# SCRIPT CONTROL LANGUAGE

# REFERENCE GUIDE

# OVERVIEW OF SCRIPT CONTROL LANGUAGE SYNTAX

The Script Control Language (SCL) is used to write *thread scripts* and *test scripts*:

- **Thread scripts** define and control the test cases and input that are to be used to test the target system.
- **Test scripts** define the structure of a test, by specifying which threads are to be executed and in what order. They also allow you to set up a dynamic test structure that uses the results of threads that have already been run, to determine which threads should be run later in the test.

Script source files consist of up to three sections – an *Environment* section, a *Definitions* section and a *Code* section, which must appear (if present) in that order.

The first section is the mandatory **Environment** section. This section defines the global attributes of the script, i.e. the script description, script mode and wait command units. It is introduced by the ENVIRONMENT command, and continues until a DEFINITIONS or CODE command is encountered.

The second section is the optional **Definitions** section. This section contains the variable, constant, timer and file definitions for the script; for test scripts, it also contains thread definitions and test declarations. It starts with the DEFINITIONS command, and continues until the CODE command.

The last section is the mandatory **Code** section, which contains the main script commands. The start of this section is marked by the CODE command; it continues until the end of the script source file.

Tabs, spaces and form-feeds may be incorporated into the code to align keywords and generally aid legibility; they have no other effect on compilation.

## CHARACTER REPRESENTATION

The text within an SCL source file falls into three broad categories:

1. SCL commands.
2. Arguments to SCL commands – variable names, integer values or quoted character strings, for example.
3. Comments, to improve legibility and maintenance.

Within character string arguments, SCL supports the use of any character with an ASCII value in the range HEX 00 to FF inclusive. However, direct specification of these characters is not always possible, for two reasons:

1. Characters with values in the ranges HEX 00 to 20 and HEX 7F to A0, and the value HEX FF, are 'non-printing' characters, and cannot easily be specified in an SCL source file.
2. Two characters are reserved for use by SCL – one as a command character and the other as a control character. The characters used for these purposes cannot be used as literal characters in a character string. The default values are "~" for the command character and "^" for the control character; these values are used throughout these instructions. They can, however, be changed within the script.

To resolve these problems, SCL provides a set of 'character commands', as described in *Representing the Command Character* and *Representing the Control Character*. In addition, to ensure there is no ambiguity within the source file, characters are rejected which have values in the ranges HEX 00 to 20, or HEX 7F to A0, or the value HEX FF, except as described in *Characters ignored by the compiler*.

Character commands are recognized within all SCL character strings (except for a small number of exceptions that are explicitly stated). Thus, for example, the character string "~<07>" always represents a single character (namely the character with a hexadecimal value of 7), **not** five characters.

**Note:** Single quotes may be included in character strings by using double quotes for the string delimiters, and vice versa.

## Character Command Using Hexadecimal ASCII Code

All characters can be represented by hexadecimal ASCII code, character command. The command format is:

~<hh>

"~" is the currently defined command character and "hh" is the hexadecimal ASCII code of the required character. This form of character command is primarily intended to represent characters that cannot be represented by any of the other forms of character command.

For example, the ASCII horizontal tabulation character is represented by "~<09>" and the null character by "~<00>".

## Character Command Using ASCII Mnemonic

SCL provides a number of character commands which give an easily identifiable representation of common control characters. These use the ASCII mnemonic of the control character in question. The following commands are available (using the default command character "~"):

| | |
|---|---|
| ~<BEL> | Bell |
| ~<BS> | Backspace |
| ~<CR> | Carriage return |
| ~<DEL> | Delete |
| ~<ESC> | Escape |
| ~<FF> | Form feed |
| ~<HT> | Horizontal tab |
| ~<LF> | Line feed |
| ~<SP> | Space |
| ~<VT> | Vertical tab |

## Control Command

All 7-bit control characters, i.e. characters with ASCII codes in the range HEX 00 to 1F inclusive, may be represented using a control command. The control command has the following format:

^c

"^" is the default control character and "c" is the control character specifier. The control character specifier is an ASCII graphics character with an ASCII code in the range HEX 40 (ASCII "@") to 5F (ASCII "_"). The compiler will apply the bottom 6 bits only, to generate an ASCII code in the range HEX 00 to 1F.

For example, the ASCII bell character (ASCII code HEX 07), is represented by "^G".

## Representing the Command Character

The command character always introduces a command and therefore cannot be used to represent the command character itself. The command character is instead represented by a command of the following format:

~~

"~" is the currently defined command character.

## Representing the Control Character

The control character is always used to represent the <CTRL> key, in combination with the character following it. It therefore cannot be used to represent the control character itself. The control character is instead represented by a command of the following format:

~^

"~" is the currently defined command character and "^" is the currently defined control character.

## CONTINUATION LINES

It is not always possible to fit a script statement or command onto one line, so SCL allows you to use ‘continuation lines’.

An SCL statement or command may be split over two or more lines by terminating all but the last line of the statement with an ampersand or hyphen character (“&” or “–”). To avoid possible confusion with the minus character, it is recommended that the ampersand be used, and that it be separated from the preceding characters on the line by at least one space.

The only things that may follow a continuation character are space characters, tab characters and comments (see the next section).

A quoted character string is continued onto another line by closing it at the end of the line and re-opening it on the next. Opening and closing quotes must match on any one line, as shown in the following example:

```
LOG "This string of text is continued "          &
       ’over two lines.’
LOG "This message contains a variable ", VAR1,   &
       ’ and is continued on this line ’,        &
       VAR2, ’ and this line’,                   &
       ’ and this line’
```

**Note:** A line that ends with an SCL command or statement terminated by “&” or “–” implies that the next line encountered will be regarded as a continuation of the original command or statement.

## COMMENTS

Scripts may incorporate comments, either on lines by themselves or embedded in statements or commands. In both cases, the comment is identified by the comment command (“!”), and terminated by the end of the line. For example:

```
!
!Get next page.
!
 SET conid = conid + 1       ! Update connection ID
 GET URL "http://abc.com" &  ! Get this URL
    ON conid &               ! use this TCP connection
    HEADER sub_header &      ! default headers
      , WITHOUT "Referer"    ! no referer
```

## CYRANO DATANAMES

The names of many items within scripts must be defined as a CYRANO dataname. For example, script names, variable names and subroutine names must all be CYRANO datanames.

A CYRANO dataname comprises between 1 and 16 alphanumeric, underscore or hyphen characters. The first character must be alphabetic; spaces are not allowed; two adjacent underscores or hyphens are not allowed; and neither is a trailing underscore or hyphen.

## MAXIMUM VALUES IN SCRIPTS

The SCL compiler and system resources impose limitations at run-time on the maximum value (number, size, level etc) allowed for a number of items which may be specified in an SCL source file.

| Description | Value |
|---|---|
| Max. **source line length** (characters) | **132** |
| Max. no. of **labels** (per subroutine/main code) | **255** |
| Max. no. of **timers** | **1020** |
| Max. no. of **variables** | **8000** |
| Max. no. of **global variables** | **8000** |
| Max. no. of **subroutines** | **255** |
| Max. no. of **parameters** passed between scripts | **8** |
| Max. no. of **external data files referenced** in script | **256** |
| Max. no. of **external data files open** concurrently | **10** |
| Max. **character variable size** (bytes) | **65535** |
| Max. **character constant/literal size** (bytes) | **65535** |
| Max. **space available for script values** (Kbytes) | **128** |
| Max. **nesting level** for **conditions** | **10** |
| Max. **nesting level** for **array expressions** | **10** |
| Max. **nesting level** for **conditional compilations** | **10** |
| Max. **nesting level** for **IF/DO commands** | **100** |
| Max. **nesting level** for **subroutines** | **10** |

## INCLUDING TEXT FROM OTHER SOURCE FILES

The INCLUDE command allows you to combine several source files into a single source file at compilation time. These included files may contain commands from any of the script sections and may span these sections. Scripts may be nested up to a depth of 10, including the main script. Care should be taken to avoid duplicating any of the script section commands (for example, ENVIRONMENT).

This command can appear at any point within the script, including before the ENVIRONMENT command.

### Format:

```
INCLUDE filename
```

### Parameter:

**filename**
A quoted character string which defines the name of the source file to be included. The location of the file will default to the current default directory.

### Example:

```
INCLUDE '\usr\payrol\tests\payroll.gbl'
```

## CONDITIONAL COMPILATION OF SOURCE CODE

SCL provides a number of commands to allow you to define that a section of code should be compiled only under certain circumstances. Conditional sections of code are marked with 'variants'.

Variants are specified on the **-V** option, on the **scl** compiler command line when you compile the source file.

Conditional compilation commands may appear at any point within the Environment, Definitions and Code sections, including before the ENTRY command and between subroutines. They cannot appear part way through a command or statement. They may be nested to a depth of 10.

### Format:

```
condition variant
```

### Parameters:

**condition**
A conditional compilation command which starts or ends a section of code. This may be one of the following:

| | |
|---|---|
| #IFDEF | Compile next section if "variant" requested |
| #IFNDEF | Compile next section if "variant" **not** requested |
| #ELIF | Otherwise compile next section if "variant" requested |
| #ELSE | Otherwise compile the next section |
| #ENDIF | End of variant section |

The #IFDEF, #IFNDEF and #ELIF commands require the "variant" parameter, to specify the condition under which the following section of code will be compiled. The #ELSE and #ENDIF commands relate to the most recently specified variant.

**variant**
A *CYRANO dataname* which identifies a section of code that is only compiled under certain conditions. The compiler processes this variant in conjunction with the **-V** option on the **scl** command line.

### Examples:

```
#IFDEF variant1
    log "This is only compiled if /VARIANT=variant1 is specified"
#ELIF variant2
    log "This is only compiled if /VARIANT=variant2 is specified"
```

```
#ELSE
    log "This is only compiled if neither variant is specified"
#ENDIF
```

# THE ENVIRONMENT SECTION

The Environment section of the SCL source code is introduced by the mandatory ENVIRONMENT command. It defines the global attributes of the script, i.e. the script description, the script mode and wait command units.

The Environment section must be the first section of the script, preceding the Definitions section (if present) and Code section. It may, however, be preceded by an INCLUDE statement. For further information, see *Including Text from Other Source Files*.

# DESCRIPTION STATEMENT

### Description:

This mandatory statement assigns a descriptive character string to a script. This descriptive string will be displayed on the CYRANO TestCommander utility's Script Properties form.

### Format:

```
DESCRIPTION string
```

### Parameter:

**string**
A quoted character string, between 1 and 50 characters in length, used as the description.

### Examples:

```
DESCRIPTION 'Create Customer Records'
DESCRIPTION "Update Customer's Record"
DESCRIPTION "Test abc.com Support Pages"
```

## MODE HTTP STATEMENT

This optional statement defines the script as an HTTP mode script. These scripts are used to issue HTTP requests to an HTTP server.

This statement must be specified in order for the HTTP-specific commands to be available to a script.

### Format:

```
MODE HTTP
```

### Parameters:

None

## WAIT UNIT STATEMENT

This optional statement defines the unit of the wait period specified in WAIT commands within a script. This does NOT apply to the wait period in the WAIT FOR SEMAPHORE command - the wait period in this command is always specified in seconds.

If this statement is omitted, the wait unit is seconds.

### Format:

```
WAIT UNIT [SECONDS | MILLISECONDS]
```

### Parameters:

None

# THE DEFINITIONS SECTION

The Definitions section of the SCL source code defines the variables, threads and constants that are used by the script. It can also contain declarations of tests, timers and files. It is optional, and is introduced by the DEFINITIONS command.

Only one Definitions section may appear in a script; if it is present, it must follow the Environment section and precede the Code section.

# TEST STATEMENT

## Description:

This statement declares a test that may be executed by a test script. All tests referenced within a test script must be declared within the Definitions section of the script, using this statement.

Up to 32767 tests may be declared and used in a test script.

## Format:

```
TEST name
```

## Parameter:

**name**
The name of a test. This must be a valid *CYRANO dataname*.

## Example:

```
TEST  pers
```

# THREAD STATEMENT

## Description:

This statement defines a thread. All threads referenced within a test script must be defined within the Definitions section of the script.

Up to 32767 threads may be defined and used in a test script.

## Format:

```
THREAD name FOR TARGET target_type
```

## Parameters:

**name**
The name of the thread. This must be a valid *CYRANO dataname*.

**target_type**
Defines the target test object type of the thread. This may be set to "COMMAND" or “WEB”

## Example:

```
THREAD pers-thr FOR TARGET COMMAND
```

# CHARACTER STATEMENT

## Description:

This statement defines a character string variable consisting of ASCII characters, including control characters. SCL supports character variables of between 1 and 65535 bytes in length.

Arrays of character variables can be defined, with a maximum of three dimensions. For further information about arrays, see *Variable Arrays*.

