

# Writing CLI Macros

The ERX system has an embedded macro language that enables you to define and run macros that will generate and execute CLI commands. Macro files—identified by the *.mac* extension—can be used to store more than one macro. Depending on your needs, you might want to store all of your macros in one file, group macros by function, or store only one macro per file.

Topic	Page
Writing Macros	7-1
Running Macros	7-16
Practical Examples	7-19

## Writing Macros

---

You must write macros on your computer, not on the ERX system. The macros can contain loops, variables, string and numeric values, and conditional statements. Macros can invoke other macros (as long as they are contained within the same macro file), including themselves, but infinite recursion is not permitted. Macros are case-insensitive.

Macros consist of *control expressions* and *noncontrol expressions*. Control expressions are enclosed by *control brackets*, which are angle-bracket and number sign pairs, like this: `<# controlExpression #>`. Examples of control expressions include the macro name and macro end statements, and *while* loops. A control expression can include multiple operation statements if you separate the statements with semicolons (;). For example:

```
<# i:=0; while i++ < 3 #>
```

All macros must have names consisting only of letters, numbers, and the underline character (  ). The first character of a macro name cannot be a number. If you include more than one macro within a macro file, each macro must have a unique name. The first line of a macro defines the macro's name:

```
<# macroName #>
```

Noncontrol expressions are not enclosed by control brackets and simply become part of the generated CLI command text.

You must end all macros with the following control expression:

```
<# endtmpl #>
```

You can add comments to your control expressions to clarify the code by prefacing the comment with forward slashes (`//`) inside the control brackets:

```
<# endtmpl //A comment in the macro end expression #>
```

Text after the `//` is ignored when the macro is run and is not displayed by the CLI.

You can also add comments outside the control expressions by prefacing the comment with an exclamation point (`!`). The CLI displays these comments if you use the **test** or **verbose** keywords with the **macro** command; the CLI never regards these comments as commands.

```
!This is a comment outside any control expression
```

You can improve the readability of a macro by using tabs to indent expressions. Leading and trailing tabs have no effect on the macro output, because they are removed when the macro is run.

**Example** The following is a simple macro that you can use to configure the IP interface on the Fast Ethernet port of the SRP module after you have restored the factory defaults:

```
<# ipInit #>
<# ipAddress := env.getline ("IP Address of System?") #>
ena
conf t
int f0/0
ip addr <# ipAddress; '\n' #>
ip route 10.0.0.0 255.0.0.0 192.168.1.1
host pk 10.10.0.166 ftp
<# endtmpl #>
```

## Environment Commands

Macros use environment commands to write data to the macro output, to determine a value, or to call other commands. Table 7-1 describes the environment commands that are currently supported.

**Table 7-1** Environment commands

Command	Description
<code>env.delay(int <i>delay</i>)</code>	Causes the macro to delay further execution for the number of seconds specified by <i>delay</i>
<code>env.getLine</code>	Prompts the user with a question mark (?) and waits for a response
<code>env.getLine(string <i>prompt-string</i>)</code>	Prompts the user with the value of <i>prompt-string</i> and waits for a response
<code>env.getLineMasked</code>	Prompts the user with a question mark (?), waits for a response, and echoes the response with an asterisk (*) for each character entered by the user
<code>env.getLineMasked(string <i>prompt-string</i>)</code>	Prompts the user with the value of <i>prompt-string</i> , waits for a response, and echoes the response with an asterisk (*) for each character entered by the user
<code>env.argc</code>	Returns the number of arguments passed to the macro
<code>env.argv(n)</code>	Returns the value of the <i>n</i> th argument, such that $1 \leq n \leq \text{env.argc}$ The returned value is a string, not a number; if you want to use this value for a subsequent numeric operation, you must first convert it to a number with the <code>env.atoi(<i>string</i>)</code> command
<code>env.argv(0)</code>	Returns the name of the macro
<code>env.atoi(<i>string</i>)</code>	Converts the specified string to a numeric value
<code>env.atoi(env.argv(n))</code>	Converts input values to integers

## Variables

A local variable enables you to store a value used by the macro while it executes. The macro can modify the value during execution. Local variables can be integers, real numbers, or strings. The initial value of local variables is zero.

Like macros, local variables must have a name consisting only of letters, numbers, or the underline character (`_`). The variable name must not begin with a number. You must not use a reserved keyword as a variable name.

*Literals*

A literal is an exact representation of numeric or string values. Every number is a literal. Place single or double quotation marks around a string to identify it as a string literal. You can specify special characters within a literal string by prefacing them with a backslash as follows:

quotation mark	\'
double quotation mark	\"
tab	\t
carriage return	\r
new line	\n
string end	\0
backslash	\\

**Examples**

```
42
98.6
'string literal'
"count"
"\t this string starts with a tab and ends with a tab \t"
```

*Operators*

You can use operators to perform specific actions on local variables or literals, resulting in some string or numeric value. Table 7-2 lists the available macro operators in order of precedence by operation type. Operators within a given row are equal in precedence.

**Table 7-2** Macro operators

Operation Type	Operators						
Extraction	substr()	rand()	round()	truncate()			
String	\$						
Multiplicative	*	/	%				
Arithmetic	+	-	++	--			
Relational	<	>	<=	>=	=	!=	
Logical		&&	!				
Assignment	:=						
Miscellaneous	[ ]	,	( )	.	;	<#	#>

Table 7-3 briefly describes the action performed by each operator.

