VU Language Reference

Version 2000.02.10

Rational
the e-development company™

VU Language Reference

Revised 04/2000

This manual prepared by:
Rational Software Corporation
20 Maguire Road
Lexington, MA 02421
U.S.A.

Phone:
800-433-5444
408-863-4000

E-mail: support@rational.com
Web: http://www.rational.com

P/N 800-023370-000

Contents

5 User Emulation

Preface

This manual describes the statements and conventions of the VU scripting language. VU includes most of the syntax rules and core statements found in the C language.

This manual is intended to help application developers and system testers read and customize virtual user scripts generated with Rational Robot. Familiarity with Robot and other Rational Suite software is assumed. Familiarity with programming language practices is also assumed.

Other Resources

- This product contains complete online Help. For information, see the following section.
- All manuals for this product are available online in PDF format. The manuals are on the *Rational Solutions for Windows* Online Documentation CD.
- For information about training opportunities, see the Rational University Web site: http://www.rational.com/university.

Using the VU Help

You can access the VU Help in a variety of ways:

- From the **Start** menu, click **VU Language Reference** in the installation directory of your Rational product (typically, Rational Test).
- From within Robot, click **Help > VU Language Reference**.
- While you are editing a script in Robot, you can display context-sensitive information about a particular VU command. To do so:

1. Place the insertion point immediately before, after, or anywhere within the command name.
2. Press F1.

   If a single Help topic is associated with the command name, reference information about that command appears immediately.

   If multiple Help topics are associated with the command, the topics are listed in the Topics Found dialog box. Select the topic you want and click **Display**.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Contacting Rational Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical Support as follows:

Rational Technical Support

Location	Contact Information	Notes
North America	Telephone: 800-433-5444 408-863-4000 E-mail: support@rational.com	Please be prepared to supply the following information: – Your name, telephone number, and company name – Computer make and model – Operating system and version number – Product release number and serial number – Your Case ID number (if you are calling about a previously reported problem)
Europe	Telephone: +31 (0) 20 4546 200 E-mail: support@europe.rational.com	
Asia Pacific	Telephone: +61-2-9419-0111 E-mail: support@apac.rational.com	
World Wide Web	http://www.rational.com	Click the Technical Support link.

Part I

Introducing VU

▸▸▸ CHAPTER 1

What Is VU?

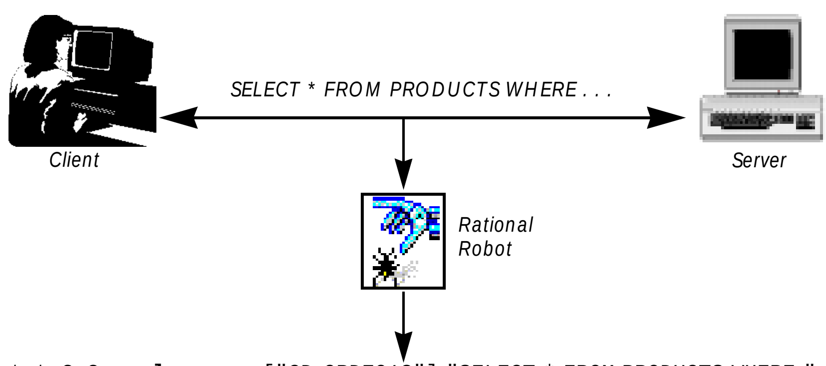
The VU language is the Rational Software corporation language for building virtual user scripts.

The VU language is based on the C programming language. In addition to supporting many C language features, VU includes commands and environment variables specifically designed for use in Rational Performance Studio scripts.

Automated Script Generation

When you record client/server conversations, Rational Robot automatically generates a script for you in the VU language. You can either play back the script as it was generated, or you can make modifications in Robot.

During virtual user recording, Robot “listens in” on the client/server conversation. Robot translates the raw conversation into a series of VU commands and stores them in the script.




```
stmt_2_2 = sqlprepare ["CD ORDE013"] "SELECT * FROM PRODUCTS WHERE "...
```

Working with Scripts

Although Robot generates complete, executable scripts, sometimes you may want to edit a recorded script — for example, to:

- Add `for`, `while`, and `do-while` loops to simplify repetitive actions.
- Add conditional branching.
- Modify think time variables.
- Respond to runtime errors.

Your Work Environment

With VU as your scripting language, you view, edit, and compile scripts in Robot.

You play back virtual user scripts through a Rational LoadTest schedule. However, if you are in Robot and want to play back a script, click **File** → **Playback**. Robot automatically creates a schedule for you and invokes LoadTest so you can play back the schedule for that script.

Source and Runtime Files

The VU language supports the following kinds of files:

File type	Extension	Location
Script files	.s	The Script directory of your repository and project.
Watch files (also called session files)	.wch	The Session directory of your repository and project.
Header files	.h	The VU.h file shipped with LoadTest is located in \Rational\Rational Test 7\Include by default.

VU Additions to the C Language

The VU language contains a number of commands in addition to standard C programming language commands. The following categories of commands are provided to help you test your applications and analyze the results:

Environment control commands – Enable you to control a virtual user's environment by changing the VU environment variables. For example, you can set the level of detail logged or the number of times to try a connection.

Flow control statements – Enable you to add conditional execution structures and looping structures to your virtual user script. The flow control statements behave like their C counterparts, with enhancements added to `break` and `continue`.

Library routines – Provide your virtual user script with predefined functions that handle file I/O, string manipulation, and conversion of data types and formats.

Send and receive emulation commands – Emulate client activity and evaluate the server's responses, as well as performing communication and timing operations. You can log emulation commands in a log file.

Emulation functions – Like emulation commands, emulation functions emulate client activity and evaluate the server's responses. However, emulation functions do not perform communication and timing operations, and they are not logged in a log file.

Datapool functions – Retrieve data from a datapool. A **datapool** is a source of data that you can use to access variable data from a script. This enables a script that is executed more than once to use different values for each execution. You create the datapool with Robot or TestManager.

VU toolkit functions – These functions, which come with PerformanceStudio, enable you to parse data returned by `sqlnrecv` into rows and columns.

SQABasic Scripting Language

Because the VU scripting language lets you capture client/server conversations, it is the language to use for testing how your client/server system performs.

But for testing GUI objects, you need to record a user's keyboard and mouse actions. You also need to insert verification points into the script to compare the way GUI objects look and work across successive builds of the application. The SQABasic scripting language is required for testing GUI objects.

For more information about the SQABasic scripting language, see the *SQABasic Language Reference*.

▸▸▸ CHAPTER 2

Functional List

This chapter organizes the VU commands into functional categories. For information on the VU commands pertaining to Jolt and SAP, see Appendixes A and B.

HTTP Emulation Commands and Functions

HTTP Send Emulation Commands

`http_request`	Sends an HTTP request to a Web server.

HTTP Receive Emulation Commands

`http_header_recv`	Receives header metadata from a Web server.
`http_nrecv`	Receives a user-specified number of bytes from a Web server.
`http_recv`	Receives data from a Web server until the specified text string occurs.

HTTP Emulation Functions

`http_disconnect`	Closes the connection to a Web server.
http_find_values	Searches for the specified values on the current connection.
`http_header_info`	Gets individual header values from header metadata.
`http_url_encode`	Prepares strings for inclusion in an HTTP request.
`expire_cookie`	Expires a cookie in the cookie cache.
`set_cookie`	Adds a cookie to the cookie cache.

SQL Emulation Commands and Functions

SQL Send Emulation Commands

`sqlclose_cursor`	Closes the indicated cursor.
`sqldeclare_cursor`	Associates a SQL statement with a cursor ID, which is required to open the cursor.
`sqldelete_cursor`	Deletes the current row using the indicated cursor.
`sqlexec`	Executes SQL statements.
`sqlopen_cursor`	Opens the specified cursor.
`sqlposition_cursor`	Positions a cursor within a result set.
`sqlprepare`	Prepares a SQL statement for execution.
`sqlrefresh_cursor`	Refreshes the result set of a cursor.
`sqlupdate_cursor`	Updates the current row of the indicated cursor.
`sqlsysteminfo`	Queries the server for system information.

SQL Receive Emulation Commands

`sqlfetch_cursor`	Fetches the requested rows from the cursor indicated.
`sqllongrecv`	Retrieves longbinary and longchar results.
`sqlnrecv`	Retrieves row results after `sqlexec` is executed.

SQL Emulation Functions

`sqlalloc_cursor`	Allocates a cursor for use in cursor-oriented SQL emulation commands and functions.
`sqlalloc_statement`	Allocates a cursor data area for Oracle playback.
`sqlcommit`	Commits the current transaction.
`sqlconnect`	Logs on to a SQL database server.
`sqlcursor_rowtag`	Returns the tag of the last row fetched.
`sqlcursor_setoption`	Sets a SQL cursor option.
`sqldisconnect`	Closes the specified connection.
`sqlfree_cursor`	Frees a cursor.

`sqlfree_statement`	Frees all of the client and server resources for a prepared statement.
`sqlinsert_cursor`	Inserts rows via a cursor.
`sqlrollback`	Rolls back the current transaction.
`sqlsetoption`	Sets a SQL database server option.

NOTE: See *VU Toolkit Functions: Data* for additional SQL emulation functions.

VU Toolkit Functions

VU Toolkit Functions: Data

`AppendData`	Adds the data returned by `sqlnrecv` to the specified data set.
`FreeAllData`	Frees all data sets saved with `SaveData` and `AppendData`.
`FreeData`	Frees specified data sets saved with `SaveData` and `AppendData`.
`GetData`	Retrieves a specific row from the data set created with `SaveData` or `AppendData`.
`GetData1`	Retrieves a value in the first row of a data set created with `SaveData` or `AppendData`.
`SaveData`	Stores the data returned by the most recent `sqlnrecv` command into a data set.

VU Toolkit Functions: File I/O

`IndexedField`	Parses the line read by the `ReadLine` function and returns the field designated by *`index`*.
`IndexedSubField`	Parses the field set by the `NextField` or `IndexedField` function and returns the subfield designated by *`index`*.
`NextField`	Parses the line read by the `ReadLine` function.
`NextSubField`	Parses the field returned by the most recent call to `NextField` or `IndexedField`.
`ReadLine`	Reads a line from the open file designated by *`file_descriptor`*.
`SHARED_READ`	Allows multiple users to share a file.

TUXEDO Emulation Commands and Functions

TUXEDO Send Emulation Commands

`tux_bq`	Queues a UNIX command for background processing.
`tux_tpabort`	Aborts the current transaction.
`tux_tpacall`	Sends a service request.
`tux_tpbroadcast`	Broadcasts notification by name.
`tux_tpcall`	Sends a service request and awaits its reply.
`tux_tpcommit`	Commits the current transaction.
`tux_tpconnect`	Establishes a conversational service connection.
`tux_tpdequeue`	Removes a message from a queue.
`tux_tpdiscon`	Takes down a conversational service connection.
`tux_tpenqueue`	Queues a message.
`tux_tpgetrply`	Gets a reply from a previous request.
`tux_tpinit`	Joins an application.
`tux_tpnotify`	Sends notification by client identifier.
`tux_tppost`	Posts an event.
`tux_tprecv`	Receives a message in a conversational service connection.
`tux_tpresume`	Resumes a global transaction.
`tux_tpsend`	Sends a message in a conversational service connection.
`tux_tpsubscribe`	Subscribes to an event.
`tux_tpsuspend`	Suspends a global transaction.
`tux_tpterm`	Leaves an application.
`tux_tpunsubscribe`	Unsubscribes to an event.

TUXEDO Receive Emulation Commands

None.

TUXEDO Emulation Functions

`tux_allocbuf`	Allocates a free buffer.
`tux_allocbuf_typed`	Allocates a TUXEDO-typed buffer.
`tux_freebuf`	Deallocates a free buffer.
`tux_getbuf_ascii`	Gets a free buffer or buffer member and converts it into a string.
`tux_getbuf_int`	Gets a free buffer or buffer member and converts it into an VU integer.
`tux_getbuf_string`	Gets a free buffer or buffer member and converts it into a string without converting nonprintable characters.
`tux_reallocbuf`	Resizes a free buffer.
`tux_setbuf_ascii`	Writes a string value into a buffer or buffer member.
`tux_setbuf_int`	Sets a free buffer or buffer member with an VU integer value.
`tux_setbuf_string`	Sets a free buffer or buffer member with an VU string value, without converting nonprintable characters.
`tux_sizeofbuf`	Returns the size of a buffer.
`tux_tpalloc`	Allocates TUXEDO-typed buffers.
`tux_tpbegin`	Begins a transaction.
`tux_tpcancel`	Cancels a call descriptor for an outstanding reply.
`tux_tpchkauth`	Checks whether authentication is required to join an application.
`tux_tpfree`	Frees a typed buffer.
`tux_tprealloc`	Changes the size of a typed buffer.
`tux_tpscmt`	Sets when `tpcommit`() should return.
`tux_tpsprio`	Sets the service request priority.
`tux_tptypes`	Provides information about a typed buffer.
`tux_typeofbuf`	Returns the type of a buffer.
`tux_userlog`	Writes a message to the TUXEDO central event log.

IIOP Emulation Commands and Functions

IIOP Send Emulation Commands

`iiop_bind`	Binds an interface name to an Object Reference pseudo-object.
`iiop_invoke`	Initiates a synchronous IIOP request to an interface implementation.

IIOP Emulation Functions

`iiop_release`	Releases storage associated with a pseudo-object.

Socket Emulation Commands and Functions

Socket Send Emulation Commands

`sock_send`	Sends data to the server.

Socket Receive Emulation Commands

`sock_nrecv`	Receives *n* bytes from the server.
`sock_recv`	Receives data until the specified delimiter string is found.

Socket Emulation Functions

`sock_connect`	Opens a socket connection.
`sock_create`	Creates a socket to which another process may connect.
`sock_disconnect`	Disconnects a socket connection.
`sock_fdopen`	Associates a file descriptor with a socket connection.
`sock_isinput`	Checks for available input on a socket connection.
`sock_open`	Waits for a socket connection from another process.

Emulation Commands That Can Be Used with Any Protocol

Send Emulation Commands

`emulate`	Provides generic emulation command services to support a proprietary protocol.

Other Emulation Commands

`start_time`	Marks the start of a block of actions to be timed.
`stop_time`	Marks the end of a block of actions being timed.
`testcase`	Checks a response for specific results, and reports and logs them.

Flow Control Commands

`break`	Stops execution of `for`, `while`, and `do-while` statements.
`continue`	Skips remaining statements in a loop and continues with the next iteration of the loop.
`do-while`	Repeatedly executes a VU statement while a condition is true.
`else-if`	Conditionally executes a VU statement.
`for`	Repeatedly executes a VU statement.
`if-else`	Conditionally executes a VU statement.
`script_exit`	Exits from a script.
`user_exit`	Exits an entire virtual user emulation from within any point in a virtual user script.
`while`	Repeatedly executes a VU statement.

I/O Routines

`close`	Writes out buffered data to a file and then closes the file.
`feof`	Returns a value indicating whether or not the end of a file has been encountered.
`fflush`	Causes any buffered data for a file to be written to that file.
`fgetc`	Provides unformatted character input capability.
`printf, fprintf, sprintf`	Writes specified output to a file, standard output, or a string variable.
`fputc, fputs`	Write unformatted output for characters or strings.
`fseek`	Repositions the file pointer.
`ftell`	Returns the file pointer's offset in the specified file.
`open`	Opens a file for reading or writing.
`scanf, fscanf, sscanf`	Reads specified input from standard input, a file, or a string expression.
`tempnam`	Generates unique temporary file names.
ungetc	Provides unformatted character input capability.
`unlink`	Removes files.

Conversion Routines

`atoi`	Converts strings to integers.
`base64_decode`	Decodes a base 64–encoded string.
`base64_encode`	Encodes a string using base-64 encoding.
`ctos`	Converts characters to strings.
`hex2mixedstring`	Returns a mixed ascii/hex version of a VU string.
`itoa`	Converts integers to strings.
`mixed2hexstring`	Returns a pure hex version of a VU string.
`stoc`	Returns a selected character from a string argument.

String Routines

`cindex`	Returns the position within *`str`* of the first occurrence of the character *`char`*.
`lcindex`	Returns the position of the last occurrence of a user-supplied character.
`match`	Determines whether a subject string matches a specified pattern.
`mkprintable`	Creates printable versions of strings that contain nonprintable characters.
`sindex`	Returns the position of the first occurrence of any character from a specified set.
`sqtrans`	Creates string expressions based on character translations of string expressions, squeezing out any repeated characters.
`strlen`	Returns the length of a string expression.
`strneg`	Creates a string expression based on character set negation (complements).
`strrep`	Creates a string expression based on character repetition.
`strset`	Creates a string expression based on user-supplied characters.
`strstr`	Searches for one string within another.
`strspan`	Returns the length of the initial segment within a string expression, beginning at the specified position.
`subfield`	Extracts substrings from string expressions based on field position.
`substr`	Extracts substrings from string expressions based on character position.
`trans`	Substitutes or deletes selected characters in a string expression.

Random Number Routines

`negexp`	Returns a random integer from a negative exponential distribution with the specified mean.
`rand`	Returns a random integer in the range 0 to 32767.
`srand`	Reseeds the random number generator, essentially resetting it to a specific starting place.
`uniform`	Returns a random integer uniformly distributed in the specified range.

Timing Routines

`delay`	Delays script execution for a specified time period.
`time`	Returns the current time in integer format.
`tod`	Returns the current time in string format.

Miscellaneous Routines

`abs`	Returns the absolute value of its argument as an integer.
`bank`	Creates bank expressions for assignments to the bank environment variables `Escape_seq` and `Logout_seq`.
display	Provides a string to the monitor for display in message view.
`getenv`	Obtains the values of Windows NT or UNIX environment variables from within a virtual user script.
`log_msg`	Writes messages to the log file with a standard header format.
`putenv`	Sets the values of Windows NT or UNIX environment variables from within a virtual user script.
`system`	Allows an escape mechanism to the UNIX shell from within a virtual user script running on a UNIX system.
`usergroup_member`	Returns the position of a virtual user within a user group
`usergroup_size`	Returns the number of members in a user group.

Synchronization Functions

`wait`	Blocks a virtual user from further execution until a user-defined global event occurs.
`sync_point`	Waits for users in a LoadTest schedule to synchronize.

Datapool Functions

`datapool_close`	Closes an open datapool.
`datapool_fetch`	Moves the datapool cursor to the next record.
`datapool_open`	Opens a datapool.
`datapool_rewind`	Resets the cursor for the datapool.
`datapool_value`	Retrieves the value of a specified column.

Environment Control Commands

`eval`	Returns the value and data type at the top of a VU environment variable's stack.
`pop`	Removes the value of a VU environment variable from the top of the stack.
`push`	Pushes the value of a VU environment variable to the top of the stack.
`reset`	Changes the current value of a VU environment variable to its default value, and discards all other values in the stack.
`restore`	Makes the saved value of a VU environment variable the current value.
`save`	Saves the value of a VU environment variable.
`set`	Sets a VU environment variable to the specified expression.
`show`	Writes the current values of the specified VU environment variables to standard output.

Statements

`COOKIE_CACHE`	Indicates the state of the cookie cache at the beginning of a session.
`DATAPOOL_CONFIG`	Provides configuration information about a datapool.
`print`	Writes to standard output when the formatting capability of `printf` is not required.

▸ ▸ ▸ Part II

Using VU

▸▸▸ CHAPTER 3

VU Fundamentals

The fundamentals of the VU scripting language are similar to the C programming language. This chapter describes the following features of VU program scripting:

- Data types
- Language elements
- Expressions
- Statements
- Comments
- Arrays
- Flow control
- Scope of variables
- Initial values of variables
- VU regular expressions
- How a VU script represents unprintable data

Data Types

The VU language supports the following data types:

- Integer
- String
- Bank

Mixing different data types in a single expression is generally not allowed. For example, an integer expression cannot be compared to a string expression, nor can a character constant be assigned to a string expression. Expressions formed with the comma (,) and conditional (?:) operators, however, do allow you to mix data types.

The data type of a variable or function can be declared or is an integer by default. The data type of an expression is predefined in the VU language or depends on its own operators and operands.

Integer

An integer can be of any class, but only integers can be shared. Characters and shared variables are special cases of the integer data type. Integer expressions, including character constants, have 32-bit integer values. Although the default type of a variable is integer, a variable can be explicitly declared integer for clarity.

```
int int_name_1, int_name2;
```

String

The string data type is a basic VU data type, just like `int`. In the C language, a string is an array of characters, but the VU programmer need not allocate or deallocate storage. The value of a string expression is a set of characters. The following statement declares two variables as the string data type:

```
string string_name_1, string_name_2;
```

Bank

A `bank` is a nonscalar (composite) data type that consists of a collection of zero or more scalar data items (integers, strings, or both). The position of data items within a bank is significant only within data items of the same data type; the position is insignificant within data items of different data types. Bank expressions are used with the environment variables `Escape_seq`, `Logout_seq`, and `Mybstack`. The VU language does not allow you to define bank variables or bank functions.

Bank expressions can be created in the following ways:

- With the built-in function `bank`.
- By evaluating the value of a bank environment variable with the `eval` environment control command.
- By creating a union of two bank expressions with the + operator.

Information about the contents of a bank expression can be determined as follows:

- *`bank_exp`*`[int]` returns the number of integer data items in *`bank_exp`*.
- *`bank_exp`*`[string]` returns the number of string data items in *`bank_exp`*.

- *bank_exp*[int][*n*] returns the *n*th integer data item in *bank_exp*, where *n* is an integer expression such that 0 < *n* ≤ *bank_exp*[int]. If *n* is outside this range, a VU runtime error is generated.
- *bank_exp*[string][*n*] returns the *n*th string data item in *bank_exp*, where *n* is an integer expression such that 0 < *n* ≤ *bank_exp*[string]. If *n* is outside this range, a VU runtime error is generated.

Language Elements

A VU script contains identifiers, constants, operators, and keywords. For a list of keywords, see Appendix A, *Keywords*.

Identifiers

Identifiers are named by the programmer. An identifier must begin with an alphabetic character, and it consists of any combination of alphabetic characters, underscores (_), and digits. Uppercase and lowercase alphabetic characters are differentiated, so, for example, `RATIONAL` and `rational` are both unique identifiers.

Identifiers are used to represent:

- Variables
- Names of functions and procedures
- Arguments of functions or procedures
- Datapools

Constants

The VU language supports integer, character, string, and array constants. For information about arrays and array constants, see *Arrays* on page 3-18.

Integer Constants

Integer constants can be specified in decimal, octal, or hexadecimal format. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal; otherwise, the integer constant is considered decimal. For example, decimal 63 written as 63 in decimal, 077 in octal, or 0x3F, 0X3F, 0x3f, or 0X3f in hexadecimal format. All integer constants are treated as 32-bit integers. Negative numbers are obtained by prefacing the integer constant with the unary negation operator (-).

Character Constants

Character constants are specified by enclosing the constant in single quotation marks. A character constant always represents a single character.

String Constants

The VU language allows two types of string constants: standard and pattern. The difference between standard and pattern string constants is in how they treat the backslash character. Pattern string constants allow you to use the backslash character to specify patterns.

To specify a standard string constant, enclose the constant in double quotation marks (`""`). To specify a pattern string constant, enclose the constant in single quotation marks (`''`). If a null character (`\0`) is placed in a string constant, the null character and all remaining characters in the string constant are ignored. A double quotation mark can be included in a standard string constant by prefacing the quotation mark with a backslash (\).

For standard string and character constants, the backslash character is represented by two backslashes (\\). A single backslash is ignored unless it occurs in a sequence. For pattern string constants, the backslash character is never ignored. If it is part of a sequence, the escape sequence (including the backslash itself) represents the corresponding ASCII character. If it precedes the single quotation mark, it indicates that the quotation mark is part of the string instead of the final string delimiter. For example, the backslash and single quotation mark represent a single quotation mark. Otherwise, the backslash and the character that follow it have no special interpretation.

Since both pattern string constants and character constants are delimited by single quotation marks, the characters inside the quotation marks determine whether the constant is a character constant or a pattern string constant. If the characters enclosed by the quotation marks can be interpreted as representing a single character, the constant is a character constant. Otherwise, it is a pattern string constant.

Adjacent string constants are concatenated at compile time as in ANSI C.

For example, `"good-bye," "cruel world"` is equivalent to `"good-bye, cruel world"`. This is useful for splitting long string constants across multiple lines, and applies to both standard and pattern string constants, or to any combination of the two types.

Examples of Constants

The following table lists examples of character constants, standard string constants, and pattern string constants:

Constant	Type	Description
`'a'`	character	Simplest form of character constant.
`'\''`	character	Represents a single quotation mark. It is preceded by a backslash.
`'ab'`	pattern string	Simple two-character pattern string constant.
`'\7'`	character	Represents the character constant with ASCII value `7` (bell). There is no way to specify the two-character pattern string `\7`. A string containing these characters can be specified with the standard string constant `"\\7"`.
`'\9'`	character	Represents the character `9` since the backslash is ignored.
`'7\\'`	pattern string	The pattern string constant contains the three characters `7\\`.
`'\\'`	character	Represents the backslash character.
`'\141'`	character	Equivalent to `'a'` since the ASCII value of `a` is 141.
`'\148'`	pattern string	The pattern string contains two characters: form feed (ASCII 014) and 8. This is not interpreted as a character constant as the previous example because 148 is not an octal number.
`'a\r\8\b'`	pattern string	The pattern string constant contains five characters: `a`, carriage return, backslash, `8`, and backspace.
`"\a\r\\8\b"`	standard string	Equivalent to the pattern string constant of the previous example.
`"\a\r" '\8\b'`	concatenated string	Also equivalent to the previous example, using string constant concatenation of a standard string constant and a pattern string constant.

(Continued)

Constant	Type	Description
`'\\\n'`	pattern string	The pattern string constant contains three characters: backslash, backslash, and newline.
`'\\n'`	pattern string	The pattern string constant contains three characters: backslash, backslash, and `n`. This is not interpreted as a backslash followed by newline, since — processing left to right — the second backslash is associated with the first backslash, and not the `n`.

Operators

The VU language offers a full range of operators for integer, string, and bank expressions. Not all operators are valid with all expressions. When used with expressions whose data type is integer, the VU operators generally perform the same as operators in C, except that VU integers are always 32 bits in size. To simplify common string operations, the VU language also defines operators on string expressions that are not provided in C.

For information about operators that work with arrays, see *Array Operators* on page 3-22. The following conventions are used in this section:

- *`int1`*, *`int2`*, and *`int3`* refer to arbitrary integer expressions.
- *`str1`*, *`str2`*, and *`str3`* refer to arbitrary string expressions.
- *`exp1`*, *`exp2`*, *`exp3`*, and *`exp4`* refer to arbitrary expressions of either integer or string type.
- *`bank_exp1`* and *`bank_exp2`* refer to arbitrary bank expressions.
- *`any_exp1`* and *`any_exp2`* refer to arbitrary expressions of any type such as:
  - integer
  - string
  - array
  - bank

Binary Arithmetic Operators

The binary arithmetic operators are + , -, *, /, and %. The data type of an expression containing a binary arithmetic operator is the same as the type of the operands. None of these operators change the values of their operands. Binary arithmetic operators require two operands of the same data type.

Operators for Integers

The binary arithmetic operators + , -, *, /, % support integer operands. They provide 32-bit addition, subtraction, multiplication, integer division, and modulus (`int1 %` *`int2`* = the remainder of *`int1`* divided by *`int2`*).

Operators for Strings

The only binary arithmetic operator to support string operands is the concatenation operator + . The string expression *`str1`* + *`str2`* returns *`str2`* concatenated to *`str1`*. The string expression *`str3`* = *`str1`* + *`str2`* is equivalent to the C statement `strcat(strcpy(`*`str3`*,*`str1`*),*`str2`*).

Operators for Bank Expressions

The only binary arithmetic operator to support bank operands is the union operator, + . The bank expression *`bank_exp1`* + *`bank_exp2`* returns a bank containing all of the integer and string data items of both *`bank_exp1`* and *`bank_exp2`*. For example, if *`bank_exp1`* is equivalent to `bank(1, "ab",2,"xy")` and *`bank_exp2`* is equivalent to `bank("def",3,4,"ghi")`, then *`bank_exp1`* + *`bank_exp2`* is equivalent to `bank(1,2,3,4,"ab","xy","def","ghi")`.

Ordering among data items of the same type is retained; therefore, the + operator is not commutative for the bank operands.

Binary Bitwise Operators

The binary bitwise operators require two integer operands and always operate on all 32 bits of each operand. The operations are identical to that of their C language counterparts when operating on unsigned 32-bit quantities. The data type of an expression containing a binary bitwise operator is integer. None of these operators change the values of their operands.

The following table shows the binary bitwise operators:

Operator	Description
&	bitwise AND *int1* & *int2* has bits set to 1 that are set to 1 in both *int1* and *int2*; the remaining bits are set to 0.
|	bitwise OR *int1* | *int2* has bits set to 1 that are set to 1 in either *int1* or *int2*; the remaining bits are set to 0.
^	bitwise exclusive OR *int1* ^ *int2* has bits set to 1 in each bit position where *int1* and *int2* have different bits; the remaining bits are set to 0.
<<	left shift *int1* << *int2* has the value of *int1* shifted left by *int2* bit positions, filling vacated bits with 0; *int2* must be positive.
>>	right shift *int1* >> *int2* has the value of *int1* shifted right by *int2* bit positions, filling vacated bits with 0; *int2* must be positive.