An asterisk may be used instead of a colon to delimit the size.

## Format:

```
CHARACTER{:n} name {[dimensions]}|{values} {, options}
```

## Parameters:

**n**
An unsigned integer value in the range 1–65535, representing the size of the variable in bytes. The default is 1.

**name**
The name of the variable. This must be a valid *CYRANO dataname*.

**dimensions**
The dimensions of the array to be allocated for this variable. Up to three dimensions can be specified, separated by commas, each comprising one or two numbers.

If a dimension has only one number, the elements in that dimension range from 1 to the number specified. If two numbers are specified, they must be separated by a colon (":"); the elements in this dimension range from the first number to the second.

Note that if "dimensions" is specified, "values" may not be.

**values**
A list of character values to be associated with the variable. Note that if "values" is specified, "dimensions" may not be. See *Variable Values* for further information on variable values.

**options**
A list of variable options. See *Variable Options* for further information on variable options.

## Examples:

```
CHARACTER:15 dept

CHARACTER:20 names ('TOM','JOHN','DICK'), SCRIPT

CHARACTER:9 months [12]

CHARACTER*20 staff-by-dept [8,101:150]
```

# CONSTANT STATEMENT

## Description:

This statement defines a variable which has a static value within a script. They may thus be translated at compilation time, and not consume memory at run-time.

The value of a constant may be either an integer value or a quoted character string.

Constants can be used in any situation where a literal of the same type (i.e. character or integer) can be used, for example in a value list. The only constraint is that the constant must have been defined before it is used.

## Format:

```
CONSTANT name = value
```

## Parameters:

**name**
The name of the constant. This must be a valid *CYRANO dataname*.

**value**
A quoted character string or an integer value.

## Examples:

```
CONSTANT  TRUE = -1
CONSTANT  PROMPT = 'Enter Value : '
CONSTANT  SEARCHSTRING = ' "TERMINATE" '
```

# FILE STATEMENT

## Description:

This statement declares an identifier (ID) for any external files that are accessed by this script. The FILE statement is mandatory for any files that are being passed as a parameter to the script, and optional otherwise. It is good practice, however, to formally declare all file IDs in this way before use.

## Format:

```
FILE input_fileid
```

## Parameter:

**input_fileid**
A *CYRANO dataname* used to identify a file that is passed as a parameter to the script.

## Example:

```
FILE datafile
```

# INTEGER STATEMENT

## Description:

This statement defines a variable with a positive or negative integral value. In SCL, integers are defined as being 4 bytes long, giving a range of –2147483648 to +2147483647.

Arrays of integer variables can be defined, with a maximum of three dimensions. For further information about arrays, see *Variable Arrays*.

## Format:

```
INTEGER name {[dimensions]}|{values} {, options}
```

## Parameters:

**name**
The name of the variable. This must be a valid *CYRANO dataname*.

**dimensions**
The dimensions of the array to be allocated for this variable. Up to three dimensions can be specified, separated by commas, each comprising one or two numbers.

If a dimension has only one number, the elements in that dimension range from 1 to the number specified. If two numbers are specified, they must be separated by a colon (":"); the elements in this dimension range from the first number to the second. Note that if "dimensions" is specified, "values" may not be.

**values**
A list or range of integer values to be associated with the variable.

Note that if "values" is specified, "dimensions" may not be. For further information on variable values, see *Variable Values*.

**options**
A list of variable options. For further information on variable options, see *Variable Options*.

## Examples:

```
INTEGER loop-count
INTEGER fred (1-99), SCRIPT
INTEGER values [50:100,20]
```

# TIMER STATEMENT

### Description:

The TIMER statement declares the name of a stop-watch timer. These timers may be used in conjunction with the START TIMER and END TIMER statements in the Code section of the script.

Up to 1020 timers may be declared and used in a script.

### Format:

```
TIMER name
```

### Parameter:

**name**
The name of the timer. This must be a valid *CYRANO dataname*.

### Examples:

```
TIMER Mf-Update
TIMER Cust-Reg
```

## VARIABLE ARRAYS

Character and integer variables declared within the Definitions section of a script may be defined as arrays. SCL supports arrays of up to three dimensions. There is no defined limit to the number of elements which may be declared in an array dimension.

If an array of two or three dimensions is specified, each dimension must be separated from the following dimension by a comma. When an array is referenced, array subscripts must be specified for each of its dimensions.

The numbering of the array elements is dependent on how the array was declared. SCL supports both start and end array subscript values within the array declaration itself. For example:

```
CHARACTER*9      MONTHS [1:12]
CHARACTER*9      MONTHS [12]
```

Both of these variable declarations declare an array of character variables each with 12 elements. The elements in the array are both numbered 1 to 12. Compare them with the following example:

```
CHARACTER*9      MONTHS [0:11]
```

This example also declares an array of 12 elements, but the array elements are numbered from 0 to 11.

Only positive values can be specified for the start and end array subscript values, and the start value must be less than or equal to the end value. If the start value is omitted, it defaults to 1.

When you want to retrieve a value from an array variable, you can use numeric literals, integer variables or complex arithmetic expressions to specify the element(s). For example:

```
SET Tax = Revenue [Office, Index + 1] * 0.175
```

## VARIABLE VALUES

A set of values may be associated with a variable, using a value clause in the variable definition. They are used by the GENERATE and NEXT commands, which allow the variable to be assigned a value from the list or range, either randomly (using GENERATE) or sequentially (using NEXT).

Values may be specified as a list (for integer and character variables) or as a range (integer variables only).

**Note:** Lists may contain only individual values, and not ranges. Variables which have been declared as an array may not have an associated value list or range.

A value list has the following format:

```
(value1{, value2, value3 ...})
```

The values must be of the same data type as the variable, i.e. integer values for integer variables and character values for character variables. They may be literals or constants which have previously been defined.

**Note:** In the case of character variables, the maximum size of a character constant or literal string is 65535 characters.

Ranges provide a shorthand method for defining a list of adjacent integer values and have the following format:

```
(start_value - end_value)
```

If the start value is less than the end value, the variable is incremented by 1 on each execution of the NEXT command, until the end value is reached. If the start value is greater than the end value, the variable is decremented by 1 on each execution of the NEXT command, until the end value is reached.

If the variable is set to the end value when the NEXT command is executed, the variable will be reset to the start value. You can also reset the variable explicitly, by using the RESET command.

In the following list of example variable definitions including values, the first two definitions are equivalent:

```
Integer       A         (4,3,2,1,0,-1)
Integer       B         (4 - -1)
Integer       C         (100 - 999)
Integer       D         (100,200,300,400)
Character*10  Language  ("ENGLISH", 'FRENCH', &
                         'GERMAN', "SPANISH")
Character     Control   ("~<CR>", "~<LF>", "^Z", &
                         "^X", "^U")
```

## VARIABLE OPTIONS

Additional attributes may be assigned to a variable using option clauses. Variable options follow the value definitions (if present), and are introduced by a comma. There are three types of option clause available: the first defines the scope of the variable; the second is used with variables with associated values, to define how random values are to be generated, if required; the third is used with variables that are defined as a parameter for the script.

The following sections describe the types of variable option clause.

## Variable Scope Options

The variable scope options define how widely accessible the variable is; they are mutually exclusive. The variable scope options are:

```
,LOCAL
,SCRIPT
,THREAD
,GLOBAL
```

These options are described below:

- **LOCAL**

  Local variables are only accessible to the thread running the script in which they are defined. They cannot be accessed by any other threads or scripts (including scripts referenced by the main script). Similarly, a script cannot access any of the local variables defined within any of the scripts it calls.

  Space for local variables defined within a script is allocated when the script is activated and deallocated when script execution completes.

  This is the default if no scope option is specified in the variable definition.

- **SCRIPT**

  Script variables are accessible to any thread running the script in which they are defined.

  Space for the script variables defined within a script is allocated when the script is activated and there are no threads currently running the script. If one or more threads are already running the script, the existing script variable data is used.

  The space for script variables is normally deallocated when the execution of a script terminates, and no other threads are running the script. In some cases, however, it may be desirable to retain the contents of script variables even if there is no thread accessing the script. This can be achieved by using the “,KEEPALIVE” clause on the EXIT command. The space allocated to script variables is only deleted when a thread is both the last thread accessing the script and has not specified the “,KEEPALIVE” clause. A particular use of this clause is where the script is being called by a number of threads, but there is no guarantee that there will be at least one thread accessing the script at all times.

- **THREAD**

  Thread variables are accessible from any script executed by the thread which declares an instance of them.

  The space for thread variables is deallocated when the thread completes.

  Thread variables cannot have associated value lists or ranges.

- **GLOBAL**

  Global variables are accessible to any thread running any script under the same Test Manager.

  The space for global variables is deallocated when the Test Manager in question is closed down.

  Global variables cannot have associated value lists or ranges.

## Random Variable Options

The random options are only valid for variables which have an associated set of values; they are mutually exclusive. The two random options are:

```
,RANDOM
,REPEATABLE {RANDOM} {, SEED = n}
```

These options function as follows:

- **RANDOM**

  This option indicates that a value is to be selected randomly from a list or range, when the variable is used in conjunction with the GENERATE command. The values will be selected in a different order each time they are generated; this is achieved by generating a different seed value for the variable each time the variable is initialized. Local variables are initialized when script execution begins. Script variables are initialized by the first thread to execute the script.

  This option is particularly useful when load-testing a system.

  This is the default if no random option is specified.

- **REPEATABLE {RANDOM}**

  This option indicates that a value is to be selected randomly from a list or range, when the variable is used in conjunction with the GENERATE command, but in the same order each time the script is run. This is achieved by using the same seed value for the variable each time the variable is initialized.

  This option is particularly useful in regression testing when reproducible input is required.

- **SEED = n**

  This option can be used in conjunction with the REPEATABLE RANDOM option, to specify the seed value that is to be used when generating the random sequence of numbers. This makes it possible to use a different sequence of random values for each repeatable random variable. “n” is a numeric literal in the range –2147483648 to +2147483647.

## File Option

The variable file option associates an ASCII text file of values - one per line - with a variable:

, FILE = <filename>

where “filename” is the name of the ASCII text file, excluding the pathname and file extension. The file must always reside in a project's data directory and the file extension must always be ".fvr".

The file is used by the NEXT command, which allows the variable to be assigned a value from the file sequentially.

Values are held in the file with one value per line. The values must be of the same data type as the variable, i.e. integer values for integer variables and character values for character variables. For example, a file for an integer variable could contain the values:

-100

0

100

A file for a character variable could contain the values:

Pele

10

Cruyff

14

Note that SCL character commands are not recognised within the file variable files - the file should contain raw ASCII characters only.

Values are retrieved from the file associated with a variable using the NEXT command. This command retrieves the next sequential value from the file. When the NEXT command is first executed, it will retrieve the first value from the file. If the variable is set to the last value in

the file when the NEXT command is executed, the variable will be reset to the first value in the file. You can also reset the variable explicitly, by using the RESET command.

The file option is NOT valid for variables which:

1. Have an associated value list
2. Have been declared as an array
3. Are part of a record

## EXAMPLE VARIABLE DEFINITIONS

This section shows a number of example variable definitions:

```
Integer          Isub      (100,200,300,400)
Integer          ERRCOUNT  ,Global
Integer          Jsub      (-400,-300,-200), Local, Random
Integer          B          ,Script, Repeatable, Seed=30352
Integer          Prdcod    ,File=”prd_codes”
Character:24     surname
Character*10     Alph      ("A","C","E"), Repeatable Random
Character*80     Prddsc    ,File=”prd_descriptions”
Constant         TAXrate = 17.5
Constant         confirm = "Confirm [Y/N] :"
```

# THE CODE SECTION

The mandatory Code section of the SCL source file contains all the commands that define the script's behavior.

A script source file must contain a (single) Code section as the last section in the file. It is introduced by the mandatory CODE command.

## CODE SECTION STRUCTURE

The Code section of an SCL source file is composed of:

- **Commands**

  SCL provides a wide range of commands that control the behavior of the script.

  A command is normally terminated by the end of the source line, but may be continued on a subsequent line by specifying the continuation character as the last character on a line - apart for any line comment. Either an ampersand or a hyphen ("&" or "–") may be used as the continuation character; this is described in *Continuation Lines*.

  Spaces and tabs are treated as separators within a command, although spaces are significant when they appear in character string arguments.

- **Characters ignored by the compiler**

  The script compiler allows any character with an ASCII value in the range HEX 00 to 20 or HEX 81 to 8F inclusive to appear at the start of a line or the end of a line. It ignores these characters, allowing tabs and form-feeds, for example, to be used to aid legibility.

  If any ASCII control character appears elsewhere, the script compiler will generate a compilation error.