**Table 7-3** Operator actions

Operation	Operator	Action
Arithmetic (binary)	+	Adds the right and left sides together
Arithmetic (binary)	-	Subtracts the element to the right of the operator from the element to the left of the operator
Assignment	:=	Evaluates the elements to the right of the operator, then assigns that value to the local variable to the left of the operator
Combine	\$	Creates a new string by joining the values of the right and left sides; converts any numeric values to strings before joining
Less than	<	Evaluates as true (returns a 1) if the element to the left of the operator is <i>less than</i> the expression to the right of the operator; otherwise the result is false (0)
Greater than	>	Evaluates as true (returns a 1) if the element to the left of the operator is <i>greater than</i> the expression to the right of the operator; otherwise the result is false (0)
Less than or equal to	<=	Evaluates as true (returns a 1) if the element to the left of the operator is <i>less than or equal to</i> the expression to the right of the operator; otherwise the result is false (0)
Greater than or equal to	>=	Evaluates as true (returns a 1) if the element to the left of the operator is <i>greater than or equal to</i> the expression to the right of the operator; otherwise the result is false (0)
Equal to	=	Evaluates as true (returns a 1) if the element to the left of the operator is equivalent to the expression to the right of the operator; otherwise the result is false (0)
Not equal to (logical NOT)	!=	Evaluates as true (returns a 1) if the element to the left of the operator is not equal to the expression to the right of the operator; otherwise the result is false (0)
Logical OR		Evaluates as true (returns a 1) if the values of either the left or right sides is nonzero; evaluation halts at the first true (1) expression
Logical AND	&&	Evaluates as true (returns a 1) if the values of the left and right sides are both nonzero; evaluation halts at the first false (0) expression
Miscellaneous	[ ]	See <i>Invoking Other Macros</i> in this chapter for usage.
Miscellaneous	,	See <i>While Constructs</i> in this chapter for usage.
Miscellaneous	( )	Groups operands and operators to achieve results different from simple precedence; effectively has the highest precedence

**Table 7-3** Operator actions (continued)

Operation	Operator	Action
Miscellaneous	.	Provides access to environment commands; see Table 7-1. Provides access to macros; see <i>Invoking Other Macros</i>
Miscellaneous	;	Separates operation statements within a control expression
Miscellaneous	<# #>	Encloses control expressions
Multiplication	*	Multiplies the expression to the left of the operator by the expression to the right
Division	/	Divides the expression to the left of the operator by the expression to the right
Modulo	%	Divides the expression to the left of the operator by the expression to the right and returns the integer remainder. If the expression to the left of the operator is less than the expression to the right, then the result is the expression to the left of the operator.
Postincrement	++	Increments the variable after the expression is evaluated
Postdecrement	--	Decrements the variable after the expression is evaluated
Preincrement	++	Increments the variable before the expression is evaluated
Predecrement	--	Decrements the variable before the expression is evaluated
Negation	!	Reverses the logical state of its operand. 0 is returned for nonzero operands. 1 is returned for operands that evaluate to zero.
Arithmetic (unary)	+	Provides the absolute value of the value
Arithmetic (unary)	-	Provides the inverse of the value
Substring	substr()	Extracts a portion of a string
Randomize	rand()	Generates a random integer between the provided endpoints, inclusive
Round	round()	Rounds the value to the nearest integer
Truncate	truncate()	Truncates a noninteger value to the value left of the decimal point

### Assignment

Use the assignment operator (`:=`) to set the value of a local variable. The expression to the right of the operator is evaluated, and then the result is assigned to the local variable to the left of the operator. The expression to

the right of the operator can include the local variable if you want to modify its current value.

**Example**

```
<# i := i + 1 #>
<# count := count - 2 #>
```

### Increment and Decrement

You can use the increment operator (`++`) to increase the value of a local variable by one. You specify when the value is incremented by the placement of the operator. Incrementing occurs after the expression is evaluated if you place the operator to the right of the operand.

Incrementing occurs before the expression is evaluated if you place the operator to the left of the operand.

**Example 1**

```
<# i := 0; j := 10 #>
<# j := j - i++ #>
```

In Example 1, the result is that *i* equals 1 and *j* equals 10, because the expression is evaluated ( $10 - 0 = 10$ ) before *i* is incremented.

**Example 2**

```
<# i := 0; j := 10 #>
<# j := j - ++i #>
```

In Example 2, the result is still that *i* equals 1, but now *j* equals 9, because *i* is incremented to 1 before the expression is evaluated ( $10 - 1 = 9$ ).

Similarly, you can use the decrement operator (`--`) to decrement local variables. Placement of the operator has the same effect as for the increment operator.

When a local variable with a string value is used with the increment or decrement operators, the value is permanently converted to an integer equal to the length in characters of the string value.

### String Operations

The combine operator (`$`) concatenates two strings into one longer string. Numeric expressions are converted to strings before the operation proceeds. The variable *local* evaluates to “want a big”:

```
<# local := "want a " $ "big" #>
```

### Extraction Operations

The extraction operations are substring (`substr`), randomize (`rand`), round, and truncate. These operators are equal in precedence, and all take precedence over the string operator.

