

21 Macros

This chapter describes how to extend the *e* language by defining macros:

- “Overview of *e* Macros” on page 21-1
- **define as** on page 21-5
- **define as computed** on page 21-14
- “Match Expressions in *e* Macros” on page 21-22
- “Tracing Macro Expansion” on page 21-31
- “Macro Error Messages” on page 21-32
- “Debugging Macros” on page 21-33

21.1 Overview of *e* Macros

You can extend *e* by adding new language constructs for any of the following syntactic categories:

- Statements
- Struct members
- Actions
- Expressions
- Commands

You can add a new construct by defining a macro. In general, a macro definition specifies:

- A syntactic pattern that defines a new syntactic construct in the language
- A replacement that specifies *e* code containing other, already existing constructs of the same category.

For example, you can create a new kind of action to calculate the maximum value between two numeric expressions. The first expression is an assignable expression such as a variable. The action then assigns the maximum value to the first expression. This new language construct has the following syntax:

```
largest lhs-exp exp
```

The result of this action is that the value of *lhs-exp* becomes the largest of the old value of *lhs-exp* and the value of *exp*. If the value of a variable *x* is 5, then using the new construct as follows

```
largest x 4
```

does not change anything. Using the construct as follows

```
largest x 7
```

changes the value of *x* to 7.

This kind of action can be added by defining the following macro:

```
define <largest' action> "largest <left'exp> <right'exp>" as {  
  if <right'exp> > <left'exp> then {  
    <left'exp> = <right'exp>;  
  };  
};
```

This **define** statement consists of:

- The macro name—<largest' action>— indicates that the new language construct belongs to the syntactic category “action”.
- The match expression—“largest<left'exp> <right'exp>”— specifies that the new construct consists of the keyword **largest** followed by two expressions separated by white space.
- The replacement— {if ...then ...}— specifies that each time a construct matching the match expression is found in *e* code, it is replaced by the specified **if...then** action.

Match expressions in macro definitions specify the syntactic pattern of the new construct, and they contain elements such as:

- Literal characters

In the example above, “largest” and the spaces separating the two expressions are literal characters.

- Syntactic arguments in the form

```
<[tag' ]syntactic-type>
```

where *tag* is an *e* name. In the example above, <left'exp> and <right'exp> are syntactic arguments.

- Regular expression operators such as options and alternatives

The syntactic types allowed in match expressions include the five syntactic categories and some other syntactic types:

```
<statement>
<struct_member>
<action>
<exp>
<command>
<name>
<file>
<num>
<block>
<type>
<any>
```

For a description of these syntactic types, see [Table 21-3 on page 21-23](#). All the possible elements of a match expression are described in detail in [“Match Expressions in e Macros” on page 21-22](#).

The handling of replacement code containing multiple semicolon-separated items by **define as** and **define as computed** macros depends on several factors, and includes the following behavior:

- For macros of the syntactic category `<exp>`, the replacement code must contain exactly one construct of the same category. See `DEPR_MACRO_MULTI_REPLACEMENT_ITEMS` in [Table 2-1 on page 2-2](#) in *Specman Deprecation and Backwards Compatibility*.
- For macros of syntactic categories `<action>`, `<struct_member>`, `<statement>`, and `<command>`, the replacement code can contain multiple numbers of constructs separated by a semicolon. The replacement code can also be empty.
- For `<action>` macros, if there are several semicolon-separated actions in the replacement code, and some of them are **var** actions, the scope of these variables is inside the replacement, and they are not seen outside the macro. However, if there is only one action, and it is a **var** action, the variable defined by it has outer scope and can be used after the macro call.
- For `<command>` macros, if there are multiple semicolon-separated items in the replacement code, it is considered a block of actions. It is impossible to define several non-action commands in one macro. The scoping rules for `<action>` macros also applies to `<command>` macros.

Once a macro is defined, the construct it introduces becomes a legal *e* construct. Each occurrence of such a construct later in the code is replaced by other constructs, as specified by the macro. The macro can be used in the replacement code of other macros.

Any new macro you define takes priority over all previously defined macros for the same category. The definition order is determined by the order of macro definitions within a module and the order in which modules are loaded. This means that if some *e* code matches more than one macro of the specified category and can be treated as more than one language construct of that category, the most recent macro definition is used. In particular, all user-defined constructs have priority over all built-in constructs.

There are two kinds of macros in *e*:

- **define as** macros
- **define as computed** macros

Both kinds of macros add a new construct to the language by defining the *e* code that replace occurrences of the match expression later in the program. But they are different in how they define the replacement. In **define as** macros, the replacement is given as template code. In **define as computed** macros, the replacement code is calculated as a string in a procedural way, just as in a regular routine.

For example, the `<largest'action>` macro can be modified to calculate either maximum and minimum. However, this cannot be done using **define as**, because the replacement code for calculating the minimum is different from the code for calculating the maximum. The replacement code cannot just be given as a template. The macro must return the appropriate code depending on which keyword, `largest` or `smallest`, is used. This macro can be defined using **define as computed**:

```
define <largest_smallest'action>
  "(largest|smallest) <left'exp> <right'exp>" as computed {
    var cmp: string = (<l> == "largest") ? ">" : "<";
    return append("if ", <right'exp>, cmp, <left'exp>, "then {",
                  <left'exp>, "=", <right'exp>, "}");
  };
```

define as computed macros give maximum flexibility in generating the replacement. However, they are more complicated and typically they are less readable.

Another difference is that constructs defined by **define as** macros can be used in the same module in which the macro is defined, after the macro definition itself, but constructs defined by **define as computed** macros become available only after the whole module is loaded, and cannot be used in the same module.

It is better to use **define as** for every construct that can be implemented with a simple syntactic template. Use **define as computed** only when there is no other choice.

See Also

- [“Syntactic Elements” on page 2-13](#)
- [“Match Expressions in e Macros” on page 21-22](#)
- [define as on page 21-5](#)
- [define as computed on page 21-14](#)
- [“Tracing Macro Expansion” on page 21-31](#)

21.2 define as

Purpose

Define a new language construct using template replacement code

Category

Statement

Syntax

define <tag'syntactic-category> “*match-expression*” **as** { *replacement* }

Syntax Example

```
<'
define <largest'action> "largest <exp> <num>" as {
    if <num> > <exp> then {<exp> = <num>};
};
'>
```

Parameters

tag'syntactic-category

The *tag* is any legal *e* name. It must start with a letter and consist of letters, digits and underscores (_). See “[Legal e Names](#)” on page 2-11 for more information.