## COMMAND TYPES

SCL offers a large number of commands to support the creation of powerful and flexible scripts. These fall into a number of distinct categories:

- **HTTP Commands**
- **Input Stream Entry Commands**
- **Output Stream Handling Commands**
- **Flow Control Commands**
- **File Handling Commands**
- **Formal Test Control Commands**
- **Synchronization Commands**
- **Statistical Data Logging Commands**
- **Diagnostic Commands**
- **Miscellaneous Commands**

## SCRIPT PROCESSING

When a script is executed, the first command in the script is selected and executed.

Commands are processed sequentially, unless a command that alters the flow of control is executed, in which case processing continues at the defined point in the script.

A script terminates when the end of the script is reached, when an EXIT, or DETACH {THREAD} command is executed, or when an error is detected and error trapping is not enabled for the script.

## VARIABLES

All variables accessed by a script **must** be pre-defined in the Definitions section of the script. If an undefined variable is accessed from within an SCL source file, a syntax error will be reported.

All integer variables are initially set to zero, and character variables are empty

## LABELS

Labels may be used to identify SCL statements. A label consists of label name followed by a colon. For example:

```
REQ_TIMEOUT:  LOG "HTTP GET", url, "timed out"
```

A label name must be a valid *CYRANO dataname*.

Any defined subroutines may not reference labels defined in other sections of the code, since labels are local to the module within which they are defined.

## SYMBOLS

During compilation, the compiler maintains symbol tables of all the symbols it has encountered, so that it may resolve references to them. There are separate symbol tables for tests, variables, timers, labels and threads.

All symbols within a symbol table must be unique. However, the use of separate symbol tables allows, for instance, the same name to be used for a label as for a variable.

Furthermore, because labels are not propagated into subroutines or vice versa, labels within a subroutine may duplicate labels within other subroutines, or within the main body of the code.

## CODE SECTION COMMANDS

This section describes the commands that can be included in the Code section of a script source file.

The Code section can also contain character commands, labels and comments. Further information on these items is given in *Overview of Script Control Language Syntax*.

For easier reference some HTTP-specific commands are documented in two places. They are grouped together under the heading “HTTP Commands” and are also found under other relevant headings.

## HTTP Commands

The HTTP commands provide facilities for issuing HTTP requests for resources, examining/ interrogating the response messages and synchronizing requests. These commands are only available in scripts which contain the MODE HTTP statement in their Environment section

## CONNECT Command

### Description:

This command may be used to establish a TCP connection to a nominated host. It is only valid within a script that has been defined as MODE HTTP.

This command specifies an ID for the TCP connection. This may be used in subsequent GET, HEAD, POST and LOAD RESPONSE_INFO commands to use this TCP connection. The TCP connection may be closed using the DISCONNECT command. It will also be terminated when the thread exits the script.

The connection ID specified must not correspond to a TCP connection already established previously using the CONNECT command. Otherwise a script error will be reported.

### Format:

```
CONNECT TO host ON conid
```

### Parameters:

**host**

A character variable, quoted character string or character expression, containing the host name or IP address of the resource to connect to and, optionally, the port number on which the connection is to be made. If a port is specified, it must be separated from the host field by a colon (":"). If the port number field is empty or not specified, the port defaults to TCP 80.

**conid**

An integer variable, integer value or integer expression defining the connection ID. This is used in all subsequent operations on this connection.

### Examples:

```
CONNECT TO "proxy.dev.mynet:3128" ON 1
CONNECT TO myhost ON 2
CONNECT TO 'abc.com' ON conid
```

## DISCONNECT Command

### Description:

This command closes one or all of the TCP connections established using the CONNECT command. It is only valid within a script that has been defined as MODE HTTP.

If the "FROM conid" clause is specified, the TCP connection identified by that Connection ID will be closed. If the "ALL" keyword is used, all TCP connections established by the current thread will be closed.

By default, the DISCONNECT command will wait until any requests on the connection(s) to be closed are complete before closing them. If the WITH CANCEL clause is specified, the connection(s) will be closed immediately.

The Connection ID specified must correspond to a TCP connection established using the CONNECT command, otherwise a script error will be reported.

### Format:

```
DISCONNECT [FROM conid | ALL ]  {,WITH CANCEL}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection to be closed.

### Examples:

```
DISCONNECT FROM 1
DISCONNECT FROM conid
DISCONNECT FROM 1, WITH CANCEL
DISCONNECT ALL
DISCONNECT ALL, WITH CANCEL
```

## GET Command

### Description:

This command issues an HTTP GET request for a specified resource. It is only valid within a script that has been defined as MODE HTTP.

The request header fields are obtained from the HEADER clause. These can be modified using the WITH and WITHOUT clauses.

The HTTP GET request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record is written when the request message is sent, and the second is written on receipt of the response request message from the server.

The response code in the response message can be retrieved by using the optional "RETURNING CODE response_code " clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause can be used to specify the integer variable to hold one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be established to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message will be written to the audit log and the thread will be aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

**Format:**

```
GET [ URI | URL ] uri-httpversion          {&}
ON conid                                   {&}
HEADER http_header                         {&}
{,WITH header_value}                       {&}
{,WITHOUT header_field}                    {&}
{,RESPONSE TIMER timer_name}               {&}
{,RETURNING STATUS response_status}        {&}
{,RETURNING CODE response_code}
```

**Parameters:**

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request header fields.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request header fields. These request-header fields are added to those specified in "http_header". If a request-header field appears in both "http_header" and "header_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

### Examples:

```
GET URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
HEADER sub_header &
,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/")
GET URI "http://abc.com/~~pascal/don.gif HTTP/1.0" ON 2 &
HEADER sub_header &
,WITH  " Host: abc.com" &
,WITHOUT "Referer Accept-Language"
```

## HEAD Command

### Description:

This command issues an HTTP HEAD request for a specified resource. It is only valid within a script that has been defined as MODE HTTP.

The request header fields are obtained from the HEADER clause. These can be modified using the WITH and WITHOUT clauses.

The HTTP HEAD request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record is written when the request message is sent, and the second is written on receipt of the response request message from the server.

The response code in the response message can be retrieved by using the optional "RETURNING CODE response_code " clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause can be used to specify the integer variable to hold one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be establised to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message will be written to the audit log and the thread will be aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

**Format:**

```
GET [ URI | URL ] uri-httpversion          {&}
ON conid                                   {&}
HEADER http_header                         {&}
{,WITH header_value}                       {&}
{,WITHOUT header_field}                    {&}
{,RESPONSE TIMER timer_name}               {&}
{,RETURNING STATUS response_status}        {&}
{,RETURNING CODE response_code}
```

**Parameters:**

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request-header fields.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request-header fields. These request header fields are added to those specified in "http_header". If a request header field appears in both "http_header" and "http_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

### Examples:

```
HEAD URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
HEADER sub_header &
,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/")
HEAD URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON 2 &
HEADER sub_header &
,WITH  " Host: abc.com" &
,WITHOUT "Referer Accept-Language"
```

## LOAD RESPONSE_INFO BODY Command

### Description:

This command loads a character variable with all or part of the data from an HTTP response message body for a specified TCP connection. It is used after a GET, HEAD or POST command.

OpenSTA will automatically wait until any request on the specified connection ID is complete before executing this command. It is not necessary for the script to do this explicitly.

If the data string is too long to fit into the target variable, it will be truncated.

The WITH clause can be used to specify a DOM (Document Object Model) signature identifying the data to be returned. If this clause is omitted, the full response message body is loaded into the target variable.

### Format:

```
LOAD RESPONSE_INFO BODY ON conid  INTO variable        {&}
{,WITH identifier}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which the HTTP response message will be received.

**variable**

The name of a character variable into which the HTTP response message body, or the selected part of it, are loaded.

**identifier**

A character variable, quoted character string or character expression containing a DOM (Document Object Model) signature identifying the data to be retrieved from the response message body.

### Example:

```
LOAD RESPONSE_INFO BODY ON 1 INTO post_body
```

## LOAD RESPONSE_INFO HEADER Command

### Description:

This command loads a character variable with all or some of the HTTP response message header fields for a specified TCP connection.

OpenSTA will automatically wait until any request on the specified Connection ID is complete before executing this command. It is not necessary for the script to do this explicitly.

If the data string is too long to fit into the target variable, it will be truncated.

The WITH clause can be used to specify the names of one or more header fields whose values are to be retrieved from the HTTP response message. If this clause is omitted, all the response message header fields are retrieved.

### Format:

```
LOAD RESPONSE_INFO HEADER ON conid  INTO variable      {&}
{,WITH identifier}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the connection ID of the TCP connection on which the HTTP response message will be received.

**variable**

The name of a character variable into which the HTTP response message headers, or the selected headers, are loaded.

**identifier**

A character variable, quoted character string or character expression containing the names of one or more response message header fields to be retrieved.

### Example:

```
LOAD RESPONSE_INFO HEADER ON 4 INTO resp_headers
```

## POST Command

### Description:

This command issues an HTTP POST request for a specified resource. It is only valid within a script which has been defined as MODE HTTP.

The request field headers to be used in the request are obtained from the HEADER clause, appropriately modified by the WITH and WITHOUT clauses, if specified.

The HTTP POST request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record will be written when the request message is sent, and the second written on receipt of the response request message from the server.

The status code in the response message may be retrieved by using the optional "RETURNING CODE response_code" clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause may be used to return one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, that defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be established to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message is written to the audit log and the thread is aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

The "ON TIMEOUT GOTO tmo_label" clause can be specified to define label to which control will be transferred if the request times out.

**Format:**

```
POST [ URI | URL ] uri-httpversion          {&}
ON conid                                    {&}
HEADER http_header                          {&}
{,{BINARY} BODY http_body}                  {&}
{,WITH header_value}                        {&}
{,WITHOUT header_field}                     {&}
{,RESPONSE TIMER timer_name}                {&}
{,RETURNING STATUS response_status}         {&}
{,RETURNING CODE response_code}             {&}
{,WITH TIMEOUT period {, ON TIMEOUT GOTO tmo_label}}
```

**Parameters:**

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request header fields.

**http_body**

A character variable, quoted character string or character expression containing the request body.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request header fields. These request header fields are added to those specified in "http_header". If a request header field appears in both "http_header" and "http_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

**period**

An integer variable, integer value or integer expression defining the number of seconds to wait before an unsatisfied request is timed out. The valid range is 0 - 32767.

**tmo_label**

A label defined within the current scope of the script, to which control branches if a time-out occurs.

### Examples:

```
POST URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
HEADER sub_header &
,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/") &
,WITH TIMEOUT 60, ON TIMEOUT GOTO post_timeout

POST URL "http://dogbert.abebooks.com/abe/IList HTTP/1.0" on
SEARCH_PAGE &
HEADER post_header &
,WITH ("Host: dogbert.abebooks.com", &
"Referer: http://dogbert.abebooks.com/abe/IList") &
,BODY "bu=New+Search"

POST URI "http://abc.com/~pascal/don.gif HTTP/1.0" ON 2 &
HEADER sub_header &
```

```
,WITH  " Host: abc.com" &
,WITHOUT "Referer Accept-Language"
```

## SYNCHRONIZE REQUESTS Command

### Description:

HTTP requests are issued asynchronously. Immediately after an HTTP request has been issued, the next command in the script is processed. OpenSTA does not wait for a response to be received for an HTTP request.

This command causes the thread currently executing to be suspended immediately, until responses have been received for all the requests that have been issued by the thread. It is only valid within a script that has been defined as MODE HTTP.

The 'ON TIMEOUT GOTO tmo_label' clause can be specified to define the label to which control will be transferred if the request times out.

### Format:

```
[SYNCHRONIZE | SYNCHRONISE]  REQUESTS {&}
{, WITH TIMEOUT period, ON TIMEOUT GOTO tmo_label}
```

### Parameters

**period**

An integer variable, integer value or integer expression defining the number of seconds to wait before the command is timed out. The valid range is 0 - 32767.

**tmo_label**

A label defined within the current scope of the script, to which control branches if a timeout occurs.

### Examples:

```
SYNCHRONIZE REQUESTS
SYNCHRONISE REQUESTS &
, WITH TIMEOUT 60, ON TIMEOUT GOTO timed_out
```

## Input Stream Entry Commands

Input stream entry commands control how the script feeds input to the system under test.

## GENERATE Command

### Description:

This command loads a random value from a set of values into a variable.

The variable must have a list or range of values associated with it in the Definitions section. If it is defined as “REPEATABLE RANDOM”, values will be retrieved in the same random order on every run. If it is defined as “RANDOM”, values will be retrieved in different random sequences each run.

### Format:

```
GENERATE variable
```

### Parameter:

**variable**
The name of the variable into which the generated value is to be loaded. The variable must have a set of values associated with it in the Definitions section.

### Example:

```
GENERATE Part-Number
```

## GET Command

### Description:

This command issues an HTTP GET request for a specified resource. It is only valid within a script that has been defined as MODE HTTP.

The request header fields are obtained from the HEADER clause. These can be modified using the WITH and WITHOUT clauses.

The HTTP GET request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record is written when the request message is sent, and the second is written on receipt of the response request message from the server.

The response code in the response message can be retrieved by using the optional "RETURNING CODE response_code " clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause can be used to specify the integer variable to hold one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be established to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message will be written to the audit log and the thread will be aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

**Format:**

```
GET [ URI | URL ] uri-httpversion          {&}
ON conid                                   {&}
HEADER http_header                         {&}
{,WITH header_value}                       {&}
{,WITHOUT header_field}                    {&}
{,RESPONSE TIMER timer_name}               {&}
{,RETURNING STATUS response_status}        {&}
{,RETURNING CODE response_code}
```

**Parameters:**

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request header fields.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request header fields. These request-header fields are added to those specified in "http_header". If a request-header field appears in both "http_header" and "header_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

### Examples:

```
GET URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
HEADER sub_header &
,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/")
GET URI "http://abc.com/~~pascal/don.gif HTTP/1.0" ON 2 &
HEADER sub_header &
,WITH  " Host: abc.com" &
,WITHOUT "Referer Accept-Language"
```

## HEAD Command

### Description:

This command issues an HTTP HEAD request for a specified resource. It is only valid within a script that has been defined as MODE HTTP.

The request header fields are obtained from the HEADER clause. These can be modified using the WITH and WITHOUT clauses.

The HTTP HEAD request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record is written when the request message is sent, and the second is written on receipt of the response request message from the server.

The response code in the response message can be retrieved by using the optional "RETURNING CODE response_code " clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause can be used to specify the integer variable to hold one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be establised to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message will be written to the audit log and the thread aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

### Format:

```
GET [ URI | URL ] uri-httpversion          {&}
ON conid                                   {&}
HEADER http_header                         {&}
{,WITH header_value}                       {&}
{,WITHOUT header_field}                    {&}
{,RESPONSE TIMER timer_name}               {&}
{,RETURNING STATUS response_status}        {&}
{,RETURNING CODE response_code}
```

### Parameters:

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request-header fields.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request-header fields. These request header fields are added to those specified in "http_header". If a request header field appears in both "http_header" and "http_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

### Examples:

```
 HEAD URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
 HEADER sub_header &
 ,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/")
 HEAD URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON 2 &
 HEADER sub_header &
 ,WITH  " Host: abc.com" &
,WITHOUT "Referer Accept-Language"
```

## NEXT Command

### Description:

This command loads a variable with the next sequential value from a set of values. This could be either a list or a range associated with that variable, or from a file associated with the variable.

When the NEXT command is first executed, it will retrieve the first value. The set is treated as cyclic: when the last value has been retrieved, the next value retrieved will be the first in the set.

This command may be used to reset the value pointer associated with a variable so that the first NEXT command to be executed after the RESET retrieves the first value in the set.

The variable must have a set of values or a file associated with it in the Definitions section.

### Format:

```
NEXT variable
```

### Parameter:

**variable**
The name of a variable into which the next value from the set is loaded. The variable must have a set of values or a file associated with it in the Definitions section.

### Example:

```
NEXT Emp-Name
```

## POST Command

### Description:

This command issues an HTTP POST request for a specified resource. It is only valid within a script which has been defined as MODE HTTP.

The request field headers to be used in the request are obtained from the HEADER clause, appropriately modified by the WITH and WITHOUT clauses, if specified.

The HTTP POST request is asynchronous. Immediately after the request is issued, the next command in the script is processed - it does not wait for a response message to be received.

There is an optional "RESPONSE TIMER" clause, which can be used to specify that a pair of response timer records are to be written to the statistics log. The first record will be written when the request message is sent, and the second written on receipt of the response request message from the server.

The status code in the response message may be retrieved by using the optional "RETURNING CODE response_code" clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional "RETURNING STATUS response_status" clause may be used to return one of two values indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, that defines SCL integer constants for both the response code and response status values.

The TCP connection used for the request depends upon whether a connection has already been established for the specified Connection ID using the CONNECT command. If it has, the request uses that connection. If it has not, a TCP connection will be established to the host identified by the uri-httpversion, on port 80.

By default, if an error occurs while establishing the TCP connection or issuing the request, an error message is written in the audit log and the thread is aborted. However, if error trapping is enabled, control will be transferred to the error-handling code.

The "ON TIMEOUT GOTO tmo_label" clause can be specified to define label to which control will be transferred if the request times out.

### Format:

```
POST [ URI | URL ] uri-httpversion         {&}
ON conid                                   {&}
HEADER http_header                         {&}
{,{BINARY} BODY http_body}                 {&}
{,WITH header_value}                       {&}
{,WITHOUT header_field}                    {&}
{,RESPONSE TIMER timer_name}               {&}
{,RETURNING STATUS response_status}        {&}
{,RETURNING CODE response_code}            {&}
{,WITH TIMEOUT period {, ON TIMEOUT GOTO tmo_label}}
```

### Parameters:

**uri-httpversion**

A character variable, quoted character string or character expression, containing the URI (Uniform Resource Identifier) of the resource upon which to apply the request, and the HTTP Version, separated by a single space character. The HTTP Version indicates the format of the message and the sender's capacity for understanding further HTTP communication.

**conid**

An integer variable, integer value or integer expression identifying the connection ID of the TCP connection on which to issue the request.

**http_header**

A character variable, quoted character string, character expression or character value list containing the request header fields.

**http_body**

A character variable, quoted character string or character expression containing the request body.

**header_value**

A character variable, quoted character string, character expression or character value list containing zero or more request header fields. These request header fields are added to those specified in "http_header". If a request header field appears in both "http_header" and "http_value", the field specified here overrides that specified in "http_header".

**header_field**

A character variable, quoted character string, character expression or character value list containing the request header field names of fields to be excluded from the request.

**timer_name**

The name of a timer declared in the Definitions section of the script.

**response_status**

An integer variable into which the response status of the HTTP response message is loaded when the HTTP response message is received.

**response_code**

An integer variable into which the response code of the HTTP response message is loaded when the HTTP response message is received.

**period**

An integer variable, integer value or integer expression defining the number of seconds to wait before an unsatisfied request is timed out. The valid range is 0 - 32767.

**tmo_label**

A label defined within the current scope of the script, to which control branches if a time-out occurs.

### Examples:

```
POST URL "http://abc.com/~~pascal/don.gif HTTP/1.0" ON conid &
HEADER sub_header &
,WITH  (" Host: abc.com", "Referer: http://abc.com/~~pascal/") &
,WITH TIMEOUT 60, ON TIMEOUT GOTO post_timeout

POST URL "http://dogbert.abebooks.com/abe/IList HTTP/1.0" on
SEARCH_PAGE &
HEADER post_header &
,WITH ("Host: dogbert.abebooks.com", &
"Referer: http://dogbert.abebooks.com/abe/IList") &
,BODY "bu=New+Search"

POST URI "http://abc.com/~pascal/don.gif HTTP/1.0" ON 2 &
HEADER sub_header &
,WITH  " Host: abc.com" &
```

```
,WITHOUT "Referer Accept-Language"
```

## RESET Command

### Description:

This command resets the value pointer for a variable to the first value in the associated value set. This could be either a list or a range associated with that variable, or from a file associated with the variable. In the case of a repeatable random variable, the variable's seed may be reset to a specified or defaulted value.

The RESET command does not alter the contents of the variable. The value to which the variable has been reset is only retrieved on execution of the first NEXT command after the RESET command.

### Format:

```
RESET variable{, SEED=value}
```

### Parameters:

**variable**
The name of the variable whose value pointer is to be reset. The variable must have a set or a file associated with it in the Definitions section.

**value**
An integer numeric literal in the range –2147483648 to +2147483647. If the "SEED" clause is omitted from the RESET command, the seed variable will be reset to the value specified when the variable was defined, or to the value specified by a previous RESET command.

### Examples:

```
RESET Emp-Name
RESET Per-Num, SEED=-8415
```

## SET Command

### Description:

This command allows a value to be assigned to an integer or character variable. The values may be any integer or character values or a function reference, but their data types must match that of the variable. The values may be derived as a result of arithmetical operations.

If the variable is an **integer variable**, the assignment expression may be another integer variable or a numeric literal, or a complex arithmetic expression consisting of two or more integer values or variables, each separated by an operator. The following operators are supported:

| | | | |
|---|---|---|---|
| + | for addition | – | for subtraction |
| * | for multiplication | / | for division |
| \| | for modulo | | |

The value resulting from a division operation will be an integer, i.e. the remainder will be ignored. The modulo calculation is the converse of this operation, i.e. the variable will be set to the value of the remainder. For example:

```
SET A = B / C
SET D = B | C
```

If B = 13 and C = 2, then A will be set to 6 and D to 1.

Parentheses may be specified to determine the order of precedence. If parentheses are not specified, then the expression is evaluated from left to right with no other order of precedence applied.

Care should be taken when using arithmetic expressions, since there is no check for integer overflow at run-time. If an integer overflow occurs a script error will be reported.

If the variable is a **character variable**, the assignment expression may consist of one or more character variables or literals. Operands are separated by the addition operator if the operands are to be added together; if the second operand is to be subtracted from the first, they are separated by the subtraction operator.

The character function ~EXTRACT may be referenced within a SET command to extract a substring from a character variable or quoted character string into a character variable.

The integer function ~LOCATE may be referenced within a SET command to load the offset of a substring within a character variable or quoted character string into an integer variable.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error. An error could occur if, for example, an ~EXTRACT function is specified with an invalid offset, or an attempt is made to divide by zero.

### Format:

```
SET variable = operand1 { operator operand &
 {operator operand...} } {ON ERROR GOTO err_label}
```

### Parameters:

**variable**
The name of an integer or character variable into which the result of the operation is to be placed.

**operand1**
The value from which the initial operation result will be taken. For a character SET command, the operand may be a character variable, quoted character string or character function reference. For integer SET commands, the operand may be an integer function reference, literal or variable.

**operator**
The operation which is to be performed upon the previous and following operands. For character SET commands, it may be "+" to add the first operand to the second, or "–" to subtract the second operand from the first. For integer SET commands, all operators are valid.

**operand**
The variable or value which is used to modify the current value for "variable". For a character SET command, the operand may be a character variable, quoted character string or character function reference. For integer SET commands, the operand may be an integer literal or variable.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
SET STRING1 = STRING2 - "ERROR"

SET STRING1 = STRING2 + STRING3 + STRING4

SET STRING1 = STRING2 - '"END MARKER"' &
            ON ERROR GOTO Error_report
```

## Output Stream Handling Commands

Output stream handling commands control how thread scripts examine and manipulate output from the system, either within the script itself or by saving the data for later comparison.

## ~EXTRACT Command

### Description:

This command is a function and can only be referenced within a SET command. It returns the portion of the source string identified by the specified offset and length.

If the string identified by the offset and length overlaps the end of the source string, only the characters up to the end of the source string will be returned.

If the offset does not lie within the bounds of the source string when the script is executed, a message will be written to the audit log, indicating that a bad parameter value has been specified. Script execution will then be aborted, or the specified action taken if error trapping is enabled via the ON ERROR command.

### Format:

```
~EXTRACT (offset, length, string)
```

### Return Value:

The character substring extracted from the source string.

### Parameters:

**offset**
An integer variable or value defining the offset in the source string of the first character that is to be extracted. The first character of the source string is at offset zero.

**length**
An integer variable or value defining the number of characters to extract to form the substring.

**string**
The character value or character variable from which the substring is to be extracted.

### Example:

```
SET NameCode = ~EXTRACT (0, 4, Name) + RunningNo
```

## FORMAT Command

### Description:

This command translates characters from one format into another. This makes it easier to manipulate character strings that have been output from the system under test, or which are to be input into that system.

In all translations, the command requires three elements:

1. The target variable that will receive the translated value. This may be either a character variable or an integer variable.
2. A format string defining the type of translation required. For an integer target variable, the format string must only contain a single format identifier; for a character variable, the format string may contain multiple identifiers and/or ordinary characters that are to be copied unchanged to the target variable.
3. One or more values to be translated; these may be specified as variables or as literal text. A single value must be specified for each of the format identifiers in the format string; the data type of each must agree with the associated format identifier and the data type of the target variable, as discussed below. Note that any discrepancies in this respect are detected at run-time and are not picked up by the compiler.

   The following types of translation are supported:

   **%U –** Translate each alphabetic character in the input string into its uppercase equivalent. Both source and target variables must be character variables. The source string if necessary is truncated to fit the target variable.