You can use the substring operator (`substr`) to extract a shorter string from a longer string. To use the substring operator, you must specify the source string, an offset value, and a count value. You can specify the string directly, or you can specify a local variable that contains the string. The offset value indicates the place of the first character of the substring to be extracted; "0" indicates the first character in the source string. The count value indicates the length of the substring. If the source string has fewer characters than the sum of the offset and count values, then the resulting substring has fewer characters than indicated by the count value.

### Example

```
<# local := "want a " $ "big" $ " string" #>
<# substr(local, 5, 12) #>The result is "a big string"
<# substr(local, 0, 10) #>The result is "want a big"
<# substr("ready", 0, 4) #>The result is "read"
```

The random operator produces a random integer value from the specified inclusive range; in the following example, the result is between 1 and 10:

```
<# number:= rand(1,10) #>
```

The round operator rounds off the number to the nearest integer:

```
<# decimal:= 4.7 #>
<# round(decimal) #>The result is decimal is now 5
```

The truncate operator truncates noninteger numbers to the value left of the decimal point:

```
<# decimal:= 4.7 #>
<# truncate(decimal) #>The result is decimal is now 4
```

## Arithmetic Operations

The arithmetic operations are multiply (\*), divide (/), modulo (%), add (+), and subtract (-). Multiply, divide, and modulo are equal in precedence, but each has a higher precedence relative to add and subtract. Add and subtract are equal in precedence.

### Example

```
<# 4 % 3 + 12 - 6 #>The result is 7
```

When a local variable with a string value is used with arithmetic operators, the value is temporarily converted to an integer equal to the length in characters of the string value. You can use the `env.atoi` commands to avoid this situation.

## Relational Operations

The relational operations compare the value of the expression to the left of the operator with the value of the expression to the right. The result of

the comparison is 1 if the comparison is true and 0 if the comparison is false.

If the expressions on both sides of the operator are strings, they are compared alphabetically. If only one expression is a string, the numeric value is used for comparison. Arithmetic operators have a higher precedence.

**Example**

```
<# i := 9; i++ < 10 #>The result is 1
<# i := 9; ++i < 10 #>The result is 0
```

### Logical Operations

You can use the logical operators AND (&&), OR (||), and NOT (!) to evaluate expressions. The result of the operation is a 1 if the operation is true and 0 if the operation is false.

For the logical AND, the result of the operation is true (1) if the values of the expressions to the left and right of the operator are both nonzero. The result of the operation is false (0) if either value is zero. The evaluation halts when an expression is evaluated as zero.

For the logical OR, the result of the operation is true (1) if the values of the expression on either the left or right of the operator is nonzero. The result of the operation is false (0) if both values are zero. The evaluation halts when an expression is evaluated as nonzero.

The NOT operator must precede the operand. The operation inverts the value of the operand; that is, a nonzero expression becomes 0, and a zero expression becomes 1. For the logical NOT, the result of the operation is true (1) if it evaluates to zero, or false if it evaluates to nonzero.

**Example**

```
<# i := 6; i >= 3 && i <= 10 #>The result is 1
<# i := 1; i >= 3 && i <= 10 #>The result is 0
<# i := 6; i >= 3 || i <= 10 #>The result is 1
<# i := 1; i >= 3 && i <= 10 #>The result is 0
<# i := 5; !i #>The result is 0
<# i := 5; j := 0; !i && !j #>The result is 0
<# i := 5; j := 0; !i || !j #>The result is 1
```

Relational operators have a higher precedence than logical AND and OR. The NOT operator is equal in precedence to the increment and decrement operators.

### Miscellaneous Operations

The positive (+) and negative (-) operations must precede the operand. The result of a positive operation is the absolute value of the operand. The

result of a negative operation is the negative value of the operand; that is, a  $+(-5)$  becomes 5 and a  $-(-2)$  becomes 2. These operators have the same precedence as the increment and decrement operators. If there is an operand on both sides of these operators, they are interpreted as the add and subtract operators.

**Example**

```
# local_abs := +local #>
<# local_neg := -local #>
```

All operations are performed in the order implied by the precedence of the operators. However, you can modify this order by using parentheses (()) to group operands and operators. Operations within parentheses are performed first. The result is that of the operation(s) within the parentheses.

**Example**

```
<# 4 % (3 + 12) - 6 #>The result is -6
<# 5 && 2 > 1 #>The result is 1
<# (5 && 2) > 1 #>The result is 0
```

Results of control expressions are written to the output stream when the expression consists of the following:

- A single local variable
- A single literal element
- An operation whose result is not used by one of the following operations:

assignment	predecrement	postdecrement	while
if	preincrement	postincrement	

**Example**

```
<# localvar #>value of localvar is written
<# " any string" #>" any string" written
<# 4 % 3 + 12 - 6 #>"7" is written
<# 4 % (3 + 12) - 6 #>"-6" is written
<# i := i + 1 #>nothing is written
<# count := (count - 2) #>nothing is written
```

## Conditional Execution

You can use *if* or *while* constructs in macros to enable conditional execution of commands.