The *syntactic-category* indicates the syntactic category of the macro and is one of the following keywords:

- *statement*— a top level declarative construct.
- *struct_member*— a part of a struct definition.
- *action*— a procedural construct that can be executed.
- *exp*— a construct that can be evaluated (an expression).
- *command*— a construct that can be entered at Specman command prompt and in ecom files.

The tag and syntactic category in combination must form a unique name for the macro within the package. For example, it is possible to have both a `<do_it'statement>` and a `<do_it'action>`, but there cannot be two `<do_it'action>` macros in the same package. However, it is possible to have macros with the same name in different packages, just as different types in different packages can have the same name.

match-expression

A quoted string containing the syntactic pattern that specifies what constructs are matched by the macro. A match expression is a sequence of literal characters and syntactic arguments, possibly with regular expression operators such as options and alternatives. Once the macro is defined, you can use the syntax specified by the match expression wherever it is legal to use a construct of the same syntactic category as the macro.

Match expression syntax and meaning are described in detail in “[Match Expressions in e Macros](#)” on page 21-22. For a list of syntactic arguments allowed in match expressions, see [Table 21-3 on page 21-23](#).

replacement

Template *e* code, usually containing replacement terms that refer to the elements of the match expression. This code must constitute a legal *e* construct of the same category, either a built-in construct or a construct defined by another macro.

When the macro is used, each replacement term is substituted by the text matched by the corresponding element of the match expression.

The kinds of replacement terms that can be used in the replacement code, and their corresponding match-expression elements, are shown in [Table 21-1](#).

The treatment of multiple semicolon-separated items in replacement code is described in “[Overview of e Macros](#)” on page 21-1.

Table 21-1 Replacement Term Syntax for define as

Replacement Term Kind	Description	Examples
Syntactic argument name	The name of a syntactic argument (with or without repetition) in the match expression. It refers to the text in the match expression that is matched by the syntactic argument. See “ Syntactic Argument Names ” on page 21-25 for more information. Note If you need to refer the separate elements of a repetition argument, you must use define as computed on page 21-14.	<action> <first'exp>

Table 21-1 Replacement Term Syntax for define as

Replacement Term Kind	Description	Examples
Submatch number	<p>A numeric constant enclosed in triangular brackets that refers to the text matched by a grouping, option or syntactic argument within the match expression. For example in the following match expression, <1> represents the entire grouping (including the <exp> and the optional “Hello”), <2> represents the <exp>, and <3> represents the optional “Hello”:</p> <pre>"My (<exp>[Hello])"</pre> <p>If the number is not positive, or if it exceeds the number of submatches in the match expression, a compile time error is issued.</p> <p>See “Submatches” on page 21-28 for more information.</p>	<pre><1> <2></pre>
Submatch label	<p>The submatch label of a grouping or an option in the match expression. It refers to the text matched by that grouping or option. See “Submatch Labels” on page 21-26 for more information.</p>	<pre><OPT> <EXP></pre>
Replacement term with a default value	<p>You can give any of the replacement terms (syntactic argument name, submatch number, submatch label) a default value. The default value appears within the same triangular brackets and is separated from the replacements term by a bar “ ”. The default value is any text that is substituted when the corresponding element does not appear in the input.</p>	<pre><num 0> <2 x></pre>
<?>	<p>An alphanumeric character sequence that is unique over all expansions of this macro. This is useful for creating unique names that do not collide in the various places this macro is used.</p>	<pre>var a<?>: int;</pre>

Description

A **define as** statement creates a new language construct of the specified syntactic category (action, command, and so on).

You give the macro name by assigning it a tag and a syntactic category. The replacement specifies what is to be done each time such a construct occurs in the program or is executed at the Specman prompt.

When Specman expects a construct of that category and finds a construct that matches the match expression of the macro, Specman replaces that construct with the replacement code, substituting replacement terms with the corresponding submatches. This process can be recursive. If the new construct also matches the match expression of some macro, its replacement is used in turn.

Any new macro you define with **define as** or **define as computed** takes priority over all previously defined macros for the same category. The order of definition is determined by the order of their definition within a module and by the order in which modules are loaded.

Example 1

This example defines a new kind of expression that finds the maximum between three given numeric expressions.

```
<'
define <max3'exp> "max3 <first'exp>,<second'exp>,<third'exp>" as {
    max(<first'exp>, max(<second'exp>, <third'exp>))
};
'>
```

Once this macro is loaded, it can be used either from the Specman command prompt or in the code. For example:

```
cmd-prompt> print (max3 1,2,3)
(max3 1,2,3) = 3
cmd-prompt> print (max3 10,15,5)
(max3 10,15,5) = 15
```

Example 2

This example defines a new Specman command that executes the UNIX “ls” command with any of its flags. The new command is invoked by “lis” with a list of flags.:

```
<'
define <dir_lis'command> "lis[ <any>]" as {
    print output_from("ls <any>") using items=UNDEF;
};
'>
```

At the Specman prompt you can enter the following commands to list files in the current directory. The “-lt” flags and the “*.e” syntax are the same as for the UNIX “ls” command:

```
cmd-prompt> lis
cmd-prompt> lis -lt *.e
```

Example 3

This example defines a new action that generates a frame with specified field values, and calls a method named “send()”:

```
<'
define <simple_frame'action> "send simple frame \
    <dest_addr'num> <source_addr'num> <size'num>" as {
    var f: frame;
    gen f keeping {
        .kind not in [SRAM,DUT];
        .size == <size'num>;
        .dest_address == <dest_addr'num>;
        .source_address == <source_addr'num>;
    };
    f.send();
};
'>
```

Here is an example of using the “send simple frame” macro:

```
<'
extend sys {
    run() is also { send simple frame 0x00fe 0x0010 0xfa };
};
'>
```

Example 4

This example defines a new action that generates a frame of a specified “kind” with an optional “dest_address” value, and calls the send() method.

The <name> argument of the match expression denotes the value of the “kind” field of the frame, and the <dest_addr'num> argument denotes the value of the “dest_address” field. If no “dest_address” is given, then the default value 0x1000 is used.

```
<'
define <configure_frame'action>
    "<name> config frame[ <dest_addr'num>]" as {
    var f: frame;
    gen f keeping {
        .kind == <name>;
        .dest_address == <dest_addr'num|0x1000>;
        .size == 64;
    };
    f.send();
};
};
```

```
'>
```

The `<dest_addr'num|0x1000>` item in the replacement has a default value. When a “dest_addr” is given, then it is used. When it is not given, then default value of “0x1000” is used.