Assignment Operators

Assignment operators require two operands of the same type. The first operand of an assignment operator must be a variable. The type and value of an expression containing an assignment operator is always equivalent to the type and value of its second (rightmost) operand.

The value on the left of the operator (*int1*) changes to the value specified; the value on the right of the operator (*int2*) does not change.

If you are reading and updating a shared variable, your read-and-update operation is mutually exclusive of any other virtual user's update of that variable.

The following table shows the assignment operators:

<table>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>=</td><td>int1= int2 changes the value of int1 to that of int2.</td></tr>
<tr><td>+ =</td><td>int1 + = int2 changes the value of int1 to that of int1 + int2.</td></tr>
<tr><td>-=</td><td>int1 -= int2 changes the value of int1 to that of int1 - int2.</td></tr>
<tr><td>*=</td><td>int1 *= int2 changes the value of int1 to that of int1 * int2.</td></tr>
<tr><td>/=</td><td>int1 /= int2 changes the value of int1 to that of int1 / int2.</td></tr>
<tr><td>%=</td><td>int1 %= int2 changes the value of int1 to that of int1 % int2.</td></tr>
<tr><td>&=</td><td>int1 &= int2 changes the value of int1 to that of int1 & int2.</td></tr>
<tr><td>| =</td><td>int1 | = int2 changes the value of int1 to that of int1 | int2.</td></tr>
<tr><td>^ =</td><td>int1 ^ = int2 changes the value of int1 to that of int1 ^ int2.</td></tr>
<tr><td>< < =</td><td>int1 < < = int2 changes the value of int1 to that of int1 < < int2.</td></tr>
<tr><td>> > =</td><td>int1 > > = int2 changes the value of int1 to that of int1 > > int2.</td></tr>
<tr><td>=</td><td>str1= str2 changes the value of str1 to that of str2; str2 is unchanged.</td></tr>
<tr><td>+ =</td><td>str1+ = str2 changes the value of str1 to the concatenation of str1 and str2; str2 is unchanged.</td></tr>
</table>

Unary Operators

Unary operators require one integer or string operand. The type of an expression containing a unary operator is the type of the operand.

The following table describes the unary operators:

Operator	Description
!	logical negation If the value of *int1* is nonzero, !*int1* equals 0; if the value of *int1* is 0, !*int1* equals 1. In either case, *int1* is unchanged.
&	address of The & operator is valid in an external C function expecting the passed address of a variable and in the following function calls: ▶ `fscanf` ▶ `scanf` ▶ `sscanf` ▶ `match` ▶ `wait` ▶ `sprintf` For integer operands, &*int1* equals the address of *int1*; *int1* is unchanged. The operand of & must be an integer variable or integer array element. Semantically, the integer operand of & must be a normal integer variable (or array element) or a shared integer variable, depending on the associated function definition. For string operands, &*str1* equals the address of *str1*; *str1* is unchanged. The operand of & must be a string variable or string array element.
+ +	increment (+ + *int1*) equals *int1+1* when evaluated in an expression; (*int1++*) equals *int1* when evaluated, and is incremented after evaluation. The operand must be a variable or integer array element. If you are reading and incrementing a shared variable, your read-and-update operation is mutually exclusive of any other virtual user's update of that variable.

(Continued)

Operator	Description
--	decrement (--`int1`) equals `int1`-1 when evaluated in an expression; (`int1`--) equals `int1` when evaluated, and is decremented after evaluation. The operand must be a variable or integer array element. If you are reading and decrementing a shared variable, your read-and-update operation is mutually exclusive of any other virtual user's update of that variable.
-	negation -`int1` equals the additive inverse of `int1`. `int1` is unchanged.
~	bitwise one's complement sets bits to one that are zero in `int1`; the remaining bits are set to zero. `int1` is unchanged.

Relational Operators

The relational operators consist of &&, `||`, >, <, >=, <=, ==, and `!=`. The data type of an expression containing a relational operator is always integer. None of the relational operators change their operands. Relational operators require two operands of the same data type.

As in C, the implementations of && and `||` guarantee left-to-right evaluation and do not perform unnecessary operand evaluation. In other words, the second operand of && is not evaluated if the first operand has the value 0; likewise, the second operand of || is not evaluated if the first operand has a nonzero value.

The following table shows the relational operators for integer operands:

Operator	Description
&&	logical AND `int1` && `int2` equals 1 if both `int1` and `int2` have nonzero values. Otherwise, it equals 0.
||	logical OR `int1` || `int2` equals 0 if both `int1` and `int2` have the value 0. Otherwise, it equals 1.
>	greater than `int1`> `int2` equals 1 if `int1` is greater than `int2`. Otherwise, it equals 0.

(Continued)

Operator	Description
<	less than *int1* < *int2* equals 1 if *int1* is less than *int2*. Otherwise, it equals 0.
> =	greater than or equal to *int1* > = *int2* equals 1 if *int1* is not less than *int2*. Otherwise, it equals 0.
< =	less than or equal to *int1* < = *int2* equals 1 if *int1* is not greater than *int2*. Otherwise, it equals 0.
= =	equality *int1* = = *int2* equals 1 if *int1* and *int2* have the same value. Otherwise, it equals 0.
!=	inequality *int1* != *int2* equals 0 if *int1* and *int2* have the same value. Otherwise, it equals 1.

The following table shows the relational operators for string operands:

Operator	Description
>	greater than *str1* > *str2* equals 1 if *str1* is greater (based on the machine's collating sequence) than *str2*. Otherwise, it equals 0. Equivalent to the C expression (1 = = *strcmp*(*str1*,*str2*)).
<	less than *str1* < *str2* equals 1 if *str1* is less (based on the machine's collating sequence) than *str2*. Otherwise, it equals 0. Equivalent to the C expression (-1 = = strcmp(*str1*,*str2*)).
> =	greater than or equal to *str1* > = *str2* equals 1 if *str1* is not less than *str2*. Otherwise, it equals 0. Equivalent to the C expression (-1 != strcmp(*str1*,*str2*)).
< =	less than or equal to *str1* < = *str2* equals 1 if *str1* is not greater than *str2*. Otherwise, it equals 0. Equivalent to the C expression (1 != strcmp(*str1*,*str2*)).

(Continued)

Operator	Description
= =	equality *str1* = = *str2* equals 1 if *str1* and *str2* have the same value. Otherwise, it equals 0. Equivalent to the C expression (!`strcmp`(*str1*,*str2*)).
!=	inequality *str1* != *str2* equals 0 if *str1* and *str2* have the same value. Otherwise, it equals 1. Equivalent to the C expression (`strcmp`(*str1*,*str2*)).

Other Operators

The VU language offers two additional operators — the comma operator (,) and the conditional operator (?:). The following table describes these operators:

Operator	Description
,	comma The **comma operator** allows operands of different types. For any two expressions *exp1* and *exp2*, the resulting value of the "*exp1*, *exp2*" is the value of *exp2*, and the resulting type is the type of *exp2*. The operands of the comma operator are not changed. The comma operator is used only in the `for` statement, as in `for` (*exp1*; *exp2*; *exp3*,*exp4*) and cannot have bank expressions as its operand. The comma is also used as a grammatical symbol in other places in the VU language — for example, to separate arguments in a function call.
?:	The **conditional operator** requires three operands. The expression *int1* ? *any_exp1* : *any_exp2* has the value and type of *any_exp1* if *int1* is nonzero. Otherwise, the expression has the value and type of *any_exp2*. *any_exp1* and *any_exp2* must have the same type. None of *any_exp1*, *any_exp2*, or *int1* are changed.

Operator Precedence and Associativity

The following table shows the operator precedence and associativity of each VU operator. ("Associativity" is the order in which operators of the same precedence are evaluated.) Operators in the same row have the same precedence. The precedence decreases with each row.

Use parentheses to change the order of evaluation of an expression. An expression inside parentheses is always evaluated first, and the extra parentheses are ignored.

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td>( )   [ ]</td><td>left-to-right</td></tr>
<tr><td>- (unary)   !   ~   & (address of) ++ --</td><td>right-to-left</td></tr>
<tr><td>*   /   %</td><td>left-to-right</td></tr>
<tr><td>+   - (binary)</td><td>left-to-right</td></tr>
<tr><td>>>   <<</td><td>left-to-right</td></tr>
<tr><td>>   >=   <   <=</td><td>left-to-right</td></tr>
<tr><td>==   !=</td><td>left-to-right</td></tr>
<tr><td>& (bitwise AND)</td><td>left-to-right</td></tr>
<tr><td>^</td><td>left-to-right</td></tr>
<tr><td>|</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>left-to-right</td></tr>
<tr><td>||</td><td>left-to-right</td></tr>
<tr><td>?:</td><td>right-to-left</td></tr>
<tr><td>= += -= *= /= %= &= |= ^= <<= >>=</td><td>right-to-left</td></tr>
<tr><td>,</td><td>left-to-right</td></tr>
</table>

Expressions

An expression contains one or more VU identifiers, constants, keywords, and operators. Every expression has a data type and a value. The data type of an expression determines how its value is interpreted. Each of the following VU language constructs is an expression:

- Constant
- Variable
- Argument
- Read-only variable
- `eval` *`environment_variable`*
- *`unary_operator expression`*
- *`expression unary_operator`*
- *`expression binary_operator expression`*
- *`expression`* ? *`expression`* : *`expression`*
- *`bank_expression`*`[int]`
- *`bank_expression`*`[string]`
- *`bank_expression`*`[int]`[*`int_expression`*]
- *`bank_expression`*`[string]`[*`int_expression`*]
- *`array_variable`*[*`int_expression`*]
- *`array_variable`*[*`int_expression`*][*`int_expression`*]
- *`array_variable`*[*`int_expression`*][*`int_expression`*][*`int_expression`*]
- Function (a function invocation or call)
- Emulation command
- `limitof` array

Statements

Statements contain one or more VU expressions. Not all statements are valid everywhere in a VU script. For example, argument assignments and `return` statements are invalid outside of function or procedures, and the `break` and `continue` statements are invalid outside of loops.

The following table shows the VU statements:

Statement	Description
;	Null statement.
variable asgn_op exp;	Variable assignment. *asgn_op* is any assignment operator; *exp* is an integer or string expression.
int_exp;	*int_exp* is an integer expression, which includes integer function calls and emulation commands. (String function calls cannot be used as VU statements by themselves, but only as a part of a VU expression.)
environment_control_command env_var;	`push`, `pop`, etc. *env_var* is any environment variable.
environment_control_command [*env_var_list*];	`push`, `pop`, etc. *env_var_list* is a comma-separated list of one or more environment variables.
`break`;	Break.
`break` *integer_constant*;	Multilevel break.
`continue`;	Continue.
`continue` *integer_constant*;	Multilevel continue.
`DATAPOOL_CONFIG`	See *DATAPOOL_CONFIG* on page 6-13 for detailed syntax.
`COOKIE_CACHE`	See *COOKIE_CACHE* on page 6-11 for detailed syntax.
`if` (*int_exp*) *statement*	*int_exp* is an integer expression; *statement* is any valid statement form, defined recursively.
`if` (*int_exp*) *statement* `else` *statement*	*int_exp* is an integer expression; *statement* is any valid statement form, defined recursively.

(Continued)

Statement	Description
procedure_name (*exp_list*);	Procedure call. *exp_list* is a comma-separated list of 0 or more expressions.
print *exp_list*;	*exp_list* is a comma-separated list of one or more expressions.
return;	Return
return *exp*;	*exp* is an integer, array, or string expression that is returned to the calling function or procedure.
sync_point *string_const*	*string_const* is the name of a synchronization point.
while (*int_exp*) *statement*	*int_exp* is an integer expression; *statement* is any valid statement form, defined recursively.
do *statement* while (*int_exp*);	*statement* is any valid statement form, defined recursively; *int_exp* is an integer expression.
for (*exp_list* ; *int_exp* ; *exp_list*) *statement*	*exp_list* is a comma-separated list of zero or more expressions; *int_exp* is an optional integer expression; *statement* is any valid statement form, defined recursively.
{ *declaration_list* *statement_list* }	*declaration_list* contains 0 or more declarations. *statement_list* contains 0 or more statements.
declaration	*class type name_list*: ▶ *class* (optional) can be: shared, persistent, or external_C. Only type int may be shared. ▶ *type* may be int or string. *type* may be omitted for integer declarations. ▶ *name_list* is a comma-separated list of one or more identifiers; each identifier is optionally followed by the initializer = *constant*, where *constant* is the same type as the identifier.

Comments

Comments are delimited by the characters `/*` and `*/`. The following example shows a one-line comment and a two-line comment:

```
/* This is the main body of the script */
/* This comment contains
more than one line */
```

Comments cannot include other comments.

Arrays

The VU language supports arrays of up to three dimensions of all scalar data types, such as integer and string.

Array elements are referenced by integer expression subscripts enclosed in brackets (`[ ]`). Array indexing is zero based. The first element of an array is referenced by index 0. Multidimensional arrays are subscripted by multiple pairs of brackets. Arrays are declared as a fixed size or as expandable. Expandable arrays grow as necessary up to an optional maximum size.

Array Constants

Array constants are specified as a list of scalar constants enclosed in braces. All scalar constants in the list must be of the same type. For example, `{ 1, 2, 3, 4 }` is an array constant of four integers. A multidimensional array constant is specified as a list of array constants enclosed in braces:

```
{ { "this", "is" },
  { "a", "two", "dimensional", "array" },
  { "of", "strings" } }
```

All arrays in a multidimensional array constant must be of the same type but not necessarily the same size.

You can use the repeat operator (:) to specify repetition of a constant element array. The array constant:

```
{ 1:5, 2:3, 3:4 }
```

contains 12 elements and is the same as the constant:

```
{1,1,1,1,1,2,2,2,3,3,3,3}
```

The repeat operator is also used to repeat array constants:

```
{ { { 1:3, 2:2 }, { 5:6 }:3 }:2 }
```

is the same as:

```
{ { { 1, 1, 1, 2, 2 },
    { 5, 5, 5, 5, 5, 5 },
    { 5, 5, 5, 5, 5, 5 },
    { 5, 5, 5, 5, 5, 5 } },
  { { 1, 1, 1, 2, 2 },
    { 5, 5, 5, 5, 5, 5 },
    { 5, 5, 5, 5, 5, 5 },
    { 5, 5, 5, 5, 5, 5 } } }
```

Array constants are allowed only as the right-hand side of an array assignment or in an array initialization.

Declaring an Array

An array declaration has the form:

```
class type name [m..M,g];
class type name [m..M,g] [m..M,g];
class type name [m..M,g] [m..M,g] [m..M,g];
```

The declaration has these parts:

- *`class`* is optional (only `persistent` and `external_C` are allowed).
- *`type`* is the scalar type, which can be `int` or `string.`
- *`name`* is the name of the array.
- *`[m..M,g]`* is a dimension specification. It indicates the minimum and maximum number of elements the array can contain, and a growth size.
  - *`m`* is an integer constant that specifies the minimum (initial) size of the array. The minimum initial size of a dimension is useful when combined with initialization as described below.
  - *`M`* is an integer constant that specifies the maximum size of the array.
  - *`g`* is an integer constant that specifies the growth size of the array. For efficiency, declare a expandable array with a growth size, which specifies the number of elements by which to grow the array.

m, *M*, *g* can be combined in the following ways:

Combination	Meaning
[*M*]	fixed size
[]	no limit, growth determined at runtime
[*m*..*M*]	initial size *m*, limit *M*, growth determined at runtime
[*M*,*g*]	no minimum, first access allocates a minimum of *g* elements
[*m*..*M*,*g*]	initial size *m*, limit *M*, grow by *g* elements
[*g*]	no limit, grow by *g* elements
[*m*..]	initial size *m*, no limit, growth determined at runtime
[*m*..,*g*]	initial size *m*, no limit, grow by *g* elements

In all cases, up to three independent sets of [*m*..*M*,*g*] are allowed, one per dimension.

Arrays can be declared persistent:

```
persistent type name [m..M,g]...;
```

Arrays cannot be declared shared.

Initializing an Array

Arrays of all types can be initialized by specifying an array constant of the appropriate type and number of dimensions in the declaration.

```
int a[5] = { 1, 2, 3, 4, 5 };
```

If the initializer has fewer elements than the array variable, the remaining elements are undefined.

Initialized arrays with a non-fixed size are created at least large enough to hold all of the elements in the initializer.

If array initializers are too large to fit in the declared array, a fatal compilation error results.

An array initializer constant can contain one or more occurrences of the colon (:) repeat operator. The repeat operator specifies repetition of a constant element. It is a binary operator with the following form:

```
constant_element : n_reps
```

The operator has these parts:

- *constant_element* is a scalar or array constant of the same type as the array initialized.
- *n_reps* is an integer constant specifying the number of times *constant_element* is repeated.

  If *n_reps* is an asterisk (*), *constant_element* is repeated as many times as necessary until the rest of the array has been initialized. With arrays of non-fixed size, *constant_element* is repeated until the rest of the minimum size of the array is initialized. If the minimum size of the array is already initialized, :* has no effect.

Example of Array Initialization

The following declaration initializes the first 5 elements of `a` to the values 1 through 5 and the next 95 elements (the rest of the array) to 0.

```
int a[100] = { 1, 2, 3, 4, 5, 0:95 };
```

The following declarations initialize all elements of the arrays to 0.

```
int a[100] = { 0:* };
int b[10..50] = { 0:* };
```

Note that `b[10..50]` declares `b` with a minimum size of 10 and a maximum of 50 elements. The initialization sets elements 0–9 of `b` to 0. All other elements of `b` are undefined.

In the following example, array `aa` above is initialized such that `aa[x][0] == 1` and `aa[x][1] == 0` for all `0 <= x <= 4`. All other elements of `aa` are undefined.

All types of array initializers can use the repeat operator, including array constants.

```
string sa[10] = { "hello", "world", "":* };
int aa[10][3] = { {1, 0}:5 };
```

The following array initialization:

```
int a[10] = { 1, 2, 0:* };
```

is the same as:

```
int a[10] = { 1, 2, 0, 0, 0, 0, 0, 0, 0, 0 };
```

The following two-dimensional array initialization:

```
int aa[7][] = { { 1, 2, 3, 4 }:3, { 0 }:* };
```

is the same as:

```
int aa[7][] = { { 1, 2, 3, 4},
                { 1, 2, 3, 4},
                { 1, 2, 3, 4},
                { 0 },
                { 0 },
                { 0 },
                { 0 } };
```

The following three-dimensional array initialization initializes all 1000 elements of `aaa` to `0`:

```
int aaa[10][10][10] = { { { 0:* }:* }:* };
```

The following string array initializations:

```
string sa[10] = { "abc", "123", "":* };
string saa[7][] = { { "one", "two", "three", "four" }:3, { "" }:* };
```

are the same as:

```
string sa[10] = { "abc", "123", "", "", "", "", "", "", "", "" };
string saa[7][] =
     { { "one", "two", "three", "four"},
       { "one", "two", "three", "four"},
       { "one", "two", "three", "four"},
       { "" },
       { "" },
       { "" },
       { "" } };
```

This declaration initializes all 1000 elements of `saaa` to `""`:

```
string saaa[10][10][10] = { { { "":* }:* }:* };
```

Array Subscripts

Array elements are selected by enclosing an integer expression in brackets ([]). The first element is selected by subscript 0. Multidimensional arrays can be subscripted by adjacent subscripts, each enclosed in brackets.

```
string saa[7][] = { { "one", "two", "three", "four" }:3, { "" }:* };
```

`saa[0]` is a one-dimensional array of strings with value `{ "one"`, `"two"`, `"three"`, `"four" }`.

`saa[4][0]` is a string with value `""`.

`saa[4][1]` is an undefined string.

Array Operators

In this section, *`ary1`* and *`ary2`* are arbitrary arrays of any type and any number of dimensions.

Binary Concatenation Operator for Arrays

The only binary arithmetic operator to support array operands is the concatenation operator + . The array expression `ary1` + `ary2` returns an array containing all of the elements of `ary1` followed by all of the elements of `ary2`. The elements of `ary1` and `ary2` are not changed. `ary1` and `ary2` must be array expressions of the same number of dimensions and same base type.

Assignment Operators for Arrays

The assignment operators that support array operands are = and + = .

`ary1` = `ary2` changes the value all elements in `ary1` to the values of the corresponding elements in `ary2`, including any undefined elements. The elements of `ary2` are not changed.

`ary1` + = `ary2` is equivalent to `ary1` = `ary1` + (`ary2`).

Unary limitof Operator for Arrays

`limitof` is the only unary operator with an array operand. It returns the value of the highest subscript of any defined element in the operand. For multidimensional arrays, `limitof` returns the maximum defined subscript of the outermost (first) dimension. When used on a subarray, `limitof` returns the maximum subscript for the subarray. If all elements of an array are undefined, `limitof` returns -1.

The maximum defined subscript returned by `limitof` means that no larger subscript has a defined value, *not* that all smaller subscripts of the same array have defined values. This VU script clarifies the use of `limitof`:

```
{
 int a[25];
 int b[][];
 a[10] = 1;
 a[8] = 2;
 b[3][20] = 5;
 b[2][15] = 7;
 printf("limitof a is %d\n", limitof a);
 printf("limitof b is %d\n", limitof b);
 printf("limitof b[3] subarray= %d\n", limitof b[3]);
 printf("limitof b[2] subarray= %d\n", limitof b[2]);
 printf("limitof b[1] subarray= %d\n", limitof b[1]);
}
```

The output is:

```
limitof a is 10
limitof b is 3
limitof b[3] subarray= 20
limitof b[2] subarray= 15
limitof b[1] subarray= -1
```

Arrays as Subroutine Arguments

User-defined functions and procedures can have array arguments. An array argument is declared the same as an array variable. Array arguments are always passed by address, not by value. Functions and procedures can freely modify the elements of any array argument.

Flow Control

The VU language offers two types of flow control: conditional execution (the `if-else` and `else-if` structures) and looping (`for`, `while`, and `do-while` structures). The VU language also features `break` and `continue` statements to allow for controlled exit from a loop. Except for enhancements added to `break` and `continue`, the VU control structures behave like their C counterparts.

Loops

VU loops allow VU statements to be executed repeatedly. Loops include `for`, `while`, and `do-while`.

Break and Continue

The VU `break` and `continue` statements allow for more flexible control over the execution of `for`, `while`, and `do-while` loops. As in C, if the `break` statement is encountered as one of the statements in a `for`, `while`, or `do-while` loop, execution of that loop stops immediately. Also, as in C, if the `continue` statement is encountered as one of the statements in a `while` or `do-while` loop, the remaining statements in the loop are skipped, and execution continues with the evaluation step of the loop.

Unlike C, however, the VU `break` and `continue` statements have an optional argument, which specifies the nested loop level where the `break` or `continue` statement is executed.

Scope of Variables

By default, the scope of a variable is limited to one runtime instance of a script for one virtual user. However, you can declare a variable as shared or persistent.

The following table lists the differences between shared variables and persistent variables:

Shared Variable	**Persistent Variable**
One copy for all virtual users to access.	Each virtual user has its own copy.
Maintains its value across all scripts.	Maintains its value across scripts of that virtual user only.
Data type must be integer.	Data type is an integer or string, or is an array of integers and strings.

Other VU variables and functions are global in scope within a runtime instance of a script but private to each virtual user. Subroutine arguments are local to that subroutine and are unknown to the rest of the script.

Shared Variables

A shared variable is an integer variable. Any discussion of integer variables also applies to shared variables, and you can use a shared variable anywhere you can use an integer variable except as the operand for the address-of operator (&).

You can use a shared variable to:

- Set loop maximums when you repeat operations, to set transaction rates, and to set average delay times.
- Block a virtual user from further execution until a global event occurs. For example, if you are re-indexing a SQL table, you would want to block access to that table until the indexing is complete. You can use the `wait` library function with a shared variable to do this.
- Pause a script's execution until a specified number of virtual users arrive at that point. However, it is simpler to use the synchronization functions to do this.

You create a shared variable within a VU script.

To declare shared variables, use the `shared` keyword. You do not need to declare the shared variable as integer because all shared variables are integer variables. The following two examples declare both `first_shared` and `second_shared` as shared variables, but the second example includes the keyword `int` for documentation:

```
shared first_shared, second_shared;
shared int first_shared, second_shared;
```

Shared variables have an initial value of 0 for a run. You can set a different initial value in the schedule, and you can modify the initial value anywhere in a VU script.

The following example modifies the value of `first_shared` to 17:

```
shared first_shared;
first_shared = 17;
```

Once you have started playing back the script, you can change the value of a shared variable when you monitor the schedule.

A variable that is not declared shared is local to both the script and the virtual user, and is unrelated to any shared variable of the same name in other scripts.

Updating a shared variable takes more time than updating a normal integer variable. This is because if two virtual users try to update a shared variable, extra communication is necessary to make sure that the variable is locked from the second user until the first user's update completes. If the schedule run involves Agent computers, further communication is necessary to coordinate access among multiple computers.

Reading a shared variable takes the same amount of time as reading a normal integer variable if the schedule is run only on the Master computer. However, if the schedule run involves Agent computers, extra communication is necessary to coordinate access among multiple computers, and thus reading a shared variable will take more time.

Persistent Variables

Persistent variables are useful when you want to retain the value of a variable between scripts. For example:

- You have opened a file in persistent mode, and you want subsequent scripts to access the file without reopening it. You could use a persistent integer variable to hold the return value from `open`.
- You want a virtual user to randomly choose a record from a file. You could declare a persistent array of integers, and load the keys into that array.

The initial value of a persistent variable in a script is determined as follows:

1. If a persistent variable has the same name (and type) in a previously executed script in the session (by that virtual user), the initial value of the persistent variable in the current script is inherited from the final value of that persistent variable in the most recently executed script in which it was declared. Otherwise:
2. If the declaration of the persistent variable included an initializer, then the initial value is taken from the initializer. Otherwise:
3. The initial value is undefined (like any non-persistent variable).

A persistent variable must be declared persistent in any script that accesses it.

A non-shared variable declared persistent without a type is integer by default.

A variable that is not declared persistent is local to that script and is unrelated to any persistent variables of the same name in other scripts.

Shared variables and function or procedure arguments cannot be declared persistent.

If a persistent variable has a type conflict with a persistent variable of the same name but in a previous instance of the same script, a fatal error occurs.

Examples

The comments in the following examples illustrate many of the points made in the preceding section. These examples are based on the assumption that the scripts are run in the order A, B, C.

Script A

```
persistent fd;
persistent string user_nickname, s1, s2;
persistent int where_am_i;
{
    fd = open("foo", "pw+"); /* open persistent */
    user_nickname = "Slick";
    s1 = "hello world";
}
```

Script B

```
persistent fd;
persistent string user_nickname, s2;
persistent p1=10;
string s1;   /* not persistent */
/* fd contains the file descriptor returned by
 * script A’s open call. user_nickname == "Slick"
 * s2 is undefined. p1==10;
 * s1 is not persistent and therefore does not
 * inherit the final value of s1 from the
 * previous script, thus it is undefined.
 */
{
    s1 = "good-bye world";
}
```

Script C

```
persistent string s1= "ignored_value";
int where_am_i;
/* s1 == "hello world" ( from script A )
 * int where_am_i is undefined and unrelated
 * to int where_am_i from script A.
 */
{ ... }
```

Initial Values of Variables

You set the initial values for unshared variables in a script. There is no default value for unshared variables.

You can initialize a variable when you declare it. In this example, `i` is `5`, `s1` and, `s2` are `"hello"`, `s3` is `"there"`, and `first_shared` is `0`:

```
    int i = 5;
    string s1, s2 = "hello", s3 = "there";
    shared first_shared;
```

You can set the initial values for shared variables when you run a schedule. However, if you do not declare a value for a shared variable, its value is 0.

You get a runtime error if an expression contains an undefined variable or an uninitialized, declared variable.

For information about initializing an array variable, see *Initializing an Array* on page 3-20.

VU Regular Expressions

A regular expression is a string that specifies a pattern of characters. The `match` library routine, for example, accepts strings that are interpreted as regular expressions.

VU regular expressions are like UNIX regular expressions. VU, however, offers two additional operators: `?` and `|`. In addition, VU regular expressions can include ASCII control characters in the range 0 to 7F hex (0 to 127 decimal).

General Rules

VU regular expressions have the following characteristics:

- The concatenation of single-character operators matches the concatenation of the characters individually matched by each of the single-character operators.
- Parentheses () can be used within a regular expression for grouping single-character operators. A *group* of single-character operators can be used anywhere one single-character operator can be used — for example, as the operand of the `*` operator.
- Parentheses and the following non-ordinary operators have special meanings in regular expressions. They must be preceded by a backslash if they are to represent themselves:
  - The `^` operator must be preceded by a backslash when it is the first operator of a regular expression or the first character inside brackets.
  - The `$` operator must be proceeded by a backslash when it is the last operator of a regular expression or it immediately follows a right parenthesis.
  - Operators inside brackets do not need to be preceded by a backslash.

Single-Character Regular Expression Operators

The following rules apply to single-character regular expression operators, which match at most a single character:

- Any ordinary character (any character not described below) is a single-character operator that matches itself.
- The `\` (backslash) operator and any following character match that character.

- The brackets operator [*str*], where *str* is a non-empty string, matches any single character contained in *str*, unless the first character of *str* is ^ (circumflex), in which case the operator matches any single character except those in *str*.

  A range of characters can be represented in *str* using a dash character (-)—for example, [a-z] matches all lowercase alphabetic characters. If - occurs either as the first (or after an initial ^) or last character of *str*, it specifies itself rather than a range. If] occurs as the first (or after an initial ^) character in *str*, it specifies itself rather than ending the brackets operator. The characters . (period), * (asterisk), \ (backslash), ? (question mark), | (pipe), ( ) (parentheses), [(left bracket), and + (plus) lose their special meanings in *str* and therefore are not preceded by a backslash.