   **%L –** Translate each alphabetic character in the input string into its lowercase equivalent. Both source and target variables must be character variables. The source string if necessary is truncated to fit the target variable.

   **%D –** Convert a character string date value into numeric format (representing the number of days since the Smithsonian base date of 17–Nov–1858). The target variable must be an integer variable, and the source variable a character string containing a valid date; this can be either in the default style for the platform on which the script is running or in the fixed format “DD–MMM–CCYY” (where “CC” is optional).

   This format identifier may also be used to convert a numeric date value (representing the number of days since the Smithsonian base date of 17–Nov–1858) into a character string in the fixed format “DD–MMM–CCYY”. The source variable must be an integer variable and the target variable a character string, which will be truncated if necessary.

   **%T –** Convert a character string time value into a numeric format (representing the number of 10 milli-second ‘ticks’ since midnight). The target variable must be an integer variable, and the source variable a character string containing a valid time;

this can be either in the default style for the platform on which the script is running or in the form “HH:MM:SS.MMM” (where “.MMM” is optional).

This format identifier may also be used to convert a numeric time value (representing the number of 10 milli-second ticks since midnight) into a character string in the fixed format “HH:MM:SS.MMM”. The source variable must be an integer variable and the target variable a character string, which will be truncated as required.

### Format:

```
FORMAT (target-variable, format-string, variable {,variable ...}) {&}
 {{,}ON ERROR GOTO err_label}
```

### Parameters:

**target-variable**
The name of an integer or character variable into which the result of the operation is placed.

**format-string**
A quoted character string containing the string to be formatted and containing a number of format identifiers. The format identifiers must be compatible with the data types of the variables that follow.

**variable**
One or more integer or character variables or literals to be translated. The number of variables must correspond with the number of format identifiers in the format string. The data type of each variable must match the corresponding format identifier and the target variable.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
FORMAT (date_string, "The date is %D today, and the time is %T",  &
         int-date, int-time), ON ERROR GOTO end

FORMAT (date_value, "%D", char-date), ON ERROR GOTO format_err

FORMAT (uppercase_string, "Name in uppercase is %U", lowercase_string)
```

## LOAD RESPONSE_INFO BODY Command

### Description:

This command loads a character variable with all or part of the data from an HTTP response message body for a specified TCP connection. It is used after a GET, HEAD or POST command.

OpenSTA will automatically wait until any request on the specified connection ID is complete before executing this command. It is not necessary for the script to do this explicitly.

If the data string is too long to fit into the target variable, it will be truncated.

The WITH clause can be used to specify a DOM (Document Object Model) signature identifying the data to be returned. If this clause is omitted, the full response message body is loaded into the target variable.

### Format:

```
LOAD RESPONSE_INFO BODY ON conid  INTO variable        {&}
{,WITH identifier}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection on which the HTTP response message will be received.

**variable**

The name of a character variable into which the HTTP response message body, or the selected part of it, are loaded.

**identifier**

A character variable, quoted character string or character expression containing a DOM (Document Object Model) signature identifying the data to be retrieved from the response message body.

### Example:

```
LOAD RESPONSE_INFO BODY ON 1 INTO post_body**
```

## LOAD RESPONSE_INFO HEADER Command

### Description:

This command loads a character variable with all or some of the HTTP response message header fields for a specified TCP connection.

OpenSTA will automatically wait until any request on the specified Connection ID is complete before executing this command. It is not necessary for the script to do this explicitly.

If the data string is too long to fit into the target variable, it will be truncated.

The WITH clause can be used to specify the names of one or more header fields whose values are to be retrieved from the HTTP response message. If this clause is omitted, all the response message header fields are retrieved.

### Format:

```
LOAD RESPONSE_INFO HEADER ON conid  INTO variable    {&}
{,WITH identifier}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the connection ID of the TCP connection on which the HTTP response message will be received.

**variable**

The name of a character variable into which the HTTP response message headers, or the selected headers, are loaded.

**identifier**

A character variable, quoted character string or character expression containing the names of one or more response message header fields to be retrieved.

### Example:

```
LOAD RESPONSE_INFO HEADER ON 4 INTO resp_headers
```

## ~LOCATE Command

### Description:

This command is a function and can only be referenced within a SET command. It returns an integer value, corresponding to the offset of the specified substring in the source string. The offset of the first character in the source string is zero. If the substring is not found, the function returns a value of –1.

By default, the matching is case sensitive. The strings “London” and “LONDON”, for example, would not produce a match, because the case of the characters is not the same. This can be overridden by specifying the “, CASE_BLIND” clause.

The source string is scanned from left to right. If the substring appears more than once in the source string, the function will always return the offset of the first occurrence.

### Format:

```
~LOCATE (substring, string)                                    {&}
        {,CASE_BLIND}
```

### Return Value:

The offset of the substring in the source string. If the substring was not found, then a value of –1 is returned.

### Parameters:

**substring**
The character value defining the substring to be located in the source string. This may be a character variable or quoted character string.

**string**
The character value to be searched for the specified substring. This may be a character variable or quoted character string.

### Example:

```
SET Offset = ~LOCATE (Separator, TEST), CASE_BLIND
```

## Flow Control Commands

Flow control commands determine which sections of a script are processed, and in what order.

## CALL Command

### Description:

This command calls a subroutine from within a script. Subroutines must follow the main code section and must not be embedded within it. They share the variable definitions of the main module.

It is not possible to branch into or out of a subroutine, because a label cannot be referenced outside of the main module or subroutine in which it occurs. This does mean, however, that each subroutine enables a script to define up to 255 labels in addition to those used in the main code.

A maximum of eight parameters may be passed from the calling code to the called subroutine. The parameters passed may be character or integer variables, literals or quoted character strings. The calling code must pass exactly the same number of parameters to the called subroutine as the called subroutine has defined in its SUBROUTINE statement. The names of the variables in the call need not be the same as in the subroutine parameter list, but the data types of each of the parameters must match. Failure to comply with these conditions will result in a script error being generated.

The values of the variables defined as parameters in the subroutine definition are not copied back to the variables in the call, on return from the subroutine. However, if the same variable names are used in the call and the subroutine parameter list, the value of the variable in the call will be changed by a change in the subroutine; this is because the calling code and the called subroutine share the same data definitions. Conversely, if different variable names are used, any changes made to variables within the subroutine will not affect the variables in the call.

### Format:

```
CALL subroutine {[parameter{, parameter ...}]}
```

### Parameters:

**subroutine**
The name of the called subroutine. The name must be a valid *CYRANO dataname*.

**parameter**
A character variable, integer variable, integer value or a quoted character string. Up to 8 parameters may be declared in the CALL command. There must be the same number of parameters in this list as are in the subroutine's definition, and the data types of the parameters must match.

### Examples:

```
CALL DATE_CHECK
CALL CREATE_FULL_NAME [char_first,char_second,char_title]
```

## CALL SCRIPT Command

### Description:

This command calls a thread script from another thread script. When the command is executed, control is transferred to the called script; when the called script exits, control is returned to the calling script, optionally returning a status from the called script. There is no limit on the number of scripts that may be referenced by any one script.

In general, a called script is considered as an extension to the calling script, and any changes made in the called script are propagated back to the calling script on exit. However, certain changes (e.g. further ON ERROR handlers) only remain in force for the duration of the called script (or scripts called by it); the original condition is re-established when control is returned to the calling script.

For thread scripts, a maximum of eight parameters may be passed from the calling script to the called script. An omitted parameter is specified by two consecutive commas ",,". The calling script must pass exactly the same number of parameters to the called script as the called script has defined in its ENTRY statement (accounting for any omitted parameters). In addition, the data types of each of the parameters must match. Failure to comply with these conditions will result in a script error being generated.

The values of the parameters are passed from the caller into the variables defined within the ENTRY statement of the called script. Any modifications to the values of the variables are copied back to the caller on return from the called script.

An optional status value can be returned from the called script by using the "RETURNING" clause to specify the integer variable which is to hold the return status value.

By default, if an error occurs in a called script, an error message is written to the audit log and the thread aborts; control is **not** returned to the calling script. However, if error trapping is enabled in the calling script and the error was a script error, then control **will** be returned to the calling script's error-handling code.

The "ON ERROR GOTO err_label" clause can be specified to define a label to which control should be transferred in the event of an error while attempting to call the script.

### Format:

```
CALL SCRIPT name                                                {&}
        {[parameter{, parameter ...}]}                          {&}
         {RETURNING status} {ON ERROR GOTO err_label}
```

### Parameters:

**name**
A character variable or quoted character string defining the name of the script to be called. The name must be a valid *CYRANO dataname*.

**parameter**
A character variable, integer variable, quoted character string, integer value or file ID to be passed to the called script. A maximum of 8 parameters may be passed between scripts.

**status**
An integer variable to receive the returned status from the called script. If no status is returned from the called script, then this variable will contain the last status returned from any called script.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
CALL SCRIPT Script-Name

CALL SCRIPT "TEST"

CALL SCRIPT "CALC_TAX" [COST, RATE, TAX]

CALL SCRIPT "GET_RESPONSE" returning Response ON ERROR GOTO Problem
```

## CANCEL ON Command

### Description:

This command terminates the automatic trapping of script errors, which is enabled with the ON ERROR command. Any script errors encountered will cause the thread to be aborted.

This command will only affect automatic trapping of script errors within the current script or scripts called by it. On exit from this script, any ON ERROR handler established by a calling script will be re-established.

### Format:

```
CANCEL ON {ERROR}
```

### Parameters:

None

### Examples:

```
CANCEL ON
CANCEL ON ERROR
```

## DETACH Command

### Description:

This command causes the current thread to exit. The program exits from any scripts or subroutines that have been called (including nested calls) until control returns to the primary script. The thread is then detached from the run-time component.

### Format:

```
DETACH {THREAD}
```

### Parameters:

None

### Examples:

```
DETACH
DETACH THREAD
```

## DO Command

### Description:

The DO and ENDDO commands allow a set of commands to be repeated a fixed number of times. The section of a script to be repeated is terminated by an ENDDO command.

### Format:

```
DO variable = value1, value2 {, step}
    command{s}
ENDDO
```

### Parameters:

**variable**
The name of the control or index variable that is adjusted each time the loop executes. The adjustment is determined by the value of the step variable. This must be an integer variable.

**value1**
The starting value of the control variable. This must be either an integer variable or an integer value.

**value2**
The terminating value of the control variable. This must be an integer variable or value, and may be either higher or lower than **value1**. When the control variable contains a value that is greater than this value (or lower if the step is negative), the loop will be terminated.

**step**
An integer variable or value determining the value by which the control variable or index variable is incremented each time the loop executes. If **value2** is less than **value1**, then the **step** value must be negative. If a step variable is not specified, then the **step** value will default to 1.

### Examples:

```
DO Empno = 1, 1000
       NEXT Name
       LOG 'Employee number: ', Empno, &
            '; Name: ', Name
ENDDO


DO Empno = START, END, 10
       NEXT Name
       LOG 'Employee number: ', Empno, &
            '; Name: ', Name
ENDDO
```

## END SUBROUTINE Command

### Description:

This command terminates a subroutine. It must follow all other executable commands within the subroutine. The only statements that may follow an END SUBROUTINE command are a comment, a new SUBROUTINE command or an INCLUDE command; the included script must contain more subroutine definitions.

A subroutine is initiated by the SUBROUTINE command.

### Format:

```
END SUBROUTINE
```

### Parameters:

None

### Example:

```
END SUBROUTINE
```

## ENTRY Command

### Description:

This command, if specified, must be the first item in the Code section of the thread script, excluding format characters and comments. It identifies which variables are to receive values passed as parameters from a calling script

It is advisable that variables declared in the ENTRY command do not have an associated value list or range or file. Values passed in this way will be overwritten when script initialization takes place following the ENTRY command.

### Format:

```
ENTRY [parameter{, parameter ...}]
```

### Parameter:

**parameter**
A character variable (of up to 50 characters in length), integer variable or file ID declared in the Definitions section of the script. Up to 8 parameters may be declared in the ENTRY command. There must be the same number of parameters in this list as are passed to the script (including omitted parameters), and the data types of corresponding parameters must match.

### Example:

```
ENTRY [DATE_PARAM, TIME_PARAM, CODE_PARAM]
```

## EXIT Command

### Description:

This command causes execution of the current script to terminate immediately. No further input will be provided from the script file and no commands executed.

An optional status value can be returned when the script in question has been called from another script. This is achieved by using the **status** variable to place a value into the return status variable specified on the call to this script. If no status is specified, but the caller is expecting one, then the status returned will be that returned by the last script which exited with a status. This allows a status to be retrieved from a deeply nested script where no explicit status returning has been employed.

At run-time, a script is automatically terminated when the end of the script is reached. It is not necessary to include an EXIT command as the last command in a script, to terminate script execution.