Here is a usage example for this macro:

```
<'
extend sys {
    run() is also {
        SRAM config frame;
        DUT config frame 0x1001;
    };
};
'>
```

Example 5

This example defines a new kind of struct member, called “bean”, that adds to a struct a new private field of a given type, as well as the public getter and setter methods for that field.

```
<'
define <bean'struct_member> "bean <name>[ ]:[ ]<type>" as {
    private <name>: <type>;
    get_<name> (): <type> is {
        return <name>;
    };
    set_<name> (val: <type>) is {
        <name> = val;
    };
};
'>
```

Here is a usage example for this macro:

```
<'
struct bean_struct {
    bean x: int;
};

extend sys {
    run () is also {
        var a: bean_struct = new;
        a.set_x(5);
        print a.get_x();
    };
};
'>
```

The “bean x: int” struct member declaration in this example adds both the “x” field of type int to the struct and two methods `get_x()` and `set_x()`. These methods return and modify the value of “x”, respectively.

Example 6

This example defines a pattern for debugging objects that consists of:

- A flag for each instance of a debuggable struct
- A list of debuggable structs per type under `sys`

```
<'
define <debuggable_struct' statement>
"debuggable (struct <name>[ like <parent' name>]) {<struct_member>;...}" as
{
  <1> {
    debug_flag: bool;
    <struct_members>;
  };
  extend sys {
    debug_monitor_<name>: list of <name>;
  };
};

debuggable struct a {
  x: int;
};

'>
```

The struct “a” defined here has a “`debug_flag`” field of type `bool` and an “x” field of type `int`. A list of such structs, called `debug_monitor_a`, is added to `sys`.

The submatch number `<1>` used in the replacement refers to the entire submatch in the parentheses:

```
(struct <name>[ like <parent' name>])
```

A “debuggable struct” statement can specify a parent to the struct.

The `<struct_members>` replacement term refers to the whole repetition argument within the curly brackets.

It is usually recommended to use submatch labels rather than submatch numbers. The above macro can be rewritten as:

```
<'
define <debuggable_struct' statement> "debuggable \
  (<HEAD>struct <name>[ like <parent' name>]) {<struct_member>;...}" as {
```

```

<HEAD> {
    debug_flag: bool;
    <struct_members>;
};
extend sys {
    debug_monitor_<name>: list of <name>;
};
};
'>

```

Example 7

In the following example, the frame struct has a when subtype that adds a Boolean field “good_frame” only for frames of kind SRAM. The macro defines a new expression that checks whether a given frame is of kind SRAM and whether its good_frame value is TRUE.

```

<'
define <good_frame'exp> "<frame'exp> is good frame" as {
    <frame'exp> is a SRAM frame (sram<?>) and sram<?>.good_frame
};

extend sys {
    f1: frame;
    f2: frame;

    run() is also {
        f1 = new SRAM frame with { .good_frame = TRUE; };
        f2 = new DUT frame;
        if (f1 is good frame) or (f2 is good frame) {
            out("One of the frames is good!");
        };
    };
};
'>

```

The <?> element attaches a prefix to the “sram” that is unique for every usage of the macro. Without <?>, the code does not compile. Because there are two uses of the macro within the expression (f1 is good frame) or (f2 is good frame), without <?> the macro generates a variable named “sram” twice in the scope of the same expression.

To see how “sram<?>” is expanded when the macro is used, enter the **trace reparse** command before the example file is loaded. The following is a printout of the results. For each call to the macro, the <?> element is replaced by a unique string.

```
D> <good_frame'exp> 'f1 is good frame'  
D> reparsed as: 'f1 is a SRAM frame (sram__frame_0__) and  
sram__frame_0__.good_frame'  
D> <good_frame'exp> 'f2 is good frame'  
D> reparsed as: 'f2 is a SRAM frame (sram__frame_1__) and  
sram__frame_1__.good_frame'
```

See Also

- [“Overview of e Macros” on page 21-1](#)
- [“Match Expressions in e Macros” on page 21-22](#)
- [define as computed on page 21-14](#)
- [“Tracing Macro Expansion” on page 21-31](#)
- [DEPR_MACRO_MULTI_REPLACEMENT_ITEMS](#) and [DEPR_MACRO_EMPTY_REPLACEMENT](#) in [Table 2-1 on page 2-2](#) in *Specman Deprecation and Backwards Compatability*

21.3 define as computed

Purpose

Define a new language construct using an action block to build the replacement code

Category

Statement

Syntax

```
define <tag'syntactic-category> “match-expression” as computed {action; ...}
```

Syntax Example

```
<'  
define <time_command'action> "time[ <any>]" as computed {  
    if <any> == "on" {  
        return( "tprint = TRUE; print sys.time;" );  
    } else if <any> == "off" {  
        return("tprint = FALSE");  
    } else {
```

```
        error("Usage: time [on|off]");
    };
};
'>
```

Parameters

tag'syntactic-category The *tag* is any legal *e* name. It must start with a letter and consist of letters, digits and underscores (_). See [“Legal e Names” on page 2-11](#) for more information.

The *syntactic-category* indicates the syntactic category of the macro and is one of the following keywords:

- *statement*— a top level declarative construct.
- *struct_member*— a part of a struct definition.
- *action*— a procedural construct that can be executed.
- *exp*— a construct that can be evaluated (an expression).
- *command*— a construct that can be entered at Specman command prompt and in ecom files.

The tag and syntactic category in combination must form a unique name for the macro within the package. For example, it is possible to have both a `<do_it'statement>` and a `<do_it'action>`, but there cannot be two `<do_it'action>` macros in the same package. However, it is possible to have macros with the same name in different packages, just as different types in different packages can have the same name.

match-expression A quoted string containing the syntactic pattern that specifies what constructs are matched by this macro. A match expression is a sequence of literal characters and syntactic arguments, possibly with regular expression operators such as options and alternatives. Once the macro is defined, you can use the syntax specified by the match expression wherever it is legal to use a construct of the same syntactic category as the macro.

Match expressions syntax and meaning are described in detail in [“Match Expressions in e Macros” on page 21-22](#). For a list of syntactic arguments allowed in match expressions, see [Table 21-3 on page 21-23](#).

action;

A block of actions that are executed when the match expression is matched. This action block is treated as the body of a method that returns a string. Thus you must use the **result** variable or the **return** action to return a result from the action block. The resulting string must contain a legal *e* construct of the same category, either built-in or defined by another macro. This construct is used as the replacement code.