### If Constructs

*If* constructs provide a means to execute portions of the macro based on conditions that you specify. An *if* construct consists of the following components:

- An opening *if* expression
- A group of any number of additional expressions
- (Optional) Any number of *elseif* expressions and groups of associated expressions
- (Optional) An *else* expression and any associated group of expressions
- An *endif* expression to indicate the end of the *if* structure

The *if* expression and any optional *elseif* expressions must include either a lone environment value command, a local variable, a literal, or some operation using one or more operators.

Only one of the groups of expressions within the *if* construct is executed, according to the following scheme:

- 1 The *if* expression is evaluated. If the result is true (nonzero), the associated expression group is executed.
- 2 If the result is false (zero), then the first *elseif* expression, if present, is evaluated. If the result is true (nonzero), the associated expression group is executed.
- 3 If the result of evaluating the first *elseif* expression is false (zero), the next *elseif* expression is evaluated, if present. If the result is true (nonzero), the associated expression group is executed.

If all *elseif* expressions evaluate to false (zero) or if no *elseif* expressions are present, then the *else* expression group—if present—is executed.

- 4 This evaluation process continues until an expression evaluates to nonzero. If there is no nonzero evaluation, then no expression group is executed.

You can write an empty expression group so that no action is performed if this group is selected for execution. You can nest *if* structures within other *if* structures or *while* structures.

The following sample macro demonstrates various *if* structures:

```
<#          if_examples          #>
<# //----- #>

<# if 1 #>
! This is always output because any nonzero value is "true."
<# endif #>

<# if 0 #>
! This is never output because a value of zero is "false."
<# endif #>

<# // Here's an example with elseif and else. #>
<# color := env.getline("What is your favorite color? ") #>
<# if color = "red" #>
! Red is my favorite color, too.
<# elseif color = "pink" #>
! Pink is a lot like red.
<# elseif color = "black" #>
! Black is just a very, very, very dark shade of red.
<# else #>
! Oh. That's nice.
<# endif #>

<# // Here's a nested if example. #>
<# sure := env.getline("Are you sure that " $ color $ " is
your favorite color? ") #>
<# if substr(sure, 0, 1) = 'y' || substr(sure, 0, 1) = 'Y' #>
<# if color != "black" && color != "white";
shade := env.getline("Do you prefer dark " $ color $
" or light " $ color $ "? ") #>
<# if shade = "dark" #>
! I like dark colors, too.
<# elseif shade = "light" #>
! I prefer dark colors myself.
<# else #>
! Hmmm, that's neither dark nor light.
<# endif #>
<# else #>
! Oh. That's nice.
<# endif #>
<# else #>
! I didn't think so!
<# endif #>
<# endtmpl #>
```

## While Constructs

*While* constructs provide a means to repeatedly execute one or more portions of the macro based on a condition that changes during the execution. A *while* construct consists of the following components:

- An opening *while* expression
- A group of any number of additional expressions
- An *endwhile* expression to indicate the end of the *while* structure

The *while* expression must include either a lone environment value command, a local variable, a literal, or some operation using one or more operators. Each time that this expression evaluates to nonzero, the associated expression group is executed.

You can place an iteration expression after the *while* expression. This optional expression is evaluated after each execution of the *while* expression group.

You can include *if* structures within a *while* structure. You can also include special control expressions indicated by the *break* or *continue* expressions. The *break* expression breaks out of the *while* structure by halting execution of the expression group and executing the first expression after the *endwhile* statement. The *continue* expression skips over the rest of the expression group, evaluates any iteration expression, then continues with the execution of the *while* structure. The *while* structure is limited to 100,000 repetitions by default. You can nest up to ten *while* structures.

**Example** The following sample macro demonstrates various *while* structures:

```

<#           while_examples           #>
<# //----- #>
<# // Remember that variables are automatically initialized to 0. #>
! Table of squares of the first 10 integers:
<# while ++i <= 10 #>
!<#i;" ";i*i;"\n"#>
<# endwhile #>

<# // Remember that the value of a string used as an integer is the number. #>
<# // of characters in the string. #>
<# stars := "*" #>
<# while stars < 10, stars := stars $ "*"#>
!<# stars;"\n" #>
<# endwhile #>
<# while stars > 0, stars := substr(stars, 0, stars-1)#>
!<# stars;"\n" #>
<# endwhile #>

<# // An example of the continue and break statements. #>
<# // Also note that many statements can be grouped. #>
! All the positive even numbers less than 11
<# i:=0; while ++i < 100 #>
  <#if i%2; continue; endif; if i > 10; break; endif; !" $ i $ "\n"; #>
<# endwhile #>

<# // While constructs will NOT iterate forever. #>
<# while 100 > 0 // This is always true, but the macro will eventually stop #>
<# ++iterations; endwhile #>
! The while loop iterated <#iterations#> times.
<# endtmpl #>

```

### Invoking Other Macros

Macros can invoke other macros within the same macro file; a macro can also invoke a macro from another macro file if the invocation takes place in literal text, that is, not within a control expression. A macro can invoke itself directly or indirectly (an invoked macro can invoke the macro that invoked it); the number of nested invocations is limited to 10 to prevent infinite recursion.

Within each macro, you can specify parameters that *must* be passed to the macro when it is invoked by another. You must specify named variables enclosed in parentheses after the macro name in the first line of the macro, as shown in this example:

```
<# macroName (count, total) #>
```

Additional parameters can be passed as well. Parameters can be local variables, environmental variables, literals, or operations. The invoking macro passes local variables by reference to the invoked macro. Passing parameters has no effect on the invoking macro unless the parameter is a local variable that is changed by the invoked macro. When the invoked macro completes execution, the local variable assumes the new value for the invoking macro.

