting struct subtype and methods of the base struct type can be extended in the inheriting struct subtype.

Additional subtypes can, in turn, be derived from a subtype. In the following example, the subtype `agp_transaction` is derived from the (previously defined) `pci_transaction` subtype. Each subtype can add fields to its base type and place its own constraints on fields of its base type.

Syntax example:

```

type AGPModeType: [AGP_2X, AGP_4X];
struct agp_transaction like pci_transaction {
    block_size: uint;
    mode: AGPModeType;
    when AGP_2X agp_transaction {
        keep block_size == 32;
    };
    when AGP_4X agp_transaction {
        keep block_size == 64;
    };
};

```

### 6.3 Extending structs: extend type

<b>Purpose</b>	Extend an existing data struct
<b>Category</b>	Statement
<b>Syntax</b>	<b>extend</b> [ <i>struct-subtype</i> ] <i>base-struct-type</i> { [ <i>struct-member</i> ; ...]}
<b>Parameters</b>	<i>struct-subtype</i> Adds struct members to the specified subtype of the base struct type only. The added struct members are known only in that subtype, not in other subtypes.
	<i>base-struct-type</i> The base struct type to extend.
	<i>struct-member</i> ; ...                    The contents of the struct. The following are types of struct members: — data fields for storing data — methods for procedures — events for defining temporal triggers — coverage groups for defining coverage points — <b>when</b> , for specifying inheritance subtypes — declarative constraints for describing relations between data fields — <b>on</b> , for specifying actions to perform upon event occurrences — <b>expect</b> , for specifying temporal behavior rules The definition of a struct can be empty, containing no members.

This adds struct members to a previously defined struct or struct subtype. Members added to the base struct type in extensions apply to all other extensions of the same struct, e.g., if a method in a base struct is extended using **is only**, it overrides that method in every one of the **like** children.

If **like** inheritance has been used on a struct type, there are limitations on further extending the original base struct type definition; see 6.4.

Syntax example:

```

type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend packet {
    keep len < 256;
};

```

## 6.4 Restrictions on inheritance

The following restrictions shall apply when using inheritance:

- Determinant fields shall not be explicitly referenced.
- Generation of a parent does not create **like** children.
- **when** subtypes shall not be added to a struct with **like** children. Similarly, a **like** child shall not be created from a struct that has **when** subtypes.

## 6.5 Extending subtypes

A *struct subtype* is an instance of the struct in which one of its fields has a particular value. For example, the `packet` struct defined in the following example has `atm packet` and `eth packet` subtypes, depending on whether the `kind` field is `atm` or `eth`.

*Example*

```

type packet_kind: [atm, eth];
struct packet {
    len: int;
    kind: packet_kind;
};
extend atm packet {
    keep len == 53;
};

```

Similar to structs, a struct subtype can be **extended**; the extension shall only apply to that subtype.

## 6.6 Creating subtypes with when

The **when** struct member creates a conditional subtype of the current struct type when a particular field of the struct has a given value. This is called “when” inheritance and is one of two techniques that *e* provides for implementing inheritance. The other is called “like” inheritance. When inheritance is described in this subclause. Like inheritance is described in 6.2.

When inheritance is the recommended technique for modeling in *e*. Like inheritance is more appropriate for procedural test bench programming. When and like inheritance are compared in Annex C.

<b>Purpose</b>	Create a subtype
<b>Category</b>	Struct member
<b>Syntax</b>	<b>when</b> [ <i>struct-subtype</i> ] <i>base-struct-type</i> { [ <i>struct-member</i> ; ...]}
<b>Parameters</b>	<i>struct-subtype</i> A subtype declaration in the form <i>type-qualifier</i> ' <i>field-name</i> . The <i>type-qualifier</i> is one of the legal values for the field named by <i>field-name</i> . If the <i>field-name</i> is a Boolean field and its value is TRUE for the subtype, the <i>type-qualifier</i> can be omitted. The <i>field-name</i> is the name of a field in the base struct type. Only Boolean or enumerated fields can be used. If the field type is Boolean, the <i>type-qualifier</i> shall be TRUE or FALSE. If the field type is enumerated, the qualifier shall be a value of the enumerated type. If the <i>type-qualifier</i> can apply to only one field in the struct, the ' <i>field-name</i> can be omitted. More than one <i>type-qualifier</i> ' <i>field-name</i> combination can be stated to create a subtype based on more than one field of the base struct type.
	<i>base-struct-type</i> The struct type of the current struct (in which the subtype is being created).
	<i>struct-member</i> ; ...                     Definition of a struct member for the struct subtype. One or more new struct members can be defined for the subtype.