The action block can contain replacement terms that refer to elements of the match expression.

When the macro is used, each replacement term is substituted by the text matched by the corresponding match-expression element. Actually, most replacement terms in the action block are expressions of type string.

The kinds of replacement terms that can be used in the action block, and their corresponding match-expression elements, are shown in [Table 21-2](#).

The treatment of multiple semicolon-separated items in replacement code is described in [“Overview of e Macros” on page 21-1](#).

Table 21-2 Replacement Term Syntax for define as computed

Replacement Term Kind	Description	Examples
Syntactic argument name	The name of a syntactic argument without repetition in the match expression. It refers to the text in the match expression that is matched by the syntactic argument. See “Syntactic Argument Names” on page 21-25 for more information.	<code><action></code> <code><first'exp></code>
Repetition syntactic argument name	The name of a repetition syntactic argument in the match expression. Unlike other replacement terms, it is an expression of type list of string , where each element of the list represents the string matched by a separate element of the repetition. If you need to refer to the entire string matched by the whole repetition, use a submatch number or submatch label. To use a submatch label, the entire repetition argument must be enclosed in parentheses in the match expression. See “Syntactic Argument Names” on page 21-25 and “Submatch Labels” on page 21-26 for more information.	<code><exprs></code> <code><my'actions></code>

Table 21-2 Replacement Term Syntax for define as computed

Replacement Term Kind	Description	Examples
Submatch number	<p>A numeric constant enclosed in triangular brackets that refers to the text matched by a grouping, option or syntactic argument within the match expression. For example in the following match expression, <1> represents the entire grouping (including the <exp> and the optional “Hello”), <2> represents the <exp>, and <3> represents the optional “Hello”:</p> <pre style="margin-left: 40px;">"My (<exp>[Hello])"</pre> <p>Notes</p> <ul style="list-style-type: none"> • For repetition arguments, the submatch number refers to the entire string. To refer to the list of strings of a repetition argument, use the repetition syntactic argument name. • If the number is not positive, or if it exceeds the number of submatches in the match expression, a compile time error is issued. <p>See “Submatches” on page 21-28 for more information.</p>	<p><1> <2></p>
Submatch label	<p>The submatch label of a grouping or an option in the match expression. It refers to the text matched by that grouping or option. See “Submatch Labels” on page 21-26 for more information.</p>	<p><OPT> <EXP></p>
<?>	<p>An alphanumeric character sequence that is unique over all expansions of this macro. This is useful for creating unique names that do not collide in the various places this macro is used.</p> <p>Unlike other replacement terms, you must insert this term into the returned string expression. It is not itself recognized as a legal string expression in define as computed. The following example illustrates an illegal use of <?> in define as computed:</p> <pre style="margin-left: 40px;">-- illegal in define as computed return append("var x", <?>, ":", <type>);</pre>	<pre>return append("var x<?>: ", <type>);</pre>

Description

A **define as computed** statement creates a new language construct of a given category (action, command, and so on).

You give the macro name by assigning it a tag and a syntactic category. The action block specifies what is to be done each time such a construct occurs in the program or is executed at the Specman prompt.

When Specman expects a construct of that category and finds a construct that matches the match expression of the macro, Specman executes the action block of the macro, substituting any replacement terms with the corresponding submatches. Then, the string returned by the action block is treated as the code to be executed. This process can be recursive. If the new code also matches the match expression of some macro, its replacement is used in turn.

Any new macro you define with **define as** or **define as computed** takes priority over all previously defined macros for the same category. The order of definition is determined by the order of their definition within a module and by the order in which modules are loaded.

For simple replacements, it is recommended that you use the **define as** statement rather than **define as computed**. **define as computed** macros are more complicated, less readable, and less easy to debug. Use them only when there is no other choice.

When compiling a file that uses a construct defined by a **define as computed** macro, you have two choices:

- By default, you must use two-phase compilation. Compile the file that defines the macro first. Then you can use the compiled executable to load or compile any file that uses this newly defined construct.
- Optionally, you can use the **sn_compile.sh -enable_DAC** option to perform single-phase compilation of **define as computed** macros.

However, if you have the following in the same compilation cluster, single-phase compilation causes an error to be issued.

- A C routine declaration
- An extension of **init()** of an instantiated struct which calls the C routine
- Computed macro definitions (but not necessarily their use), unless they occur in the top module

Notes

- Language constructs defined by **define as computed** macros cannot be used in the same file they are defined in. To use a such a construct in a loaded file, you must also load the file that contains the **define as computed** statement.
- You can debug **define as computed** macros after you set the macro debugging mode to **expansion**.

```
cmd-prompt> config debug -macro_debug_mode=expansion
```

Example 1

This example defines a new “soft_extend” statement, which adds a new item to an existing enumerated type but is more flexible than the usual “extend”. When the type already has such item, it does nothing and does not produce an error message. This might be useful, for example, if you have several different variants of item sets for the enumerated type in different modules, and these modules can be loaded in different combinations and in different order.

Note You can use this macro to extend a particular type by a specific item name only once in a single *e* module. However, you can use this macro multiple times in a single *e* module if you specify a unique item name with each use.

```
soft_extend1.e
<

define <soft_extend' statement>
"soft_extend <type' name> by <item' name>" as computed {
    var rfe: rf_enum =
        rf_manager.get_type_by_name(<type' name>).as_a(rf_enum);
    if rfe == NULL {
        error("There does not exist enum type called ", <type' name>);
    };
    if not rfe.get_items().has(it.get_name() == <item' name>) {
        return appendf("extend %s: [%s]", <type' name>, <item' name>);
    };
};

'>
```

The first syntactic argument in this macro is the name of an enumerated type; the second is the name of the item to be added. Using the reflection interface, the macro checks whether such an enumerated type exists, and if not, issues an error message. Otherwise, again using reflection, the macro checks whether the enumerated type already has the item to be added (<item' name>). If not, the returned string contains an **extend** statement that adds the item. If such an item already exists, the returned string contains a statement that does nothing.

For more information on the reflection interface, see [Chapter 23 “Reflection Interface for e”](#) in the *Usage and Concepts Guide for e Testbenches*.