The invoked macro can use the **param**[*n*] expression to access parameters passed to it, where *n* is the number of the parameter passed. This is useful if optional parameters can be passed to a macro or if the same iterative algorithm needs to process the parameters.

Use the expression **param**[0] to return the total number of parameters passed to the macro. Use the **return** keyword to halt execution of the invoked macro and resume execution of the invoking macro. Use the **exit** keyword to halt execution of all macros.

**Example 1** The following sample macro demonstrates macro invocation:

```
<#          invoking_examples          #>
<# //----- #>
<# name := env.getline("What is your first name? ") #>
! First, <#name#>, we will invoke the if_examples and
! the while_examples macros...
<# tmpl.if_examples; tmpl.while_examples #>
! Hey <#name#>, have you noticed that your name backwards is:
!<# eman:= ""; tmpl.reversestring(name, eman); eman; "\n"#>
<#   tmpl.argumentlist("a", "b", "c")#>
<# endtmpl #>

<# argumentlist #>
<# if param[0] = 0; return; endif #>
! argumentList() was called with the following arguments:
<# while ++i <= param[0]#>
!   <#param[i];"\n"#>
<# endwhile #>
<# endtmpl #>

<# reversestring (string, gnirts) #>
<# i := 0 + string; // i is now equal to the number of
   characters in string. #>
<# while --i >= 0; gnirts := gnirts $ substr(string, i, 1);
   endwhile #>
<# endtmpl #>
```

**Example 2** The following macro in file `macro1.mac` invokes a macro from within another file, `macro2.mac`:

```
<# callAnotherMacro #>
  <# localVar := 5 #>
    macro macro2.mac macroName2 <# localVar #> string1
  <# endtmpl #>
```

This macro passes the value of `localVar` to `macroName2`. The value of `localVar` remains at 5 for `callAnotherMacro`, regardless of any operations upon that variable in the second macro. In other words, an invoked macro in another file cannot return any values to the invoking macro.

The output of `callAnotherMacro` looks like this:

```
host1#macro verbose macro1.mac callAnotherMacro
host1#!Macro 'callAnotherMacro' in the file 'macro1.mac'
starting execution
macro macro2.mac macroName2 5 string1
!Macro 'macroName2' in the file 'macro2.mac' starting
execution
!Macro 'macroName2' in the file 'macro2.mac' ending
execution
host1#!Macro 'callAnotherMacro' in the file 'macro1.mac'
ending execution
```

The invoked macro cannot invoke a third macro from another file. Only a single level of invocation is supported.

## Running Macros

---

Although you must write macros on a computer, you can copy them to the system. Issue the **macro** command from the CLI to execute both local macros and macros stored remotely.

You can display the commands that are generated by the macro file without executing the commands by using the **test** keyword. It is good practice to confirm that the test display matches your expectations before you execute the macro to run the commands.

You can terminate a macro while it is running by entering `<Ctrl+C>`. You can close Telnet and SSH windows while a macro is running, but the macro does not terminate until it completes the current command.

### **macro**

- Use to execute a macro that generates—and can execute—CLI commands. This command is available in all command modes.
- This command invokes a hidden FTP client and takes place in the context of the current virtual router (VR) rather than the default VR. You must configure

the FTP server so that any traffic destined for the VR can reach the VR; typically, you configure the FTP server to reach the default address of the system, which will always be able to reach the VR.

- You can specify both a macro filename and a macro contained within that file. For example, the following command looks for the file “confatm.mac” and runs the macro named “atm0verDs3” contained within the file:

```
host1(config)#macro confatm.mac atm0verDs3
```

- You can specify only a macro filename. The command searches in the specified file for a macro named “start.” The command fails if the “start” macro does not exist. For example, the following command looks for the file “confatm.mac” and runs the macro named “start” contained within the file:

```
host1(config)#macro confatm.mac
```

- You can specify only the macro name, using the **name** keyword, if the macro file is stored locally in NVS and has the same name as the included macro you wish to invoke. For example, the following command looks for the file “confatm.mac” and runs the macro named “confatm” contained within the file:

```
host1(config)#macro name confatm
```

- You must specify a macro filename for remotely stored macro files, as in the following example:

```
host1(config)#macro pc:/macros.mac atm0verDs3
```

- You can pass arguments to the macro; if the argument contains a space or other special character, you must enclose the argument within double quotation marks.
- Use the **test** keyword to specify that the macro generate, but not execute, the commands. You can check the output to verify that it is what you want. The test mode is verbose and displays comments.
- Use the **verbose** keyword to echo commands to the display and display comments as the macro executes. By default the command executes in nonverbose mode.
- There is no **no** version.