Use the **when** construct to create families of objects, in which multiple subtypes are derived from a common base struct type. A subtype is a struct type in which specific fields of the base struct have particular values, e.g.,

- a) If a struct type named `packet` has a field named `kind` that can have a value of `eth` or `atm`, then two subtypes of `packet` are `eth packet` and `atm packet`.
- b) If the `packet` struct has a Boolean field named `good`, two subtypes are `FALSE'good packet` and `TRUE'good packet`.
- c) Subtypes can also be combinations of fields, such as `eth TRUE'good packet` and `eth FALSE'good packet`.

Struct members defined in a **when** construct shall only be accessed in the subtype, not in the base struct. This provides a way to define a subtype that has some struct members in common with the base type and all of its other subtypes, but has other struct members that belong only to the current subtype.

If like inheritance is used to create a subtype of a base struct type, the base type shall not be extended by using **when**.

Syntax example:

```

struct packet {
    len: uint;
    good: bool;
    when FALSE'good packet {
        pkt_msg() is {
            out("bad packet");
        };
    };
};

```

## 6.7 Extending when subtypes

1

There are two general rules governing the extensions of **when** subtypes:

- a) If a struct member is declared in the base struct, it shall not be re-declared in any **when** subtype, but it can be extended. 5
- b) With the exception of coverage groups and the events associated with them, any struct member defined in a **when** subtype does not apply or is unknown in other subtypes, including:
  - 1) fields 10
  - 2) constraints
  - 3) events
  - 4) methods
  - 5) **on**
  - 6) **expect** 15
  - 7) **assume**

### 6.7.1 Coverage and when subtypes

All coverage events shall be defined in the base struct. Attempts to do so within a subtype, however, shall result in a load time error. Coverage groups shall be defined in the base struct or in the subtype. 20

### 6.7.2 Extending methods in when subtypes

A method defined or extended within a **when** construct is executed in the context of the subtype and can freely access the unique struct members of the subtype with no need for any casting. 25

When a method is declared in a base type, each extension of the method in a subtype shall have the same parameters and return type as the original declaration. Attempts to do otherwise shall result in a load time error. However, if a method is not declared in the base type, each definition of the method in a subtype can have different parameters and return type. 30

If more than one method of the same name is known in a **when** subtype, any reference to that method is ambiguous and shall result in a load-time error. To remove the ambiguity from such a reference, use the **as\_a()** type casting operator (see 5.7.1) or the **when** subtype qualifier syntax. 35

Method calls are checked when the *e* code is parsed. If there is no ambiguity, the method to be called is selected and all similar references are resolved in the same manner. 40

40

45

50

55

## 6.8 Defining fields: field

<b>Purpose</b>	Define a struct field	
<b>Category</b>	Struct member	
<b>Syntax</b>	<code>[package   protected   private] [const] [!] [%] field-name[: type] [[min-val .. max-val]] [[(bits   bytes):num]]</code>	
<b>Parameters</b>	<b>package   protected   private</b>	Sets any access restriction: which code, at what scope, can access this struct member. Otherwise, the default setting is all code has access to this struct member. See 22.3 for the keyword definitions.
	<b>const</b>	Denotes this field shall retain a constant value throughout its lifetime.
	<b>!</b>	Denotes an ungenerated field. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<b>%</b>	Denotes a physical field. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<i>field-name</i>	The name of the field being defined.
	<i>type</i>	The type for the field. This can be any scalar type, string, struct, or list. If the field name is the same as an existing type, the <i>: type</i> part of the field definition can be omitted. Otherwise, the type specification is required.
	<i>min-val..max-val</i>	An optional range of values for the field. If no range is specified, the range is the default range for the field's type.
<b>(bits   bytes: num)</b>	The width of the field in bits or bytes. This syntax can be used for any scalar field, even if the field has a type with a known width.	

This defines a field to hold data of a specific type. It can be a constant value (`const`), a physical field (`%`) or a virtual field (the default), and generated (the default) or not generated (`!`). For scalar data types, the size of the field can also be specified in bits or bytes.

Syntax example:

```
private const %packet: packet_t;
```

### 6.8.1 Constant values

**const** identifies a field whose value is kept constant throughout the lifetime of the object and enforces this constant value. The *e* compiler takes advantage of **const** declarations to optimize memory use. A significant reduction in memory consumption may result from declaring a **when** determinant field (see 6.6) as **const**.