If the script has been called, using the CALL SCRIPT command, execution of the calling script will resume at the command immediately following the CALL SCRIPT command.

When an EXIT command is processed and there are no other threads executing the script, the script data is discarded. However, if the ",KEEPALIVE" option is specified on the EXIT command, then the script data that will not be deleted even if there are no other threads executing it. This allows subsequent threads to execute the script and access any script data set up by a previous thread.

### Format:

```
EXIT {status} {,KEEPALIVE}
```

### Parameter:

**status**
An integer variable or integer value to be returned as the status from this script to the caller. The status will be returned into the integer variable specified on the CALL command.

### Examples:

```
EXIT
EXIT RETURN-STATUS
```

## GOTO Command

### Description:

This command transfers control to a specified script label. The transfer of control is immediate and unconditional.

Conditional branches may be made using the IF command.

### Format:

```
GOTO label
```

### Parameter:

**label**
A label defined within the current scope of the script.

### Examples:

```
GOTO Start
GOTO End-Of-Script
```

## IF Command

### Description:

This command performs tests on the values of variables against other variables or literals, and transfers control to a specified label depending upon the outcome of the tests.

Alternatively, structured IF commands may be used to perform one or more commands depending upon the success or failure of the tests.

By default, the matching is case sensitive. The strings “London” and “LONDON”, for example, would not produce a match, because the case of the characters is not the same. This can be overridden by specifying the “, CASE_BLIND” clause.

### Format:

```
1.    IF condition GOTO label
2.    IF condition THEN
        commands{s}
    { ELSEIF condition THEN
        command{s} }
            :
            :
    { ELSEIF condition THEN
        command{s} }
    { ELSE
        command{s} }
      ENDIF
```

**Parameters:**

**condition**
A condition of the following format:

```
{NOT}(operand1 operator operand2 {, CASE_BLIND}) &
 {AND/OR condition ...}
```

The two operands may each be a variable, a quoted character string or an integer value.

The option “CASE_BLIND” may be specified for “operand2”, to request a case-insensitive comparison of the operands.

“NOT” inverts the result of the bracketed condition that it precedes.

The binary operators are:

| | |
|---|---|
| **=** | operand1 equals operand2 |
| **<>** | operand1 does not equal operand2 |
| **<** | operand1 is less than operand2 |

| | |
|---|---|
| **<=** | operand1 is less than or equal to operand2 |
| **>** | operand1 is greater than operand2 |
| **>=** | operand1 is greater than or equal to operand2 |
| **^** | operand1 contains operand2 |
| **CONTAINS** | operand1 contains operand2 |
| **<^>** | operand1 does not contain operand2 |
| **NOT CONTAINS** | operand1 does not contain operand2 |
| **NOT_CONTAINS** | operand1 does not contain operand2 |

All conditions are evaluated from left to right.

**label**
A label defined in the current scope of the script.

**command**
Any number of script commands – including further IF or DO commands, provided that the maximum nesting level of 100 is not exceeded.

### Example:

```
IF ( NOT(isub=10) AND (NOT(isub=99)) ) THEN
      LOG "...continued"
ELSE
      LOG " Completed loop"
ENDIF
```

## ON ERROR Command

### Description:

This command allows script errors – which would normally cause the thread being executed to abort – to be captured, and script execution to be resumed at a predefined label. The ON ERROR handler is global to all sections of the script; it is propagated into all called subroutines and scripts.

The ON ERROR command captures any errors which occur either in the script within which it was declared or within any lower level scripts called by it. All script errors, such as a bad parameter error on the ~EXTRACT command, or an attempt to call a non-existent script, may be intercepted and dealt with by this command.

If a script error is encountered, then a message will be written to the audit log, identifying and locating where the error occurred. If the error has occurred in a script at a lower level than that within which the ON ERROR command was declared, then all scripts will be aborted until the required script is found.

An ON ERROR handler may be overridden by the “ON ERROR GOTO” or “ON TIMEOUT GOTO” clause for the duration of a single command. It may also be overridden by the ON ERROR command within a called script or subroutine; such a modification will affect only those scripts and subroutines at that nesting level or lower. On exit from the script or subroutine, the previously defined ON ERROR handler will be re-established.

When ON ERROR checking is established, it can be disabled by using the CANCEL command, as follows:

```
CANCEL ON ERROR
```

### Format:

```
ON ERROR GOTO label
```

### Parameter:

**label**
The name of the label within the current scope of the script, to which control branches if a script error is encountered.

### Example:

```
ON ERROR GOTO SCRIPT-ERROR
```

## RETURN Command

### Description:

This command returns control from a called subroutine to the instruction following the call to that subroutine.

### Format:

```
RETURN
```

### Parameters:

None

### Example:

```
RETURN
```

## SUBROUTINE Command

### Description:

This command defines the start of a discrete section of code which is bounded by the SUBROUTINE and END SUBROUTINE commands.

Subroutines are called from the main code with a command of the format “CALL name”. They return control to the main code by use of the RETURN command. A maximum of 255 subroutines may be defined within a script.

Subroutines share the same variable definitions as the main code but have their own labels. A label may not be referenced outside the main module or outside the subroutine in which it occurs. This has the effect of disabling branching into and out of subroutines, and also means that each subroutine may use a further 255 labels in addition to those used in the main code.

### Format:

```
SUBROUTINE name {[parameter{, parameter..}]}
```

### Parameters:

**name**
The name of the subroutine. This must be a valid *CYRANO dataname*, and must be unique within the script.

**parameter**
A character variable or integer variable declared in the Definitions section of the script. Up to 8 parameters can be declared in the SUBROUTINE command. There must be the same number of parameters in this list as there are in the subroutine call, and the data types of the parameters must match.

### Examples:

```
SUBROUTINE GET_NEXT_VALUE

SUBROUTINE CREATE_FULL_NAME [sub_char_1, sub_char_2, sub_char_3]
   SET full_name = sub_char_3 + &
            sub_char_1 + sub_char_2
   RETURN

END SUBROUTINE
```

## File Handling Commands

File handling commands help scripts and external data files exchange data.

## CLOSE Command

### Description:

This command closes an external data file. The file must have already been opened by the OPEN command.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error.

### Format:

```
CLOSE fileid {{,}ON ERROR GOTO err_label}
```

### Parameters:

**fileid**
The name associated with the file when it was opened.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Example:

```
CLOSE datafile ON ERROR GOTO Close_error
```

## OPEN Command

### Description:

This command opens an external data file (or creates a new one) and associates a *CYRANO dataname* with it, for future reference.

The data file can be opened for INPUT, OUTPUT or APPEND access. While the file is open for OUTPUT or APPEND access, it may not be opened by any other threads, unless the ",SHARED" clause is specified. This clause allows multiple threads to open an external file concurrently; they use the same 'stream', so that reads from or writes to that file by the threads are interleaved. A shared file may be open for INPUT access by one group of threads and for OUTPUT or APPEND access by another group at the same time. Note that more than one thread may open a file for INPUT access, even if the ",SHARED" clause is not specified; in this case, reads from the file are totally independent for each thread.

If the ",COMMA-SEPARATED" clause is specified, the file will be opened as a comma-separated data file. This means that comma characters within the record are treated as being significant; they are used to break the record up into its constituent fields. This clause can be used when opening SCL data files for input or output. External data files that are created from an OPEN command using the ",COMMA-SEPARATED" clause, contain records that are all ASCII text – including integer fields. Integer fields are automatically converted back into their binary representation by the READ command. Conversely, integers are converted into ASCII when written to the file using the WRITE command.

If the ",BINARY" clause is specified, then the file will be opened as a binary file rather than a text file. Binary files differ from text files in that there is no delimiting of individual records. The data is organized as a continuous stream and is written to and read from the file as such.

For a text file, a newline character will be added to the end of each record written to the file to act as a record delimiter. When reading records from a text file, data will be read up to but not including the newline character. The newline character will be skipped over to position the file at the start of the next record to be read.

The record read will be truncated as required to fill the specified variable.

Attempting to write to a data file opened for INPUT access will cause a script error. In the same way, a data file opened for OUTPUT or APPEND access cannot be read from.

If "OUTPUT" access is specified, then under normal circumstances, a new file is created. However, if the ",SHARED" clause is also specified, a new file will only be created by the thread which first executes the command. All other threads will write to this file.

A maximum of 10 external data files may be open for each thread at any one time. Attempting to open more than this number will result in a script error being reported.

The "ON ERROR GOTO err_label" clause can be specified to define a label to which control should be transferred in the event of an error. This must be the last clause in the statement.

### Format:

```
OPEN filename AS fileid {FOR access} ,Sharing {,BINARY}      {&}
        {,COMMA-SEPARATED | NO-COMMA}                          {&}
        {ON ERROR GOTO err_label}
```

### Parameters:

**filename**
A character variable or quoted character string containing the filename of the file to open.

**fileid**
A *CYRANO dataname* associated with the file when it is opened; it is used to identify the file in future references. The “fileid” must be declared in a FILE statement in the Definitions section of the script.

**access**
The type of access which is to be allowed to the file. This can be “INPUT” (the default), “OUTPUT” or “APPEND”.

**sharing**
Can be set to “SHARED” or “NOSHARED”, and defaults to “NOSHARED”.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
OPEN "\usr\james\data.dat" AS datafile &
      FOR APPEND
```

```
OPEN NEW_DATA AS datafile FOR INPUT &
      ON ERROR GOTO file-error
```

## READ Command

### Description:

This command reads a single record from an external file that is currently open for input, into a variable. If the file record is longer than the variable, the record data is truncated.

If the external data file was opened using the “COMMA-SEPARATED” clause on the OPEN command, the compiler checks that data is read as comma-separated records; it reports an error if data is not read into a record variable.

If the file was opened with the “BINARY” option specified, then a record consists of sufficient data to fill the specified variable.

If the “BINARY” option was **not** specified when the file was opened, then the record read will be delimited by a newline character in the file. This newline character is used purely as a record delimiter and does not form part of the record.

By default, the file will be rewound when an “End-of-File” status is returned by the READ command. This action may be modified by use of the “AT END GOTO label” clause.

The file is read sequentially.

### Format:

```
READ variable FROM fileid
{AT END GOTO label}  {ON ERROR GOTO err_label}
```

### Parameters:

**variable**
A character variable into which the next record from the file is read.

**fileid**
The dataname associated with the file when it was opened.

**label**
A label within the current scope of the script, to which script execution will branch if the “End-of-File” status is encountered.

**err_label**
A label within the current scope of the script, to which script execution will branch if an error occurs.

### Examples:

```
READ data_record FROM datafile
```

```
READ data FROM datafile AT END GOTO EXIT_LABEL &
      ON ERROR GOTO read_error
```

## REWIND Command

### Description:

This command causes an external data file to be rewound. The file must already have been opened by the OPEN command. If the file is open for APPEND or OUTPUT access, it will be re-initialized when the first Write operation following the REWIND command is made – providing the file has not been closed and reopened in the interim.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error.

### Format:

```
REWIND fileid {ON ERROR GOTO err_label}
```

### Parameters:

**fileid**
The name associated with the file when it was opened.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
REWIND datafile
REWIND datafile ON ERROR GOTO Error_report
```

## WRITE Command

### Description:

This command writes the specified variable, as a record, to the specified file.

The WRITE command may only be used on external data files which have been opened for APPEND or OUTPUT access. Attempting to write to an external data file which has not been opened for either of these access types will cause a script error to be reported.

If the external data file was opened using the “COMMA-SEPARATED” clause on the OPEN command, the compiler checks that data is written as comma-separated records; it reports an error if the data to be written is not a record variable.

If the file has been opened as a text file (i.e. the “BINARY” option was not specified on the OPEN command), then a newline character will be appended to the record before it is written to the file, to act as a record delimiter. For this reason, records written to text files should not themselves contain newline characters, because they will be treated as delimiters when the record is read again.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error.

### Format:

```
WRITE variable                                                    &
        TO fileid                                                 {&}
        {ON ERROR GOTO err_label}
```

### Parameters:

**variable**
The name of a character variable, or a quoted character string, which is to be written to the specified file.

**fileid**
The name associated with the file when it was opened.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Examples:

```
WRITE data_record TO datafile
WRITE ’"Date" "Time" "Order" "Description" "Qty"’ TO datafile
```

## Formal Test Control Commands

Formal test control commands provide formal support for tracking the results of each test, so that it is possible to see easily how well the testing is going.

## END TEST-CASE Command

### Description:

The END TEST-CASE command terminates a section of the script that starts with a START TEST-CASE command, to create an individual test case.

If the END TEST-CASE command is reached during execution of the script, the test case is considered to have succeeded, and the message specified in the test definition is sent to the report log.

**Note:** Test cases cannot be nested. However, there is no restriction on calling another script that contains test cases, from within a test case section.