- The . (period) operator matches any single character.

Other Regular Expression Operators

The following rules apply to all other regular expression operators, which operate on single-character operators or groups of single-character operators:

- The ^ (circumflex) operator, only when it is the first operator, indicates that the next operator must match the first character of the string matched.
- The $ (dollar sign) operator, only when it is the last operator, indicates that the preceding operator must match the last character of the string matched.
- The * (asterisk) operator and a preceding single-character operator match zero or more occurrences of any character matched by that operator.
- The + (plus) operator and a preceding single-character operator match one or more occurrences of any character matched by that operator.
- The {*m,n*} (braces) operator, where *m* <= *n* <= 254, and a preceding single-character operator match from *m* to *n* occurrences of any character matched by that operator. Matching exactly *m* occurrences of the operators specified by {*m*}. {*m*,} indicates *m* or more occurrences.
- The ? (question mark) operator and a preceding single-character operator match zero or one occurrence of any character matched by that operator. Therefore, ? is equivalent to {0,1}.
- The | (pipe) operator indicates alternation. When placed between *n* groups of operators, it matches the characters matched by the left group of operators that matches a non-empty set of characters.

Regular Expression Examples

The following examples show the use of VU regular expressions:

VU Regular Expression	Matches
`"ab?c"`	The strings `"abc"` and `"ac"`, as well as the strings `"defabcghi"` and `"123acc"`, since the regular expression need not specify the entire string to match. However, the strings `"ab"` and `"abbc"` do not match.
`"^ab?c$"`	Only the strings `"abc"` and `"ac"`.
`" [A-Za-z]{1,5}ly "`	Any blank-surrounded word of three to seven characters ending with `ly`.
`"^[^aeiouAEIOU]+$"`	Any sequence of characters that does not contain a vowel.
`"[0-9]+"`	Any integer.
`"^[dr]etract$"`	Only the words detract and retract.
`"((Mon)|(Tues)|(Wednes)|(Thurs)|(Fri)|(Satur)|(Sun))day"`	Any day of the week.
`"(abc\\(){1,2}"`	One or two occurrences of the string `"abc("`. Because the pattern is specified as a standard string constant, two backslashes must be used to escape the special meaning of (. The pattern could also be specified as `'(abc\(){1,2}'` using a pattern string constant.
`"((abbcc)|(a+b+c)|(abc+))$0$"`	If the string matched is `"abc"`, the second alternative (`"a+b+c"`) is matched and the string `"abc"` is returned. If the string matched is `"aabbcc"`, the first alternative is matched, and the string `"abbcc"` is returned. If the string matched is `"abcccc"`, the third alternative is matched and the string `"abcccc"` is returned. If the string matched is `"abbbcc"`, none of the alternatives match.

(Continued)

VU Regular Expression	Matches
`"(to+ chea[pt].*){2}"`	The strings `"We would rather sell too cheap than to cheat"` and `"Expect one to cheat who is too cheap"`, as well as `"'too cheat' makes no more sense than 'to cheap'"`.
`"^$(([0-9]{200}){50}){100,}"`	Any sequence of a million or more digits starting with $.
`"[a-fA-F0-9]{1,4}"`	Any hexadecimal number with a decimal value in the range 0 to 65535.
`"[ACF-IK-PR-W][a-y]{2,4} [a-y][CDIJMVY]?[a-z]{0,7}"`	The name of any state in the United States.
`"((K[a-zA-Z]*)$0 (D[a-zA-Z]*)$1 (S[a-zA-Z]*)$2) |((S[a-zA-Z]*)$0 (J[a-zA-Z]*)$1 (D[a-zA-Z]*)$2)"`	The full name (first, middle, and last names) of anyone with the initials KDS or SJD, provided the name contains only alphabetic characters. Strings matching the first, middle, and last names are returned.
`"^(([a-zA-Z]+)|([0-9]+))$"`	Any string containing only alphabetic or only numeric characters. The outermost set of parentheses is necessary because the $ operator has precedence over the | operator.

Regular Expression Errors

If a VU regular expression contains an error, when you run a schedule, LoadTest writes the message to `stderr` output prefixed with the following header:

`sqa7vui`*`#xxx`*`: fatal `*`orig type`*` error: `*`tname`*`: `*`sname`*`, line `*`lineno`*

where *`#xxx`* identifies the user ID (not present if 0), `fatal` signifies that error recovery is not possible (otherwise not present), *`orig`* specifies the error origination (user, system, server, or program), and *`type`* specifies the general error category (initialization, argument parsing, script initialization, or runtime).

If the error occurred during execution of a script (run-time category), *`tname`* specifies the name of the script being executed when the error occurred, *`sname`* specifies the name of the VU source file that contains the VU statement causing the error, and *`lineno`* specifies the line number of this VU statement in the source file. Note that the source file information will not be available if the script's source cross-reference section has been stripped.

If a run-time error occurs due to an improper regular expression pattern in the `match` library function, a diagnostic message of the following form follows the header:

`Regular Expression Error = `*`errno`*
where *errno* is an error code which indicates the type of regular expression error. The following table lists the possible *errno* values and explains each.

errno	Explanation
2	Illegal assignment form. Character after `)$` must be a digit. Example: `"([0-9]+)$x"`
3	Illegal character inside braces. Expecting a digit. Example: `"x{1,z}"`
11	Exceeded maximum allowable assignments. Only $0 through $9 are valid. Example: `"([0-9]+)$10"`
30	Missing operand to a range operator `(? {m,n} + *).` Example: `"?a"`
31	Range operators `(? {m,n} + *)` must not immediately follow a left parenthesis. Example: `"(?b)"`
32	Two consecutive range operators `(? {m,n} + *)` are not allowed. Example: `"[0-9]+?"`
34	Range operators `(? {m,n} + *)` must not immediately follow an assignment operation. Example: `"([0-9]+)$0{1-4}"` Correction: `"(([0-9]+)$0){1-4}"`
36	Range level exceeds `254`. Example: `"[0-9]{1-255}"`
39	Range nesting depth exceeded maximum of 18 during matching of subject string.
41	Pattern must have non-zero length. Example: `""`
42	Call nesting depth exceeded 80 during matching of subject string.
44	Extra comma not allowed within braces. Example: `"[0-9]{3,4,}"`
46	Lower range parameter exceeds upper range parameter. Example: `"[0-9]{4,3}"`

errno	Explanation
49	`'\0'` not allowed within brackets, or missing right bracket. Example: `"[\0] or [0-9"`
55	Parenthesis nesting depth exceeds maximum of 18. Example: `"(((((((((((((((((((x)))))))))))))))))))"`
56	Unbalanced parentheses. More right parentheses than left parentheses. Example: `"([0-9]+)$1)"`
57	Program error. Please report.
70	Program error. Please report.
90	Unbalanced parentheses. More left parentheses than right parentheses. Example: `"(([0-9]+)$1"`
91	Program error. Please report.
100	Program error. Please report.

How a VU Script Represents Unprintable Data

A VU script can contain unprintable data. For example, you can include a carriage return in a string or character constant. A session that recorded HTTP or socket traffic can generate scripts that contain binary data. The following sections describe how unprintable data is represented within a VU script.

Unprintable String and Character Constants

The following table shows how you represent unprintable characters in a string or character constant. The VU compiler interprets the character sequence as a single character:

Character Sequence	Description	ASCII value (octal)
`\r`	A single character representing a carriage return.	ASCII 015
`\f`	A single character representing a form feed.	ASCII 014
`\n`	A single character representing a newline.	ASCII 012

(Continued)

Character Sequence	Description	ASCII value (octal)
`\t`	A single character representing a horizontal tab.	ASCII 011
`\b`	A single character representing a backspace.	ASCII 010
`\0`	The null character (the character with value 0).	
`\`*`ddd`*	A single character representing the character *`ddd`*.	*`ddd`* represents 1, 2, or 3 digits; for example, `\141` represents the character `a`

Unprintable HTTP or Socket Data

If you are working with HTTP data or raw socket data, in addition to carriage returns and form feeds, you can send or receive binary data — images, sounds, and so on. With string arguments in the following HTTP and socket emulation commands, binary data can be represented within the string data through embedded hex strings:

- `http_request`, `http_recv`, and `http_nrecv`
- `sock_send`, `sock_recv`, and `sock_nrecv`

An embedded hex string represents binary characters by their two-character hexadecimal values. The entire hexadecimal string is delimited by grave accent (`‘`) characters.

Similarly, if you are coding a VU script by hand, you can represent binary characters by using a two-character hex format and delimiting the string with a grave accent. The string can contain these characters: `0123456789ABCDEFabcdef`. To represent a grave accent, use `\\‘` or `‘60‘`.

▸ ▸ ▸ CHAPTER 4

Scripts, Subroutines, and C Libraries

This chapter describes the script and header files that Robot compiles after recording or editing. It also describes the external library files that you can create and maintain outside of the Robot environment, as well as the subroutines that you can add to scripts and external files. The chapter includes the following topics:

- Program structure
- Preprocessor features
- Defining your own subroutines
- Accessing external C data and functions

Program Structure

VU program structure is similar to the structure of the C programming language.

The following sample of code shows the structure of a VU script. Your script is not required to have all of the elements in the sample. For example, if your script does not include another source file, it would not use the `#include` file name directives. If your script does not contain any user-defined procedures, it would not include the `proc` section.

```
#include <VU.h>
#include <VU_tux.h>
/* Use either of these forms to include another source file */
#include <filename>
#include "filename"
#define orig_ident new_token
/* Any user-defined procedures would be here*/
proc proc_name()
{ /* body of procedure */ }
/* Any user-defined functions would be here*/
func function_name()
{ /* body of function */ }
/* additional procedures and functions */
/* main body of script follows: */
{
string declarations;
shared declarations;
/* VU code goes here*/
}
```

You must define all subroutines before they are referenced; otherwise, you will get a syntax error. Subroutines included after the main body of the script are not referenced. They are ignored if they are syntactically correct.

Header Files

VU header files contain a collection of definitions and macros. `VU.h` is automatically included in scripts generated from recording HTTP, SQL, and socket sessions. `VU_tux.h` is automatically included in scripts generated from recording a TUXEDO session.

If you are manually writing a script, include the following preprocessor statement:

```
#include <VU.h>
```

If you are manually writing a script that accesses a TUXEDO application, include both `VU_tux.h` and `VU.h` as preprocessor statements:

```
#include <VU.h>
#include <VU_tux.h>
```

VU.h

The `VU.h` file includes definitions for:

- The EOF value returned by various VU functions.
- The file descriptors for the standard files.
- `ENV_VARS`, which lets you operate on the environment variables as a unit.
- The `HOURS`, `MINUTES`, and `SECONDS` macros, which enable you to specify time units other than milliseconds.
  - `HOURS(`*A*`)` returns the milliseconds in *A* hours.
  - `MINUTES(`*A*`)` returns the milliseconds in *A* minutes.
  - `SECONDS(`*A*`)` returns the milliseconds in *A* seconds.

  The value *A* must be an integer expression.
- All error codes (`_error`) that are not provided by the SQL database server.
- All options recognized by `sqlsetoption()`.

  Some constants defined in `VU.h` are vendor-specific. For example, the names of Oracle-specific values begin with `ORA_`; the names of Sybase-specific values begin with `SYB_`.

VU_tux.h

The VU_tux.h file includes definitions for symbolic constants and flag values used with TUXEDO emulation commands and functions.

sme/data.h

The sme/data.h file includes definitions for functions that come with Rational Suite PerformanceStudio. These functions let you parse data returned by sqlnrecv into rows and columns. Typically, this is useful in dynamic data correlation for SQL, where you extract data from queries and use that data in subsequent statements.

sme/file.h

The sme/file.h file includes definitions for functions that read a line of data from a file, parse the line that was read, and then reset the pointer to the next line of data, so that each emulated user can parse a line. Typically, this is useful as an alternative to datapools.

Preprocessor Features

LoadTest comes with the GNU C preprocessor. The preprocessor commands enable you to:

- Replace tokens.
- Include more than one source file in a script.
- Compile parts of a script.

Token Replacement

Token replacement and macro substitution can be specified with the #define preprocessor command. To indicate simple replacement throughout the entire script, use a command of the form:

```
#define orig_ident new_token
```

This replaces all occurrences of the identifier *orig_ident* with the token *new_token*.

To specify a macro definition with arguments, use a command of the form:

```
#define macro_name (arg1,arg2,...) macro_defn
```

Subsequent occurrences of *macro_name*(*var1, var2,* ...) are replaced by *macro_defn*, and occurrences of *arg1, arg2*,... inside the macro definition are replaced by the corresponding *varx*. To continue a definition on the next line, put a backslash (\) at the end of the line.

Example

This example substitutes `var1` for `x`, `var2` for `y`, and assigns `var3` the greater of `var1` and `var2`:

```
#define greater(x,y) (((x)>(y))?(x):(y))
#define lesser(x,y)  (((x)<(y))?(x):(y))
...
var3 = greater(var1,var2);
```

Creating a Script That Has More than One Source File

The `#include` preprocessor command lets you include another source file in your script at compile time. This command has two forms:

```
#include <filename>
#include "filename"
```

The first form looks only in a standard location for *filename*. The standard location is not specified in the VU language; it is the same set of directories used by the C preprocessor. The second form checks the current directory for *filename* before searching the standard location. In both cases, the contents of *filename* are inserted into the script at the point where the `#include` appears.

Compiling Parts of a Script

Conditional compilation commands allow you to conditionally compile parts of a script. There are three ways to specify conditional compilation:

- `#if-#else-#endif`
- `#ifdef-#else-#endif`
- `#ifndef-#else-#endif`

The first has the form:

```
#if const1
t_stmnt1
...
t_stmntn
#else
f_stmnt1
...
f_stmntm
#endif
```

where *const1* must be a constant (or an expression which has a value at compile time), and *t_stmnt1* through *t_stmntn* and *f_stmnt1* through *f_stmntm* are any VU statements or preprocessor commands. If the value of *const1* is nonzero, *t_stmnt1* through *t_stmntn* are compiled; otherwise: *f_stmnt1* through *f_stmntn* are compiled. You can omit the #else and *f_stmnt1* through *f_stmntn* if no compilation is desired when *const1* has the value 0.

The other two forms compile a portion of code if the token has been set through a #define or through LoadTest's **Tools** → **Options**. Click the **VU Compilation** tab and enter the name of the tokens under **Defines**. They are:

```
#ifdef  token1
d_stmnt1
...
d_stmntn
#else
n_stmnt1
...
n_stmntm
#endif
```

and

```
#ifndef token1
n_stmnt1
...
n_stmntn
#else
d_stmnt1
...
d_stmntm
#endif
```

token1 must be an identifier and *d_stmnt1* through *d_stmntn* and *n_stmnt1* through *n_stmntn* are any VU statements or preprocessor commands.

If the #ifdef format is used, *d_stmnt1* through *d_stmntn* are compiled if *token1* was defined; otherwise, *n_stmnt1* through *n_stmntm* are compiled.

If the #ifndef format is used, *n_stmnt1* through *n_stmntn* are compiled if *token1* has not been defined; otherwise, *d_stmnt1* through *d_stmntm* are compiled.

As in the #if command, you can omit the #else portion in either of these forms.

Defining Your Own Subroutines

The VU language lets you define the following kinds of subroutines:

- **Functions** – Subroutines that return a value through a `return` statement. You define functions with the `func` keyword.
- **Procedures** – Subroutines that do not return a value. You define procedures with the `proc` keyword.

Defining a Function

You can declare an integer function, which returns an integer value, or a string function, which returns a string value. An array function can return a value which is an array of integers or strings.

To define a function, use the following format:

```
[type] func fname [array_spec] (arg_list)
arg_declar;
{
    stmnt1;
    stmnt2;
    ...
    stmntn;
    return ret_exp;
}
```

You can define `type` as `int` or `string`. The default is `int`, so you can omit it if you are declaring an integer function.

`fname` is the name of the function you want to define.

`array_spec`, used only in array functions, is a list of integer constants that specify the size of the first, second, and third dimensions of the array. Each integer constant is enclosed in brackets. A one-dimensional array is [c1], a two-dimensional array is [c1] [c2], and a three-dimensional array is [c1] [c2] [c3].

`arg_list` lists the function's arguments. If the function has more than one argument, separate them by commas. If the function has no arguments, follow the name of the function with a pair of empty parentheses, such as `func1()`.

`arg_declar` is the declaration of the arguments. Arguments whose data type is not integer must be declared before the opening brace of the function.

`stmnt1`, *`stmnt2`*, *`stmntn`* are the VU language statements in the function. If the function contains only one statement, you can omit the braces.

A function must have at least one `return` statement. If a function has more than one `return` statement, only one is executed per call. The return is executed before the function completes execution.

ret_exp is an expression whose type matches the type of the function. If you have defined an array function, the number of dimensions of *ret_exp* must match the number of dimensions of the function. Use a null *ret_exp* (`return "";`) to return a null string from a string function.

The order and data type of the arguments in the function call must coincide with the order and data type of the arguments in the function definition. If they do not coincide, a compilation error results.

You might get a warning message if the number of arguments in the function call and function definition do not match. If you have extra arguments in the function definition, you are not able to reference them while the function is executing. If there are extra arguments in the function call, they are ignored.

The value returned by a function must match the type of the function. For example, the expression following the `return` must have an integer value if the function is an integer function and a string value if the function is a string function.

Calling a Function

To call a function, simply use the function name and the argument list:

```
fname (arg_list)
```

where *fname* specifies the name of the function, and *arg_list* lists the arguments of the function call.

Example

The following example defines a function with more than one `return` statement. The function, called `strcomp`, compares two strings:

```
func strcomp(str1, str2)
string str1, str2;
{
    if (str1 == str2)
        return 0;
    else if (str1 < str2)
        return -1;
    else
        return 1;
}
```

Defining a Procedure

To define a procedure, use the following format:

```
proc pname (arg_list)
arg_declar;
{
    stmnt1;
    stmnt2;
    ...
    stmntn;
)
```

pname is the name of the procedure you want to define.

arg_list is a list of the procedure's arguments. If the procedure has more than one argument, separate them by commas. If the procedure has no arguments, follow the name of the procedure with a pair of empty parentheses, such as `proced1()`.

arg_declar is the declaration of the arguments. Arguments whose data type is not integer must be declared before the opening brace of the procedure.

stmnt1, *stmnt2*, *stmntn* are the VU language statements in the procedure. If the procedure contains only one statement, you can omit the braces.

Although procedures do not return values, you can include the statement `return`; to return control to the caller.

Calling a Procedure

To call a procedure, simply use the procedure name and the argument list:

```
pname (arg_list)
```

Example

The following example defines the procedure `dis_time`, which displays the time and sounds a tone (ASCII 007). The procedure then returns control to the calling program:

```
proc dis_time(time_str)
string time_str;
{
    printf("At the tone%c, the time will be %s", '\007', time_str);
    return;
}
```

Accessing External C Data and Functions

The VU language supports access to external C data and functions. A VU script can call functions written in C and pass values as arguments to the C functions.

C functions can return values to VU scripts. External C objects are declared in VU using the keyword `external_C`.

VU integers are signed 32-bit integers. These are usually declared in C as `int` or `long` (this section refers to them as C type `int`). VU strings have no exact C counterpart but are accessed as C character pointers (`char *`). VU arrays are accessed in C as a pointer to a block of data of the appropriate type. Multidimensional arrays are passed as a pointer to a block of contiguous memory containing the data in row-major (normal C) order.

External C Variables

A C pointer can access a VU array of 1, 2, or 3 dimensions.

The following table shows the C data types that can be accessed by VU. Other data types are not supported and give unpredictable results.

C Variable Type	VU Variable Type
`int`	`int`
`char *`	`string /* read only */`
`char *`	`string:`*`maxsize`* `/* writable */`
`int *`	`int []`, `int[][]`, `int[][][]`
`char **`	`string []`, `string [][]`, `string [][][]`

An external C `char *`, or array of `char`, must be null terminated. VU interprets these as strings. VU does not perform memory management on external C strings or external C string arrays.

In a script an external C string is read-only *unless* its VU declaration includes its maximum size. The C code must allocate space for the string greater than or equal to *`maxsize`* bytes. The maximum size must include room for the C null-terminator character `'\0'`; it is specified with a colon and an integer constant, as in:

```
external_C string:81 extc_line;
```

Space for the string might be declared in C as:

```
char extc_space[81];
char *extc_line = extc_space;
```

In the preceding example, VU could write up to 80 characters to `extc_line`. An attempt to write more than 80 characters causes a runtime error.

VU declarations of C variables that are pointers to `int` or `char *` must be declared as VU arrays with a fixed size and must have no more than 3 dimensions. The data pointed to by the C variable is interpreted as a VU array of the declared type. VU does not perform memory management on the C pointers.

External C data cannot be declared `persistent` or `shared`. Values of external C variables persist for the duration of the run.

Declaring External C Subroutines

An external C subroutine is declared the same way as a VU function or procedure, with an empty statement block for the body.

The following VU declarations:

```
external_C func foo(i, s)
string s;
{}
external_C proc bar(limit, ia)
int limit;
int ia[];
{}
external_C int func[10][20] afunc()
{}
```

are used for the C functions whose prototypes are:

```
int foo(int, char *);
void bar(int, int *);
int *afunc(void);
```

The VU compiler performs type and number checking for argument variables between their declaration and their use.

An external C function is called in the same way that a VU function or procedure is called. Any VU data type can be passed to an external C subroutine.

Accessing Values Returned from C Functions

A C function returns a pointer accessed as a VU array of 1, 2, or 3 dimensions.

The following table shows the only C data types that can be returned from an external C function. Other data types are explicitly not supported, and give unpredictable results.

C Return Type	VU Return Type
`void`	`proc`
`int`	`int func`
`char *`	`string func`
`int *`	`int func[]`, `int func[][]`, `int func[][][]`
`char **`	`string func[]`, `string func[][]`, `string func[][][]`

A `char *` returned by a C function must point to a null terminated block of characters. VU interprets this as a string and does not attempt to perform memory management on strings returned from C functions.

VU declarations of C functions that return pointers to `int` or `char *` must be declared as VU functions that return arrays with a fixed size, and have no more than three dimensions. The data pointed to by the actual return value is interpreted as a VU array of the declared type. VU does not attempt to perform any memory management on the returned pointers.

Passing Arguments to External C Functions

Arguments are passed to external C functions by value or by reference. The default is to pass arguments by value. Arguments declared with the keyword `reference` are passed by reference (address). Reference arguments are passed as pointers to the appropriate types. Arrays are always passed as a pointer to a block of data of the appropriate type. Arguments declared `reference` are passed with the & operator, allowing the VU compiler to type-check the arguments.

Arrays are always passed by reference; you should not use the `reference` keyword and the & operator with array arguments.

When passing VU arguments to external C functions, the data type of the corresponding C argument must match this list. Other data types are not supported, and yield unpredictable results.

The following table shows how VU arguments are passed:

VU Data Type	Is Passed as C Data Type
`int`	`int`
`string`	`char *`
`reference int`	`s32 *`
`reference string`	`char **`
`int [ ]`	`s32 *`
`string [ ]`	`char **`
`int [ ][ ]`	`s32 *`
`string [ ][ ]`	`char **`
`int [ ][ ][ ]`	`s32 *`
`string [ ][ ][ ]`	`char **`

Integers

Integer arguments behave exactly as in C, except for integer arrays.

Strings

The nearest equivalent C type to a VU string is a `char *`.

A nonreference string argument is passed as a pointer to a copy of the null-terminated string data. The external C function can locally change characters in this copy, but these changes do not affect the original string value upon return to the VU script. In addition, the external C function must not attempt to modify storage beyond the end of the string, including the null terminator.

A reference string argument allows the C function to change the VU string's characters and also to reassign the actual pointer. If you want the external C function to modify the contents of the VU string, you must pass the string by reference. You must also pass a string by reference if the C function reassigns the string's pointer in order to cause a VU string to become longer. For more information, see *Memory Management of VU Data* on page 4-13.

An array of strings is passed as a pointer to a block of character pointers.

Arrays

An array is passed as a pointer to a block of data of the appropriate type (`int`, `char *`) just as C programmers expect to pass arrays.

A multidimensional array is passed as a pointer to a block of contiguous memory containing the data in row-major (normal C) order.

Memory Management of VU Data

Data created in VU is “owned” by VU. VU performs memory management on all of its data.

Strings that VU creates point to `malloc`’ed data, and VU can free them at any time. C functions that use VU strings as arguments must not save the value of a VU string in static (global) C variables, or unpredictable results occur. In addition, a C function modifying a reference argument originating from a string created by VU should free or reallocate the original pointer, and the new value must be the result of a call to `realloc` or `malloc`.

The same is true for pointers to VU array data. The storage is managed by VU, and C functions must not save the values of such pointers in static variables. The elements of a VU array are essentially passed by reference, and may be treated as such. String array elements may be treated as reference strings.

All VU variables and scalar array elements are created in an undefined state and have no value. When passed to C functions as reference arguments, these values are converted to default values. Undefined strings are passed as NULL, integers as 0. Upon return from the C function, strings with value NULL are again considered undefined. Upon return from the C function, *all* integers are considered defined. If the C function did not assign a value to such an argument, it retains the default value of 0.

Memory Management of C Data

Data created in C modules, and all pointer values returned from C functions or external C variables, are “owned” by C. VU does not perform any memory management on this data — all memory management must be performed by C modules.

Specifying External C Libraries

You can specify external C libraries for use by all VU scripts in a PerformanceStudio project. In LoadTest, select **Tools** → **Options**, and then click the **VU Compilation** tab. Under **External C Libraries**, select the libraries you want to add and click **>** .

To make a library available to a particular script, modify the script properties for that script. You can modify script properties using TestManager, Robot, or LoadTest. In LoadTest, open a schedule that includes the script, right-click on the script, and then select **Script Properties** from the menu. Click the **VU Compilation** tab. Under **External C Libraries**, click **Add,** and then enter the name of the library you want to add.

It is recommended that you enter the name of the library without the `.DLL` extension. This way the script can be run on UNIX Agent computers by posting the library to the Agent.

Creating a Dynamic-Link Library on Windows NT

To access C code and data from a VU script, compile the C code into a dynamic-link library (DLL).

NOTE: On Windows NT systems, in order for VU scripts to access data items defined in .DLLs, you must provide a function that returns the address of the data item. The function must be named the same as the data item with `addr_` added to the beginning of the function name.

There are three steps involved in creating a DLL:

1. Write and compile the C source code to be called from your VU script.
2. Examine the VU script, and note which functions and variables the script needs to access.
3. Create the DLL, and export the necessary symbols.

The following are the general steps you take to create the external library file `c_prog` and make it available to a script:

1. Write `c_prog.c`, which contains code that you want to call from your script, `script.s`. Invoke the Microsoft C compiler to compile `c_prog.c` and produce `c_prog.obj`:

   ```
   cl /c c_prog.c
   ```

2. Examine your VU script `script.s`. The example script on page 4-16 uses external C notation to indicate that the symbols `s_func`, `afunc`, and `addr_message` are defined in a C module.

3. Issue the `link` command to create a DLL and export the external C symbols. The following command produces `c_prog.lib`, `c_prog.exp` and `c_prog.dll`, and exports `s_func`, `afunc`, and `addr_message`:

   ```
   link c_prog.obj /dll /export:s_func /export:afunc
   /export:addr_message
   ```

4. Once you have created the DLL, copy it to each project that needs to access it. The directory location is:

   `Project\project_name...\Script\externC`

For more detailed information on creating a DLL, consult the documentation for a Microsoft C development tool such as Microsoft Visual Studio.

Creating a Shared Library on UNIX

To access C code and data from a VU script, compile the C code into a shared library or shared object. C source (`.c`) files are compiled into object (`.o`) files by **cc**(1), then one or more object files are combined into a shared library (`.so`) by **ld**(1). The **cc** and **ld** options are system-dependent; see **cc**(1) and **ld**(1) for more information. The following example shows how to compile a program and create a shared library:

```
$ cc -Kpic -O -c foo.c
$ cc -Kpic -O -c bar.c
$ ld -dy -G -Bsymbolic foo.o bar.o -o foo.so -lc
$
```

Or, equivalently (on most systems),

```
$ cc -KPIC -O -dy -G -Bsymbolic foo.c bar.c -o foo.so -lc
$
```

The `-c` option specifies that **cc** generates an `.o` file, and the `-KPIC` option requests position-independent code. The `-dy` option of **ld** specifies dynamic linking; the `-G` option specifies that **ld** should produce a shared object; the `-Bsymbolic` option binds references to global symbols to their definitions within the object; and the `-lc` option is needed in conjunction with the `-Bsymbolic` option to resolve references to the C library.

Once you have created the shared library, copy it to each UNIX Agent that needs to access it. The default directory location is `/tmp/externC`. You can change the directory through Load Test. Open a schedule, click the **Computers** button, and change the **Local Directory** name. You must create an `externC` subdirectory under the local directory name.

Libraries can be shared only across the same UNIX operating system vendor's agents. You must create a shared library version for each distinct UNIX operating system type.

Examples

NOTE: DLLs on Windows NT systems cannot print directly to the virtual user's `stdout` or `stderr` files. Therefore, the following script produces different output on UNIX Agents than on Windows NT Agents.