Example 2

This example improves the macro from the previous example so that it is possible to provide several items to soft_extend, similar to the usual **extend**. The macro checks which items already exist and which do not, and adds only the new ones.

```
soft_extend.e
<
```

```
define <soft_extend' statement>
    "soft_extend <type' name>:[ ]\[\<item' name>,...\]" as computed {
    var rfe: rf_enum =
rf_manager.get_type_by_name(<type' name>).as_a(rf_enum);
    if rfe == NULL {
        error("There does not exist enum type called ", <type' name>);
    };
    var new_names: list of string =
        <item' names>.all(it not in rfe.get_items().apply(it.get_name()));
    return
        appendf("extend %s: [%s]", <type' name>, str_join(new_names, ", "));
};

'>
```

In this macro, the syntactic argument `<type' name>` holds the name of an enumerated type, and a repetition syntactic argument `<item' name>,...` holds a list of item names. The name of the repetition, when used inside the macro body, is in the plural form — `<item' names>`.

If the enumerated type exists, the macro returns a string containing an **extend** statement that adds only those items that do not already exist in the enumerated type.

The following is an example of using this macro. In file `color.e`, an enumerated type called `color` is defined, with three values. In file `color1.e`, this type is extended to add the values `red`, `green` and `blue`, without knowing which values are already defined. As a result, the type `color` has 5 values: `black`, `white`, `green`, `red` and `blue`.

```
color.e
<'

type color: [black, white, green];

'>
color1.e
<'

import color;
import soft_extend;

soft_extend color: [red, green, blue];

'>
```

Example 3

This example changes the macro “`configure_frame`” shown in “[Example 4](#)” on page 21-10, so that if no “`dest_address`” is given, then a generated value is used, rather than a predefined default value.

define as computed is required here because the constraint on the “`dest_address`” field must appear only when “`dest_address`” is given, and must not appear otherwise. **define as** cannot be used here because the code is not a simple template, but must instead be calculated.

```
<'
define <configure_frame'action>
    "<name> config frame[ <dest_addr'num>]" as computed {
    var address_constraint: string;
    if !str_empty(<dest_addr'num>) then {
        address_constraint =
            append(".dest_address == ", <dest_addr'num>, ";");
    };
    return append(" \
        var f: frame; \
        gen f keeping { \
            .kind == ", <name>, "; \
            .size == 64;"
            address_constraint,
            "); \
            f.send(); \
        ");
};
'>
```

The returned string contains the sequence of actions. The **gen** action has a constraint for “`dest_address`” only if `<dest_addr'num>` is non-empty.

See Also

- “[Overview of e Macros](#)” on page 21-1
- **define as** on page 21-5
- “[Match Expressions in e Macros](#)” on page 21-22
- “[Tracing Macro Expansion](#)” on page 21-31
- `DEPR_MACRO_MULTI_REPLACEMENT_ITEMS` and `DEPR_MACRO_EMPTY_REPLACEMENT` in [Table 2-1](#) on page 2-2 in *Specman Deprecation and Backwards Compatability*

21.4 Match Expressions in e Macros

A match expression is used in a macro definition to specify the syntactic structure of the new language construct. Elements of the match expressions can be referred to from the macro body. A macro body is one of the following:

- The replacement code of a **define as** macro
- The action block of a **define as computed** macro

A match expression consists of literal characters and syntactic arguments, possibly with regular expression operators such as alternative, optional subsequence, and a restricted form of repetition. The elements of a match expression are defined in the following sections:

- [“Literal Characters” on page 21-22](#)
- [“Syntactic Arguments” on page 21-23](#)
- [“Regular Expression Operators in Match Expressions” on page 21-25](#)
- [“Submatch Labels” on page 21-26](#)
- [“Special Literal Characters” on page 21-26](#)
- [“Submatches” on page 21-28](#)
- [“Problematic Match Expressions” on page 21-28](#)
- [“Examples” on page 21-29](#)

21.4.1 Literal Characters

Any ASCII character or sequence of characters that should appear in the new construct is specified literally in the match expression string.

The following characters have special meaning in the match expression syntax, and they must be escaped (preceded by a backslash) in order to be taken literally:

() [] < > | \

If any other character is escaped, the backslash is simply ignored.

For example, a match expression for the letter “x”, followed by “<” and then the letter “y”, is:

```
"x\<y"
```

The meaning of literal characters in a match expression is as follows:

- Any character other than white space denotes an appearance of that specific single character in a matching construct.

Some characters have a special meaning and can be used only in specific situations. For more information, see “[Special Literal Characters](#)” on page 21-26.

- The space character in the match expression denotes an appearance of any non-empty number of subsequent white space characters in a matching construct.

White space characters are spaces, tabs and new lines. Subsequent white space characters in *e* code are always collapsed to a single space character (unless they appear inside double quotes). For convenience, it is allowed to put several subsequent spaces in a match expression. They are treated as a single space. The other white space characters (`\n` and `\t`) cannot appear in a match expression.

21.4.2 Syntactic Arguments

Syntactic arguments are the non-literal elements in a match expression, and they denote something within a construct that corresponds to a certain syntactic type.

Syntactic Argument Syntax

A syntactic argument has the following form:

```
<[tag' ]syntactic-type>
```

where:

- The optional *tag* is any legal *e* name. It must start with a letter and consist of letters, digits and underscores (`_`).
- The *syntactic-type* is one of the syntactic categories or other types shown in [Table 21-3](#).

The following are examples of syntactic arguments:

```
<action>
<left' exp>
<param' name>
<any>
```

[Table 21-3](#) lists the syntactic categories and other syntactic types that can be used as arguments in match expressions. See “[Syntactic Elements](#)” on page 2-13 for more information on syntactic categories.

Table 21-3 Syntactic Types in Match Expressions

Name	Description
statement	A top level declarative construct.
action	A procedural construct that can be executed.

Table 21-3 Syntactic Types in Match Expressions

Name	Description
command	A construct that can be entered at Specman command prompt and in ecom files.
struct_member	A part of a struct definition.
exp	An expression— a construct that can be evaluated.
name	A legal <i>e</i> name. It must start with a letter and consist of letters, digits and underscores (_).
file	A file name, possibly with a UNIX-style file path.
num	A literal numeric constant.
block	A series of actions, delimited by semicolons and enclosed between curly braces. Note For defined as computed macros, if you need to access the individual actions defined in the block, rather than the block as a whole (especially in a define as computed macro), use {<action>;...} instead of <block> in the match expression, and then you can use <actions> in the macro body.
type	A legal <i>e</i> type name, simple or compound.
any	Any non-empty sequence of characters.