**Example** A typical macro application is to iteratively generate a series of commands, as shown in the following macro, *atm0verDs3*:

```
<# atm0verDs3 #>
<# i:=0; while i++ < 3 #>
  controller t3 9/<#i;\n'#>
  no shut
  clock source internal module
  framing cbitadm
  ds3-scramble
!
  interface atm 9/<#i;\n'#>
  atm vc-per-vp 256
!
<# endwhile #>
```

```

!
interface atm 9/1.1
encap pppoe
!
<# i:=1; while i < 100 #>
  interface atm 9/1.1.<#i;'\n'#>
  encap ppp
  no ppp shut
  no ppp keep
  atm pvc <# i #> 1 <# i #> aal5mux ip
  ip addr 10.1.<#i#>.1 255.255.255.0
!
<# i++ #>
<# endwhile #>
!
<# endtmpl #>

```

If you stored this macro remotely in the macro file, *pc:/macros.mac*, you issue the following commands to execute the macro:

```

host1>enable
host1#conf t
host1(config)#macro pc:/macros.mac atmOverDs3

```

Alternatively, if you stored this macro locally in the macro file *atmOverDs3.mac*, you issue the following commands to execute the macro:

```

host1>enable
host1#conf t
host1(config)#macro verbose atmOverDs3

```

A portion of the output resulting from executing the *atmOverDs3* macro from a local file is shown below (the starting and ending comments would vary for a remote macro):

```

host1(config)#!Macro 'atmOverDs3' in the file
'atmOverDs3.mac' starting execution
host1(config)#controller t3 9/1
host1(config)#no shut
host1(config)#clock source internal module
host1(config)#framing cbitadm
host1(config)#ds3-scramble
host1(config)#interface atm 9/1
host1(config)#atm vc-per-vp 256
host1(config)#controller t3 9/2
host1(config)#no shut
host1(config)#clock source internal module
host1(config)#framing cbitadm

```

```
host1(config)#ds3-scramble
host1(config)#interface atm 9/2
host1(config)#atm vc-per-vp 256
host1(config)#controller t3 9/3
host1(config)#no shut
host1(config)#clock source internal module
host1(config)#framing cbitadm
host1(config)#ds3-scramble
host1(config)#interface atm 9/3
host1(config)#atm vc-per-vp 256

host1(config)#interface atm 9/1.1
host1(config)#encap pppoe

host1(config)#interface atm 9/1.1.1
host1(config)#encap ppp
host1(config)#no ppp shut
host1(config)#no ppp keep
host1(config)#atm pvc 1 1 1 aal5mux ip
host1(config)#ip addr 10.1.1.1 255.255.255.0
```

[display omitted]

```
host1(config)#interface atm 9/1.1.99
host1(config)#encap ppp
host1(config)#no ppp shut
host1(config)#no ppp keep
host1(config)#atm pvc 99 1 99 aal5mux ip
host1(config)#ip addr 10.1.99.1 255.255.255.0
host1(config)#!Macro 'atmOverDs3' in the file
'atmOverDs3.mac' ending execution
```

## Practical Examples

---

You can use the macros in this section for configuring your system or as examples of useful macros you can build yourself.

### *Configuring Frame Relay*

You can organize your macros in many different ways to suit your needs. The first sample macro in this section, *ds1mac.mac*, shows a typical method of organization. It consists of a number of related macros for configuring interfaces on CT1 and CE1 modules, as described in Table 7-4.

Some of the macros provide a single configuration function, like configuring the controller. These are invoked by other macros that are

executable from the command line. A high-level macro invokes several of the executables, acting much like a script to provide greater functionality.

**Table 7-4** Contents of ds1mac.mac

Macro Name	Description
Help	Lists the executable macros in ds1mac.mac
controllerDs1	Executable macro that configures Cx1 ports; calls macro cntrDs1
ds1Encap	Executable macro that configures Frame Relay encapsulation on Cx1 serial interfaces; calls macro cx1Encap
ds1FrCir	Executable macro that configures Frame Relay circuits on Cx1 subinterfaces; calls macro cx1FRCir
configCx1	Executable macro that configures Cx1 serial Frame Relay interfaces; calls macros cntrDs1, cx1Encap, and cx1FrCir
cntrDs1	Configures the Cx1 controller; called by other macros
cx1Encap	Configures Frame Relay encapsulation on serial interfaces; called by other macros
cx1FrCir	Configures Frame Relay circuits on the subinterfaces; called by other macros

Table 7-5 lists the complete set of macros contained in ds1mac.mac. You can run the Help macro to list the other executable macros contained in ds1mac.mac. To configure Frame Relay on your system with ds1mac.mac, you can do one of the following:

- Run the controllerDS1, ds1Encap, and ds1FrCir macros in that order
- Run the configCx1 macro

In either case, to run the macros you must provide the required values described in the macros.