C module: `lib/c_script.c`

```
# include <stdlib.h>
static int table[10][20];
char msg_data[100];
char *message = msg_data;
char **addr_message()
{
return &message;
}
int foo(int i, char **s)
{
*s = *s? realloc(*s, 18): malloc(18);
strcpy(*s, "hello from C land");
return 10 * i;
}

void bar(int max, int *a)
{
int i;
printf("message in bar(): [%s]\n", message);
for (i = 0; i <= max; i++)
    a[i] = i;
}

char *s_func(char *s)
{
printf("C output: [%s]\n", s);
return "s_func return value";
}

int *afunc(void)
{
return &(table[0][0]);
}
```

VU module: `script.s`

```
external_C string:100 message;
external_C func foo(i, s)
reference string s;
{}

external_C proc bar(limit, ia)
int limit;
int ia[];
{}

external_C int func[10][20] afunc()
{}

external_C string func s_func(s)
string s;
{}
```

```
string vs, s;
int ary[10][100];

{
vs = "hello world";
s = s_func(vs);

message = s + ", this is a test.";

ary = afunc();

foo_res = foo(5, &vs);
printf("result of foo: %d\n", foo_res);
printf("message = [%s]\n", message);

size = limitof ary[5];
bar(size, ary[5]);

for (i = 0; i <= size; i++)
printf("ary[5][%d] = %d\n", i, ary[5][i]);
}
```

Create the shared library:

```
$ cd lib
$ cc -KPIC -O -dy -G -Bsymbolic c_script.c -o c_script.so -lc
$ cd ..
```

Run the schedule.

Contents of user output on UNIX Agents:

```
C output: [hello world]
result of foo: 50
message = [s_func return value, this is a test.]
message in bar(): [hello world, this is a test.]
ary[5][0] = 0
ary[5][1] = 1
ary[5][2] = 2
ary[5][3] = 3
ary[5][4] = 4
ary[5][5] = 5
ary[5][6] = 6
ary[5][7] = 7
ary[5][8] = 8
ary[5][9] = 9
ary[5][10] = 10
ary[5][11] = 11
ary[5][12] = 12
ary[5][13] = 13
ary[5][14] = 14
ary[5][15] = 15
ary[5][16] = 16
ary[5][17] = 17
ary[5][18] = 18
ary[5][19] = 19
```

Contents of user output on NT Agents:

```
result of foo: 50
message = [s_func return value, this is a test.]
ary[5][0] = 0
ary[5][1] = 1
ary[5][2] = 2
ary[5][3] = 3
ary[5][4] = 4
ary[5][5] = 5
ary[5][6] = 6
ary[5][7] = 7
ary[5][8] = 8
ary[5][9] = 9
ary[5][10] = 10
ary[5][11] = 11
ary[5][12] = 12
ary[5][13] = 13
ary[5][14] = 14
ary[5][15] = 15
ary[5][16] = 16
ary[5][17] = 17
ary[5][18] = 18
ary[5][19] = 19
```

▸▸▸ C H A P T E R 5

User Emulation

In addition to its C-like features, VU provides features designed to emulate actual users running client applications and sending requests to a server. This chapter describes these features in the following topics:

- Emulation commands
- Emulation functions
- VU environment variables
- Read-only variables
- Supplying a script with meaningful data

Emulation Commands

An emulation command causes a script to communicate with a server in the same manner as an actual client application. The communication may involve sending one or more requests to a server, or involve receiving and evaluating responses from it. The specific communication is recorded during script creation, or possibly added during subsequent script editing. An example is to query for a row of data. Emulation commands consist of:

- HTTP emulation commands for emulating HTTP clients.
- SQL emulation commands for emulating database client applications.
- TUXEDO emulation commands for emulating TUXEDO clients.
- IIOP emulation commands for emulating IIOP clients that communicate with CORBA components.
- Socket emulation commands for emulating raw socket clients.

There are a number of emulation commands that you can add to your scripts. These provide measurement timers, customization of test cases, and inclusion of external C. These emulation commands are independent of the particular type of client (for example, HTTP, SQL, or socket), so the same additions can be used for all script types. The commands are as follows:

- The `start_time` and `stop_time` commands. You can insert these commands during recording through the Robot **Insert** menu. With these commands, you can time a block of user actions, typically for a single user level transaction.
- The `testcase` command. This command lets you customize your own test cases. For example, you can check a response for specific results and have the success or failure logged in the LoadTest report output.
- The `emulate` command. This command lets you use external C linkage to support a proprietary protocol or interface. You can wrap VU or external C function calls with the `emulate` command, and thus obtain the full set of services normally associated with the standard emulation commands, including time stamping and reporting on success or failure.

Emulation commands that succeed return a value of 1 or greater. Emulation commands that fail return a value of 0 or less.

HTTP Emulation Commands

If you have recorded Web traffic, your resulting script will contain VU emulation commands and functions pertaining to HTTP. These commands and functions have the prefix `http`.

In general, you will not have to alter an HTTP script extensively; it should typically run without errors.

HTTP Commands that You Insert Manually

LoadTest also provides HTTP emulation commands and functions that you can insert manually into your script. These are:

- `http_header_info`. This function lets you retrieve the values of the header information. For example, you can retrieve the content length of the page or when the page was last modified.
- `http_recv`. This command enables the script to receive data until a specified string appears in the data. At the end of the specified string, the script stops reading data.

Monitoring Computer Resources

To monitor computer resources for HTTP servers, you must add an `INFO SERVER` declaration for that computer in at least one VU script in the schedule.

The syntax for this statement is as follows:

```
INFO SERVER label=addr [, label=addr]
```

`label` is a string that gives the logical name of the server. This is the name you see associated with the resource data in LoadTest reports and graphs.

`addr` is a string that gives the network name or IP address of the Web server.

Although you can add this line in the script anywhere you can declare a VU variable, you should generally add it at the start of the script (after the opening brace) or immediately before the first `http_request` that communicates to that server. If you add it before the first `http_request`, enclose the `INFO SERVER` declaration in braces.

You need to add a declaration for each different HTTP server you want to monitor. You can declare the same `INFO SERVER` in different scripts; however the definitions must be consistent for all scripts included in a LoadTest schedule. There is no requirement that the `INFO SERVER` declaration occur in each HTTP script, or for that matter in an HTTP script at all (as long as it occurs in at least one VU script included in the schedule). In fact, you could create a special "servers" script just for this purpose, and assign that "declaration-only" script to any (or all) user groups in the schedule. However, the advantage of putting the appropriate `INFO SERVER` declarations in each HTTP script is that less maintenance is involved when creating schedules since you don't have to be concerned with which scripts access which HTTP servers.

Example

The following example shows a portion of an HTTP script, with comments and two `INFO SERVER` declarations added. One `INFO SERVER` declaration is at the start of the script and one is before the first `http_request` (enclosed in braces).

Each server makes two requests — one for each page of data received. Only the first request contains the connection parameters. The second request uses the existing connection specified by the `Server_connection` environment variable.

```
{
INFO SERVER "CAPRICORN_WEB" = "capricorn-web";
CAPRICORN_WEB_80 = http_request "CAPRICORN-WEB:80", "",
HTTP_CONN_DIRECT,
    "GET / HTTP/1.0\r\n"
    "Accept: application/vnd.ms-excel, application/mswo"
    "rd, application/vnd.ms-powerpoint, image/gif, imag"
    "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
    "Accept-Language: en\r\n"
    "UA-pixels: 1152x864\r\n"
    "UA-color: color8\r\n"
    "UA-OS: Windows NT\r\n"
    "UA-CPU: x86\r\n"
    "User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)\r\n"
    "Host: capricorn-web\r\n" "Connection: Keep-Alive\r\n\r\n";
set Server_connection = CAPRICORN_WEB_80;
http_header_recv 200;/* OK */
/* more data (4853) than expected >> 100 % */
http_nrecv 100 %% ; /* 4853/4051 bytes */
http_disconnect(CAPRICORN_WEB_80);
{
INFO SERVER "GEMINI_WEB" = "gemini-web";
}
GEMINI_WEB_80 = http_request "GEMINI-WEB:80", "",
HTTP_CONN_DIRECT,
    "GET / HTTP/1.0\r\n"
    "Accept: application/vnd.ms-excel, application/mswo"
    "rd, application/vnd.ms-powerpoint, image/gif, imag"
    "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
    "Accept-Language: en\r\n"
    "UA-pixels: 1152x864\r\n"
    "UA-color: color8\r\n"
    "UA-OS: Windows NT\r\n"
    "UA-CPU: x86\r\n"
    "User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)\r\n"
    "Host: capricorn-web\r\n" "Connection: Keep-Alive\r\n\r\n";
set Server_connection = GEMINI_WEB_80;
http_header_recv 200;/* OK */
/* more data (4853) than expected >> 100 % */
http_nrecv 100 %% ; /* 4853/4051 bytes */
http_disconnect(GEMINI_WEB_80);
}
```

SQL Emulation Commands

If you have recorded a SQL application, your resulting script contains VU emulation commands and functions pertaining to SQL. These commands and functions have the prefix `sql`.

A script that simply reads records will probably play back without errors. However, if you read the same record from the database over and over, your script technically "works," but may not reflect a realistic workload. This is because the database will cache the record, which may or may not be desirable, depending on whether or not cached records reflect the workload you are emulating.

You probably need to alter a script that inserts records in or deletes records from a database before it plays back as intended. This is because if one virtual user deletes a record and does not restore the database, the second virtual user's delete fails because the record is already deleted.

Processing Data from SQL Queries

The `sqlnrecv` command reads the data returned from the database, but it does not parse it into rows and columns. The following VU toolkit functions, which come with Rational Suite PerformanceStudio, enable you parse data returned by `sqlnrecv` into rows and columns.

- `proc SaveData(`*`data_name`*`)`
- `proc AppendData(`*`data_name`*`)`
- `proc FreeData(`*`data_name`*`)`
- `proc FreeAllData()`
- `string func GetData(`*`data_name`*`, `*`row`*`, `*`column`*`)`
- `string func GetData1(`*`data_name`*`, `*`column`*`)`

`SaveData` stores the data returned by the most recent sqlnrecv command, tagging it with the value of the *`data_name`* argument.

`AppendData` adds data to an existing named data set. `FreeData` and `FreeAllData` release the data and associated storage for the named set of data or for all sets of data respectively. `GetData` retrieves the specified row and column from the data associated with *`data_name`*.

`GetData1` is similar to `GetData`, but `GetData1` always retrieves the specified column from the first data row.

SQL Error Conditions

SQL emulation commands return a value of `>=1` if execution was normal, or `<=0` if an error occurred (that is, `Timeout_val` expired or `_error` has a nonzero value). SQL emulation commands set `_error` and `_error_text` to indicate the status of the emulated SQL statements. If `_error` has a nonzero value and `Log_level` is set to `"ALL"` or `"ERROR,"` the log file entry indicates that the command failed, and the values of `_error` and `_error_text` are logged.

You can also set the SQL emulation commands to “expect” certain errors. The `EXPECT_ERROR` clause causes the emulation command to “pass” (match the expected response) if the expected error occurs. Conversely, if the SQL statement produces no error, but an error is expected, the emulation command “fails” (does not match the expected response), and is logged and recorded accordingly.

VU Toolkit Functions: File I/O

A common task in performance testing is to read a set of data from a file, parse the line read, and then use the fields of data as send parameters. The VU toolbar functions provide a set of routines and variables to implement this process, and include the capability of processing comments in the input file. The variables are:

- `string` *`Last_Line`*
- `string` *`Last_Field`*
- `string` *`Last_Subfield`*

These contain the most recently read line, field, and subfield as produced by the following functions:

- `func ReadLine(`*`file_descriptor`*`)`
- `string func NextField()`
- `string func IndexedField(`*`index`*`)`
- `string func NextSubField()`
- `string func IndexedSubField(`*`index`*`)`
- `SHARED_READ(`*`filename, prefix`*`)`

The `ReadLine` function reads a line from the currently open file designated by *`file descriptor`*. The function has many options to define comment lines, field delimiters, and end-of-file behavior.

The `NextField` function parses the line read by `ReadLine`. Each successive call returns the next field on the line. The variable `Last_Field` contains the string returned by the most recent call to this function.

The `IndexedField` function parses the line read by `ReadLine` and returns the field indicated by the *index* argument. A call to `IndexedField` resets the field pointer so that a subsequent call to `NextField` returns the field following the index. The variable `Last_Field` contains the string returned by the most recent call to this function.

The `NextSubField` function parses the field returned by the most recent call to `NextField` or `IndexedField`. Each successive call returns the next subfield within the field. The variable `Last_Subfield` contains the string returned by the most recent call to this function.

The `IndexedSubField` function parses the field returned by the most recent call to `NextField` or `IndexedField`, returning the subfield indicated by *index*. A call to `IndexedSubField` resets the field pointer so that a subsequent call to `NextField` returns the field following the index. The variable `Last_Subfield` contains the string returned by the most recent call to this function.

SHARED_READ allows multiple users to share *filename*, so that no two users read the same line. It depends on two externally defined shared variables named *prefix*_lock and *prefix*_offset.

TUXEDO Emulation Commands

If you recorded a TUXEDO application, your resulting script contains VU emulation commands and functions pertaining to TUXEDO.

The names for VU emulation commands follow the names of the TUXEDO API calls, but they have the preface `tux_`. So, for example, the VU emulation command `tux_tpacall` corresponds to the TUXEDO API call `tpacall`.

There are two basic types of commands:

- Commands that return a pass/fail indicator. These commands return 1 (logical true) if the commands succeeds, and 0 (logical false) if it fails.
- The commands that return a value that other commands use later. If these commands fail, they return -1.

How VU Represents TUXEDO Pointers

Some TUXEDO API calls use pointers. However, pointers are not supported in the VU language. Therefore, the VU language uses *free buffers* to represent pointers.

A free buffer can be **simple**, representing a single buffer member, or **composite**, containing many individually named buffer members. Within VU and TUXEDO, free buffers can represent simple data types, such as pass-by-reference long integers, as well as composite data types, such as nested C structures and TUXEDO typed buffers.

Since simple buffers have no members, you should use an empty string ("") whenever a simple buffer member name is required.

For composite buffers, use the following syntax to specify a member:

```
name [ "." name [ "." name ] ... ] [ ":" instance ]
```

where *name* is the name string given to the member, and *instance* is an integer value representing the cardinal occurrence of a multiply defined member name. Instance numbers begin with zero.

The following example loads the "msgid" string of the "qctl" member of a BUFTYP_TPEVCTL buffer for tux_tpsubscribe:

```
tpevctl = tux_allocbuf(BUFTYP_TPEVCTL);
tux_setbuf_string(tpevctl, "qctl.msgid", "somevalue");
...
```

The following example loads the fourth occurrence of the field named "QUANTITY" (converting value to an integer) from an FML buffer named odata_ populated by tux_tpcall:

```
quantity = tux_getbuf_int(odata_, "QUANTITY:3");
```

With FML buffers, omitting *instance* implies the first occurrence of that member name. For example, "QUANTITY:0" and "QUANTITY" are equivalent.

The free buffer types, their member names, and the corresponding VU data types are as follows:

Buffer Type/Member Names	VU Data Type Equivalent
BUFTYP_CARRAY	string (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
BUFTYP_CLIENTID "clientdata0" "clientdata1" "clientdata2" "clientdata3"	(composite) int int int int
BUFTYP_FML User-defined field names and values	(composite)
BUFTYP_FML32 User-defined field names and values	(composite)
BUFTYP_REVENT	int
BUFTYP_STRING	string (user-defined maximum length)

(Continued)

Buffer Type/Member Names	**VU Data Type Equivalent**
BUFTYP_SUBTYPE	string (maximum length = 15)
BUFTYP_TPEVCTL	(composite)
"flags"	int
"name1"	string (maximum length = 31)
"name2"	string (maximum length = 31)
"qctl"	string. Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
"qctl.flags"	int
"qctl.deq_time"	int
"qctl.priority"	int
"qctl.diagnostic"	int
"qctl.msgid"	string (maximum length = 31)
"qctl.corrid"	string (maximum length = 31)
"qctl.replyqueue"	string (maximum length = 15)
"qctl.failurequeue"	string (maximum length = 15)
"qctl.cltid"	string. Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
"qctl.cltid.clientdata0"	int
"qctl.cltid.clientdata1"	int
"qctl.cltid.clientdata2"	int
"qctl.cltid.clientdata3"	int
"qctl.urcode"	int
"qctl.appkey"	int
BUFTYP_TPINIT	(composite)
"usrname"	string (maximum length = 30)
"cltname"	string (maximum length = 30)
"passwd"	string (maximum length = 30)
"grpname"	string (maximum length = 30)
"flags"	int
"datalen"	int
"data"	string (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.

(Continued)

Buffer Type/Member Names	VU Data Type Equivalent
BUFTYP_TPQCTL "flags" "deq_time" "priority" "diagnostic" "msgid" "corrid" "replyqueue" "failurequeue" "cltid" "cltid.clientdata0" "cltid.clientdata1" "cltid.clientdata2" "cltid.clientdata3" "urcode" "appkey"	(composite) int int int int string (maximum length = 31) string (maximum length = 31) string (maximum length = 15) string (maximum length = 15) string. Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters. int int int int int int
BUFTYP_TPTRANID "info0" "info1" "info2" "info3" "info4" "info5"	(composite) int int int int int int
BUFTYP_TYPE	string (maximum length = 7)
BUFTYP_VIEW	string (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
BUFTYP_VIEW32	string (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
BUFTYP_X_C_COMMON	string (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.

(Continued)

Buffer Type/Member Names	VU Data Type Equivalent
BUFTYP_X_C_TYPE	`string` (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.
BUFTYP_X_OCTET	`string` (user-defined maximum length). Nonprintable characters are converted to hexadecimal strings delimited by grave accent characters.

Free buffers are allocated with the `tux_allocbuf` and `tux_allocbuf_typed` functions, which return a buffer handle that can be used to reference the allocation by other API calls. Once a free buffer is no longer needed, deallocate it with the `tux_freebuf` function. Functions for loading, unloading, resizing, and describing buffers and buffer members also are available.

TUXEDO Error Conditions

Error conditions differ slightly between TUXEDO and the VU language. Consistent with the VU language, TUXEDO emulation commands set the `_error` and `_error_text` read-only variables. They also set `_error_type`, a variable used only with TUXEDO. Although you need to check the value of `_error` or the return value to determine whether an error occurred, you should then check the `_error_type`, which indicates how to interpret the value in `_error`. For example, `_error_type` tells you if the value in `_error` is a TUXEDO system error code or an FML error code. To see the actual message, you read `_error_text`, just as with any other VU emulation command.

Four VU emulation commands (`tux_tpcall`, `tux_tpgetrply`, `tux_tprecv`, and `tux_tpsend`) update the read-only variable `_tux_tpurcode`. This variable contains the same information as the TUXEDO global variable `tpurcode`, and will help diagnose playback errors related to a failure in the server.

IIOP Emulation Commands

This section describes how the VU language emulates Internet Inter-ORB Protocol (IIOP) activity. VU's IIOP emulation commands and functions currently support the CORBA model.

Interfaces, Interface Implementations and Operations

CORBA (Common Object Request Broker Architecture) defines an architecture for remote method invocation between distributed objects. The methods of an object in the CORBA model are exposed to other objects via its IDL interface definition, or interface. Once a reference to an object is obtained, operations (methods) may be invoked on that object. Remote invocation occurs via IIOP request messages.

Within this section the terms **object** and interface implementation may be used interchangeably. Likewise the terms method and operation are equivalent. However, VU/IIOP is concerned only with the CORBA/IIOP interface model and not the larger CORBA object model. Therefore object model terminology is only used when it serves to clarify a subject.

Request Contexts and Result Sets

Within VU/IIOP, every operation invocation is associated with a request context that encapsulates all of the information required to perform the operation. This includes all of the information needed to construct an IIOP Request message (object key, operation name, parameters, service context, requesting principal, and so on) as well as the information required to retrieve the response (request ID, and so on).

The operation's response data, known as the result set, is also encapsulated within its associated request context. This includes any operation out parameters, the return value and any exception information that may have been returned in the response.

Therefore all interactions with an interface implementation are done through a request context. VU/IIOP implements request contexts via Request pseudo-objects.

VU/IIOP Pseudo-Objects

VU uses a number of abstract data types to represent collections of data that cannot be represented by the native VU language scalars (such as ints and strings). These types, called **pseudo-objects**, are referenced by their pseudo-object handles.

Handles are integer values that uniquely identify pseudo-objects and their associated variables.

Two pseudo-objects supporting IIOP messaging are:

- Object Reference
- Request

Object Reference Pseudo-Objects

An Object Reference pseudo-object represents a reference to an interface implementation that implements the operations of a specific interface. Once an interface specification is bound to an active interface implementation by the `iiop_bind` emulation command, a pseudo-object representing this binding is created and assigned a unique handle. The handle may then be used by the emulation commands to send operation requests to the interface implementation.

When an interface binding is no longer needed, that Object Reference pseudo-object may then be released by the `iiop_release` emulation function. Once released, the binding to the object implementation is destroyed.

Request Pseudo-Objects

A Request pseudo-object represents an active request context. They are created by the `iiop_invoke` emulation command.

Once created a Request pseudo-object persists until it is explicitly destroyed by a call to `iiop_release`, after which all request context information associated with that pseudo-object is destroyed and its handle becomes invalid.

Parameter Expressions

A parameter expression is a string expression used to specify the names, input values and output binding variables for an operation's argument list and corresponding result set members (collectively known as the operation's parameters). Parameter expressions are used by all emulation commands that invoke operations on an interface implementation.

The syntax for a parameter expression is:

```
parameter-name-expr ":" [input-bind-expr] [":" &output-bind-var]
```

where

`parameter-name-expr` is a string naming the parameter to be bound.

`input-bind-expr` is an optional VU language expression specifying the input value to the named parameter, which must be an IDL "in" or "inout" parameter.

`output-bind-var` is an optional VU variable that will contain the output value of the named parameter, which must be an IDL "inout" or "out" parameter.

Parameter Name Expressions

Parameters that represent single data values are known as **scalar parameters**. Parameters that represent data structures containing multiple data values are known as **aggregate parameters**. VU/IIOP can address any IDL basic data type, or any IDL basic data type member of any IDL constructed data type, used as a scalar or aggregate operation argument, result value or exception when identified with a parameter name expression.

The parameter name expression form for a scalar operation argument or exception member is simply:

`parameter-name`

where *`parameter-name`* is the IDL operation argument or exception member name. The name for an operation result value is the empty string (`""`).

There are four aggregate IDL constructed data types: struct, union, array, and sequence. The expression form for identifying an aggregate parameter's member is:

`member-expr[`*`member-expr`*`...]`

where *`member-expr`* has four possible forms:

- For IDL basic types the form is:

  `member-name`

  where *`member-name`* is the name of the member, which may be the name of the parameter if it is the topmost node.

- For struct types the form for identifying struct members is:

  `struct-name``"."`*`member-expr`*

  where *`struct-name`* is the name of the struct, which may be the name of the parameter if it is the topmost node or the name of a member if it is embedded.

- For union types the expression form for identifying union members is:

  `union-name``":"`*`discriminator-value`*`"."`*`member-expr`*

  where *`union-name`* is the name of the union, which may be the name of the parameter if it is the topmost node or the name of a member if it is embedded, and *`discriminator-value`* is the value of the IDL union *`switch_type_spec`* for the member being referenced.

- For array and sequence types the member expression form for identifying array and sequence members is:

  `member-expr``"["`*`element-id`*`"]"`

  where *`element-id`* is an integer identifying the ordinal position of the member within the array or sequence, starting at 0.

Interface Definition Language (IDL)

You must provide access to the IDL for your application to LoadTest. The IDL for an application usually consists of several files with a .idl extension. These files describe the operations and parameters that the objects of your application support. Developers can create the IDL manually using a text editor. The IDL can also be generated from a modeling tool such as Rational Rose.

Without access to the IDL, LoadTest can create only opaque scripts. An opaque script shows the names of the operations, but it does not show parameter names. For example, the command below specifies that the deposit operation is to be invoked, but it does so opaquely:

```
iiop_invoke ["deposit"] "deposit", objref_2,
    "IIOP_RETURN" : : &iiop_return,
    "*" : "‘010000007d000000‘";
```

If you load the IDL by clicking **Tools** → **Interfaces** in Robot, before recording a script, Robot will create more meaningful scripts. The following is an example of an operation created with an IDL available:

```
iiop_invoke ["deposit"] "deposit", objref_2,
    "account_number" : "2938845",
    "amount" : "125";
```

If explicit path information is not provided within `#include` directives in IDL files, not all IDL may be loaded. To ensure that all IDL files are loaded, create a user environment variable called `IDLINCLUDE`. Set `IDLINCLUDE` to the path for IDL files accessed by `#include`. For example:

```
d:\idl; d:\sysidl
```

Exceptions and Errors

Any operation may return an exception instead of its normal result set.

Error reporting takes advantage of the three error-related VU read-only variables: `_error`, `_error_type` and `_error_text`:

`_error` contains the status code of the most recent VU/IIOP emulation command. If the command completes successfully, `_error` is set to `IIOP_OK`. If the command fails, `_error` contains a value greater than 0.The exact interpretation of _error is then determined by the value of `_error_type`. `_error_text` contains a textual definition of a non-zero `_error` code.

The VU language recognizes three types of errors:

- server-reported CORBA system exceptions.

  CORBA defines a set of standard exception definitions used by ORBs to report system-level error events.

- server-reported CORBA user exceptions.
- LoadTest-reported errors. These errors are in the `_error` read-only variable,.

  LoadTest reports error conditions that do not fall under the classification of CORBA exceptions.

The following table lists the server-reported CORBA system exceptions.

if _error_type is 1 and _error is	then _error_text is
1 IIOP_BAD_PARAM	an invalid parameter was passed
2 IIOP_NO_MEMORY	dynamic memory allocation failure
3 IIOP_IMP_LIMIT	violated implementation limit
4 IIOP_COMM_FAILURE	communication failure
5 IIOP_INV_OBJREF	invalid object reference
6 IIOP_NO_PERMISSION	no permission for attempted operation
7 IIOP_INTERNAL	ORB Internal error
8 IIOP_MARSHAL	error marshalling parameter/result
9 IIOP_INITIALIZE	ORB initialization failure
10 IIOP_NO_IMPLEMENT	operation implementation unavailable
11 IIOP_BAD_TYPECODE1	bad typecode
12 IIOP_BAD_OPERATION	invalid operation
13 IIOP_NO_RESOURCES	insufficient resources for request
14 IIOP_NO_RESPONSE	response to request not yet available
15 IIOP_PERSIST_STORE	persistent storage failure
16 IIOP_BAD_INV_ORDER	routine invocations out of order
17 IIOP_TRANSIENT	transient failure, reissue request
18 IIOP_FREE_MEM	cannot free memory
19 IIOP_INV_IDENT	invalid identifier syntax

(Continued)

if _error_type is 1 and _error is	then _error_text is
20 IIOP_INV_FLAG	invalid flag was specified
21 IIOP_INTF_REPOS	error accessing interface repository
22 IIOP_BAD_CONTEXT	error processing context object
23 IIOP_OBJ_ADAPTER	failure detected by object adapter
24 IIOP_DATA_CONVERSION	data conversion error
25 IIOP_OBJECT_NOT_EXIST	nonexistent object, delete reference
26 IIOP_TRANSACTION_REQUIRED	transaction required
27 IIOP_TRANSACTION_ROLLEDBACK	transaction rolled back
28 IIOP_INVALID_TRANSACTION	invalid transaction
29 IIOP_UNKNOWN	unknown exception

The following table lists the server-reported CORBA user exceptions:

if _error_type is 2 and _error is	then _error_text is
1 IIOP_USER_EXCEPTION	user exception

The following table lists the LoadTest-reported errors:

if _error_type is 3 and _error is	then _error_text is
1 IIOP_TIMEOUT	command timed out
2 IIOP_BINDFAIL	unable to bind with any modus
3 IIOP_OP_UNKNOWN	operation not found in IDL information

Socket Emulation Commands

If you have recorded an unsupported protocol as a stream of bytes, your resulting script will contain VU emulation commands and functions pertaining to raw socket data. These commands and functions have the prefix `sock`.

Although socket recording will capture network traffic, you need to be familiar with the network protocol to obtain a script you can work with and understand. If the protocol is clear text, the process is fairly straightforward. If the protocol is not clear text, you must understand the structure of the protocol messages.

NOTE: VU supports the Jolt protocol by using macros and user-defined VU functions that call socket emulation commands. For information about the Jolt protocol, see Appendix A.

Emulation Functions

Like emulation commands, the VU emulation functions are related to virtual user emulation. However, emulation functions differ from emulation commands in the following ways:

- Emulation functions do not increment the emulation command count (`_cmdcnt`).
- Emulation functions are neither logged in the standard log file nor recorded in the standard result files; hence they are not available to LoadTest reports.
- Emulation functions do not generate think time delays nor do they time out.

VU Environment Variables

Environment variables specify the virtual users' environments. For example, you can use an environment variable to specify:

- A virtual user's average think time, the maximum think time, and how the think time is mathematically distributed around a mean value.
- How long to wait for a response from the server before timing out.
- The level of information that is logged and is available to reports.