Repetition Syntactic Argument Syntax

A repetition syntactic argument has the following form:

<[*tag*']*syntactic-type*>*separator*...

where:

- The optional *tag* is any legal *e* name. It must start with a letter and consist of letters, digits and underscores (_).
- The *syntactic-type* is one of the syntactic categories or other syntactic types shown in [Table 21-3](#).
- The *separator* is any single literal character except special characters and alphanumerics. For a list of special characters, see “[Special Literal Characters](#)” on page 21-26.

A repetition syntactic argument in a match expression denotes multiple occurrences of a syntactic type, separated by a specified separator character. For example, the following denotes a sequence of expressions separated by a comma:

```
<exp>, . . .
```

Note When the repetition appears inside parentheses, square brackets or curly brackets, then an empty string (repetition with zero elements) is also matched. Otherwise, an empty string is not matched, and there must be at least one occurrence of the syntactic type.

Syntactic Argument Names

The syntactic argument name is the name you use in the macro body to refer to the syntactic argument. In case of a regular syntactic argument (without repetition), its name is as it appears in the match expression. For repetition syntactic arguments, the syntactic type part of the name is written in the plural form. For example, in the following match expression:

```
"my <exp> <other'exp> \(<repeated'exp>, . . .\)"
```

the names of the three syntactic arguments are:

```
<exp>
<other'exp>
<repeated'exps> --note the plural form
```

21.4.3 Regular Expression Operators in Match Expressions

A match expression is a regular expression over literal characters and syntactic arguments, and can include the following regular expression operators:

- Grouping—Items enclosed in parentheses

Grouping has no effect on the syntax defined by the macro, but is used for associativity, for readability, or for referencing a submatch. For example, the following denotes a name, followed by a colon, a white space, and a type:

```
(<name>: <type>)
```

- Option—Items enclosed in square brackets.

Enclosing an item or a sequence of items in square brackets denotes an optional occurrence of that item or sequence. For example, the following denotes an optional occurrence of two expressions (constructs of the category *exp*) with two asterisks between them:

```
[<first'exp>**<second'exp>]
```

- Alternative—A number of items, separated by a bar “|”.

A bar denotes a choice between several options. In other words, the matching text must include exactly one of them. For example, either the word “Hello” or an expression matches the following:

```
Hello|<exp>
```

21.4.4 Submatch Labels

You can give groupings and options in a match expression a submatch label. A submatch label serves as a symbolic name for the text matched by the grouping or the option, and lets you refer to it from the macro body by name rather than by number.

A submatch label has the following form:

```
(<LABEL>item...)
```

```
[<LABEL>item...]
```

where

LABEL is a sequence of capital letters and underscores only, beginning with a capital letter.

For example, the following grouping has submatch label <WORD>:

```
(<WORD>Hello|Bye) .
```

In this example, the option has submatch label <OPT_EXP>:

```
[<OPT_EXP><first' exp>**<second' exp>]
```

See Also

- [“Submatches” on page 21-28](#)

21.4.5 Special Literal Characters

Parentheses, square brackets, curly brackets, double quotes and semicolons have a fixed syntactic function in *e*. They cannot appear just anywhere as literal characters in a match expression. They must appear only as described below.

21.4.5.1 Parentheses, Square Brackets and Curly Brackets

These characters signify subordination of one construct to another unless they appear inside double quotes. In a match expression they must

- Appear in balanced pairs. (For example, a left parenthesis cannot appear without a right parenthesis)

- Enclose a single syntactic argument (with or without repetition)

The following match expression is legal and denotes the word “Hello”, followed by a comma, a white space, and an expression enclosed in parentheses:

```
"Hello, \(<exp>)"
```

The following match expression is also legal and denotes a sequence of actions separated by commas, enclosed in curly brackets.

```
"{<action>,...}"
```

The following match expression is illegal because has a left curly bracket without a right bracket:

```
"abc { 123"
```

The following match expression is illegal because what appears inside the parentheses is not a syntactic argument:

```
"\ (Hello\)"
```

21.4.5.2 Double Quotes

Double quotes can be used in only one way—to enclose the syntactic type “any”. Thus, the only legal use of double quotes as literals is as follows:

```
\ "<any>"
```

Because `<any>` matches any sequence of characters, the example above matches anything enclosed in double quotes.

Of course, the syntactic argument `<any>` can have a tag, for example:

```
\ "<some' any>"
```

21.4.5.3 Semicolons

A semicolon can be used only as the separator of a repetition syntactic argument inside curly brackets. Any other use of a semicolon in a match expression is illegal.

The following match expression is legal:

```
"{<command>;...}"
```

The following match expressions are illegal:

```
"Hello;world"  
"\ [<exp>;... \]"
```

21.4.6 Submatches

When a match expression matches a string in the code, certain substrings of that string are submatches of that match. The submatches are identified by the following match expression elements:

- Groupings
- Options
- Syntactic arguments (with or without repetition)

If any of the above is not actually matched, then the corresponding submatch is empty. For example, if a syntactic argument appears inside an option in the match expression, but does not appear in the matched string, then the corresponding submatch is empty.

Submatches are numbered by their order of occurrence in the match expression, from left to right, starting from 1. For example in the following match expression, the entire grouping (including the `<exp>` and the optional “Hello”) is submatch number 1, the `<exp>` is submatch number 2, and the optional “Hello” is submatch number 3:

```
"My (<exp>[Hello])"
```

You can use submatch numbers to refer to submatches from the macro body. However, it is better to use submatch labels and syntactic argument names instead.

Note The maximum total number of allowed submatches in a macro is 50.

21.4.7 Problematic Match Expressions

Following are the causes of some common problems with match expressions:

- Specifying literal backslash characters incorrectly in match expressions

Within a macro definition, a match expression appears as an *e* literal string inside a pair of double quotes. That string is first treated by the Specman parser as any usual string, including the special treatment of escaped characters such as “\n” or “\””, and only then is passed on to the match expression parser. This might cause confusion, in some cases. For example, if you write a macro to match a backslash followed by the letter “a”, you have to write it three times rather than two:

```
"\\a"
```

and if it is followed by the letter “t”, you have to write the backslash four times:

```
"\\t"
```

See “[Literal String](#)” on page 2-9 for more information on the escape sequences allowed in *e* strings.

- Failing to escape double quotes in match expressions

A non-escaped double quote is treated as the closing quote of the match expression itself, rather than a literal within it.

- Failing to escape literal parenthesis and square brackets

Parentheses and square brackets need to be escaped in the match expression to be taken as literals. Otherwise, they are taken as grouping and option operators.

- Attempting to match two or more subsequent spaces

Such usage is never matched. For example, the following match expression:

```
"Hello, [ world]!"
```

matches “Hello, !” but does not match “Hello, world!”.