#### 6.8.1.1 Initializing const fields

Initialization of fields declared as **const** shall be completed during the creation phase of a struct.

- a) A value can be assigned to a **const** field:
  - 1) during generation, using **pre\_generate()** (see 9.4.2) or **post\_generate()** (see 9.4.3);
  - 2) during unpacking, using **do\_unpack()** (see 19.4.1.1.2);
  - 3) by using an assignment action while creating an object, with a **new...with** action block (see 4.16.2) or the **init()** method (see 27.2.2.1).

In these contexts, a **const** field of the newly created struct may be used on the left side of an assignment operator. 1

- b) A **const** field can also be initialized by using a built-in initialization mechanism, such as the **copy()** (see 27.4.1) or **read\_binary\_struct()** method (see 29.5.2). 5

\*\*All the [above referenced subclauses](#) (e.g., **copy()**) should also include [x-refs](#) to the new 6.8.1.\*\*

A **const** field can only be assigned a value once; otherwise a run-time error shall occur. 10

### 6.8.1.2 Restrictions

- The **const** modifier cannot be applied to fields that are accessed by tick access notation.
- Fields of **list** type (see 6.9) cannot be declared **const**.
- Fields declared under **when** subtypes with a non-constant determinant cannot be declared **const**. 15
- Fields of **enum** types (see 5.6) that are declared **const** have no default value upon creation of the struct (even if zero (0) is a possible enumerated value); they need to be initialized as specified in 6.8.1.1.
- Constant fields cannot be passed by reference.
- A compile-time error occurs if a **const** field is initialized with any construct other than those listed in 6.8.1.1. 20

### 6.8.2 Physical fields

A field defined as a physical field (with the % option) is packed when the struct is packed. Fields that represent data to be sent to the HDL device in the simulator or that are to be used for memories need to be physical fields. Nonphysical fields are called *virtual fields* and are not packed automatically when the struct is packed, although they can be packed individually. 25

If no range is specified, the width of the field is determined by the field's type. For a physical field, use the (**bits : num** or **bytes : num**) syntax to specify the width when the field's type does not have a known width. 30

### 6.8.3 Ungenerated fields

A field defined as ungenerated (with the ! option) is not generated automatically. This is useful for fields that are to be explicitly assigned during a test or whose values involve computations that cannot be expressed in constraints. 35

Ungenerated fields have default initial values (0 for scalars, NULL for structs, and an empty list for lists). An ungenerated field whose value is a range (such as [0 . . 100]) gets the first value in the range. If the field is a struct, its values remains NULL; therefore, the referenced struct is not allocated and none of the fields in it are generated. 40

### 6.8.4 Assigning values to fields

Unless a field is defined as ungenerated, a value is generated for it when the struct is generated, subject to any constraints that exist for the field. However, even for generated fields, values can be assigned in user-defined methods or predefined methods, such as **init()**, **pre\_generate()**, or **post\_generate()**. The ability to assign a value to a field is not affected by either the ! option or any generation constraints. 45

## 6.9 Defining list fields

This subclause defines list fields. 50

55

### 6.9.1 list of

<b>Purpose</b>	Define a list field	
<b>Category</b>	Struct member	
<b>Syntax</b>	[!][%] <i>list-name</i> [ <i>length-exp</i> ]: <b>list of</b> <i>type</i>	
<b>Parameters</b>	<b>!</b>	Do not generate this list. The ! and % options can be used together, in either order.
	<b>%</b>	Denotes a physical list. The ! and % options can be used together, in either order.
	<i>list-name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list. The expression shall evaluate to a non-negative integer.
	<i>type</i>	The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

This defines a list of items of the specified type.

An initial size can be specified for the list; the list initially contains that number of items. The size shall conform to the initialization rules, the generation rules, and the packing rules. Even if an initial size is specified, the list size can change during a test if the list is operated on by a list method that changes the number of items.

All list items are initialized to their default values when the list is created. For a generated list, the initial default values are replaced by generated values. For information about initializing list items to particular values, see 5.3.3.6 and 9.2.7.3.