The following table summarizes the VU environment variables:

Environment Variable	Category	Values	Default
`CS_blocksize`	client/server	integer 1 - 32767	1
`Check_unread`	reporting	string `"FIRST_INPUT_CMD"` `"OFF"` `"EVERY_INPUT_CMD"`	`"FIRST_INPUT_CMD"`
`Column_headers`	client/server	string `"ON"` `"OFF"`	`"ON"`
`Connect_retries`	connect	integer 0-2000000000	100

(Continued)

Environment Variable	Category	Values	Default
`Connect_retry_interval`	connect	integer 0-2000000000 ms	200
`Cursor_id`	client/server	integer: a value returned by `sqldeclare_cursor`, `sqlopen_cursor`, or `sqlalloc_cursor`	0
`Delay_dly_scale`	think time	integer 0-2000000000 percent	100
`Escape_seq`	exit sequence	any bank expression; two optional integer expressions	null bank expression

`Http_control`	HTTP-related	integer indicating 0 or more of:
		`0` (exact match)
		`HTTP_PARTIAL_OK`
		`HTTP_PERM_REDIRECT_OK`
		`HTTP_TEMP_REDIRECT_OK`
		`HTTP_REDIRECT_OK`
		`HTTP_CACHE_OK`
			`0`

`Iiop_bind_modi`	IIOP-related	colon-separated list of one or more of the following strings: `"File"` `"Nameservice"` `"IOR"` `"Visibroker"`	null string
`Line_speed`	HTTP-related	integer indicating bits per second: 0-2000000000	`0` (no delay)
`Log_level`	reporting	string `"ALL"` `"TIMEOUT"` `"OFF"` `"ERROR"` `"UNEXPECTED"`	`"TIMEOUT"`
`Logout_seq`	exit sequence	any bank expression; two optional integer expressions	null bank expression
`Max_nrecv_saved`	reporting	integer 0-2000000000	2000000000
`Mybstack`	private	a bank expression	`NULL` (empty)
`Mysstack`	private	a string expression	`""`
`Mystack`	private	an integer expression	0
`Record_level`	reporting	`"MINIMAL"` `"TIMER"` `"FAILURE"` `"COMMAND"` `"ALL"`	`"COMMAND"`
`Server_connection`	client/server	A value returned by `sqlconnect`	1

(Continued)

Environment Variable	Category	Values	Default
Sqlexec_control_oracle	client/server	string "" "STATIC_BIND"	""
Sqlexec_control_ sqlserver	client/server	string "LANGUAGE" "RPC"	"LANGUAGE"
Sqlexec_control_sybase	client/server	string "LANGUAGE" "RPC" "IMMEDIATE"	"LANGUAGE"
Sqlnrecv_long	client/server	integer 0-2000000000	20
Statement_id	client/server	integer 0, or a value returned by sqlprepare or sqlalloc_statement	0
Suspend_check	reporting	string "ON" "OFF"	"ON"
Table_boundaries	client/server	string "ON" "OFF"	"OFF"
Think_avg	think time	integer 0-2000000000 ms	5000
Think_cpu_threshold	think time	integer 0-2000000000 ms	0
Think_cpu_dly_scale	think time	integer 0-2000000000 ms	100
Think_def	think time	string "FS" "LS" "FR" "LR" "FC" "LC"	"LR"
Think_dist	think time	string "CONSTANT" "UNIFORM" "NEGEXP"	"CONSTANT"
Think_dly_scale	think time	integer 0-2000000000 ms	100
Think_max	think time	integer 0-2000000000 ms	2000000000
Think_sd	think time	integer 0-2000000000 ms	0
Timeout_act	response timeout	string "IGNORE" "FATAL"	"IGNORE"
Timeout_scale	response timeout	integer 0-2000000000 ms	100
Timeout_val	response timeout	integer 0-2000000000 ms	120000 ms

Changing Environment Variables Within a Script

Environment control commands allow a VU script to control a virtual user's environment by operating on the environment variables. The environment control commands are `eval`, `pop`, `push`, `reset`, `restore`, `save`, `set`, and `show`.

Every environment variable has, instead of a single value, a group of values: a default value, a saved value, and a current value.

- **default** – The value of an environment variable before any commands are applied to it. Environment variables are automatically initialized to a default value, and, like persistent variables, retain their values across scripts. The `reset` command resets the default value, as listed in the previous table.
- **saved** – The saved value of an environment variable can be used as one way to retain the present value of the environment variable for later use. The `save` and `restore` commands manipulate the saved value.
- **current** – The VU language supports a last-in-first-out "value stack" for each environment variable. The current value of an environment variable is simply the top element of that stack. The current value is used by all of the commands. The `push` and `pop` commands manipulate the stack.

Initializing Environment Variables through a Schedule

In addition to changing VU environment variables directly within a script, you can also initialize the most commonly used environment variables through a schedule. To initialize the values through a schedule, display the schedule and click the **User Settings** button.

The environment variable values that you set in a schedule are in effect until the script changes that value.

Client/Server Environment Variables

This section discusses the client/server environment variables.

Column_headers

This string environment variable, used by `sqlnrecv` and `sqlfetch_cursor`, indicates whether column headers should be included with the retrieved data. Values are `"ON"` (the default) or `"OFF."` When the value is `"ON,"` `sqlnrecv` or `sqlfetch_cursor` includes column names in `_alltext` and in the log file. `_response` never includes column headers.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Cursor_id

This integer environment variable has a default value of 0 and may contain any value returned by `sqldeclare_cursor`, `sqlopen_cursor`, or `sqlalloc_cursor`.

If the value of `Cursor_id` is zero, then `sqldeclare_cursor` allocates new resources for a cursor and returns the cursor id associated with those resources. If the value of `Cursor_id` is non-zero, `sqldeclare_cursor` does not allocate new resources, and instead reuses the resources associated with that cursor.

The `sqlopen_cursor` command behaves the same way when it is given a SQL statement. If `sqlopen_cursor` is given a `Cursor_id` argument, `Cursor_id` has no effect.

CS_blocksize

This integer environment variable, used by `sqlnrecv` and `sqlfetch_cursor`, specifies the maximum number of rows to receive with a single SQL database request. If `sqlnrecv` or `sqlfetch_cursor` must retrieve more than the number of rows specified by `CS_blocksize`, the rows are retrieved by multiple requests.

The minimum and default value is 1 row. Although the maximum value is 32767 rows, your system resources or database server may limit you to a considerably smaller maximum value.

This environment variable affects system performance and response time measurements. You should set it to the same value that the client application uses. This may vary from one command to another.

If you set `CS_blocksize` too small, your system performs too many fetch commands. If you set it too large, your system performs too few fetch commands.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Server_connection

This integer environment variable identifies the current server connection over which emulation commands operate. Values are integer expressions obtained by calling `sqlconnect`, `http_request`, or `sock_connect`.

If `Record_level` is `"COMMAND"` or `"ALL,"` `Server_connection` is recorded. This is to inform LoadTest reports which `Server_connection` an emulation command uses.

You can initialize this environment variable only by editing a script.

Sqlexec_control variables

These string environment variables, used by `sqlexec`, control the method used to transmit the SQL statement to the SQL database server.

The `Sqlexec_control` variables are as follows:

Variable	Description
`Sqlexec_control_sybase`	Values can be: ▶ `LANGUAGE`. Default. Commands are sent as regular SQL text. ▶ `RPC`. Commands are initiated and executed as a remote procedure call. Arguments are optional. ▶ `IMMEDIATE`. Commands are executed as dynamically prepared statements, with or without arguments.
`Sqlexec_control_sqlserver`	Values can be: ▶ `LANGUAGE`. Default. Commands are sent as regular SQL text. ▶ `RPC`. Commands are initiated and executed as a remote procedure call. Arguments are optional.
`Sqlexec_control_oracle`	Values can be: ▶ `""`. Default. Arguments are bound for each call to `sqlexec`. ▶ `STATIC_BIND`. Arguments are bound to a static memory location, and argument values are copied to that location for execution by `sqlexec`.

You can initialize this environment variable only by editing a script.

Sqlnrecv_long

This integer environment variable, which is used by `sqlnrecv` and `sqlfetch_cursor`, specifies the number of bytes of longbinary and longchar columns to be fetched from the server, and included in the `_response` read-only variable and logged.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Statement_id

`Statement_id` allows you to reuse cursor structures. You can allocate it once (using `sqlalloc_statement`) and then prepare different SQL statements on the same structure, by setting the `Statement_id` environment variable to the value returned from `sqlalloc_cursor`. This improves performance on the database by taking up fewer resources.

`Statement_id` holds the statement IDs returned by `sqlprepare` and `sqlalloc_statement`. These IDs can be used by `sqlexec`, as well as the sqlcursor commands, in place of a string representation of a SQL statement. `Statement_id` is also used by `sqlfree_statement`, and affects `sqlnrecv` and `sqllongrecv`.

Example 1

```
stmtid_1 = sqlalloc_statement();
set Statment_id = stmtid_1;
/* since we set Statement_id = stmtid_1, sqlprepare will operate on
that id
instead of creating a
new one */
sqlprepare "select * from employees";
sqlexec stmtid_1;
/* this statement will also operate on the stmtid_1 instead of
creating a
new structure since Statement_id is still set */
sqlprepare "select * from users";
sqlexec stmtid_1;
```

Example 2

The `Statement_id` also allows you to interleave `sqlexec` and `sqlnrecv` commands. Up until now, it has always been a requirement that `sqlnrecv commands` immediately follow `sqlexec` commands. If you use the `Statement_id` environment variable, you can do an exec on one statement (`stmtid_1`), do a prepare, exec, and fetch on another statement (`stmtid_2`), and then go back and do a fetch on `stmtid_1`.

For example:

```
stmtid_1 = sqlalloc_statement();
stmtid_2 = sqlalloc_statement();

set Statement_id = stmtid_1;

/* this operates on stmtid_1 */
sqlprepare "select * from employees";

sqlexec stmtid_1;

set Statement_id = stmtid_2;

/* this operates on stmtid_2 */
sqlprepare "select * from users";

sqlexec stmtid_2;

/* this operates on stmtid_2 since that is what Statement_id is set
to */
sqlnrecv ALL_ROWS;

set Statement_id = stmtid_1;

/* this operates on stmtid_1 since that is what Statement_id is now
set to
*/
sqlnrecv ALL_ROWS;
```

Table_boundaries

This string environment variable, used by `sqlnrecv` and `sqlfetch_cursor`, halts data retrieval at table boundaries. Values are `"ON"` or `"OFF."`

When the value is `"ON"`:

- `sqlnrecv` halts at the end of the current table, even if fewer than *n* rows were retrieved. The next call to `sqlnrecv` retrieves the next table.
- `sqlfetch_cursor` does not cross table boundaries when fetching from a multitable result set.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Connect Environment Variables

This section describes the connect environment variables. This group of environment variables applies to HTTP and socket schedule runs only. Specifically, these environment variables apply to the `http_request` and `sock_connect` emulation commands.

Connect_retries

`Connect_retries` is the number of retries before giving up the connection. Its values are 0–2000000000; the default is 100.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Connect_retry_interval

`Connect_retry_interval` is the delay (in milliseconds) after a connection failure before the next connection attempt. Its values are 0–2000000000; the default is 200.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Exit Sequence Environment Variables

This section describes the exit sequence environment variables. This group of environment variables applies to SQL schedule runs only.

Escape_seq and Logout_seq

The environment variables `Escape_seq` and `Logout_seq` make the exit from an SQL schedule run as graceful as possible. These variables each specify a SQL command to be sent to the SQL database server by `sqlexec` when certain circumstances occur.

The values of `Escape_seq` and `Logout_seq` are bank expressions, consisting of

- A required string expression containing the SQL statements that you want to send to the server through `sqlexec`.
- An optional integer expression to temporarily override the `Think_avg` value when sending the sequence to the SQL database server.
- A second optional integer expression to temporarily override the `Server_connection` value, which is the default value used if there are multiple open connections.

`Escape_seq` and `Logout_seq` both have as a default value a bank expression containing a null string and no optional integer override values.

Example

This example begins a database transaction and then pushes an escape sequence of `"rollback work"` using a think time value of `0` seconds. After the transaction is complete, the escape sequence is restored to its original value by `pop`.

```
#include <VU.h>
. . .
sqlexec "begin transaction";
push Escape_seq = bank("rollback work", 0);
. . .
sqlexec "commit work";
pop Escape_seq;
```

When Exit Sequence Variables Are Sent

The following list describes the circumstances under which `Escape_seq` and `Logout_seq` are sent to the server.

- Both `Escape_seq` and `Logout_seq` are sent if:
  - The virtual user is executing a script when a schedule run is to be terminated and the LoadTest option `User_term_mode` has the value `"COMMAND."` The sending of `Escape_seq` and `Logout_seq` is delayed until the virtual user completes the current or next emulation command.
  - The library routine `user_exit` is called with a negative status value.
- Only `Logout_seq` is sent if:
  - The virtual user terminates normally after completing his last assigned script.
  - The virtual user is executing a script when a schedule run is to be terminated and the LoadTest option `User_term_mode` has the value `"SCRIPT."` The sending of `Logout_seq` is delayed until the virtual user completes the current script.
  - The library routine `user_exit` is called with a zero status value.
- Neither `Escape_seq` nor `Logout_seq` are sent if:
  - Emulation has not started before the termination is triggered; that is, an initialization error occurred before the first instruction in the first script was executed.
  - The virtual user has not run any SQL emulation commands.
  - A fatal runtime error, other than a fatal receive command time-out, occurs.
  - The library routine `user_exit` is called with a positive status value.

- Escape_seq or Logout_seq may be sent partially or not at all if a schedule run is terminating, the virtual user is executing a script, and the time period specified in the Cleanup_time option expires before or during the time that Escape_seq or Logout_seq are sent. To avoid this, be sure that Cleanup_time is set long enough.

Given that either or both of the sequences are sent to the server, the following conditions apply:

- If both Escape_seq and Logout_seq are sent, Escape_seq is sent first.
- Escape_seq is executed via sqlexec for the connection indicated by each Server_connection if a non-null Escape_seq string is defined. The current value of Escape_seq is executed first, followed by each successive Escape_seq string on the stack until the Escape_seq environment stack is empty.
- Logout_seq is executed via sqlexec for each connection for which a non-null Logout_seq string is defined. The current value of Logout_seq is executed first, followed by each successive Logout_seq string on the stack until that Logout_seq environment stack is empty.
- The sqlexec command uses the current environment variables (Think_avg, Think_dist, Think_def, Think_sd, Think_dly_scale, Think_max, Log_level, and Record_level), when it submits the sequences, except:
  - If an optional Think_avg override value was provided with the sequence, it temporarily replaces the current Think_avg value and additionally enforces a Think_dist of "CONSTANT" (for the specific sequence only).
  - No attempt is made to receive any results from the SQL database server for the sequences. Therefore, for both sequences, if Think_def is "LR" or "FR," it is changed to "CONSTANT" after the very first string of either sequence was sent.

HTTP-Related

This section discusses the HTTP environment variables.

Http_control

This integer environment variable controls which status values are acceptable when a virtual user script is played back. A value of 0, the default, indicates that only exact matches are accepted. However, you can set this variable so that a script plays back successfully even if

- The response was cached during record or playback.
- The server responds with partial or full page data during record or playback.
- The script was redirected to another http server during playback.

`Http_control` can have one or more of the following values:

A value of	Indicates that playback script will accept
`0`	exact matches only
`HTTP_PARTIAL_OK`	206 for 200 and 200 for 206
`HTTP_PERM_REDIRECT_OK`	301 for 200 and 200 for 301
`HTTP_TEMP_REDIRECT_OK`	302 for 200 and 200 for 302
`HTTP_REDIRECT_OK`	301 and 302 for 200, and 200 for 301 and 302
`HTTP_CACHE_OK`	304 for 200 and 200 for 304

You can set `Http_control` to accept multiple values — for example:

```
Http_control = HTTP_REDIRECT_OK | HTTP_CACHE_OK;
```

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or in a script—either by editing the script or by setting the option before you record. The values you set in the schedule are in effect until the script changes that value.

NOTE: For information on how to set this option before you record, see *Controlling the Values Accepted When an HTTP Script Is Played Back* on page 3-20 of the *Using Rational LoadTest* manual.

Line_speed

When you play back a script, the data is sent and received at network speed, with no delays. This integer environment variable enables you to emulate a user who is sending and receiving data through a modem.

Different users can use different line speeds; in fact different connections can be set up with different line speeds. This variable is useful to gauge the effect of dial-up versus direct network connection line speeds on user response times.

You can set `Line_speed` to any integer from 0 to 2000000000 bits per second. A value of 0 means that the data is sent and received at network speed.

IIOP-Related

This section discusses the IIOP-related environment variables.

Iiop_bind_modi

To send requests to an interface implementation, it must be bound to the requestor. The VU emulation command `iiop_bind` establishes a binding method, called a bind modus, for all subsequent emulation commands. The default bind modus for `iiop_bind` is IOR (Interoperable Object Reference), which depends on the optional argument `ior`.

The string environment variable `Iiop_bind_modi` contains a list of bind modi to be used. Each item in the list is separated with a vertical bar. Each modus is tried in the order given. If a mapping is found, it is used and the search ends.

The following table lists the values of `Iiop_bind_modi`:

Value	Description
`File (`*`Filename`*`)`	A CSV-formatted file of interface name/IOR pairs.
`IOR`	An IOR specification (that is, a string representation of an object reference).
`NameService (IOR)`	A CORBA-compliant Name Service interface implementation.
`Visibroker`	Visibroker osagent locator service (vendor-specific).
`VisibrokerNameService`	Uses the Visibroker osagent location service to access the NameService.

Private Environment Variables

This section describes the private environment variables.

Mystack, Mybstack, and Mysstack

The environment variables `Mystack`, `Mybstack`, and `Mysstack` are private stack variables for each of the three VU data types (integer, bank, and string). These three variables are not used by any of the emulation commands, allowing you complete freedom in their use. These variables can be manipulated and accessed by the environment control commands in a manner identical to the other environment variables.

Like persistent variables, private stack variables are an effective means to preserve data values for a virtual user across scripts, since environment variables are maintained across scripts for the duration of the emulation. This example measures a turn-around time that spans multiple scripts:

```
/* start time of EV1 is recorded & saved on stack */
set Mystack = start_time ["EV1"];
... /* one or more script executions elapse */ ...
endtime = time(); /* actual end time of "EV1": */
/* start time re-recorded from stack to satisfy
   "same script" requirement: */
start_time eval Mystack;
/* "EV1" start/end times recorded: */
stop_time ["EV1"] endtime;
```

Although arrays are recommended as more convenient and efficient, a potential use of `Mybstack` is for quick access to small tables of integer or string data. For example, the following code fragment sets up a table of 20 user names:

```
/* initialize table; preserve Mybstack with push*/
push Mybstack = bank("RUSSELL", "EADIE", "BRIGGS", "RYAN", "COUNTS",
"KWOR", "ALLAN", "BROWN", "WALTON", "HARDING");

/* prepare query */
sqlprepare "select * from Student where Surname = ?";
for ( i = 1; i <= 10; i++)
{
    /* run the query with the selected name */
    sqlexec _statement_id, eval Mybstack[string][i];
}

/* return to old environment */
pop Mybstack;
```

As indicated in this example, you can initialize and access one table in a given environment. By using the `save` and `restore` environment control commands, you can initialize, maintain, and access two tables per environment. However, you cannot access data from more than two tables per environment.

Reporting Environment Variables

This section discusses the reporting environment variables.

Check_unread

`Check_unread` controls when `sqlexec` commands should check for unread row results from the previous `sqlexec`.

The value of `Check_unread` is one of three string expressions:

- **"OFF"** – Do not check for unread row results.
- **"FIRST_INPUT_CMD"** (default) – The first `sqlexec` following a `sqlnrecv` checks for unread row results from the previous `sqlexec`.
- **"EVERY_INPUT_CMD"** – Every `sqlexec` checks for unread row results from the previous `sqlexec`.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Max_nrecv_saved

`Max_nrecv_saved` lets you control the maximum number of rows (SQL) or bytes (HTTP and socket) saved by the receive emulation commands.

`Max_nrecv_saved` is an integer environment variable that affects the behavior of the `sqlnrecv`, `sqllongrecv`, `sqlfetch_cursor`, `http_header_recv`, `http_recv`, `http_nrecv`, `sock_recv`, and `sock_nrecv` emulation commands.

Its default value is 2000000000; the range is 0–2000000000.

The typical reason for using `Max_nrecv_saved` is to save memory and disk space by not having to store and log the results of a very large database query — for example, one that returns thousands of rows.

`Max_nrecv_saved` does not affect the data actually retrieved from the server. Therefore:

- The `_nrecv` read-only variable still contains the number of rows or bytes processed by the last receive emulation command
- `_total_rows` still contains the total number of rows actually received
- `_total_nrecv` still holds the total number of bytes actually received.

If the number of rows or bytes you receive exceeds Max_nrecv_saved:

- The emulation command does not necessarily fail.
- If your Log_level is ALL, the log file entry will note both the number of rows or bytes received and the number of rows or bytes logged.
- Any excess rows are discarded instead of being saved in _response.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Log_level

The value of Log_level determines what information is written to the standard log file, in the log's perfdata directory. The log file is called l*xxx*, where *xxx* is a user ID.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

The values of Log_level are as follows:

- **"OFF"** – Nothing is logged. Log_level can also be given the value "OFF" during a portion of the emulation so that no log entries are made for that portion.
- **"TIMEOUT"** (default) – Logs emulation command timeouts. If a receive emulation command fails due to a timeout, the preceding sqlexec, http_request, or sock_send command is logged, followed by an entry for the failed receive emulation command. If the Log_level is "TIMEOUT" and if the scripts for a virtual user contain no emulation commands that timed out, no log file is created.

  For the testcase and emulate commands, *fail_string* is logged. If there is no *fail_string*, *log_string* is logged.
- **"UNEXPECTED"** – Logs timeouts and unexpected responses from SQL emulation commands.

  For all other emulation commands, "UNEXPECTED" is equivalent to "TIMEOUT."
- **"ERROR"** – Logs all SQL emulation commands that set _error to a nonzero value. All timeouts also are logged, as described in TIMEOUT. All log entries include _error and _error_text. Their values typically are supplied by the SQL database server.

For all other emulation commands, `"ERROR"` is equivalent to `"TIMEOUT."`

- **`"ALL"`** – Signifies that complete logging is to be done. A log entry is made for every emulation command. This log entry contains the following:
  - The type of emulation command and any command ID associated with it.
  - Identification of the VU script and source file containing the command.
  - The line number of the command in the source file and the emulation command count of the VU script. The emulation command count is incremented for every emulation command. When you monitor a schedule, it is useful to distinguish between executions of the same command on different loop iterations, since the script line number would be identical for each iteration.
  - The command-specific information listed in the following table. If the scripts for a virtual user contain no emulation commands, no log file is created.

Command	Specific Information Logged
`http_nrecv`	The response from the server. If response is unexpected, the number of `EXPECTED` characters and the number of `RECEIVED` characters are both logged.
`http_recv`	The response from the server. If response is unexpected, the number of `EXPECTED` characters and the number of `RECEIVED` characters are both logged.
`http_request`	One line after the header indicating the success or failure of the connection, and one line containing the request data transmitted to the server.
`http_header_recv`	One line containing the status from the HTTP header.
`iiop_bind`	The repository id string, the instance id string, the IOR string if present, and the modus actually used to create the binding.
`iiop_invoke`	Connection information if a connection was established for this operation, followed by the operation, all input (or input/output) parameter values, and either the values of all output (or input/output) parameters, or the values of all exception parameters.
Jolt-related VU commands	Jolt emulation is implemented by the emulation commands `sock_send` and `sock_nrecv`.
SAP-related VU commands	SAP emulation is implemented by external C functions and the `emulate` command.

(Continued)

Command	Specific Information Logged
sock_send	The characters submitted to the server. Any data that is not printable and cannot be represented by a standard C escape sequence (graphic images, for example) is represented as an embedded hex string.
sock_nrecv	The response from the server. If a response is unexpected, the number of EXPECTED characters and the number of RECEIVED characters are both logged. Any data that is not printable and cannot be represented by a standard C escape sequence (graphic images, for example) is represented as an embedded hex string.
sock_recv	The response from the server. If a response is unexpected, the expected characters (in standard string constant format) are preceded by EXPECT= , and the actual response is preceded by ACTUAL= . Any data that is not printable and cannot be represented by a standard C escape sequence (graphic images, for example) is represented as an embedded hex string.
sqlprepare	The statement ID returned and the SQL statements that were prepared.
sqlclose_cursor	The cursor ID and the SQL statements (including the statement ID for prepared statements).
sqldeclare_cursor sqldelete_cursor	The SQL statements (including the statement ID for prepared statements), any arguments supplied, the number of rows processed (_total_rows), and the cursor ID.
sqlexec	The SQL statements (including the statement ID for prepared statements), any arguments supplied, and the number of rows processed (_total_rows). If present, the arguments are logged as a comma-separated list of values enclosed in brackets [ ]. String arguments are enclosed in single quotation marks (*'value'*) and integer arguments are shown in decimal without quotation marks (12345). The values of named arguments are preceded by their names; positional argument values are logged without any prefix.
sqlfetch_cursor	The SQL statements (including the statement ID for prepared statements), any arguments supplied, the number of rows processed (_total_rows), the cursor ID, the number of rows received, the number of rows logged if different from the number received, and the number of tables read to fetch the requested number of rows.

(Continued)

Command	Specific Information Logged
`sqlinsert_cursor`	The SQL statements (including the statement ID for prepared statements), any argument supplied, the argument values, the number of rows processed (`_total_rows`), and the cursor ID.
`sqlopen_cursor`	The SQL statements (including the statement ID for prepared statements), any arguments supplied, the argument values, the number of rows processed (`_total_rows`), the cursor ID, and the number of rows received.
`sqlnrecv`	The number of rows received, a two-line column header (`_column_headers`) if the value of the environment variable `Column_headers` is `"ON,"` and a character representation of the rows received (`_response`). If the number of rows received (`_nrows`) exceeds the value of `Max_nrecv_saved`, the log file entry notes both the number of rows received and the number of rows logged. For example: `10439 rows received (1000 logged) from 1 table`
`sqlposition_cursor`	The SQL statements (including the statement ID for prepared statements), the number of rows processed (`_total_rows`), and the cursor ID.
`sqlrefresh_cursor`	The SQL statements (including the statement ID for prepared statements), the number of rows processed (`_total_rows`), and the cursor ID.
`sqlsysteminfo`	The operation, all the argument values given for that operation, the number of rows processed (`_total_rows`), and the cursor ID.
`sqlupdate_cursor`	The SQL statements (including the statement ID for prepared statements), any arguments supplied, the argument values, the number of rows processed (`_total_rows`), and the cursor ID.
TUXEDO commands	Any arguments supplied and their argument values. Tuxedo buffer commands include the type and value of the buffer.

(Continued)

Command	Specific Information Logged
`start_time` `stop_time`	No logging done.
`testcase` `emulate`	If no *`log_string`* is specified, nothing is logged. If *`log_string`* but no *`fail_string`* is specified, *`log_string`* is logged. If both are specified, *`log_string`* is logged if the command succeeds; otherwise, *`fail_string`* is logged.