- Placing white spaces at the beginning or end of a match expression

Because Specman removes any white spaces preceding or trailing a construct before attempting to match it, no construct matches a match expression with spaces at either end of it. For example, the following match expression is illegal:

```
" Hello, world!"
```

Empty match expressions are also illegal.

- Failing to use a line continuation character in multi-line match expressions

To continue a match expression over multiple lines, put the line continuation character “\” at the end of each line except the last, just as you do with any string in *e*.

- Failing to use unique names for syntactic arguments

Any syntactic argument referenced from the macro body must have a name that is unique within the same match expression. However, it is not necessary for each referenced syntactic argument to have a tag. For example, the syntactic arguments `<exp>` and `<other'exp>` have unique names.

Also, if some syntactic arguments are not referenced from the macro body, they can have the same name. For example, it is OK to use `<exp>` without a tag more than once in the same match expression if neither expression is referenced from the macro body.

See Also

- [“Special Literal Characters” on page 21-26](#)
- [“Regular Expression Operators in Match Expressions” on page 21-25](#)

21.4.8 Examples

Below are some examples for legal match expressions.

Example 1

This match expression matches a construct that consists of an optional keyword, either “private” or “public”, followed by white spaces, then a name followed by a colon, optional white spaces, a type, and then optionally followed by an equal sign, possibly surrounded by white spaces, and an expression.

```
"[<MOD>(private|public) ]<name>:[ ]<type>[[ ]=[ ]<exp>]"
```

The label “<MOD>” is a submatch label for the optional “private” or “public” keyword, including the white space after them.

Note The space at the end of the first option cannot be replaced by a space outside the option, because then the match expression does not match an input without one of those keywords.

Some inputs that are matched by this match expression:

```
x:int
private y: uint
public abc:string ="Hello"
```

Example 2

This match expression matches a construct that consists of a list of expressions separated by commas and enclosed in square brackets, followed by white spaces, anything within double quotes, and another sequence of expressions separated by white spaces.

```
"\[<exp>, ... \] \["<any>\["<more' exp> ..."
```

Some inputs that are matched by this match expression:

```
[ x : y+5] "Hello, world" 5 6 7
[ ] "xyz"i j
```

See Also

- [Overview of e Macros](#) on page 21-1
- [define as](#) on page 21-5
- [define as computed](#) on page 21-14
- “Tracing Macro Expansion” on page 21-31

21.5 Tracing Macro Expansion

To see how macros are expanded by the parser:

1. Type the following Specman command at the *cmd-prompt*> prompt:

```
trace reparse
```

2. If necessary, reload the *e* code that contains the macro usage that you want to see expanded.

For commands typed at the prompt and for files loaded into Specman, any macro expansion that is done after you enter the **trace reparse** command is printed to the screen.

Below are shown the results of using the macro <max3'exp> defined in “Example 1” on page 21-9.

```
cmd-prompt> var x: int
cmd-prompt> x = max3 1, 2, 3
D> <max3'exp> 'max3 1, 2, 3'
D> reparsed as: 'max(1, max( 2,  3))'
cmd-prompt> print x
x = 3
cmd-prompt> x = max3 10, 15, 5
D> <max3'exp> 'max3 10, 15, 5'
D> reparsed as: 'max(10, max( 15,  5))'
cmd-prompt> print x
x = 15
```

Note **trace reparse** shows how macros are actually being expanded during parsing and works for commands as well as loaded *e* code. However, you can also use the **show macro_call** command at run time to display macro expansion for loaded *e* code. Use **show macro_call** for run-time macro debugging.

See Also

- “Overview of *e* Macros” on page 21-1
- **define as** on page 21-5
- **define as computed** on page 21-14
- “Match Expressions in *e* Macros” on page 21-22
- **show macro** on page 17-2 in the *Specman Command Reference*

21.6 Macro Error Messages

When a macro definition is loaded, Specman does not check the correctness of the replacement code itself, except for the correctness of the replacement terms and—only for **define as** macros—some general problems, like unbalanced parentheses. Most errors in the replacement code are discovered and reported only when the macro is actually expanded, at load/compile time or run time

When an error is caused by *e* code, and that *e* code is the result of a macro expansion, then the error message includes a reference to the macro definition code, as well as a reference to the place where the macro was expanded.

Example 1 **define as** Macro

In the case of a **define as** macro, the error message identifies the problematic line within the macro definition code.

macro_definition.e

```
<'
define <bean'struct_member> "bean <name>[ ]:[ ]<type>" as {
  private <name>: <type>;
  get_<name> (): <type> is {
    return <name>;
  };
  set_<name> (val: <type>) is {
    <name> = value;           --error here at line 8
  };
};
'>
```

macro_usage.e

```
<'
import macro_definition;

extend sys {

  bean a_number: int;       --macro expanded here at line 6

};
'>
```

Results

```
Specman> load macro_usage
Loading macro_definition.e (imported by macro_usage.e) ...
read...parse...update...patch...h code...code...clean...
```

```

Loading macro_usage.e ...
read...parse...update...patch...h code...code...
  *** Error: No such variable 'value'
          at line 8 in macro_definition.e
  <name> = value;
          expanded at line 6 in macro_usage.e
  bean a_number: int;

```

Example 2 **define as computed Macro**

In the case of a **define as computed** macro, the error message points to the line where the macro definition begins, not to any problematic line within the definition. For example, if you call the ‘config frame’ macro as defined below from the Specman prompt, you see the following message:

```

Specman> SRAM config frame
*** Error: 'gen' action cannot be invoked interactively, only from a method
    in code generated by macro defined at line 2 in @macro_definition
define <configure_frame'action>
    expanded from command

```

macro_definition.e

```

<'
define <configure_frame'action>
    "<name> config frame[ <dest_addr'num>]" as computed {
    var address_constraint: string;
    if !str_empty(<dest_addr'num>) then {
        address_constraint =
            append(".dest_address == ", <dest_addr'num>, ";");
    };
    return append("{ \
        var f: frame; \
        gen f keeping { \
            .kind == ", <name>, "; \
            .size == 64;";
            address_constraint,
            "}; \
            f.send(); \
        } ");
    };
'>

```

21.7 **Debugging Macros**

Debugging macros is very similar to debugging methods in *e* source code. Debugging of nested macros, the macro call chain stack, and both **define as** and **define as computed** macros are supported. Each macro call appears to open a new frame, enabling you to follow the history of calls.