### Format:

```
END TEST-CASE
```

### Parameters:

None

### Example:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate < minimum) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

## EXECUTE TEST Command

### Description:

This command is only valid in test scripts; it initiates execution of a named test.

Execution of the current script continues immediately at the next command – the script does not wait for the named test execution to complete. To specify that the script should wait, use the WAIT FOR TEST command. For more information on the WAIT FOR TEST command, see *WAIT FOR TEST Command*.

### Format:

```
EXECUTE TEST name {, ON NODE node}
```

### Parameters:

**name**
The test name.

**node**
A character variable, or quoted character string, containing the name of the node on which the test is to be executed. The default is to initiate execution of the test by a Test Manager on the local node.

### Examples:

```
EXECUTE TEST payroll
EXECUTE TEST pers, ON NODE "Beta1"
EXECUTE TEST pers, ON NODE test-node
```

## EXECUTE THREAD Command

### Description:

This command is only valid in test scripts; it initiates execution of a named script on a named thread.

Execution of the current script continues immediately at the next command – the script does not wait for the named thread execution to complete. To specify that the script should wait, use the WAIT FOR SCRIPT command. For more information on this command, see *WAIT FOR SCRIPT Command*

The thread must be defined within the Definitions section of the current test script.

### Format:

```
EXECUTE THREAD thread USING SCRIPT script          {&}
{, ON NODE node}                                   {&}
{, NUMBER number}                                  {&}
{, INTERVAL interval}                              {&}
{, STARTUP DELAY value}
```

### Parameters:

**thread**
The thread name.

**script**
A quoted character string, containing the name of the script to execute.

**node**
A character variable, or quoted character string, containing the name of the node on which the thread is to be executed. The default is to initiate execution of the thread by a Test Executer on the local node.

**number**
An integer variable or value defining the number of threads to attach for this thread execution. If the option is omitted, the number defaults to 1. The value must be in the range 0–65535 inclusive.

**interval**
An integer variable or value defining the number of seconds delay between the start of script execution on one thread and the next, in a set of threads. It allows script executions to be staggered. This option may only be specified if the number of threads defined by the NUMBER option is greater than 1. If the parameter is omitted, the value defaults to 0 seconds. The value must be in the range 0–32767 inclusive.

**value**
An integer variable or value defining the number of seconds that the Test Manager should

delay between attaching a thread or set of threads, and beginning to execute scripts on those thread(s). If this parameter is omitted, the value defaults to 0 seconds. The value must be in the range 0–32767 inclusive.

### Examples:

```
EXECUTE THREAD pay-load USING SCRIPT "pay0100", &
NUMBER 30, INTERVAL 10

EXECUTE THREAD pers USING SCRIPT "pers001", &
STARTUP DELAY 30

EXECUTE THREAD pers USING SCRIPT "change_dept"
```

## FAIL TEST-CASE Command

### Description:

This command indicates that the current test case has failed. The test case failure message is sent to the report log, and the test case anomaly count is incremented.

Script execution is resumed at the first instruction following the end of the test case section (i.e. the END TEST-CASE command). If a “GOTO” clause is specified, script execution is resumed at the point identified by the clause label. If a valid command immediately follows the FAIL TEST-CASE command that would not be executed because of the jump in script execution, the script compiler outputs a warning message when the script is compiled, but still produces an object file (assuming there are no errors).

**Note:** This command is only valid within a test case section of a script. It can be repeated as often as required within an individual test case.

### Format:

```
FAIL TEST-CASE {GOTO label}
```

### Parameter:

**label**
A label defined within the current scope of the script, to which control branches.

### Example:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate < minimum) THEN
        FAIL TEST-CASE
    ELSEIF (dist_rate > maximum) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

## HISTORY Command

### Description:

History logs contain a history of the executions of a test. Therefore, the programme always attempts to open an existing history log each time the test is executed.

The HISTORY command allows you to specify a message to be logged in this file. Each message will have a date, time and thread name associated with it in the history log.

A history message may consist of any number of individual values separated by commas. Any non-printable ASCII characters in character values are replaced with periods (“.”) Integer values are written as signed values, using only as many characters as necessary.

### Format:

```
HISTORY value {, value...}
```

### Parameters:

**value**
The value or variable to be written to the history log. This may be a variable or quoted character string.

### Examples:

```
HISTORY "Test Run Completed." &
         ’ Actions = ’, action_count

HISTORY "This message contains a character command " &
         "to represent the tilde character ~~"

HISTORY "This message contains a ’single quoted section’" &
         ’and "a double one here".’
```

## PASS TEST-CASE Command

### Description:

This command indicates that the current test case has succeeded. The test case success message is sent to the report log.

If no GOTO clause is specified, script execution is resumed at the first command following the end of the test case section (i.e. the END TEST-CASE command). If a GOTO clause is specified, script execution is resumed at the point identified by the clause label. If a valid command immediately follows the PASS TEST-CASE command that would not be executed because of the jump in script execution, the compiler outputs a warning message when the script is compiled, but still produces an object file (assuming there are no errors).

**Note:** This command is only valid within a test case section of a script. It can be repeated as often as required within a test case.

If the END TEST-CASE command is reached during execution of the script, the test case is automatically considered to have succeeded, and the success message is sent to the report log.

### Format:

```
PASS TEST-CASE {GOTO label}
```

### Parameter:

**label**
A label defined within the current scope of the script, to which control branches.

### Example:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate >= minimum) THEN
        PASS TEST-CASE
    ELSE
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

## REPORT Command

### Description:

Report logs contain transient information relating to the execution of a test.

The REPORT command allows the user to specify a message to be logged in this file. Each message will have a date, time and thread name associated with it in the report log.

A report message may consist of any number of individual values separated by commas.

Any non-printable ASCII characters in character values are replaced with periods (“.”). Integer values are written as signed values, and use only as many characters as are necessary.

### Format:

```
REPORT value{, value...}
```

### Parameters:

**value**
The value or variable to be written to the report log. This may be a variable or quoted character string.

### Examples:

```
REPORT "Section 1 Completed after ", loops, &
        ’ Iterations’

REPORT "This is a long message ", &
        "that is continued on this line ", &
        "and this line"

REPORT "This message contains a character command " &
        "to represent the tilde character ~~"

REPORT "This message contains a ’single quoted section’" &
        ’and "a double one here".’
```

## START TEST-CASE Command

### Description:

The START TEST-CASE command introduces a section of code that is grouped together into a test case. The section is terminated by an END TEST-CASE command.

The START TEST-CASE command must include a description of the test case. The test case description and test case status are written to the report log when the test case is executed.

**Note:** Test cases cannot be nested, so a test case must be terminated with an END TEST-CASE command before a new test case section can be started. However, there is no restriction on calling another script that contains test cases, from within a test case section.

### Format:

```
START TEST-CASE description
```

### Parameter:

**description**
A character variable or quoted literal string containing text that describes the test case.

### Examples:

```
START TEST-CASE "Checking for appearance of UNITS field"
    IF (no_units = 0) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

```
SET tc_desc_str = "Checking for appearance of UNITS field"
START TEST-CASE tc_desc_str
    IF (no_units = 0) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

## Synchronization Commands

These commands address events that scripts may have to wait for before continuing their execution.

## ACQUIRE MUTEX Command

### Description:

This command acquires exclusive access to a shared resource, known as a *mutex*. The mutex is identified by its name and scope (which must be either “LOCAL” or “TEST-WIDE”). A test-wide mutex is one that is shared by all scripts running as part of a distributed test; a local mutex is only shared between scripts running on the local node.

By default, if an attempt is made to acquire a mutex that has already been acquired by another script (within the same scope), then the thread will be suspended until the mutex is released. However, if a time-out period is specified, this represents the maximum number of seconds that OpenSTA will wait for the mutex to be released before timing out the request. A period of zero indicates that the request should be timed out immediately if the mutex has been acquired by another script.

The “ON TIMEOUT GOTO tmo_label” clause can be specified to define a label to which control should be transferred if the request times out. In addition, the “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error, or if the request times out and there was no “ON TIMEOUT GOTO tmo_label” clause.

### Format:

```
ACQUIRE {scope} MUTEX mutex_name                                    {&}
        {,WITH TIMEOUT period {,ON TIMEOUT GOTO tmo_label}}         {&}
        {,ON ERROR GOTO err_label}
```

### Parameters:

**scope**
The scope of the mutex to be acquired. This must be either “LOCAL” or “TEST-WIDE”, and defaults to “LOCAL”.

**mutex-name**
A character variable, or quoted character string, containing the name of the mutex which is to be acquired. “mutex-name” must be a valid *CYRANO dataname*.

**period**
An integer variable or value, defining the number of seconds to wait before an unsatisfied request is timed out. The valid range is 0–2147483647.

**tmo_label**
A label defined within the current scope of the script, to which control branches if a time-out occurs.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs, or the command times out and “tmo_label” is not specified.

**Example:**

```
ACQUIRE LOCAL MUTEX "MUMPS-SERVER", ON ERROR GOTO mumps-error
```

## CLEAR SEMAPHORE Command

### Description:

This command resets a named semaphore to its “Clear” state. The semaphore is identified by its name and scope (which must be either “LOCAL” or “TEST-WIDE”). A test-wide semaphore is one that is shared by all scripts running as part of a distributed test; a local semaphore is only shared between scripts running on the local node.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error.

### Format:

```
CLEAR {scope} SEMAPHORE semaphore-name                    {&}
        {,ON ERROR GOTO err_label}
```

### Parameters:

**scope**
The scope of the semaphore to clear. This must be either “LOCAL” or “TEST-WIDE”, and defaults to “LOCAL”.

**semaphore-name**
A character variable, or quoted character string, containing the name of the semaphore to clear.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Example:

```
CLEAR LOCAL SEMAPHORE "SERVER-RUNNING"
```

## RELEASE MUTEX Command

### Description:

This command releases a named mutex. The mutex to be released is identified by its name and scope, which must correspond to the values specified on the corresponding ACQUIRE MUTEX command.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error. Note that an error always occurs if the script that issues the RELEASE MUTEX request has not previously acquired it.

### Format:

```
RELEASE {scope} MUTEX mutex_name                                {&}
           {,ON ERROR GOTO err_label}
```

### Parameters:

**scope**
The scope of the mutex to release. This must be either “LOCAL” or “TEST-WIDE”, and defaults to “LOCAL”.

**mutex-name**
A character variable, or quoted character string, containing the name of the mutex to release.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Example:

```
RELEASE LOCAL MUTEX "MUMPS-SERVER"
```

## SET SEMAPHORE Command

### Description:

This command sets a named semaphore to its “Set” state. The semaphore is identified by name and scope (which must be either “LOCAL” or “TEST-WIDE”). A test-wide semaphore is one that is shared by all scripts running as part of a distributed test; a local semaphore is only shared between scripts running on the local node.

The “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error.

### Format:

```
SET {scope} SEMAPHORE semaphore-name                          {&}
        {,ON ERROR GOTO err_label}
```

### Parameters:

**scope**
The scope of the semaphore to be set. This must be either “LOCAL” or “TEST-WIDE”, and defaults to “LOCAL”.

**semaphore-name**
A character variable, or quoted character string, containing the name of the semaphore to be set.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs.

### Example:

```
SET LOCAL SEMAPHORE "SERVER-RUNNING"
```

## SYNCHRONIZE REQUESTS Command

### Description:

HTTP requests are issued asynchronously. Immediately after an HTTP request has been issued, the next command in the script is processed. OpenSTA does not wait for a response to be received for an HTTP request.

This command causes the thread currently executing to be suspended immediately, until responses have been received for all the requests that have been issued by the thread. It is only valid within a script that has been defined as MODE HTTP.

### Format:

```
[SYNCHRONIZE | SYNCHRONISE]  REQUESTS
```

### Parameters:

None.

### Examples:

```
SYNCHRONIZE REQUESTS
SYNCHRONISE REQUESTS
```

## WAIT Command

### Description:

This command suspends the script execution for the specified number of seconds. The unit is either seconds or milliseconds depending upon the value of the Environment statement WAIT UNIT p23.

### Format:

```
WAIT period
```

### Parameter:

**period**
An integer variable or value defining the number of seconds for which script execution is to be suspended. The valid range is 0–2147483647.

### Examples:

```
WAIT 5
WAIT Wait-Period
```

## WAIT FOR SCRIPT Command

### Description:

This command is applicable to test scripts only; it halts the script until the specified script has completed. The script must have been initiated by an EXECUTE THREAD command on the same Test Manager.

If a period is specified, this represents the maximum number of seconds that the script will wait before timing out the current script. When a test script times out, a message is written to the audit log, the script execution is either aborted or the specified action carried out when error trapping is enabled via the ON ERROR command.