Example

The sample SQL script for `sqlexec` (page 6-123) produces the following log file. In this example, the log file entries are designed to be easily accessible. The script is `doc` and the source file is `doc.s`. When the value of `_error` is not zero, <<< and >>> are replaced by `***`, so that these occurrences are quickly located. The command ID (if any) is shown in brackets after the command. The numbers in parentheses after the script and script names are the emulation command count and the source line number. In this example, the first emulation command began on source line 22.

```
<<< sqlexec[school]: script = doc(1), source = doc.s(22) >>>
use school
0 rows processed
<<< sqlexec[]: script = doc(2), source = doc.s(24) >>>
select Empnum, Empname, Roomnum from Employee where Rank='TUTOR'
0 rows processed
<<< sqlnrecv[Tutors]: script = doc(3), source = doc.s(28) >>>
10 rows received from 1 table
Empnum      Empname             Roomnum
----------- ------------------- -------
78062       CRESSMAN            2005
79069       PEARSON             2220
80075       BOSTMAN             2220
80079       ROWLANDS            2005
80166       WOODLEY             1307
81494       DIXON               1180
81931       CAMPBELL            2111
82631       FESSERMAN           2111
83418       PORTER              1307
84229       KRAEMER             1307
*** sqlnrecv[Tutors]: script = doc(4), source = doc.s(28) ***
5 rows received from 1 table
EXPECTED 10 rows
```

```
ERROR 40012:  End of results
Empnum      Empname             Roomnum
----------- ------------------- -------
84555       SEARLE              2005
85082       NORRIS              2111
85609       O'DONNELL           1180
85718       ASHE                1180
86080       PALMER              2220
<<< sqlexec[]: script = doc(5), source = doc.s(35) >>>
select * from Dept
0 rows processed
<<< sqlnrecv[dept (a)]: script = doc(6), source = doc.s(36) >>>
4 rows received from 1 table
DEPTNO DNAME         LOC
------ ------------- -------------
    10 ACCOUNTING    NEW YORK
    20 RESEARCH      DALLAS
    30 SALES         CHICAGO
    40 OPERATIONS    BOSTON
<<< sqlprepare[prep inser]: script = doc(7), source = doc.s(39) >>>
1= insert into Dept values (:no, :name, :place)
<<< sqlexec[]: script = doc(8), source = doc.s(42) >>>
(1) insert into Dept values (:no, :name, :place) [ :no='50',
:name='testing', :place='Raleigh' ]
1 row processed
<<< sqlexec[]: script = doc(9), source = doc.s(42) >>>
(1) insert into Dept values (:no, :name, :place) [ :no='60',
:name='shipping', :place='Durham' ]
1 row processed
<<< sqlexec[]: script = doc(10), source = doc.s(42) >>>
(1) insert into Dept values (:no, :name, :place) [ :no='70',
:name='receiving', :place='Chapel Hill' ]
1 row processed
<<< sqlexec[]: script = doc(11), source = doc.s(45) >>>
select * from Dept
0 rows processed
<<< sqlnrecv[dept (b)]: script = doc(12), source = doc.s(46) >>>
7 rows received from 1 table
DEPTNO DNAME         LOC
------ ------------- -------------
    10 ACCOUNTING    NEW YORK
    20 RESEARCH      DALLAS
    30 SALES         CHICAGO
    40 OPERATIONS    BOSTON
    50 testing       Raleigh
    60 shipping      Durham
    70 receiving     Chapel Hill
<<< sqlexec[]: script = doc(13), source = doc.s(49) >>>
delete from Dept where deptno >= 50
3 rows processed
<<< sqlexec[]: script = doc(14), source = doc.s(51) >>>
select * from Dept
0 rows processed
<<< sqlnrecv[dept (c)]: script = doc(15), source = doc.s(52) >>>
4 rows received from 1 table
DEPTNO DNAME          LOC
------ -------------  -------------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

Record_level

The value of `Record_level` determines what information is written to the standard result file, in the log's `perfdata` directory. The result file is called `rxxx`, where *xxx* is a user ID. Since the result file is in binary form, it is not directly readable; instead, it is input to LoadTest reports.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

`Record_level` can be set to one of the following strings:

- `"MINIMAL"` – Record only items necessary for reports to run. However, the reports will contain no real data. Use this value when you do not want the user's activity included in the reports.
- `"TIMER"` – `MINIMAL` plus `start_time` and `stop_time` emulation commands. Your reports will not contain response times for each emulation command, and an emulation command failure will not show up as a a failure. In addition, the result file for each virtual user will be small. A small result file means that disk consumption and CPU overhead for each virtual user is less, results are retrieved quickly from Agent computers, and you can run reports in a relatively short time. Set `Record_level` to this value if you are not concerned with the response times or pass/fail status of an individual emulation command.
- `"FAILURE"` – `TIMER` plus emulation command failures and some environment variable changes. Set `Record_level` to this value if you want the advantages of a small result file but you also want to make sure that no emulation command failed.
- `"COMMAND"` – `FAILURE` plus emulation command successes and some environment variable changes (default).
- `"ALL"` – `COMMAND` plus all environment variable changes. Complete recording is done. A binary entry is written to the result file for every emulation command and for the `set`, `reset`, `restore`, `push`, and `pop` environment control commands. You can view these entries in Trace report output.

NOTE: Most report output is the same with `"ALL"` or `"COMMAND."` The exception is the Trace report output. With `"ALL,"` the Trace report output includes every emulation command as well as the `set`, `reset`, `restore`, `push`, and `pop` environment control commands. With `"COMMAND,"` the Trace report output includes every emulation command but includes the `set`, `reset`, `restore`, `push`, and `pop` environment control commands only when they affect the `Server_connection` environment variable.

Suspend_check

The string environment variable `Suspend_check` controls whether you can suspend a virtual user from a Monitor view. The value of `Suspend_check` must be one of the following strings:

- **"ON"** (default) – Normal suspend checking is performed (A suspend request is checked before beginning the think time interval by each send emulation command.)
- **"OFF"** – Disables suspend checking. Checking resumes only after the value of `Suspend_check` is changed to `"ON,"` and the next think time interval is encountered.

You can use `Suspend_check` to encapsulate a critical portion of the script where you do not want it to stop. You can also use `Suspend_check` on a script run by a single virtual user and then suspend all virtual users through the Monitor. The single virtual user is not suspended.

Use `Suspend_check` carefully. In particular, be careful to pair `push` and `pop` operations, and to set `Suspend_check` back to `"ON"` after temporarily changing it to `"OFF."`

Response Timeout Environment Variables

This section describes the response timeout environment variables. This group of environment variables applies to HTTP, SQL, IIOP, and socket schedule runs.

Specifically, the response timeout environment variables affect the following commands:

- **HTTP send emulation commands**: `http_request`
- **HTTP receive emulation commands**: `http_header_recv`, `http_recv`, `http_nrecv`
- **SQL send emulation commands**: `sqlprepare`, `sqlexec`, `sqldeclare_cursor`, `sqlopen_cursor`, `sqldelete_cursor`, `sqlupdate_cursor`, `sqlclose_cursor`, `sqlposition_cursor`, `sqlrefresh_cursor`, `sqlinsert_cursor`.
- **SQL receive emulation commands**: `sqlnrecv`, `sqllongrecv`, `sqlfetch_cursor`
- **IIOP send emulation commands**: `iiop_bind`, `iiop_invoke`

- **Socket receive emulation commands**: `sock_recv`, `sock_nrecv`
- **Other send emulation commands**: `emulate`

> **NOTE:** The socket send emulation command, `sock_send`, does not wait for a server response, and therefore the response timeout environment variables do not affect it.

An emulation command generally waits for a response from the server. If a response is received, the appropriate logging and recording is done, and the emulation continues with the execution of the next statement. On the other hand, if the elapsed time an emulation command has been waiting exceeds the value of `Timeout_val` (subject to scaling by `Timeout_scale`), the emulation command times out. In this case, after appropriate logging and recording is done, the value of `Timeout_act` is examined to determine whether this timeout is ignored and emulation continued normally, or whether this timeout is considered a fatal error, resulting in steps taken to end the emulation.

Timeout_act

The values for `Timeout_act` are the strings `"IGNORE"` and `"FATAL."`

If the value of `Timeout_act` is `"IGNORE,"` the emulation continues normally, after the appropriate logging and recording, when a timeout occurs. Recall that an emulation command that returns 0 signals that a timeout has occurred, allowing the script to dynamically react as appropriate to an unexpected response.

If the value of `Timeout_act` is `"FATAL,"` the time out of an emulation command is considered a fatal runtime error. The appropriate logging and recording is done, followed by termination of the virtual user.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Timeout_scale

This integer environment variable controls the percentage multiplier applied to the time-out delay (`Timeout_val`). The default value of 100% represents no change. A value of 50% means one-half the delay, which is twice as fast; 200% means twice the delay, which is half as fast as the original.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Timeout_val

The value of Timeout_val can be any integer in the range 0 to15000000. This value specifies in milliseconds, starting from when the emulation command begins communication with the server, the time an emulation command waits for a server response before it times out. The default value of Timeout_val is 120000 milliseconds (2 minutes).

Choose the value of Timeout_val with care. If it is too small, commands requesting large amounts of data or complex operations time out, even though the server may respond correctly.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think Time Variables

The think time environment variables control the virtual user's "think time" behavior. This is simply the time that a typical user would delay, or think, between submitting commands.

In a virtual user script, the Think_avg is usually set before each http_request emulation command, each sqlexec and sqlprepare emulation command, all TUXEDO emulation commands, and each sock_send emulation command. You need to decide whether to preserve the think times as is, or vary the think times. To preserve the think times, simply run the script.

You can truncate think times that are too long. For example, you might examine a script and see a few very long settings of Think_avg. To truncate these think times, set the value of Think_max to your maximum acceptable think time.

If you are using the script for a multiuser run, you may also want to set the Think_dist environment variable to "NEGEXP" rather than "CONSTANT" so that each virtual user does not pause the same amount of time between each command.

You may decide to further refine your script by dividing the think time into user think time and CPU think time. To do this, set the cpu_threshold environment variable.

Delay_dly_scale

This integer environment variable globally scales the delay times of all `delay` library routines by applying a percentage multiplier. A value of 100%, which is the default, means no change. A value of 50% means one-half the delay, which is twice as fast as the original, 200% means twice the delay, which is half as fast. A value of zero means no delay.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_avg

Specifies the duration, in milliseconds, of the "average" think time interval. The value of `Think_avg` can be any integer in the range 0-2000000000. The default value is 5000 milliseconds.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_cpu_dly_scale

This integer environment variable enables you "change" from a slower computer to a faster computer, and vice versa by multiplying the CPU think time value by a percentage. A value of 100%, which is the default, means no change. A value of 50% means one-half the delay, which is twice as fast as the original; 200% means twice the delay, which is half as fast. A value of zero means no delay. Delay scaling is performed before truncation (if any) by `Think_max`.

For user think times (`Think_avg` is greater that or equal to `Think_cpu_threshold`), `Think_dly_scale` is used instead.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_cpu_threshold

There are actually two kinds of delays — user think time and CPU processing time.

User think time is the time a typical user delays, or thinks, between submitting commands. CPU processing time is the time it takes for the application to generate internal commands from the user's data.

For example, an actual user may pause to think before selecting a student name from a SQL database. This is recorded as user think time. Once the user clicks on the student name, the time spent generating the SQL command and accessing the database is a CPU delay.

Similarly, when a user thinks about which Web page to access, this delay is user think time. Once the user provides the URL for the desired Web page, the CPU must issue commands to get that Web page and display it to the user. This delay is a CPU processing delay.

The environment variable `Think_cpu_threshold` lets you to divide delay time into user think time delays and CPU processing time delays. You then scale each time individually with the environment variables `Think_cpu_delay_scale` and `Think_dly_scale`.

If the value of `Think_avg` is greater than `Think_cpu_threshold`, the delay is considered user think time. The value of `Think_dly_scale` is used to calculate the think time.

If the value of `Think_avg` is less than `Think_cpu_threshold`, the delay is considered CPU think time. With CPU think time:

- The value of `Think_cpu_dly_scale` is used to calculate the delay. This allows CPU processing delays to be scaled differently from user think time delays. For example, typical usage would be to "change" the CPU from a 486 to a Pentium by scaling the CPU processing delays downward.
- The value of `Think_dist` is ignored. All application CPU processing delays are assumed to be `"CONSTANT."` This allows user think time distributions to be used without affecting the calculation of CPU processing delays.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

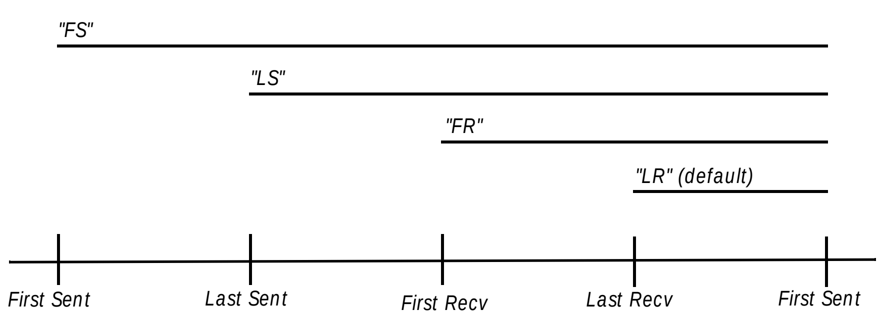
Think_def

Specifies the starting point of the think time interval. The values for `Think_def` can be the following string expressions:

- **"FS" –** The think time interval for the current send emulation command begins at the time the previous send emulation command is submitted.
- **"LS" –** The think time interval for the current send emulation command begins at the time the previous send emulation command is completed.
- **"FR" –** The think time interval for the current send emulation command begins at the time the first data of the previous receive emulation command is received. If there was no intervening receive emulation command, the think time interval begins when the previous send emulation command is completed.
- **"LR" –** The think time interval for the current send emulation command begins at the time the last data of the previous receive emulation command is received. If there was no intervening receive emulation command, the think time interval begins when the previous send emulation command is completed.
- **"FC" –** The think time interval for the current send emulation command begins at the time the previous HTTP connection (`http_request` with address information) or socket connection (`sock_connect`) is submitted. `"FC"` ("first connect") uses the `_fc_ts` integer read-only variable.
- **"LC" –** The think time interval for the current send emulation command begins at the time the previous HTTP connection (`http_request` with address information) or socket connection (`sock_connect`) is completed. `"LC"` ("last connect") uses the `_lc_ts` integer read-only variable.

If you are running SQL-based script, you will probably not want to change the default value of `Think_def`. This is because the values `FS`, `LS`, and `FR` for `sqlexec` and `sqlprepare` are usually almost equivalent.

The following figure shows how the different starting points produce a longer or shorter think time interval:



You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_dist

Specifies think time distribution for virtual user think times. It has no effect for CPU think times. The `Think_dist` environment variable can have the following values:

- **"CONSTANT" –** Sets a constant think time interval equal to the value of `Think_avg`. This is the default value.
- **"UNIFORM" –** Sets a random think time interval distributed uniformly in the range: [`Think_avg` - `Think_sd`, `Think_avg` + `Think_sd`]
- **"NEGEXP" –** This is the recommended setting for multiuser runs. It provides a random think time interval and approximates a bell curve around the think average that you have set. The average think time and standard deviation are equal. In mathematical terms, this setting supplies a random think time interval from a negative exponential distribution with a mean equal to the value of `Think_avg`.

The random number generator used to generate think times for the `"UNIFORM"` and `"NEGEXP"` think time distributions is *not* reseeded by default at each script invocation with an identical seed for each virtual user. To modify default behavior of the random number generator, set the **Seed** and **Seed Flags** options in the schedule. By default, **Seed** generates the same sequence of random numbers. However, it sets unique seeds for each virtual user so that each virtual user will have a different random number sequence.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_dly_scale

This integer environment variable controls the percentage multiplier to be applied to the user think time value. A value of 100%, which is the default, means no change. A value of 50% means one-half the delay, which is twice as fast as the original; 200% means twice the delay, which is half as fast. A value of zero means no delay. Delay scaling is performed before truncation (if any) by `Think_max`.

For CPU think times (`Think_avg` is less than `Think_cpu_threshold`), `Think_cpu_dly_scale` is used instead.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_max

Provides a maximum threshold for think times. `Think_max` specifies, in milliseconds, the maximum value that a generated think time can have. If the normally generated think time value (as defined by `Think_avg`, `Think_dist`, `Think_dly_scale`, and optionally `Think_sd`) exceeds `Think_max`, it is set to the value of `Think_max`. The default value of `Think_max` is 2,000,000,000 milliseconds, which effectively disables the truncation.

`Think_max` is useful with scripts that mimic the actual user think times. You can truncate longer-than-desired think times, which speeds up playback, without having to search for and edit each long think time. `Think_max` has the additional benefit of keeping the original think times. To restore these times, simply remove or comment out the lines that modified the default value of `Think_max`.

`Think_max` is also useful with the `Think_dist` value of `"NEGEXP"` (which ordinarily produces negative exponentially generated think times) to instead produce truncated negative exponentially generated think times.

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Think_sd

Specifies the think time standard deviation. `Think_sd` has meaning only when the value of `Think_dist` is `"UNIFORM."` Otherwise, `Think_sd` has no effect.

The value of `Think_sd` is an integer in the range 0-2000000000. The default value is 0. This value specifies a range around the mean think time interval (`Think_avg`). The actual think time intervals are distributed uniformly throughout this range.

If the value of `Think_dist` is `"UNIFORM"` and the value of `Think_sd` is greater than the value of `Think_avg`, then the think time intervals are still distributed uniformly throughout the range, and any resulting negative think time intervals are treated as having a zero value (no delay).

You can initialize this environment variable by clicking the **User Settings** button in a schedule, or by editing the script. The values you set in the schedule are in effect until the script changes that value.

Examples of Think Time Variables

The following examples further describe the use of the think time variables.

```
sqlexec "select * from publishers";
sqlnrecv ALL_ROWS;
set Think_avg = 3000;
set Think_def = "LS";
set Think_dist = "CONSTANT";
sqlexec "select * from authors";
sqlnrecv ALL_ROWS;
```

Assume that the `sqlexec "select * from publishers"` command was completed at time 12000 and that the `sqlexec "select * from authors"` command was invoked at time 13750. Therefore, the second `sqlexec` would wait approximately 1250 milliseconds (that is, 3000 - (13750 - 12000)) before submitting the `select * from authors` command.

The following example uses the macros `SECONDS` and `MINUTES` defined in the `VU.h` header file. `SECONDS` converts its argument from seconds to milliseconds; `MINUTES` converts its argument from minutes to milliseconds. For details, see *VU.h* on page 4-2.

```
#include <VU.h>

sqlexec "select * from publishers";
sqlnrecv ALL_ROWS;
set Think_avg = MINUTES(2);
set Think_dist = "UNIFORM";
set Think_sd = SECONDS(30);
sqlexec "select * from authors";
sqlnrecv ALL_ROWS;
sqlexec "select * from titles";
sqlnrecv ALL_ROWS;
```

The think time intervals for the `select * from authors` and `select * from titles` commands is uniformly distributed in the range [90000,150000] milliseconds (90000 = 120000 - 30000, 150000 = 120000 + 30000). Since the default value of `"LR"` is used for `Think_def`, the think time intervals for these two commands begin when the end of the result set is received by the previous `sqlnrecv` command.

Read-Only Variables

The VU read-only variables provide access to data items collected during the schedule run. These data items provide information about the commands and responses submitted and received during the emulation, plus information about the progress of the emulation. In fact, all of the log file information in *`stdlog`* and most of the result file information in *`stdrec`* is maintainable directly from the read-only variables. Therefore, by using the read-only variables, you can customize log or result files to perform detailed logging and recording.

All read-only variables begin with the underscore character (_). They can be used in expressions in the same way a variable of the same type could be used, except that they cannot be used as the first operand of any assignment operator, nor as the operand of the &, ++, or -- operators.

The following table shows the string-valued read-only variables:

Variable	Contains
`_alltext`	The same as `_response`.
`_cmd_id`	The ID of the most recent emulation command.
`_command`	The text of the most recent: ▶ `http_request` ▶ `sqlprepare`, `sqlexec`, `sqldeclare_cursor`, `sqlfetch_cursor`, `sqlopen_cursor`, `sqldelete_cursor`, `sqlupdate_cursor`, `sqlclose_cursor` ▶ `tux_bq`, `tux_tpabort`, `tux_tpacall`, `tux_tpbroadcast`, `tux_tpcall`, `tux_tpconnect`, `tux_tpdequeue`, `tux_tpenqueue`, `tux_tppost`, `tux_tpsubscribe` ▶ `sock_send` ▶ The operation of the most recent `iiop_invoke`
`_column_headers`	The two-line column header if `Column_headers` is `ON`; otherwise, it contains `""`.
`_error_text`	The full text of the error from the last emulation command. If `_error` is 0, `_error_text` returns `""`. For an SQL database or TUXEDO error, the text is provided by the server.
`_host`	The host name of the computer on which the script is running.
`_response`	The text of up to the value of `Max_nrecv_saved` ▶ rows received in the most recent `sqlnrecv`, `sqllongrecv`, or `sqlfetch_cursor` ▶ bytes received in the most recent `http_header_recv`, `http_recv`, `http_nrecv` ▶ bytes received in the most recent `sock_nrecv` or `sock_recv` This read-only variable is the same as `_alltext`.

(Continued)

Variable	Contains
`_script`	The name of the VU script currently being executed.
`_source_file`	The name of the file that was the source for the portion of the VU script being executed.
`_user_group`	The name of the user group (from the schedule) of the user running the script.
`_version`	The full version string of LoadTest (for example 7.5.0.1045).

The following table shows the integer-valued read-only variables:

Variable	Contains
`_cmdcnt`	A running count of the number of emulation commands the script has executed.
`_cursor_id`	The last cursor declared by `sqldeclare_cursor` or opened by `sqlopen_cursor`.
`_error`	The status of the last emulation command. Most values for `_error` are supplied by the server.
`_error_type`	If you are emulating a TUXEDO session and `_error` is nonzero, `_error_type` contains one of the following values: 0 (no error) 1 VU/TUX Usage Error 2 TUXEDO System/T Error 3 TUXEDO FML Error 4 TUXEDO FML32 Error 5 SUT Error 6 VU/TUX Internal Error If you are emulating an IIOP session and `_error` is nonzero, `_error_type` contains one of the following values: 0 (no error) 1 IIOP_EXCEPTION_SYSTEM 2 IIOP_EXCEPTION_USER 3 IIOP_ERROR
`_fc_ts`	The "first connect" timestamp for `http_request` and `sock_connect`.

(Continued)

Variable	Contains
`_lc_ts`	The "last connect" timestamp for `http_request` and `sock_connect`.
`_total_nrecv`	The total number of bytes received for all HTTP and socket receive emulation commands issued on a particular connection.
`_fr_ts`	The timestamp of the first received data of `sqlnrecv`, `http_nrecv`, `http_recv`, `http_header_recv`, `sock_nrecv`, or `sock_recv`. For `sqlexec` and `sqlprepare`, `_fr_ts` is set to the time the SQL database server responded to the SQL statement.
`_fs_ts`	The time the SQL statement was submitted to the server by `sqlexec` or `sqlprepare`, or the time when the first data was submitted to the server by `http_request` or `sock_send`.
`_lineno`	The line number in `_source_file` of the previously executed emulation command.
`_lr_ts`	The timestamp of the last received data for `sqlnrecv`, `http_nrecv`, `http_recv`, `http_header_recv`, `sock_nrecv`, or `sock_recv`. For `sqlexec` and `sqlprepare`, `_lr_ts` is set to the time the SQL database server responded to the SQL statement.
`_ls_ts`	The time the SQL statement was submitted to the server by `sqlexec` or `sqlprepare`, or the time the last data was submitted to the server by `http_request` or `sock_send`.
`_nrecv`	The number of rows processed by the last `sqlnrecv`, or the number of bytes received by the last `http_nrecv`, `http_recv`, `sock_nrecv`, or `sock_recv`.
`_nusers`	The number of total users in the current LoadTest session.
`_nxmit`	The total number of characters contained in the SQL statements transmitted to the server in the last `sqlexec` or `sqlprepare` command, or the number of bytes transmitted by the last `http_request` or `sock_send`.
`_statement_id`	The value assigned as the prepared statement ID, which is returned by `sqlprepare and sqlalloc_statement`.

(Continued)

Variable	Contains
`_tux_tpurcode`	TUXEDO user return code, which mirrors the TUXEDO API global variable `tpurcode`. It can be set only by the `tux_tpcall`, `tux_tpgetrply`, `tux_tprecv`, and `tux_tpsend` emulation commands.
`_total_rows`	Set to the number of rows processed by the SQL statements. If the SQL statements do not affect any rows, `_total_rows` is set to 0. If the SQL statements return row results, `_total_rows` is set to 0 by `sqlexec`, then incremented by `sqlnrecv` as the row results are retrieved.
`_uid`	The numeric ID of the current virtual user.

Initialization of Read-Only Variables

At the beginning of a schedule run, before the execution of the first script:

- The timestamp variables, `_fs_ts`, `_ls_ts`, `_fr_ts`, `_lr_ts`, `_fc_ts`, and `_lc_ts`, are initialized to the current time.
- `_uid` is initialized to the correct user ID. All other integer read-only variables are initialized to 0.
- All string read-only variables are initialized to null strings.

After a script executes, read-only variables are reinitialized, except for the timestamp variables. By default, timestamp variables carry over their values from the previous script. However, the timestamp variables are reinitialized if you open a schedule, click the **Runtime** button, and check **Initialize timestamps for each script**.

Example

Besides supporting customized logging and recording, the read-only variables serve other purposes within a script. For example, a particularly useful application of `_uid` is to create a common script with commands and responses tailored to specific virtual users. The following example shows a common login script, which is identical for each user except for SQL database user IDs and passwords:

```
string name;
name = "usr"+itoa(_uid);
con=sqlconnect ("", name, "pswd" +itoa(_uid),"","");
set Server_connection = con;
...
sqlexec "insert into sales values ("+name +", 12, 10.00)";
```

In this segment, it is assumed that `usr`*`xxx`* and `pswd`*`xxx`* are the SQL database server ID and password strings for user *`xxx`*. For example, the login ID and password of virtual user 12 would be `usr12` and `pswd12`.

Supplying a Script with Meaningful Data

When you play back a script, the script uses the exact values that you recorded. Assume, for example, that you record a script that adds a record with a primary key of John Doe to a database. When you play back the script, to emulate thousands of users, you will get errors after the first John Doe is added. To correct this situation, you use **datapools**, which supply unique test values to the server.

Although varying test values may work for those transactions that depend on the result of an earlier transaction, other transactions may depend on values received from the server. If a script contains these transactions, you must manually edit the script to replace some of the missing client logic so that the values correlate dynamically. This is called **dynamic data correlation**.

Datapools

A datapool is a convenient way to supply variable data values to a script. Typically, you use a datapool with a script so that:

- Each virtual user that runs the script can send realistic values, including unique values, to the server.
- A single virtual user that performs the same transaction multiple times can send realistic values to the server in each transaction.

If you do not use a datapool with a script, each virtual user sends the same values to the server (which are the values you provided when you recorded the script).

Usually, you create a datapool immediately after you record a virtual user script, using the datapool capability in Rational Robot.

For more information about creating and managing datapools, see the *Using Rational LoadTest* manual.

Dynamic Data Correlation

Dynamic data correlation is a technique to supply variable data values to a script when the transactions in a script depend on values supplied from the server.

For example, when you record an http script, the Web server may send back a unique string, or session ID, to your browser. The next time your browser makes a request, it must send back the same session ID to authenticate itself with the server.

The session ID can be stored in three places:

- In the Cookie field of the HTTP header.
- In an arbitrarily named field of the HTTP header.
- In an arbitrary hidden field in an actual HTML page.

Rational Suite PerformanceStudio finds the session IDs (and other correlated variables) and, when you run the schedule, automatically generates the proper script commands to extract their actual values.

Before you record a script, you can choose whether PerformanceStudio correlates all possible values (the default), does not correlate any values, or correlates only a specific list of variables that you provide.

Part III

Command Reference

▸▸▸ CHAPTER 6

Command Reference

This command reference contains the following categories of information:

- **Environment control commands** – Enable you to control a virtual user's environment by changing the VU environment variables. For example, you can set the level of detail logged or the number of times to try a connection.
- **Flow control statements** – Enable you to add conditional execution structures and looping structures to your virtual user script. The flow control statements behave like their C counterparts, with enhancements added to `break` and `continue`.
- **Library routines** – Provide your virtual user script with predefined functions that handle file I/O, string manipulation, and conversion of data types and formats.
- **Send and receive emulation commands** – Emulate client activity and evaluate the server's responses. These commands also perform communication and timing operations. You can log emulation commands in a log file.
- **Emulation functions** – Like emulation commands, emulation functions emulate client activity and evaluate the server's responses. However, emulation functions do not perform communication and timing operations, and they are not logged in a log file.
- **Synchronization statement** – Causes a script to pause execution until all participating virtual users rendezvous. Generally, you control synchronization points through a LoadTest schedule, but you can use the VU `sync_point` statement to insert a synchronization point anywhere in a script.
- **Datapool functions** – Retrieve data from a datapool and assign the individual values to script variables. This enables a script that is executed more than once to use different values in each execution.
- **VU toolkit functions** – These functions, which come with Rational Suite PerformanceStudio, enable you to parse data returned by `sqlnrecv` into rows and columns.

abs

Library Routine

Description Returns the absolute value of its argument.

Syntax `int abs (int)`

Syntax Element	Description
int	The integer expression for which to return an absolute value.

Comments None.

Example This example prints the absolute values of the integers 34 and -10:

```
int var1 = 34;
int var2 = -10;
int result;
result = abs(var1)
printf ("The absolute value of %d is %d\n", var1, result);
result = abs(var2)
printf ("The absolute value of %d is %d\n", var1, result);
```

See Also None.

AppendData

VU Toolkit Function: Data

Description Adds the data returned by `sqlnrecv` to the specified data set.

Syntax

```
#include <sme/data.h>
string func AppendData(data_name)
string data_name;
```

Syntax Element	Description
data_name	The name of the data set to receive the data from `sqlnreceive`.

Comments The `AppendData` function adds the data returned by the most recent `sqlnrecv` command to the data set specified by the *`data_name`* argument. Before data can be added to a set, the set must be created with a call to `SaveData`. No check is made to ensure that the data to be added has the same structure as the existing data stored under that name. If they do not match, a valid return is generated, but subsequent results are undefined.

If the specified data set does not exist, the function calls `SaveData` to create a data set with the matching characteristics. In either case, it returns the length of the data set including the data just appended.

Because data is stored using only the results of the most recent `sqlnrecv` command, any VU environment variables that affect the data returned also affect this function. In particular, it assumes that only one table was fetched. If `Table_boundaries` is set to `"OFF"` and multiple tables are retrieved, the results of this function and subsequent data commands on the stored data have undefined results.

Example

This example first frees any previously saved data from the “parts” text buffer. A loop is started to query the database five times. The script then obtains the next record from a file being shared by all virtual users that execute this script. The record is parsed by selection of the first field and direct selection of the third field, skipping the second field. The third field is composed of four or more subfields. Parsing of the third field continues by selection of the first subfield, which provides a count of the number of remaining subfields. One of the remaining subfields is selected at random to form a part of the query. After the query is performed, the returned rows are saved. If this is the first iteration of the loop, the rows are saved to the “parts” text buffer. Subsequent iterations of the loop append the data from the returned rows to the “parts” text buffer.