Syntax example:

```
packets: list of packet;
```

## 6.9.2 list(key) of

<b>Purpose</b>	Define a keyed list field
<b>Category</b>	Struct member
<b>Syntax</b>	![%]list-name: list(key: key-field) of type
<b>Parameters</b>	<b>!</b> Do not generate this list. For a keyed list, the <b>!</b> is required, not optional.
	<b>%</b> Denotes a physical list. The <b>%</b> option can precede or follow the <b>!</b> .
	<i>list-name</i> The name of the list being defined.
	<i>key-field</i> The key of the list. For a list of structs, it is the name of a field of the struct. For a list of scalar or string items, it is the item itself, represented by the <b>it</b> variable. This is the field or value that the keyed list pseudo-methods check when they operate on the list.
	<i>type</i> The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

Keyed lists are used to enable faster searching of lists by designating a particular field or value to use during the search. A keyed list can be used, for example, in the following ways:

- As a hash table, in which searching only for a key avoids the overhead of reading the entire contents of each item in the list.
- For a list that has the capacity to hold many items, but only contains a small percentage of its capacity, randomly spread across the range of possible items, e.g., a sparse memory implementation.

Besides the **key** parameter, the keyed list syntax differs from the regular list syntax in the following ways:

- a) The list shall be declared with the **!** do-not-generate operator. This means a keyed list needs to be built item-by-item, since it cannot be generated.
- b) The `[exp]` list size initialization syntax is not allowed for keyed lists, i.e., `list[exp]: list(key: key) of type` is not legal. Similarly, **keep** shall not be used to constrain the size of a keyed list.
- c) A keyed list is a distinct type, different from a regular list. This means a keyed list cannot be assigned to a regular list or vice versa, e.g., if `list_a` is a keyed list and `list_b` is a regular list, `list_a = list_b` shall result in an error.

If the same key value exists in more than one item in a keyed list, the keyed list pseudo-methods use the latest item in the list (the one with the highest list index number). Other items with the same key value are ignored. The keyed list pseudo-methods (see 26.7) only work on lists that were defined and created as keyed lists. Conversely, restrictions apply when using regular list pseudo-methods or other operations on keyed lists (see 26.7.4).

Syntax example:

```
!locations: list(key: address) of location;
```

## 6.10 Projecting list of fields

<b>Purpose</b>	Specifying a field from all items in a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.field-name</i>	
<b>Parameters</b>	<i>list</i>	A list of structs.
	<i>field-name</i>	A name of a field or list in the struct type.

This returns a list containing the contents of the specified *field-name* for each item in the *list*. If the *list* is empty, it returns an empty list. This syntax is the same as *list.apply(field)* (see 26.4.1).

An error shall be issued if the *list* is not a list of structs or the struct type does not have a field named *field-name*.

Syntax example:

```
s_list.fld_nm
```

## 6.11 Defining attribute fields

<b>Purpose</b>	Define the behavior of a field when copied or compared	
<b>Category</b>	Unit member	
<b>Syntax</b>	<b>attribute</b> <i>field-name attribute-name = exp</i>	
<b>Parameters</b>	<i>field-name</i>	The name of a field in the current struct.
	<i>attribute-name</i>	<i>attribute-name</i> is one of the following: <ul style="list-style-type: none"> <li>a) <b>deep_copy</b>—controls how the field is copied by the <b>deep_copy()</b> routine.</li> <li>b) <b>deep_compare</b>—controls how the field is compared by the <b>deep_compare()</b> routine.</li> <li>c) <b>deep_compare_physical</b>—controls how the field is compared by the <b>deep_compare_physical()</b> routine.</li> <li>d) <b>deep_all</b>—controls how the field is copied by the <b>deep_copy()</b> routine or compared by the <b>deep_compare()</b> or <b>deep_compare_physical()</b> routines.</li> </ul>
	<i>exp</i>	<i>exp</i> is one of the following: <ul style="list-style-type: none"> <li>a) <b>normal</b>—performs a deep (recursive) copy or comparison.</li> <li>b) <b>reference</b>—performs a shallow (non-recursive) copy or comparison.</li> <li>c) <b>ignore</b>—do not copy or compare.</li> </ul>

Defining attributes controls how a field behaves when it is copied or compared. These attributes are used by **deep\_copy()**, **deep\_compare()**, and **deep\_compare\_physical()**. For a full description of the behavior specified by each expression, see 28.1.1, 28.1.2, or 28.1.3, respectively.

To determine which attributes of a field are valid, all extensions to a unit or struct are scanned in the order they were loaded. If several values are specified for the same attribute of the same field, the last attribute specification loaded is the one that is used.

The **attribute** construct can appear anywhere, including inside a **when** construct or an **extend** construct.

Syntax example:

```
attribute channel deep_copy = reference;
```

1

5

10

15

20

25

30

35

40

45

50

55

1

5

10

15

20

25

30

35

40

45

50

55