**Table 7-5** ds1mac.mac

```
<# Help #>
! This file contains the following executable macros:
! controllerDs1
! ds1Encap
! ds1FrCir
! configCx1
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# controllerDsl #>
<# if env.argc = 0 #>
! This macro configures your Cxl controller.
! This macro will configure e1 ports as unframed.
! This macro should be called with 4 arguments.
! The argument list should be as follows:
! type; number of numPorts; slot; port; clock; framing;
  lineCoding
<# return #>
<# endif #>
<# type := env.argv(1) #>
<# ifCount := env.argv(2) #>
<# slot := env.argv(3) #>
<# port := env.argv(4)#>
<# clock := env.argv(5) #>
<# framing := env.argv(6) #>
<# coding := env.argv(7) #>

<# if clock = 'internal' #>
<# clock := 'internal mod' #>
<# endif #>

<# tmpl.cntrDsl(type, ifCount, slot, port, clock, framing,
  coding) #>
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# dslEncap #>
<# if env.argc = 0 #>
! This macro configures Frame Relay encapsulation on Cx1
  serial
! interfaces.
! This macro must be called with 4 arguments.
! If the protocol is Frame Relay (fr), then specify the
  type (DTE
! or DCE) and the lmi type.
! The argument list should be as follows:
! number of numPorts; slot; port; proto; frType; frLmi
<# return #>
<# endif #>
<# ifCount := env.argv(1) #>
<# slot := env.argv(2) #>
<# port := env.argv(3) #>
<# proto := env.argv(4) #>
<# if proto = 'fr' #>
<# proto := 'frame-relay ietf' #>
<# endif #>
<# tmpl.cxlEncap(ifCount, slot, port, proto) #>
<# endtmpl #>
<# dslFrCir #>
<# if env.argc = 0 #>
! This macro configures Frame Relay circuits on Cx1
  subinterfaces.
! This macro must be called with 4 arguments.
! The argument list should be as follows:
! number of numPorts; slot; port; numCirs; dlci
<# return #>
<# endif #>
<# ifCount := env.argv(1) #>
<# slot := env.argv(2) #>
<# port := env.argv(3) #>
<# numCirs := env.argv(4) #>
<# dlci := env.argv(5) #>
<# tmpl.cxlFrCir(ifCount, slot, port, numCirs, dlci) #>
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# configCx1 #>
<# if env.argc = 0 #>
! This macro configures Cx1 serial Frame Relay interfaces.
! This macro must be called with 4 arguments.
! The argument list should be as follows:
! type; number of numPorts; slot; port; clock; framing;
! coding; proto; frType; frLmi; numCirs; dlci
<# return #>
<# endif #>
<# type := env.argv(1) #>
<# ifCount := env.argv(2) #>
<# slot := env.argv(3) #>
<# port := env.argv(4) #>
<# clock := env.argv(5) #>
<# framing := env.argv(6) #>
<# coding := env.argv(7) #>
<# proto := env.argv(8) #>
<# tmpl.cntrDsl(type, ifCount, slot, port, clock, framing,
coding) #>
<# if proto = 'fr' #>
<# frType := env.argv(9) #>
<# frLmi := env.argv(10) #>
<# numCirs := env.argv(11) #>
<# dlci := env.argv(12) #>
<# tmpl.cx1Encap(ifCount, slot, port, proto, frType, frLmi)
#>
<# tmpl.cx1FrCir(ifCount, slot, port, numCirs, dlci) #>
<# else #>
<# tmpl.cx1Encap(ifCount, slot, port, proto, type, type) #>
<# endif #>
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# cntrDs1 #>
<# //This macro is called by other macros to configure DS1
ports #>
<# //Parameters in order are interface Type; numPorts;
slot; port; clock; framing; lineCoding #>
!
! Configure Cx1 Controller
!
<# type := param[1] #>
<# ifCount := env.atoi(param[2]) #>
<# slot := param[3] #>
<# port := env.atoi(param[4]) #>
<# clock := param[5] #>
<# framing := param[6] #>
<# coding := param[7] #>
<# while ifCount-- > 0 #>
controller <# type;' '; slot;'/' ;port;'\n' #>
<# if framing = 'unframed' #>
unframed
<# else #>
framing <# framing;'\n' #>
linecoding <# coding;'\n' #>
<# endif #>
clock source <# clock;'\n' #>
no shutdown

<# port++ #>
<# endwhile #>
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# cxlEncap #>
<# //This macro is called by other macros to configure
  Frame Relay encapsulation on serial interfaces. #>
<# //Parameters in order are interface Type; numPorts;
  slot; port; clock; framing; lineCoding #>
!
! Configure Encapsulation
!
<# ifCount := env.atoi(param[1]) #>
<# slot := param[2] #>
<# port := env.atoi(param[3]) #>
<# proto := param[4] #>
<# if proto = 'fr' #>
<# proto := 'frame-relay ietf' #>
<# endif #>
<# while ifCount-- > 0 #>
interface serial <# slot; '/' ; port ; ':1'; '\n' #>
encapsulation <# proto; '\n' #>
<# if proto = 'frame-relay ietf' #>
frame-relay intf-type <# param[5]; '\n' #>
frame-relay lmi-type <# param[6]; '\n' #>
<# endif #>

<# port++ #>
<# endwhile #>
<# endtmpl #>
```

**Table 7-5** ds1mac.mac (continued)

```
<# cxlFrCir #>
<# //This macro is called by other macros to configure
Frame Relay circuits on subinterfaces. #>
<# //Parameters in order are interface numPorts; slot;
port; numCirs; dlci #>
!
! Configure Frame Relay Circuits
!
<# ifCount := env.atoi(param[1]) #>
<# slot := param[2] #>
<# port := env.atoi(param[3]) #>
<# numCirs := env.atoi(param[4]) #>
<# startDlci := env.atoi(param[5]) #>
<# id := env.atoi('1') #>
<# while ifCount-- > 0 #>
<# cirs := numCirs #>
<# id := env.atoi('1') #>
<# dlci := startDlci #>
<# while cirs-- > 0 #>
interface serial <# slot;'/';port;'1.';id;' \n' #>
frame-relay interface-dlci <# dlci #> ietf

<# id++; dlci++ #>
<# endwhile #>
<# port++ #>
<# endwhile #>
<# endtmpl #>
```

## Configuring ATM Interfaces

The sample macro in Table 7-6 configures ATM interfaces based on the inputs you provide when prompted by the macro.