```
#include <VU.h>
#include <sme/data.h>
#include <sme/fileio.h>

{
   shared int file_tag_lock, file_tag_offset;
   string product_id, part_id, subassm_id;
   string temp_str;
   int subassm_cnt;

   /* This script assumes a connection was made to the database. */

  /* Record layout of "myfile"                                    */
  /* product | part | subassm_cnt ; subassm_1; subassm_2 ; subassm_3;
... */

  /* There will be a minimum of three subassemblies in each record. */

   FreeData("parts");

   /* Perform 5 queries for parts. */

   for (i=0; i<=4; i++)
      {
      SHARED_READ ("myfile", file_tag);

      /* Parse the record. */
      product_id = NextField();

      temp_str = IndexedField(3);
      /* Note: The entire unparsed field is returned but it is not
       used directly. So the returned text string is not used.     */
```

```
            subassm_cnt = atoi(NextSubField());
            subassm_id = IndexSubField(uniform(2,subassm_cnt+1));

            /* Query for the part. */
            sqlexec ["test_001"]
               "select part_name from product_db "
               "where product='"+product_id+"' "
               "and subassembly='"+subassm_id+"'";
            sqlnrecv ["test_002"] ALL_ROWS;

            if i = 0
               SaveData("parts");
            else
               AppendData("parts");
            }
         }
```

See Also FreeAllData FreeData GetData GetData1 SaveData

atoi

Library Routine

Description Converts strings to integers.

Syntax `int atoi (str)`

Syntax Element	Description
str	A string expression of digits to convert.

Comments The `atoi` routine behaves like the C `atoi` function, returning an integer corresponding to a sequence of ASCII digits (0 to 9).

The `atoi` routine begins the conversion with the first character in *str* and continues converting until it encounters the end of the string *str* or until a non-digit is found. If the first character is a negative sign, `atoi` returns a negative integer. Leading tabs, spaces, and zeros in *str* are ignored. If the first character of *str* is not a digit, space, tab, or negative sign, `atoi` returns the integer value 0. In all other cases it returns the integer corresponding to the digit string.

The `atoi` routine is also useful for stripping leading zeros from a string. Execute `atoi` on the string, and then run `itoa` on the value returned.

Example This example returns the integer value 9302:

```
atoi(" 9302");
```

This example returns the integer value 32:

```
atoi("32.1");
```

This example returns the integer value 1023:

```
atoi("102" + "3yz");
```

See Also itoa

bank

Library Routine

Description Creates bank expressions for assignments to the bank environment variables `Escape_seq` and `Logout_seq`.

Syntax `bank bank (expr1, expr2,... exprN)`

Syntax Element	Description
expr1, expr2, exprN	A collection of zero or more integer expressions, string expressions, or both.

Comments The `bank` routine returns a bank expression consisting of the collection of its arguments. The position of arguments is important only within the same expression type (that is, integer or string). For example, in the following three calls to `bank`, the first two calls return equivalent bank expressions; the third call does not:

```
bank(int1, int2, str1, str2)
bank(str1, int1, int2, str2)
bank(int1, int2, str2, str1)
```

A single call to `bank` is limited by the maximum number of arguments per VU subroutine. Use the arithmetic operator (+) to create a union of `bank` expressions.

Example These two examples return a bank expression containing the three strings `"ab"`, `"cd"`, and `"ef"` (in that specific order) and the single integer 4:

```
bank("ab", 4, "cd", "ef");
bank("ab") + bank (4) + bank ("cd", "ef");
```

This example returns an empty (null) bank expression:

```
bank();
```

This example returns a bank expression containing no strings and the integer 149:

```
bank(atoi("149"));
```

See Also None.

break

Flow Control Statement

Description Stops execution of `for`, `while`, and `do-while` statements.

Syntax **break** [*level_constant*]

Syntax Element	Description
level_constant	An optional integer that specifies the number of nested loop levels to break out of.

Comments The `break` statement enables you to control the execution of `for`, `while`, and `do-while` loops. As in C, if the `break` statement is encountered as one of the statements in a `for`, `while`, or `do-while` loop, execution of that loop stops immediately.

Unlike C, however, `break` can be specified with an optional argument, which allows it to affect a specified level of nested looping structures. Without this argument, or if the argument is 1, it behaves like its counterpart in C.

Example In this example, if the value of `level_constant` is 1, execution of the `break` statement causes the `do-while` loop to end, and the next statement executed is `print "Completed do-while."` If the value of `level_constant` is 2, execution of both the `do-while` and `while` loops stops and the next statement executed is the `printf` statement. If the value of `level_constant` is 3 or greater, execution of the `do-while`, `while`, and `for` loops stops and the next statement executed is `cnt *= 7`.

```
cnt = inner_cnt = 0;
for (i = 0; i < 10; i++) {
    cnt++;
    j = 0;
    while (j < cnt) {
        j++;
        inner_cnt = j;
        do {
            inner_cnt++;
            break level_constant;
        }   while (inner_cnt <= 4);
        print "Completed do-while";
    }
    printf ("Now on iteration %d", i);
}
cnt *= 7;
```

See Also `for` `while` `do-while` `continue`

cindex

Library Routine

Description Returns the position within `str` of the first occurrence of the character `char`.

Syntax `int cindex (str, char)`

Syntax Element	Description
str	The string to search.
char	The character to search for within *str*.

Comments The `cindex` (character index) routine returns the integer zero if no occurrences of *char* are found.

The `cindex`, `lcindex`, `sindex`, and `lsindex` routines return positional information about either the first or last occurrence of a specified character or set of characters within a string expression. The `strspan` routine returns distance information about the span length of a set of characters within a string expression.

Example This example returns the integer value 1, because `a` is the first letter in the string `aardvark`:

```
cindex("aardvark", 'a');
```

This example returns the integer value 0, because the letter `b` does not occur in the string `aardvark`:

```
cindex("aardvark", 'b');
```

See Also `lcindex` `sindex` `lsindex` `strspan` `strstr`

base64_decode()

Library Routine

Description Decodes a base 64–encoded string.

Syntax `string base64_decode(str)`

Syntax Element	Description
str1	A string expression containing the encoded text.

Comments The base64_decode() function returns the clear text string equivalent of the given base64–encoded string. If base64_decode() fails, it returns an empty string, "".

Example This example uses base64_decode() to extract the login ID and password contained in the given request text.

```
string auth_str, key, log_pass, request_text;
int start, end;

key = "Authorization:Basic";
start = strstr(request_text, key);
start += strlen(key);
auth_str = substr(request_text, start, 10000);
end = strstr(auth_str, "\r\n");
auth_str = substr(auth_str, 1, end – 1);
log_pass = base64_decode(auth_str);
```

See Also base64_encode()

base64_encode()

Library Routine

Description Encodes a string using base-64 encoding.

Syntax string **base64_encode**(*str*)

Syntax Element	Description
str	A string expression containing the clear text.

Comments The base64_encode() function returns the base 64–encoded string equivalent of the given string. If base64_encode() fails, it returns an empty string, "".

This function allows users to parameterize http login IDs and passwords.

Example This example uses base64_encode() to build an authorization string for a login ID and password and then incorporates the result into an http_request.

```
string auth_str;
auth_str = base64_encode("mylog" +":"+ "mypass");
if (auth_str == “”)
{
user_exit(1,"Can’t convert login/password\n");
}
rational_com_80 = http_request["HTTP_lo~004"]
"rational.com:80", HTTP_CON_DIRECT,
"GET/HTTP/1.0\r\n",
. . .
"Authorization:Basic" + auth_str + "\r\n"
"\r\n";
```

See Also base64_decode()

close

Library Routine

Description Writes out buffered data to a file and then closes the file.

Syntax `int close(file_des)`

Syntax Element	Description
file_des	An integer expression specifying the file to close. *file_des* is the file descriptor returned by `open`.

Comments The `close` routine returns 0 when it closes a file successfully; otherwise, a runtime error is generated. Specifying an arbitrary integer not corresponding to a file descriptor as *file_des* causes `close` to generate a runtime error.

Any non-persistent open files not closed by `close` are automatically closed when the virtual user script completes. All open files, including persistent files, are closed at the end of a run. Your script cannot close standard input, output, error, record, and log files; any attempt to close one of them generates a runtime error.

Example This example declares the variable `theline` as a string. It then does the following:

- Opens `data_file` for reading and assigns it the file descriptor `file1`.
- Positions the character pointer so that each user reads a different line. File pointer for user 1 is 80 (`_uid*80`) bytes from the beginning of the file, file pointer for user 2 is 160 bytes from the beginning of the file, and so on.
- Reads an entire line (anything but a new line followed by a new line) and stores it in `theline`.

```
string theline;
file1=open("data_file","r");
fseek(file1, (_uid*80),0);
fscanf(file1, "%[^\n]\n", &theline);
close(file1);
```

See Also `open`

continue

Flow Control Statement

Description Skips remaining statements in a loop and continues with the next iteration of the loop.

Syntax

continue [*level_constant*]

Syntax Element	Description
level_constant	An optional integer that specifies how many nested loop levels to break out of.

Comments

The `continue` statement enables you to control the execution of `for`, `while`, and `do-while` loops.

As in C, if the `continue` statement is encountered in a `while` or `do-while` loop, the remaining statements in the loop are skipped, and execution continues with the evaluation step of the loop. If the `continue` statement is encountered in a `for` loop, the remaining statements in the loop are skipped, and execution continues with the increment step.

Unlike C, however, `continue` is specified with an optional argument, which allows it to affect a specified level of nested looping structures. Without this argument, or if the argument is 1, it behaves like its counterpart in C.

Example

In this example, if the value of `level_constant` is 1, the `continue` statement causes the program execution to skip execution of `loop_cnt = inner_cnt`. Execution continues at `inner_cnt <= 4`.

If the value of `level_constant` is 2, the `do-while` loop ends, the `print "Completed do-while"` statement is skipped, and execution continues at `j < cnt`.

If the value of `level_constant` is 3, both the `do-while` and `while` loops stop, the `printf` statement is skipped, and execution continues at `i++`.

```
cnt = inner_cnt = 0;
for (i = 0; i < 10; i++) {
    cnt++;
    j = 0;
    while (j < cnt) {
        j++;
        inner_cnt = j;
        do {
            inner_cnt++;
            continue level_constant;
            loop_cnt = inner_cnt;
        } while (inner_cnt <= 4);
        print "Completed do-while";
    }
    printf ("Now on iteration %d", i);
}
cnt *= 7;
```

See Also

`for`   `while`   `do-while`   `break`

COOKIE_CACHE

Statement

Description

Indicates the state of the cookie cache at the beginning of a session.

Syntax

```
COOKIE_CACHE
{
  name = value, domain, path [, secure];
  ...
}
```

Syntax Element	Description
name	A string constant giving the name of the cookie.
value	A string constant giving the value of the cookie.
domain	A string constant giving the domain for which the cookie is valid.
path	A string constant giving the path for which the cookie is valid.
secure	An optional string expression that, if given, provides the secure modifier for the cookie. The value of this parameter should be `"secure"`.

Comments

When you begin recording an http session, LoadTest queries your browser for any cookies that it has stored. These cookies are loaded into memory during script playback, thus making playback more accurate with respect to initial cookie values. This occurs automatically, but your VU script will contain a `COOKIE_CACHE` section.

This `COOKIE_CACHE` section reflects the state of the cookie cache at the beginning of a recording session. Automatically generated scripts have this section at the end of the script, but it may appear anywhere outside the main body of the script.

The cookies in the `COOKIE_CACHE` section are added to the user's cookie cache automatically before any commands in the script are executed. Cookies are created with expiration dates sufficiently in the future to ensure that they do not expire when you play back the script.

Example A cookie with the following data:

```
      Name: <AA002>
     Value: <00932743683-101023411/933952959>
      Path: <avenuea.com/>
    Secure: <0>
   Comment: <*>
    Expire: <Monday, 20-Jul-2009 00:00:00 GMT>
    Create: <Friday, 23-Jul-1999 15:27:31 GMT>
```

Appears in the `COOKIE_CACHE` as:

```
COOKIE_CACHE
{
    "AA002" = "00932743683-101023411/933952959",
   "avenuea.com", "/";
}
```

See Also `expire_cookie` `set_cookie`

ctos

Library Routine

Description Converts characters to strings.

Syntax `string` **`ctos`** `(`*`char`*`)`

Syntax Element	Description
char	An integer expression representing the character to convert.

Comments The `ctos` (character to string) routine returns a string of length one, containing the character *`char`* if *`char`* is nonzero; otherwise, `ctos` returns a string of length zero (`""`).

The `stoc` routine is the converse of `ctos`; `stoc` converts strings to characters.

Example These examples return the string `"a"`:

```
ctos("a");
ctos(256 + 'a');
```

This example returns the string `"\n"`:

```
ctos('\n');
```

These examples return the string `""`:

```
ctos('\0');
ctos(0);
```

See Also `stoc`

datapool_close

Datapool Function

Description Closes an open datapool.

Syntax

```
int datapool_close( datapool_id )
```

Syntax Element	Description
datapool_id	An integer expression returned by `datapool_open` specifying the datapool to close.

Comments If `datapool_close` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0.

Example This example opens `repo_pool` in the repository and then closes it:

```
dp_id = datapool_open ("repo_pool");
datapool_close (dp_id);
```

See Also `datapool_open`

DATAPOOL_CONFIG

Statement

Description Controls datapool creation and datapool access.

Syntax

```
DATAPOOL_CONFIG datapool_name flags
{
   directive, "col_name" [,"data_type" [,"data_value"]];
   ...
   directive, "col_name" [,"data_type" [,"data_value"]];
}
```

Syntax Element	Description
datapool_name	A string constant specifying the datapool name.

(Continued)

Syntax Element	Description
`flags`	Values that define the datapool access method. Choose at most one value from each of the following four groups: **`DP_WRAP`** or **`DP_NOWRAP`** Specifies what happens after the last row in the datapool row access order is reached: ▶ `DP_NOWRAP` – End access to the datapool. This is the default. If you attempt to retrieve a datapool value after the end of the datapool is reached, a runtime error occurs. ▶ `DP_WRAP` – Resume at the beginning of the access order. To ensure that unique datapool rows are fetched, specify `DP_NOWRAP`, and make sure that the datapool has at least as many rows as the number of users (and user iterations) that will request rows at runtime. **`DP_SHARED`** or **`DP_PRIVATE`** Specifies whether the datapool cursor is shared by all users accessing the datapool (`DP_SHARED`) or is unique to each user (`DP_PRIVATE`): ▶ `DP_SHARED` – With a shared cursor, all users work from the same access order. For example, if the access order for a Colors column is Red, Blue, and Green, the first user to request a value is assigned Red, the second is assigned Blue, and the third is assigned Green. This is the default. A shared cursor can also be persistent across schedule runs. Use the `DP_PERSISTENT` flag to make a shared cursor persistent. ▶ `DP_PRIVATE` – With a private cursor, each user starts at the top of its access order. With `DP_RANDOM` **or** `DP_SHUFFLE`, the access order is unique for each user and operates independently of the others. With `DP_SEQUENTIAL`, the access order is the same for each user (ranging from the first row in the file to the last).

(Continued)

Syntax Element	Description
	▶ DP_SEQUENTIAL, DP_RANDOM, or DP_SHUFFLE ▶ Determines datapool row access order (the sequence in which datapool rows are accessed): ▶ DP_SEQUENTIAL – Rows are accessed in the order in which they are physically stored in the datapool file, beginning with the first row in the file and ending with the last. This is the default. ▶ DP_RANDOM – Rows are accessed in any order, and any given row can be accessed multiple times or not at all. ▶ DP_SHUFFLE – Each time LoadTest rearranges, or "shuffles," the access order of all datapool rows, a unique sequence results. Each row is referenced in a shuffled sequence only once. **DP_PERSISTENT** Specifies that the datapool cursor is persistent across schedule runs. For example, if both the `DP_PERSISTENT` and `DP_SEQUENTIAL` flags are set, and datapool row number 100 was the last row accessed in the last schedule run, the first row accessed in the next schedule run is 101. A persistent cursor resumes row access based on the last time the cursor was accessed as a *persistent* cursor. For example, suppose a cursor is persistent, and the last row accessed for that cursor in a schedule run is 100. Then, the same schedule is run again, but the cursor is now private. Row access ends at 50. If the cursor is set back to persistent the next time the schedule is run, row access resumes with row 101, not 51. `DP_PERSISTENT` is only valid when the `DP_SHARED` flag exists and when either the `DP_SEQUENTIAL` or `DP_SHUFFLE` flag exists.
	OVERRIDE or **EXCLUDE** Specifies whether you want to use an optional global directive to override the individual directives specified in *`directive`*: ▶ `OVERRIDE` – The `OVERRIDE` directive is applied globally to all datapool columns. This is the default. ▶ `EXCLUDE` – The `EXCLUDE` directive is applied globally to all datapool columns. These values allow the script to ignore `datapool_open` and `datapool_fetch` calls. As a result, these values let you run the script even if the datapool file is missing. See the *`directive`* argument for more information about these values.

(Continued)

Syntax Element	Description
directive	A keyword that specifies the columns to add to the datapool as well as the source of values returned by the function `datapool_value`: ▶ `INCLUDE` – During datapool creation, creates a datapool column for *col_name*. The column is assigned the same name. – During schedule runtime, `datapool_value` returns a value for *col_name* from the corresponding datapool column. ▶ `EXCLUDE` – During datapool creation, does not create a datapool column for *col_name*. When the *flags* value contains `EXCLUDE`, no datapool is created. – During schedule runtime, `datapool_value` returns a value for *col_name* from the recorded value in *data_value*, not from the datapool. ▶ `OVERRIDE` – During datapool creation, creates a datapool column for *col_name*. The column is assigned the same name. – During schedule runtime, `datapool_value` returns a value for *col_name* from the recorded value in *data_value*, not from the datapool. You can override all of the directives in this column by specifying the *flags* value `OVERRIDE` or `EXCLUDE`. These global values treat all columns in the configuration section as either `OVERRIDE` or `EXCLUDE`.
col_name	The name of the datapool item. If a datapool column is created for this item (if *directive* is either `INCLUDE` or `OVERRIDE`), the datapool column is assigned the same name.
data_type	The data type of the value in *data_value* column. The value is always `string`.
data_value	A value that was provided during recording. The function `datapool_value` supplies *col_name* with a recorded value rather than a datapool value if the directive `OVERRIDE` or `EXCLUDE` is specified.

Comments

If you select **Use datapools** on the **Generator** tab of the Virtual User Record Options dialog box, Robot automatically includes a `DATAPOOL_CONFIG` statement in the script that it generates after recording.

To edit a DATAPOOL_CONFIG statement through the Robot user interface, click **Edit** → **Datapool Information**.

Think of non-sequential access order (DP_SHUFFLE and DP_RANDOM) as being like a shuffled deck of cards. With DP_SHUFFLE, each time you pick a card (access a row), you place the card at the bottom of the pack. But with DP_RANDOM, the selected card is returned anywhere in the pack — which means that one card might be selected multiple times before another is selected once.

Also, with DP_SHUFFLE, after each card has been selected once, you either resume selecting from the top of the same access order (DP_WRAP), or no more selections are made (DP_NOWRAP).

With DP_RANDOM, you never reach the end of the pack (there is no end-of-file condition, so DP_WRAP and DP_NOWRAP are ignored).

In a private cursor with DP_SEQUENTIAL access order, you typically have each user run multiple instances of the script. If each user runs a single iteration of the script, each would access the same datapool row (the first row in the datapool).

The following are the possible *flags* combinations that affect datapool access. These combinations include all *flags* values except OVERRIDE and EXCLUDE.

- DP_SHARED DP_SHUFFLE DP_WRAP

  LoadTest calculates a unique row access order for all users to share. After a user reaches the last row in the access order, the next user resumes access with the first row.

- DP_SHARED DP_SHUFFLE DP_WRAP DP_PERSISTENT

  Same as above, but the cursor is also persistent across schedule runs. For example, suppose row number 14 immediately follows row number 128 in the shuffled access order. If the last row accessed in the current schedule run is row 128, the first row accessed in the next schedule run is 14.

- DP_SHARED DP_SHUFFLE DP_NOWRAP

  LoadTest calculates a unique row access order for all users to share. After the last row in the access order is reached, access to the datapool ends.

- DP_SHARED DP_SHUFFLE DP_NOWRAP DP_PERSISTENT

  Same as above, but the cursor is also persistent across schedule runs. For example, suppose row number 14 immediately follows row number 128 in the shuffled access order. If the last row accessed in the current schedule run is row 128, the first row accessed in the next schedule run is 14.

- `DP_PRIVATE DP_SHUFFLE DP_WRAP`

  Loadtest calculates a unique row access order for each user. After a user reaches the last row in its access order, it resumes access with the first row.

- `DP_PRIVATE DP_SHUFFLE DP_NOWRAP`

  LoadTest calculates a unique row access order for each user. After a user reaches the last row in its access order, access to the datapool ends.

- `DP_SHARED DP_RANDOM`

  LoadTest calculates a random access order that all users share. A given row can appear in the access order multiple times. Because no end-of-file condition is possible, `DP_WRAP` and `DP_NOWRAP` are ignored.

- `DP_PRIVATE DP_RANDOM`

  LoadTest calculates a unique random access order for each user. A given row can appear in the access order multiple times. Because no end-of-file condition is possible, `DP_WRAP` and `DP_NOWRAP` are ignored.

- `DP_SHARED DP_SEQUENTIAL DP_WRAP`

  LoadTest provides all users with the same sequential access to datapool rows, starting with the first row in the datapool file and ending with the last. After a user reaches the last row in the datapool, the next user resumes access with the first row.

- `DP_SHARED DP_SEQUENTIAL DP_WRAP DP_PERSISTENT`

  Same as above, but the cursor is also persistent across schedule runs. For example, if the last row accessed in the current schedule run is row 128, the first row accessed in the next schedule run is 129.

- `DP_SHARED DP_SEQUENTIAL DP_NOWRAP`

  LoadTest provides all users with the same sequential access to datapool rows, starting with the first row in the datapool file and ending with the last. After the last row in the sequence is reached, access to the datapool ends.

- `DP_SHARED DP_SEQUENTIAL DP_NOWRAP DP_PERSISTENT`

  Same as above, but the cursor is also persistent across schedule runs. For example, if the last row accessed in the current schedule run is row 128, the first row accessed in the next schedule run is 129.

- `DP_PRIVATE DP_SEQUENTIAL DP_WRAP`

  LoadTest provides each user with individual sequential access to datapool rows, starting with the first row in the datapool file and ending with the last. After a user accesses the last row in the sequence, it resumes access with the first row in the sequence.

- `DP_PRIVATE DP_SEQUENTIAL DP_NOWRAP`

  LoadTest provides each user with individual sequential access to datapool rows, starting with the first row in the datapool file and ending with the last. After a user accesses the last row in the sequence, the user's access to the datapool ends.

Comments are not allowed in the `DATAPOOL_CONFIG` section of a script.

Commas (,) double-quotes ("), and carriage return and line feed characters cannot be used in keywords, names, or recorded values in the `DATAPOOL_CONFIG` section of a script.

Example

This example shows a `DATAPOOL_CONFIG` statement for a datapool named `CD_ORDER`. The datapool is accessed by an application that lets a customer order CDs from a music retailer.

This first line of the example contains the datapool name and the flags that define how the datapool is accessed when the script is played back in LoadTest.

Each subsequent line has four columns of information, separated by commas. These lines serve as a datapool blueprint, giving Robot the information it needs to create the datapool. During script playback, these lines also tell Loadtest where to look for values to assign the variables in the script.

In this example, a datapool column is generated for every variable listed except the last one, *xV010*. Also, during script playback, LoadTest assigns a datapool value to each variable listed except for *xV006* and *xV010*. These two variables are assigned the values `12/31/99` and `Order Initiated`, respectively, each time the script is executed.

```
DATAPOOL_CONFIG "CD ORDER"DP_NOWRAP DP_SEQUENTIAL DP_SHARED
{
    INCLUDE, "CUSTID", "string", "329781";
    INCLUDE, "PRODUCTS_COMPOSER", "string", "Bach";
    INCLUDE, "PRODUCTS_COMPOSER_4", "string", "Schubert";
    INCLUDE, "PRODUCTS_COMPOSER_3", "string", "Mozart";
    INCLUDE, "PRODUCTS_COMPOSER_2", "string", "Haydn";
    INCLUDE, "PRODUCTS_COMPOSER_1", "string", "Beethoven";
    INCLUDE, "xV001", "string", "33822";
    INCLUDE, "xV001_2", "string", "87";
    INCLUDE, "xV001_1", "string", "99383";
    INCLUDE, "xV002", "string", "2";
    INCLUDE, "xV003", "string", "10-APR-1998";
    INCLUDE, "xV004", "string", "MasterCard";
    INCLUDE, "xV005", "string", "1234567890000";
    OVERRIDE, "xV006", "string", "12/31/99";
    INCLUDE, "xV007", "string", "99383";
    INCLUDE, "xV008", "string", "2";
    INCLUDE, "xV009", "string", "$35.98";
    EXCLUDE, "xV010", "string", "Order Initiated";
}
```

See Also `datapool_open`

datapool_fetch

Datapool Function

Description Moves the datapool cursor to the next row.

Syntax `int` **`datapool_fetch`**`(`*`datapool_id`*`)`

Syntax Element	Description
datapool_id	An integer expression returned by `datapool_open` and representing an open datapool.

Comments If `datapool_fetch` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0.

`datapool_fetch` retrieves the next row in the datapool. The “next row” in the datapool is determined by the *`flags`* you set in the `DATAPOOL_CONFIG` section of the script or in the `datapool_open` command.

If cursor wrapping is disabled, and the last row of the datapool has been retrieved, a call to `datapool_fetch` returns an error. If `datapool_value` is then called, a runtime error occurs. (Cursor wrapping is disabled when the *flags* argument of `DATAPOOL_CONFIG` or `datapool_open` includes `DP_NOWRAP`.)

Example This example opens a datapool, fetches the next record in the datapool, and then closes the datapool:

```
dp_id = datapool_open ("repo_pool");
datapool_fetch(dp_id);
datapool_close (dp_id);
```

See Also `datapool_open` `datapool_rewind` `datapool_value`

datapool_open

Datapool Function

Description Opens the specified datapool and defines the datapool's row access order.

Syntax `int` **`datapool_open`** `(`*`datapool_name`*` [, `*`flags`*` ])`

Syntax Element	Description
datapool_name	The name of the datapool to open.
flags	Flags that define how the datapool is accessed when the script is played back in a LoadTest schedule. If you do not specify any values for *flags*, row access order is determined by the *flags* value of `DATAPOOL_CONFIG`. This is the preferred method for providing *flags* values. If you do define *flags* in `datapool_open`, it cannot contradict the values set in `DATAPOOL_CONFIG`. For example, if `DATAPOOL_CONFIG` does not specify the datapool access method (`DP_SEQUENTIAL` or `DP_RANDOM`), you can specify it as DP_SHUFFLE in the `datapool_open`. However, if `DATAPOOL_CONFIG` declares a datapool cursor as `DR_PRIVATE`, you cannot open it with `DP_SHARED`. For details about *flags* values, see the description of this argument in the `DATAPOOL_CONFIG` statement.

Comments

datapool_open returns a datapool identifier that other datapool functions use to perform operations on the datapool. Upon failure, the function returns 0.

The cursor for a datapool opened for shared access (DP_SHARED) is initialized by LoadTest once for an entire schedule run. When initializing a datapool cursor opened for both shared and persistent access (DP_SHARED and DP_PERSISTENT), LoadTest sets the row pointer to the next row in the row access order — that is, to the row that immediately follows the last row accessed in the last schedule run where the cursor was persistent.

The cursor for a datapool opened for private access (DP_PRIVATE) is initialized by each user once for an entire schedule run. When initializing a datapool cursor opened for private access, LoadTest sets the row-pointer to the first datapool row in the row access order.

With a private-access datapool, closing the datapool with datapool_close, and then reopening the same datapool with another call to datapool_open with the same flags and in the same or a subsequent script, resumes access to the datapool as if it had never been closed.

If multiple users (GUI users and/or virtual users) access the same datapool in a LoadTest schedule, the datapool cursor is managed as follows:

- For shared cursors, the first call to datapool_open initializes the cursor. In the same schedule run (and, with the DP_PERSISTENT flag, in subsequent schedule runs), users that subsequently call datapool_open to open the same datapool share the initialized cursor.
- For private cursors, the first call to datapool_open initializes the user's private cursor. In the user's subsequent calls to datapool_open in the same schedule run, the cursor is set to the last row accessed by that user.

Example

This example declares a datapool from the customer file. At declaration, access to the datapool is sequential, and DP_WRAP or DP_NOWRAP is unspecified. The datapool is opened to reuse records:

```
DATAPOOL_CONFIG "repo_pool" DP_SHARED DP_SEQUENTIAL
{
     INCLUDE, "column1", "string";
     INCLUDE, "column2", "string";
     INCLUDE, "column3", "string";
}

dp_id = datapool_open ( "repo_pool", DP_WRAP );
```

See Also

DATAPOOL_CONFIG	datapool_fetch	datapool_value
datapool_close	datapool_rewind	

datapool_rewind

Datapool Function

Description Resets the datapool cursor to the start of the datapool access order.

Syntax `int datapool_rewind( datapool_id )`

Syntax Element	Description
datapool_id	An integer expression returned by `datapool_open` and representing an open datapool.

Comments This command rewinds the private cursor for the datapool referenced by the datapool_id. If `datapool_rewind` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0.

The datapool is rewound as follows:

- For datapools opened `DP_SEQUENTIAL`, `datapool_rewind` resets the cursor to the first record in the datapool file.
- For datapools opened `DP_RANDOM` or `DP_SHUFFLE`, `datapool_rewind` restarts the random number sequence.
- For datapools opened `DP_SHARED`, `datapool_rewind` has no effect.

At the start of a schedule, datapool cursors always point to the first row.

If you rewind the datapool during a schedule run, previously accessed rows are fetched again.

Example This example shows a datapool configured with the defaults, opened for private access, and then rewound.