For more information on the commands available for debugging macros, see:

- **configure debugger** on page 6-18 in the *Specman Command Reference*
- **step** on page 16-4 in the *Specman Command Reference*
- **next** on page 16-6 in the *Specman Command Reference*
- **finish** on page 16-7 in the *Specman Command Reference*
- **break on call** on page 18-2 in the *Specman Command Reference*
- **break on line** on page 18-14 in the *Specman Command Reference*
- **stack** on page 22-10 in the *Specman Command Reference*
- **show macro** on page 17-2 in the *Specman Command Reference*

There are two macro debugging modes.

The **definition** macro debugging mode is the default macro debugging mode. It enables debugging of the macro's original source code.

The **expansion** macro debugging mode enables debugging of the macro's expanded code.

You can change the macro debugging mode during run-time or during debugging by any of the following methods:

- Use the **config debugger** command to select the macro debugging mode. For example:

```
cmd-prompt> config debug -macro_debugging_mode=expansion
```

- When in GUI mode, you can select the macro debug mode by using the **Macro Debug Mode** button or the **Tools » Macro Debug Mode** command in the debugger GUI.

Notes

- You must use the **expansion** macro debugging mode to debug **define as computed** macros.
- Breakpoints on expanded macro code are relevant only for the specified macro expansion, and do not apply to the macro code in general. Breakpoints on the original macro code apply to all of its macro expansions.
- While in **definition** macro debugging mode, using the **step** command on a **define as computed** macro call, execution continues until stopping on the next debuggable action line (either a call from inside the macro, or the next line after the macro call).
- While in **expansion** macro debugging mode, when execution hits a breakpoint defined in the macro definition code, the debugger stops on the matching line in the macro's expanded code.

- While in **definition** macro debugging mode, when execution hits a breakpoint defined in the macro expansion code, the debugger stops on the matching line in the definition code. If the macro was a **define as computed** macro, the debugger would stop on the macro call.

Example 1 Using the Debug Commands in a Macro

This example demonstrates the use of the **step**, **next**, **finish**, and **break on line** debugger commands.

```
<'
  define <send_simple_frame'action> "send simple frame" as {
    var f: frame;    // line 3
    gen f;          // line 4
    f.send();
  };    // line 6
extend sys {
  run() is also {
    send simple frame;    // line 9
  };
};
'>
```

During debug, if you stop on the call to `send simple frame` in line 9, the **step** command directs debugger focus to the first action in the `send_simple_frame` macro at line 3. At this point, the **step** or **next** command will direct debugger focus to the next action in line 4. The **finish** command causes execution to continue to the end of the macro, where it stops after executing the last action (in line 6). The **stack up** command shows the call to `send simple frame` in line 9.

If you issue the command

```
(stopped) cmd-prompt> break on line 3
```

each call to `send simple frame` causes execution to stop at line 3.

Example 2 Debugging Nested Macros

This example shows the debug flow used with nested macros.

```
...
<'
  define <send_simple_frames'action> "send <num> simple frames" as {
    for i from 1 to <num> do {
      send simple frame;    // line 12
    };
  };
};
...
'>
```

Nested macro calls are seen in the call chain stack. When you **step** into the `send_simple_frames` macro, and then **step** into `send_simple_frame` in line 12, a new macro call is added to the debugger's call stack. Use the **stack up** and **stack down** debugging command to browse the call chain.

Example 3 The break on ... Commands and the Macro Debugging Modes

This example shows the use of the **break on line** and **break on call** commands. Also shown is the difference between the macro debugging modes **definition** and **expansion**.

```
<'
    define <bean'struct_member> "bean <name>[ ]:[ ]<type>" as {
        private <name>: <type>;
        get_<name> (): <type> is {
            return <name>; // line 5
        };
        set_<name> (val: <type>) is {
            <name> = val;
        };
    };

    struct bean_struct {
        bean x: int;
    };
'>
```

A method defined inside a macro can be debugged.

Using the command **break on line 5**, would set a breakpoint on the line within the macro, and the debugger would stop before each execution of that line.

Using the command **break on call *.get***, would set a breakpoint on the first action in `get_x()`.

Using the **definition** macro debugging mode, stepping into the `get_x()` method would step into the original code. Note that **definition** is the default macro debugging mode.

Using the **expansion** macro debugging mode, the expanded code for debugging would be:

```
get_x(): int is {
    return x;
};
```

Example 4 Debugging Macro Expansion Code

This example shows the use of the **expansion** macro debugging mode and the use of the **break on line @#expansion_index** command.

```
// time_cmd_macro_def.e
```

```

<'
  define <time_command'action> "time[ <any>]" as computed {
    if <any> == "on" {
      return("{tprint = TRUE; print sys.time;}");
    } else if <any> == "off" {
      return("tprint = FALSE");
    } else {
      error("Usage: time [on|off]");
    }
  };
};
'>

// time_cmd_macro_use.e
<'
  extend sys {
    tprint: bool;
    run() is also {
      time on; // line 6
      time off; // line 7
      time on; // line 8
    };
  };
'>

```

Using the **show macro** command we can obtain information on the macro expansions in the code:

```

cmd-prompt> show macro in line 6 @time_cmd_macro_use

code:                "time on"
macro:                <time_command'action>
match expression:    "time[ <any>]"

submatch name      matched sub-string
-----
<any>              "on"

expansion #1
-----
1    {
2      tprint = TRUE;
3      print sys.time;
4    };

cmd-prompt> show macro of line 7 @time_cmd_macro_use

code:                "time off" at line 7 in @time_cmd_macro_use

```

```
macro:          <time_command'action> at line 3 in @time_cmd_macro_def
match expression: "time[ <any>]"

submatch name   matched sub-string
-----
<any>           "off"

expansion #2
-----
1      tprint = FALSE;
```

Stepping into lines 5 and 6 steps into different expansion code.

Enable debugging of a macro's expanded code:

```
cmd-prompt> config debug -macro_debug_mode=expansion
```

Now, using the command:

```
cmd-prompt> break on line 3 @#1
```

causes the debugger to stop on line 3 in the macro call on source line 6 only, and not on source line 8, even though their expansions are identical.

See Also

- [show macro on page 17-2](#) in the *Specman Command Reference*
- [configure debugger on page 6-18](#) in the *Specman Command Reference*
- [break on line on page 18-14](#) in the *Specman Command Reference*
- [break on call on page 18-2](#) in the *Specman Command Reference*
- [“The Debugger GUI” on page 4-4](#) in the *Usage and Concepts Guide for e Testbenches*