**Table 7-6** Sample macro to configure ATM interfaces

```
<# atmIf #>
<# slotPort:=env.getline("slot/port?") #>
<# while (vcType != 1 && vcType != 2);
  vcTypeStr :=env.getline("VC type (1 = AAL5MUX IP, 2 = AAL5SNAP)?");
  vcType := env.atoi(vcTypeStr);
endwhile #>
<# if vcType = 1; vcTypeStr := "aal5mux ip"; else; vcTypeStr := "aal5snap"; endif
#>
<# encapRouted:=1; encapBridged:=2; encapPPP:=3 #>
<# while (encapType < encapRouted || encapType > encapPPP );
  encapTypeStr :=env.getline("encapsulation (1 = routed, 2 = bridged, 3 =
ppp)?");
  encapType := env.atoi(encapTypeStr);
endwhile #>
<# if encapType = encapPPP #>
  <# authNone:=1; authPap:=2; authChap:=3; authPapChap:=4; authChapPap:=5 #>
  <# while (authType < authNone || authType > authChapPap );
    authTypeStr :=env.getline("authentication (1 = None, 2 = PAP, 3 = CHAP, 4 =
PAP/CHAP; 5 = CHAP/PAP)?");
    authType := env.atoi(authTypeStr);
  endwhile #>
  <# endif #>
<# vpStartStr := env.getline("Starting VP number?");
  vpStart:=env.atoi(vpStartStr)#>
<# vpEndStr := env.getline("Ending VP number?"); vpEnd
:=env.atoi(vpEndStr)#>
<# vcStartStr := env.getline("Starting VC number?");
  vcStart:=env.atoi(vcStartStr)#>
<# vcEndStr := env.getline("Ending VC number?"); vcEnd
:=env.atoi(vcEndStr)#>
```

**Table 7-6** Sample macro to configure ATM interfaces (continued)

```
<# loopbackStr := env.getline("Loopback interface number or <cr>?") #>
<# vp := vpStart; while vp <= vpEnd, ++vp #>
  <# vc := vcStart; while vc <= vcEnd, ++vc #>
    interface atm <#slotPort $ '.' $ ++i;\n'#>
    atm pvc <# i; ' '; vp; ' '; vc; ' '; vcTypeStr;\n'#>
    <# if encapType = encapPpp #>
      encap ppp
      <# if authType = authPap#>
        ppp authentication pap
      <# elseif authType = authPapChap#>
        ppp authentication pap chap
      <# elseif authType = authChapPap#>
        ppp authentication chap pap
      <# elseif authType = authChap#>
        ppp authentication chap
      <# endif #>
    <# elseif encapType = encapBridged #>
      encap bridged1483
    <# endif #>
    <# if loopbackStr != "" #>
      ip unnumbered loopback <# loopbackStr;\n" #>
    <# endif #>
    !
  <# endwhile #>
!
<# endwhile #>
```

**Table 7-6** Sample macro to configure ATM interfaces (continued)

```
<# if encapType = encapPPP #>
  <# authNone:=1; authPap:=2; authChap:=3; authPapChap:=4; authChapPap:=5 #>
  <# while (authType < authNone || authType > authChapPap );
  authTypeStr :=env.getline("authentication (1 = None, 2 = PAP, 3 = CHAP, 4 =
  PAP/CHAP; 5 = CHAP/PAP)?");
  authType    := env.atoi(authTypeStr);
  endwhile #>
  <# endif #>
<# vpStartStr := env.getline("Starting VP number?");
  vpStart:=env.atoi(vpStartStr)#>
<# vpEndStr    := env.getline("Ending    VP number?"); vpEnd
:=env.atoi(vpEndStr)#>
<# vcStartStr := env.getline("Starting VC number?");
  vcStart:=env.atoi(vcStartStr)#>
<# vcEndStr    := env.getline("Ending    VC number?"); vcEnd
:=env.atoi(vcEndStr)#>
<# loopbackStr := env.getline("Loopback interface number or <cr>?") #>
<# vp := vpStart; while vp <= vpEnd, ++vp #>
  <# vc := vcStart; while vc <= vcEnd, ++vc #>
  interface atm <#slotPort $ '.' $ ++i;\n'#>
  atm pvc <# i; ' '; vp; ' '; vc; ' '; vcTypeStr;\n'#>
  <# if encapType = encapPpp #>
  encap ppp
  <# if authType = authPap#>
  ppp authentication pap
  <# elseif authType = authPapChap#>
  ppp authentication pap chap
  <# elseif authType = authChapPap#>
  ppp authentication chap pap
  <# elseif authType = authChap#>
  ppp authentication chap
  <# endif #>
  <# elseif encapType = encapBridged #>
  encap bridged1483
  <# endif #>
  <# if loopbackStr != "" #>
  ip unnumbered loopback <# loopbackStr;"\n" #>
  <# endif #>
!
  <# endwhile #>
!
  <# endwhile #>
<# endtmpl #>
```