Alternatively, the "ON TIMEOUT GOTO tmo_label" clause can be specified, to define a label to which control should be transferred in the event of a time-out.

If no period is specified, the script will only resume when the specified thread has completed.

### Format:

```
WAIT {period} FOR SCRIPT script_name {AND script_name}{&}
{, ON TIMEOUT GOTO tmo_label}
```

### Parameters:

**period**
An integer variable or value defining the number of seconds to wait. The valid range is 0–32767.

**script_name**
A character variable, or quoted character string, containing the name of the script that must complete before script processing is resumed. You may specify more than one script name, separated by the "AND" operator; to aid legibility, you may enclose the script names in parentheses.

**tmo_label**
A label defined within the current scope of the script, to which control branches if a time-out occurs.

**Examples:**

```
WAIT FOR SCRIPT “payroll“
WAIT FOR SCRIPT (“pers“ AND “payroll“)
WAIT FOR SCRIPT ’payroll’, ON TIMEOUT GOTO Timed-out
```

## WAIT FOR SEMAPHORE Command

### Description:

This command halts the script until the specified semaphore is in its “Set” state. The semaphore is identified by its name and scope (which must be either “LOCAL” or “TEST-WIDE”). A test-wide semaphore is one that is shared by all scripts running as part of a distributed test; a local semaphore is only shared between scripts running on the local node.

By default, if the semaphore is in its “Clear” state when the WAIT FOR SEMAPHORE command is issued, the thread will be suspended until it is set into its “Set” state. However, if a time-out period is specified, this represents the maximum number of seconds that OpenSTA will wait for the semaphore to be set before timing out the request. A period of zero indicates that the request should be timed out immediately if the semaphore is not set.

The “ON TIMEOUT GOTO tmo_label” clause can be specified to define a label to which control should be transferred if the request times out. In addition, the “ON ERROR GOTO err_label” clause can be specified to define a label to which control should be transferred in the event of an error, or if the request times out and there was no “ON TIMEOUT GOTO tmo_label” clause.

### Format:

```
WAIT {period} FOR {scope} SEMAPHORE semaphore-name              {&}
        {,ON TIMEOUT GOTO tmo_label}                            {&}
        {,ON ERROR GOTO err_label}
```

### Parameters:

**period**
An integer variable or value defining the number of seconds to wait. The valid range is 0–2147483647.

**scope**
The scope of the semaphore to wait for. This must be either “LOCAL” or “TEST-WIDE”, and defaults to “LOCAL”.

**semaphore-name**
A character variable, or quoted character string, containing the name of the semaphore to wait for.

**tmo_label**
A label defined within the current scope of the script, to which control branches if a time-out occurs.

**err_label**
A label defined within the current scope of the script, to which control branches if an error occurs, or the command times out and “tmo_label” is not specified.

**Example:**

```
WAIT 10 FOR SEMAPHORE "SERVER-RUNNING"
```

## WAIT FOR TEST Command

### Description:

This command is applicable to test scripts only; it halts the script until the specified test has completed. The test must have been initiated by an EXECUTE TEST command on the same Test Manager. If more than one test with the specified name is running, the script is halted until **all** the tests have completed.

If a period is specified, this represents the maximum number of seconds that the script will wait before timing out the current script. When a test script times out, a message is written to the audit log, the script execution is either aborted or the specified action carried out when error trapping is enabled via the ON ERROR command.

Alternatively, the "ON TIMEOUT GOTO tmo_label" clause can be specified, to define a label to which control should be transferred in the event of a time-out.

If no period is specified, the script will only resume when the test completes.

### Format:

```
WAIT {period} FOR TEST test_name {AND test_name}{&}
{, ON TIMEOUT GOTO tmo_label}
```

### Parameters:

**period**
An integer variable or value defining the number of seconds to wait. The valid range is 0–32767.

**test_name**
The name of the test that must complete before script processing is resumed. You may specify more than one test name, separated by the "AND" operator; to aid legibility, you may enclose the test names in parentheses.

**tmo_label**
A label defined within the current scope of the script, to which control branches if a time-out occurs.

### Examples:

```
WAIT FOR TEST payroll
WAIT FOR TEST (pers AND payroll)
WAIT FOR TEST payroll, ON TIMEOUT GOTO Timed-out
```

## Statistical Data Logging Commands

Diagnostic commands help you to analyze scripts in order to diagnose an anomaly.

## END TIMER Command

### Description:

This command switches off the named stop-watch timer and writes an ‘end timer’ record to the statistics log, even if the timer is already switched off.

A stop-watch timer is switched on by the START TIMER command.

### Format:

```
END TIMER name
```

### Parameter:

**name**
The timer name. The timer must be declared in a TIMER statement in the Definitions section of the script.

### Example:

```
END TIMER Transaction
```

## START TIMER Command

### Description:

This command switches on the named stop-watch timer and writes a ‘start timer’ record to the statistics log.

There is no limit to the number of stop-watch timers that can be switched on at the same time. However, if a timer is switched on twice without being stopped in the interim, the first timer is effectively cancelled and thrown away when it is re-started.

A stop-watch timer is switched off by the END TIMER command.

### Format:

```
START TIMER name
```

### Parameter:

**name**
The timer name. The timer must be declared in a TIMER statement in the Definitions section of the script.

### Example:

```
START TIMER Transaction
```

## Diagnostic Commands

During test development, there is occasionally a need to find out more about what a script is doing in order to diagnose an anomaly. The diagnostic commands assist in this process.

## LOG Command

### Description:

OpenSTA maintains an audit trail of its activity and related events. The LOG command allows the user to specify a message to be written to the audit log. Each message in this file will have a date, time and thread name associated with it.

A log message may consist of any number of individual values separated by commas.

Any non-printable ASCII characters in character values are replaced with periods (“.”). Integer values are written as signed values, using only as many characters as are necessary.

### Format:

```
LOG value{, value...}
```

### Parameters:

**value**
The value or variable to be logged. This may be a variable or quoted character string.

### Examples:

```
LOG "Customer Name = ", Cust-Name, &
     ’ Customer Code = ’, Cust-Code

LOG "This is a long message " &
     "that is continued on this line " &
     "and this line"

LOG "This message contains a ’single quoted section’" &
     ’and "a double one here".’
```

## NOTE Command

### Description:

This command associates a list of variables or quoted character strings with the current thread. The current value(s) can be viewed on the Thread Summary screen in CYRANO TestCommander.

### Format:

```
NOTE value{,char_value,...}
```

### Parameters:

**value**
The value or variable to be logged. This may be a variable or quoted character string.

### Examples:

```
NOTE Emp-Name
NOTE "Searching for 'End Of File' failures"
```

## TRACE Command

### Description:

This command writes user-definable messages to the script tracing log.

### Format:

```
TRACE value{,value...}
```

**Parameters:**

**value**
The value or variable to be written to the trace log. This may be a variable or quoted character string.

### Examples:

```
TRACE ’Trace point following "overflow" condition’
TRACE "Trace point ", trcpos
```

## Miscellaneous Commands

Miscellaneous commands provide other functionality that has been found to be useful when creating scripts.

## CONNECT Command

### Description:

This command may be used to establish a TCP connection to a nominated host. It is only valid within a script that has been defined as MODE HTTP.

This command specifies an ID for the TCP connection. This may be used in subsequent GET, HEAD, POST and LOAD RESPONSE_INFO commands to use this TCP connection. The TCP connection may be closed using the DISCONNECT command. It will also be terminated when the thread exits the script.

The connection ID specified must not correspond to a TCP connection already established previously using the CONNECT command. Otherwise a script error will be reported.

### Format:

```
CONNECT TO host ON conid
```

### Parameters:

**host**

A character variable, quoted character string or character expression, containing the host name or IP address of the resource to connect to and, optionally, the port number on which the connection is to be made. If a port is specified, it must be separated from the host field by a colon (":"). If the port number field is empty or not specified, the port defaults to TCP 80.

**conid**

An integer variable, integer value or integer expression defining the connection ID. This is used in all subsequent operations on this connection.

### Examples:

```
CONNECT TO "proxy.dev.mynet:3128" ON 1
CONNECT TO myhost ON 2
CONNECT TO 'abc.com' ON conid
```

## DISCONNECT Command

### Description:

This command closes one or all of the TCP connections established using the CONNECT command. It is only valid within a script that has been defined as MODE HTTP.

If the "FROM conid" clause is specified, the TCP connection identified by that Connection ID will be closed. If the "ALL" keyword is used, all TCP connections established by the current thread will be closed.

By default, the DISCONNECT command will wait until any requests on the connection(s) to be closed are complete before closing them. If the WITH CANCEL clause is specified, the connection(s) will be closed immediately.

The Connection ID specified must correspond to a TCP connection established using the CONNECT command, otherwise a script error will be reported.

### Format:

```
DISCONNECT [FROM conid | ALL ]  {,WITH CANCEL}
```

### Parameters:

**conid**

An integer variable, integer value or integer expression identifying the Connection ID of the TCP connection to be closed.

### Examples:

```
DISCONNECT FROM 1
DISCONNECT FROM conid
DISCONNECT FROM 1, WITH CANCEL
DISCONNECT ALL
DISCONNECT ALL, WITH CANCEL
```

## LOAD ACTIVE_THREADS Command

### Description:

This command allows the number of threads which are currently active on the current Test Manager to be loaded into an integer variable for later use.

The count of active threads includes all threads which are executing either their primary script or a secondary script. It does **not** include threads which are processing a start-up delay or which are currently suspended.

### Format:

```
LOAD ACTIVE_THREADS INTO variable
```

### Parameter:

**variable**
An integer variable into which the count of active threads is loaded.

### Example:

```
LOAD ACTIVE_THREADS INTO active-count
```

## LOAD DATE Command

### Description:

This command loads an integer variable with the number of days since the system base date, or a character variable with the system date.

For character variables, the system date will be loaded in the system default format (for example, “DD–MMM–CCYY”); the date will be truncated as required to fit into the target variable.

### Format:

```
LOAD DATE INTO variable
```

### Parameter:

**variable**
The name of a character or integer variable into which the date is loaded.

### Examples:

```
LOAD DATE INTO INT-DATE
LOAD DATE INTO CHAR-DATE
```

## LOAD NODENAME Command

### Description:

This command loads the current node name into a variable.

### Format:

```
LOAD NODENAME INTO variable
```

### Parameter:

**variable**
A character variable into which the node name is loaded. The node name will be truncated as required, to fit into the target variable.

### Example:

```
LOAD NODENAME INTO Node-name
```

## LOAD SCRIPT Command

### Description:

This command loads the name of the script being executed, into a character variable.

### Format:

```
LOAD SCRIPT INTO variable
```

### Parameter:

**variable**
A character variable into which the script name is loaded. The script name will be truncated as required, to fill the target variable.

### Example:

```
LOAD SCRIPT INTO Scriptname
```

## LOAD TEST Command

### Description:

This command loads the name of the test of which the script is a part, into a variable. The test name will be truncated as required to fit into the target variable. The maximum size of the string returned by this command is 16 characters.

### Format:

```
LOAD TEST INTO variable
```

### Parameter:

**variable**
A character variable into which the name of the test is loaded.

### Example:

```
LOAD TEST INTO Testname
```

## LOAD THREAD Command

### Description:

This command loads the name of the thread on which the script is currently executing, into a character variable.

The variable should be 16 bytes long, since thread names may be up to 16 bytes long. The thread name will be truncated as required to fill the target variable.

### Format:

```
LOAD THREAD INTO variable
```

### Parameter:

**variable**
A character variable into which the thread name is loaded.

### Example:

```
LOAD THREAD INTO Thread-Name
```

## LOAD TIME Command

### Description:

This command loads a variable with either the number of 10ms 'ticks' since midnight (if the variable is an integer variable), or the system time (if the variable is a character variable).

For character variables, the system time will be loaded in the system default format, truncated if the variable is not long enough to hold it.

### Format:

```
LOAD TIME INTO variable
```

### Parameter:

**variable**
The name of a character or integer variable into which the time is loaded.

### Examples:

```
LOAD TIME INTO Int-time
LOAD TIME INTO Char-time
```

## LOAD TIMER Command

### Description:

This command loads an integer variable with the current value – as a number of 10ms ticks – of the specified timer. The current value of a timer is calculated by taking the time for the latest **stop timer** and subtracting from it the time for the preceding **start timer**. If no start timer / stop timer commands have been executed for the specified timer by the current thread an error will occur. This will either abort script execution, or take the specified action if error trapping is enabled via the ON ERROR command.

### Format:

```
LOAD TIMER name INTO variable
```

### Parameters:

**name**
The timer name. The timer must be declared in a TIMER statement in the Definitions section of the script.

**variable**
The name of an integer variable into which the timer value – in 10ms ticks – is loaded.

### Example:

```
LOAD TIMER Transaction INTO Timval
```