```
DATAPOOL_CONFIG "repo_pool" DP_NOWRAP DP_SEQUENTIAL
{
     INCLUDE, "column1", "string";
     INCLUDE, "column2", "string";
     INCLUDE, "column3", "string";
}

dp_id = datapool_open ( "repo_pool", DP_PRIVATE );
datapool_rewind (dp_id);
```

See Also `datapool_fetch`

datapool_value

Datapool Function

Description Retrieves the value of the specified datapool column.

Syntax `string` **`datapool_value`**`(` *`datapool_id, column`* `)`

Syntax Element	Description
datapool_id	An integer expression returned by `datapool_open` and representing an open datapool.
column	A string that specifies the name of the datapool column to retrieve. The name must match a datapool column name listed in the TestManager Datapool Specification dialog box. Column names are case sensitive.

Comments `datapool_value` returns the string value of the specified column.

If cursor wrapping is disabled, and the last row of the datapool has been retrieved, a call to `datapool_fetch` returns an error. If `datapool_value` is then called, a runtime error occurs. (Cursor wrapping is disabled when the *`flags`* argument of `DATAPOOL_CONFIG` or `datapool_open` includes `DP_NOWRAP`.)

You can retrieve a value even if the datapool column has been excluded from the datapool (through the `EXCLUDE` directive in `DATAPOOL_CONFIG`). In this case, the value retrieved is the recorded value contained in the *`data_value`* argument of the `DATAPOOL_CONFIG` statement.

Example This example retrieves the value of `"column3"` and stores it in *`dp_value`*:

```
DATAPOOL_CONFIG "repo_pool" DP_NOWRAP DP_SHARED DP_SEQUENTIAL
{
     INCLUDE, "column1", "string";
     INCLUDE, "column2", "string";
     INCLUDE, "column3", "string";
}

dp_id = datapool_open ( "repo_pool" DP_WRAP );
datapool_fetch(dp_id);
dp_value = datapool_value(dp_id, "column3");
```

See Also `datapool_fetch`

delay

Library Routine

Description Delays script execution for a specified time period.

Syntax `int **delay** (m_time)`

Syntax Element	Description
m_time	An integer expression specifying the delay in milliseconds. This is subject to scaling by the environment variable `Delay_dly_scale`.

Comments The `delay` routine returns, as an integer, the number of milliseconds actually delayed. If *m_time* is ≤ 0, `delay` returns 0 immediately.

The `delay` routine delays script execution for a specified time period before continuing. When this time period has elapsed, execution continues with the next statement.

Your system may round the delay to a lower resolution, typically in the range of 10 to 20 milliseconds.

Example This example sets a random delay. It first defines a maximum delay of 10 seconds, and then delays a random amount of time from 0 to 10 seconds:

```
#define MaxDelay 10

(
    delay_time = rand() % (MaxDelay + 1);
    delay(delay_time * 1000);
}
```

See Also None.

display

Library Routine

Description Provides a string to the monitor for display in message view.

Syntax `int **display** (str)`

Syntax Element	Description
str	A string expression to be displayed by monitor.

Comments The `display` routine always returns 1 for success. `display` accepts any string expression, but the length of the string is truncated to 20 characters when monitoring a schedule.

This function is most useful as a script debugging tool because it allows a short message to be easily viewed in real time.

Example

```
display ("beginning transaction");
```

See Also None.

do-while

Flow Control Statement

Description Repeatedly executes a VU statement while a condition is true.

Syntax

```
do
        statement1;
while (exp1);
```

Syntax Element	Description
statement1	One or more VU language statements enclosed in braces.
exp1	The integer expression to evaluate.

Comments The `do-while` loop is executed in the following steps:

1. *statement1* is executed.
2. *exp1* is evaluated.
3. If the value of *exp1* is not 0, steps 1 and 2 are repeated. If the value of *exp1* is 0, execution of the `while` loop ends.

Example This example reads and prints a string from a file whose file descriptor is `file_des.` Execution continues until the end of the file is reached.

```
do
{
  if (fscanf(file_des, "%s", &key)==1)
       printf("Key is <%s>\n" key);
}
while (!feof(file_des))
```

See Also `for` `while`

else-if

Flow Control Statement

Description Conditionally executes a VU statement.

Syntax

```
if (exp1)
          statement1;
else if (exp2)
          statement2;
...
else if (expn)
         statementn;
else
         statementx;
```

Syntax Element	Description
exp1, exp2, expn	An integer expression whose value determines whether the corresponding statement is executed. If the value is 0, the statement is not executed.
statement1, statement2, statementn, statementx	VU language statements that are executed conditionally.

Comments The `else-if` structure follows these conventions:

- If the value of *exp1* is not 0, only *statement1* is executed.
- If *exp1* is 0 and the value of *exp2* is not 0, only *statement2* is executed.
- If *exp1, exp2 ... expn-1* are 0 and the value of *expn* is not 0, only *statementn* is executed.
- If all of *exp1, exp2 ... expn* are 0, then only *statementx* is executed. The final `else` is omitted if no action is required when all of *exp1, exp2 ... expn* are 0.

As with the `if-else` structure, if a statement is replaced by multiple VU language statements, all statements are enclosed in braces.

The indentation is optional but recommended.

Example In this example, one of three options are possible. If x is less then target, the string “too small” is printed. If x is greater than target, the string “too large” is printed; otherwise, the string “just right!” is printed.

```
if (x < target)
     printf("too small\n");
else if (x > target)
     printf("too large\n");
else
     printf("just right!\n");
```

See Also if-else

emulate

Send Emulation Command

Description Provides generic emulation command services to support a proprietary protocol.

Syntax int **emulate** [*cmd_id*] *condition* [, *log_string* [, *fail_string*]]

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
condition	An integer expression. If the value of *condition* is > 0, the emulate command passes; otherwise, it fails. emulate returns the value of *condition.* Typically, *condition* is a VU function or an external C function. *condition* is executed before evaluation of *log_string* and *fail_string*. Therefore, either string could contain variables set during execution of *condition.*
log_string	An optional string expression used when logging a passed emulate command, or a failed, emulate command if *fail_string* is not provided. If *log_string* is not specified, no log entry is generated for emulate. Either *log_string* or *fail_string* is evaluated, but not both.
fail_string	An optional string expression used when logging a failed emulate command. If *fail_string* is not specified, *log_string* is used for both pass and fail cases. Either *log_string* or *fail_string* is evaluated, but not both.

Comments

The `emulate` command returns the value of *`condition`*.

The `emulate` command provides generic emulation command services to VU or external C function calls. This extends VU emulation support to proprietary protocols or interfaces. You can use the `emulate` command as a wrapper for external C function calls, and thus obtain the full set of services associated with the standard emulation commands.

> **NOTE:** VU supports the SAP protocol by using external C functions and the `emulate` command. For information about the SAP protocol, see Appendix B.

The external C dynamic-link library (shared library on UNIX Agents) contains the C functions to perform the desired client-side API functions that access the server. These C functions are wrapped in the `emulate` command, so that the results and timing of the API functions are paced, recorded, logged, and made available for analysis by LoadTest reports.

The C code generally performs response verification and error detection, and passes an integer return code to `emulate`.

The `emulate` command is affected by the following VU environment variables: the think time variables, `Log_level`, `Record_level`, `Suspend_check`, `Timeout_val`, `Timeout_scale`, and `Timeout_act`.

For more information, see *Accessing External C Data and Functions* on page 4-9.

Example

In this simple example, `api_x` is called with two string constants and an integer constant. No logging is performed, but if `api_x` returns a value > 0, the command is recorded as passed in the virtual user's record file; otherwise, it is recorded as failed. The label associated with the command is `action 1`. The response time is the time from calling `api_x` until it returns.

```
emulate["action 1"] api_x("John Doe", "$100.43", 4);
```

In this more complete example, an API has been linked into a dynamic-link library. The virtual user script calls the API with an `emulate` wrapper.

The API is a simple interface to a school database. The API consists of:

- An `open` function, which takes a student's name and returns a handle to that student's record.
- A `cmd` function, which performs operations on the records.
- A `close` function, which releases the record handle.

The actual C code for the shared library includes a wrapper C function for each API call; each call has the prefix `my`. The dynamic-link library creates the log message for each API call.

The header file, myAPI.h, is included in the virtual user script. The header file defines three constants that are used by the API, and makes the C string api_logmsg, and functions myapi_open, myapi_cmd, and myapi_close available to the virtual user script:

```
#define REGISTER_CLASS 1
#define ASSIGN_GRADE   2
#define REVISE_GRADE   3

external_c string api_logmsg;

external_c func myapi_open(name, student_handle)
    string name;
    reference int student_handle;
{}

external_C func myapi_cmd(student_handle, command, sval, ival)
    int student_handle;
    int command;
    string sval;
    int val;
{}

external_C func myapi_close(student_handle)
    int student_handle;
{}
```

The virtual user script has an emulate command for each API call, and references the shared external C string api_logmsg to log the results. The script opens the record for Joe Smith, returns the handle needed by subsequent calls (handle_1), assigns two grades, and closes the record. A think time has been added to simulate user processing:

```
#include <VU.h>
#include <myAPI.h>

{
set Think_avg = 3000;
emulate ["step001"] myapi_open("Joe Smith", &handle_1), api_logmsg;
emulate ["step002"] myapi_cmd(handle_1, ASSIGN_GRADE, "Biology",
94), api_logmsg;
emulate ["step003"] myapi_cmd(handle_1, ASSIGN_GRADE, "Chemistry",
82), api_logmsg;
emulate ["step004"] myapi_close(handle_1), api_logmsg;
}
```

See Also testcase

eval

Environment Control Command

Description Returns the value and data type at the top of a VU environment variable's stack.

Syntax *type* **eval** *env_var*;

Syntax Element	Description
type	int, string, or bank depending on type of *env_var*.
env_var	Any VU environment variable defined as a integer, string, or bank.

Comments The eval command returns an expression having the same type as *env_var* (integer, string, or bank) and the current value of *env_var*. The value of *env_var* is not altered.

Example In this example, values for Timeout_val and Log_level are set. The integer value 2000 is assigned to the variable t. Then, the integer value 1 is assigned to the variable e, because the expression (eval Log_level == "ALL") is true. The value of Timeout_val and Log_level remain unchanged.

```
set [Timeout_val = 2000, Log_level="ALL"];
t = eval Timeout_val;
e=(eval Log_level=="ALL");
```

See Also None.

expire_cookie

Emulation Function

Description Expires a cookie in the cookie cache.

Syntax **expire_cookie**(*name*, *domain*, *path*)

Syntax Element	Description
name	A string expression that specifies the name of the cookie.
domain	A string expression that specifies the domain for which this cookie is valid.
path	A string expression that specifies the path for which this cookie is valid.

Comments The `expire_cookie` function causes the named cookie to no longer be valid for the given domain and path. This effectively removes the cookie from the cache.

Example This example expires the cookie named `AA002` for domain `avenuea.com` and path `/`.

```
expire_cookie("AA002", ".avenuea.com", "/");
```

See Also `COOKIE_CACHE` `set_cookie`

feof

Library Routine

Description Determines if the end of a file was encountered.

Syntax `int` **`feof`** `(`*`file_des`*`)`

Syntax Element	Description
file_des	The integer file descriptor of the file to check. The file descriptor was returned from `open`.

Comments The `feof` routine returns a nonzero value if the end of file has previously been detected reading the named input file; otherwise, `feof` returns zero.

The related routines `fseek` repositions the file pointer and `ftell` returns information on the file pointer.

Example In this example, if the file with the descriptor *`infile_des`* contains the characters `abcde`, then the characters `abcde` are written to the file whose descriptor is `outfile_des` ten times. At the end of the example, the variables *`copies`* and *`total`* have values of 10 and 50, respectively:

```
fseek(file_des, 0, 2);
for (copies = total = 0; copies < 10; copies++)
{
    while (1)
    {
        c = fgetc(infile_des);
        if (feof(infile_des))
        {
            total += ftell(infile_des);
            fseek(infile_des, 0, 0); /* rewind */
            break;
        }
       else
           fputc(c, outfile_des);
    }
}
```

See Also `fseek` `ftell`

fflush

Library Routine

Description

Causes any buffered data for a file to be written to that file.

Syntax

`int` **`fflush`** `(`*`file_des`*`)`

Syntax Element	Description
file_des	The integer file descriptor, obtained from the `open`, the file to flush.

Comments

The `fflush` routine returns zero for success, or EOF (as defined in the standard VU header file) upon encountering an error. All VU files except standard error are buffered for efficiency.

`fflush` temporarily overrides the buffering mechanism by writing the buffered data to the named file. This is particularly useful for ensuring timely output of status messages, as shown in the following example.

Example

This example writes the strings `"Processing Phase 1", "2 ", "3 ", "4 ", "5 "`, and `"DONE\n"` to be successively written to the standard output file immediately as each respective phase is processed, instead of waiting until the file is closed or the current output buffer is filled.

```
for (phase_no = 1; phase_no <= 5; phase_no++)
{
    if (phase_no == 1)
        printf("Processing Phase ");
    printf("%d ", phase_no);
    fflush(stdout);
    do_phase(phase_no);
}
printf("DONE\n");
fflush(stdout);
```

See Also

None.

fgetc

Library Routine

Description Provides unformatted character input capability.

Syntax `int` **`fgetc`** `(`*`file_des`*`)`

Syntax Element	Description
file_des	The integer file descriptor, obtained from `open`, that refers to the file to read.

Comments The `fgetc` routine returns the next character, as an integer, from the named file. This provides a shortened, more efficient alternative to the `fscanf` routine for the case where only a single character needs input. `fgetc` returns EOF (as defined in the standard VU header file) at end-of-file or upon an error.

Example In this example, assume the file with the descriptor `infile_des` contains the characters `ABZ14`. The characters `ABZ` are written to the file whose descriptor is `outfile_des`, and the character 1 is returned to the input buffer associated with `infile_des`.

```
#include <VU.h>
while ((c = fgetc(infile_des)) != EOF)
if (c >= 'A' && c <= 'Z')
    fputc(c, outfile_des);
else
{
    ungetc(c, infile_des);
    break;
}
```

See Also `ungetc`

for

Flow Control Statement

Description Repeatedly executes a VU statement.

Syntax

```
for (exp1; exp2; exp3)
        statement1;
```

Syntax Element	Description
exp1, exp3	A VU language expression.
exp2	An integer expression to evaluate.
statement1	A VU language statement. You can include multiple VU language statements if all of the statements are enclosed in braces and terminated by semicolons.

Comments

The execution of the `for` loop occurs in the following steps:

1. *exp1* is evaluated.
2. *exp2* is evaluated and if its value is not 0, *statement1* is executed. If its value is 0, execution of the `for` loop ends.
3. If the execution of the `for` loop has not ended, *exp3* is evaluated.
4. Steps 2 and 3 are repeated until execution of the `for` loop ends.

Example

This example prints out a line 10 times:

```
for (i=0; i<10; i++)
    printf ("this line is displayed 10 times\n");
```

See Also

`while` `do-while`

fputc, fputs

Library Routine

Description

Writes unformatted output for characters or strings.

Syntax

```
int fputc (out_char, file_des)
int fputs (out_str, file_des)
```

Syntax Element	Description
out_char	An integer expression (interpreted as a character) that specifies the character to write.
out_str	A string expression that specifies the string to write.
file_des	The integer file descriptor, obtained from `open`, of the file to receive the output.

Comments The `fputc` and `fputs` routines provide a shortened, more efficient alternative to the `fprintf` routine when only a single character or string needs to be output.

Example In this example, assume that the value of `char1` is `M`. Therefore, the character `M` is written to the file whose descriptor is `outfile_des`.

```
fputc(char1, outfile_des);
```

In this example, assume that the value of the string expression `str1` is `xyz`. Therefore, the characters `xyz` are written to the file whose descriptor is `outfile_des`.

```
fputs(str1, outfile_des);
```

See Also `fprintf`

FreeAllData

VU Toolkit Function: Data

Description Frees all data sets saved with `SaveData` and `AppendData`.

Syntax

```
#include <sme/data.h>
proc FreeAllData()
```

Comments The `FreeAllData` procedure frees all data sets saved using `SaveData` and `AppendData`.

Example This example saves the data in the `tmp_results` buffer, stores the second field in `accessprofile_id`, then frees all the data.

```
#include <VU.h>
#include <sme/data.h>

{
  string accessprofile_id;

  sqlexec ["test_gr003"]
    "select PASSWORD, ACCESSPROFILEID, INACTIVE, "
    "PW_UPDATE_DT from USERACCOUNT where NAME = 'davidj'";
  sqlnrecv  ["test_gr004"] ALL_ROWS;

  SaveData ("tmp_results");
  accessprofile_id = GetData1("tmp_results", 2);
  FreeAllData ();

  sqlexec  ["test_gr005"]
     "select LOGONNAME, LOGONPASSWORD, EXP_DAYS from "
     "ACCESSPROFILE where ACCESSPROFILEID = "
     + accessprofile_id;
}
```

See Also `AppendData` `FreeData` `GetData` `GetData1` `SaveData`

FreeData

VU Toolkit Function: Data

Description Frees specified data sets saved with `SaveData` and `AppendData`.

Syntax

```
#include <sme/data.h>
proc FreeData(data_name)
string data_name;
```

Syntax Element	Description
data_name	The name of the data set to free.

Comments The `FreeData` function frees the data set associated with *data_name*, where the named data set was created using the `SaveData` or `AppendData` functions.

Example This example saves the data in the `tmp_results` buffer, stores the second field in `accessprofile_id`, then frees `tmp_results`.

```
#include <VU.h>
#include <sme/data.h>

{
  string accessprofile_id;

  sqlexec ["test_gr003"]
    "select PASSWORD, ACCESSPROFILEID, INACTIVE, "
    "PW_UPDATE_DT from USERACCOUNT where NAME = 'davidj'";
  sqlnrecv  ["test_gr004"] ALL_ROWS;

  SaveData ("tmp_results");
  accessprofile_id = GetData1("tmp_results", 2);
  FreeData ("tmp_results");

  sqlexec  ["test_gr005"]
     "select LOGONNAME, LOGONPASSWORD, EXP_DAYS from "
     "ACCESSPROFILE where ACCESSPROFILEID = "
     + accessprofile_id;
}
```

See Also `AppendData` `FreeAllData` `GetData` `GetData1` `SaveData`

fseek

Library Routine

Description

Repositions the file pointer.

Syntax

`int` **`fseek`** `(`*`file_des`*`, `*`offset`*`, `*`position`*`)`

Syntax Element	Description
file_des	The integer file descriptor, obtained from `open`, of the file whose pointer you want to reposition.
offset	An integer expression that indicates the number of bytes that the file pointer is to move. The offset can be a negative number.
position	An integer expression that indicates whether the offset is from the beginning of the file (if *position* equals 0), from the current position (if *position* equals 1), or from the end of the file (if *position* equals 2).

Comments

The `fseek` routine returns zero for successful seeks, and nonzero for unsuccessful seeks.

The related routines `feof` and `ftell` return information about the file pointer.

Example

In this example, `fseek` repositions the file pointer of the file whose descriptor is `file_des` to the beginning of the file:

```
fseek(file_des, 0, 0);
```

In this example, if the current file pointer offset is 45, `fseek` repositions the file pointer of the file whose descriptor is `file_des` to an offset of 35:

```
fseek(file_des, -10, 1);
```

In this example, `fseek` repositions the file pointer of the file whose descriptor is `file_des` to the end of the file:

```
fseek(file_des, 0, 2);
```

See Also

`feof`   `ftell`

ftell

Library Routine

Description

Returns the file pointer's offset in the specified file.

Syntax

`int **ftell** (*file_des*)`

Syntax Element	Description
file_des	The integer file descriptor, obtained from `open`, of the file whose pointer you want to obtain.

Comments

The `ftell` routine returns the current byte's offset on the named file. This offset is relative to the beginning of the file.

The related routines `fseek` repositions the file pointer and `feof` returns information on the file pointer.

Example

In this example, if the file with the descriptor `infile_des` contains the characters `abcde`, then the characters `abcde` are written to the file whose descriptor is `outfile_des` ten times. At the end of the example, the variables `copies` and `total` have values of 10 and 50, respectively:

```
fseek(file_des, 0, 2);
for (copies = total = 0; copies < 10; copies++)
{
    while (1)
    {
        c = fgetc(infile_des);
        if (feof(infile_des))
        {
            total += ftell(infile_des);
            fseek(infile_des, 0, 0); /* rewind */
            break;
        }
        else
            fputc(c, outfile_des);
    }
}
```

See Also

`feof`   `fseek`

GetData

VU Toolkit Function: Data

Description Retrieves a specific row from the dataset created with `SaveData` or `AppendData`.

Syntax

```
#include <sme/data.h>
string func GetData(data_name, row, column)
string data_name;
int row;
int column;
```

Syntax Element	Description
data_name	The name of the data set to retrieve.
row	The row of *data_name* to retrieve.
column	The column of *data_name* to retrieve.

Comments The `GetData` function retrieves a data value from a specific row and column of a data set created with the `SaveData` or `AppendData` functions. Regardless of the database definition of the column, the returned value is a string. Returned values are of variable length, with any trailing white space trimmed from the end of the value.

A null string is returned if no data is saved under this name, or if the row or column values exceed the limits of the stored data.

Example This example saves the data in the `tmp_results` buffer, and gets the second field in the first row of `tmp_results`.

```
#include <VU.h>
#include <sme/data.h>

{
  string accessprofile_id;

  sqlexec ["test_gr003"]
    "select PASSWORD, ACCESSPROFILEID, INACTIVE, "
    "PW_UPDATE_DT from USERACCOUNT where NAME = ’davidj’";
  sqlnrecv  ["test_gr004"] ALL_ROWS;

  SaveData ("tmp_results");
  accessprofile_id - GetData("tmp_results", 1, 2);
  FreeData ("tmp_results");

  sqlexec  ["test_gr005"]
     "select LOGONNAME, LOGONPASSWORD, EXP_DAYS from "
     "ACCESSPROFILE where ACCESSPROFILEID = "
     + accessprofile_id;
}
```

See Also `AppendData` `FreeAllData` `FreeData` `GetData1` `SaveData`

GetData1

VU Toolkit Function: Data

Description

Retrieves a value in the first row of a data set created with `SaveData` or `AppendData`.

Syntax

```
#include <sme/data.h>
string func GetData1(data_name, column)
string data_name;
int column;
```

Syntax Element	Description
data_name	The name of the data set to retrieve.
column	The column of *data_name* to retrieve.

Comments

The `GetData1` function retrieves a data value from a specific column of the first row of a data set created with the `SaveData` or `AppendData` functions. To retrieve data from a different row, use the `GetData` function. Regardless of the database definition of the column, the returned value is a string. Returned values are of variable length, with any trailing white space trimmed from the end of the value.

A null string is returned if no data is saved under this name, or if the row or column values exceed the limits of the stored data.

Example

This example saves the data in the `tmp_results` buffer, and gets the second field in the first row of `tmp_results`.

```
#include <VU.h>
#include <sme/data.h>

{
  string accessprofile_id;

  sqlexec ["test_gr003"]
    "select PASSWORD, ACCESSPROFILEID, INACTIVE, "
    "PW_UPDATE_DT from USERACCOUNT where NAME = 'davidj'";
  sqlnrecv  ["test_gr004"] ALL_ROWS;

  SaveData ("tmp_results");
  accessprofile_id - GetData1("tmp_results", 2);
  FreeData ("tmp_results");

  sqlexec  ["test_gr005"]
     "select LOGONNAME, LOGONPASSWORD, EXP_DAYS from "
     "ACCESSPROFILE where ACCESSPROFILEID = "
     + accessprofile_id;
}
```

See Also

`AppendData` `FreeData` `FreeAllData` `GetData` `SaveData`

getenv

Library Routine

Description Obtains the values of Windows NT or UNIX environment variables from within a virtual user script.

Syntax `string` **`getenv`** `(`*`name`*`)`

Syntax Element	Description
name	A string expression specifying the environment variable whose value is returned as a string.

Comments The `getenv` routine behaves like the C routine of the same name.

If a string of the form *`name=value`* is not found in the virtual user's environment list or if *`value`* is null (zero-length), `getenv` returns a string of zero length.

Example This example prints a random number in the range 1 to `limit`, where `limit` is the value (after conversion to an integer) of the `LIMIT` environment variable if defined; otherwise, `limit` equals 100:

```
string value;

if ((value = getenv("LIMIT")) == "")
    /* set default value if LIMIT is undefined */
    limit = 100;
else
    limit = atoi(value);
print uniform(1, limit);
```

See Also `putenv`

hex2mixedstring

Library Routine

Description Returns a mixed ASCII/hexadecimal version of a VU string.

Syntax `string` **`hex2mixedstring`**`(`*`str`*`)`

Syntax Element	Description
str	VU string expression

Comments The returned string consists of printable ASCII characters mixed with hexadecimal characters where a string of consecutive hexadecimal characters are surrounded by grave accent (‘) characters. Strings used (and returned) by VU with socket and HTTP emulation commands are in mixed ASCII and hexadecimal format.

Example

```
#include <VU.h>
string func build_new_request(s)
      string s;
     {
  /* code to create a request out of an earlier response */
     }
{
     string hexstr;
     string mixstr;
     calvin_700 = http_request ["cal001"] "calvin:700", "", 2,
    "GET / HTTP/1.0\r\n"
    "Connection: Keep-Alive\r\n"
    "User-Agent: Mozilla/4.03 [en] (X11; I; SunOS 5.5.1 sun4u)\r\n"
    "Pragma: no-cache\r\n"
    "Host: calvin:700\r\n"
    "Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
        */*\r\n"
    "Accept-Language: en\r\n"
    "Accept-Charset: iso-8859-1,*,utf-8\r\n"
    "\r\n";

    set Server_connection = calvin_700;
    http_header_recv ["cal002"]  200;/* OK */
    http_nrecv ["cal003"]  100 %% ; /* 1316 bytes */
    hexstr = mixed2hexstring(_response);
    hexstr = build_new_request(hexstr);
    mixstr = hex2mixedstring(hexstr);
   calvin_700 = http_request ["cal011"] "calvin:700", "", 2, mixstr;
    set Server_connection = calvin_700;
    http_header_recv ["cal012"]  200;/* OK */
    http_nrecv ["cal013"]  100 %% ;
    http_disconnect(calvin_700);
}
```

See Also `mixed2hexstring` `http_request` `http_nrecv` `http_recv`

http_disconnect

Emulation Function

Description Closes the connection to a Web server.

Syntax `int` **`http_disconnect`** `(`*`connection_id`*`)`

Syntax Element	Description
connection_id	An integer expression specifying a connection number returned by `http_request`, and not previously disconnected with `http_disconnect`().

Comments The `http_disconnect` function returns 1 for success and 0 for failure. If *`connection_id`* is invalid, `http_disconnect` generates a fatal runtime error.

Example This example connects to a Web server, sets the server connection, and then closes the connection:

```
#include <VU.h>
{
CAPRICORN_WEB_80 = http_request "CAPRICORN-WEB:80",
     HTTP_CONN_DIRECT,
     "GET / HTTP/1.0\r\n"
     "Accept: application/vnd.ms-excel, application/mswo"
     "rd, application/vnd.ms-powerpoint, image/gif, imag"
     "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
     "Accept-Language: en\r\n"
     "UA-pixels: 1152x864\r\n"
     "UA-color: color8\r\n"
     "UA-OS: Windows NT\r\n"
     "UA-CPU: x86\r\n"
    "User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)\r\n"
     "Host: capricorn-web\r\n"
     "Connection: Keep-Alive\r\n\r\n";
set Server_connection = CAPRICORN_WEB_80;
http_header_recv 200;/* OK */
/* more data (4853) than expected >> 100 % */
http_nrecv 100 %% ; /* 4853/4051 bytes */
http_disconnect(CAPRICORN_WEB_80);
}
```

See Also None.

http_find_values

Emulation Function

Description Searches for the specified values on the current connection.

Syntax `string[] http_find_values(name, type, tag [, name, type, tag ... ])`

Syntax Element	Description
name	A string expression that specifies the name of the desired value.
type	An integer expression that specifies the type of the value. The value of type should be one of: `HTTP_FORM_DATA`, `HTTP_HREF_DATA`, or `HTTP_COOKIE_DATA`. These values are defined in VU.h
tag	An integer expression that specifies which instance of the value is requested.

Comments

The `http_find_values`() function may occur in a VU script if you have told Robot to correlate all or some of your http data. You typically will not need to program this function yourself.

This function returns an array of strings containing the values specified. Each set of name, type and tag specifies a single requested value. Up to 21 values may be requested in a call to `http_find_values`(). If any of the requested values cannot be found, the corresponding element of the results array is `""`.

The `http_find_values`() function can be used to extract `FORM`, `HREF`, or `Set-Cookie` values.

`FORM` data appears in the response as:

```
<INPUT TYPE=xxx [xxx ]NAME=yyy [xxx ]VALUE=zzz[ xxxxxxxx]>
```

Given the above data in the response, `http_find_values("yyy", HTTP_FORM_DATA, 1)` returns `{"zzz"}`.

`HREF` data appears in the response as:

```
<A HREF=\"xxxx?nnnnn=&yyy=zzz [&y1y1=z1z1 ...]\">
```

Given the above data in the response, `http_find_values("yyy", HTTP_HREF_DATA, 1, "y1y1", HTTP_HREF_DATA, 1)` returns `{"zzz","z1z1"}`.

`Set-Cookie` data appears in the response as:

```
Set-Cookie: yyy=zzz[; y1y1=z1z1]\r\n
```

Given the above data in the response, `http_find_values("yyy", HTTP_COOKIE_DATA, 1, "y1y1", HTTP_COOKIE_DATA, 1)`returns `{"zzz","z1z1"}`.

All available data for the current connection (specified by the `Server_connection` VU environment variable) is searched regardless of whether or not that data has been processed by an http receive command.

Example

This example finds the first occurrence of the `FORM` data identified by `foo` and the second occurrence of the `HREF` data identified by `homepage`. Assuming that the response data for the current connection contains:

```
<INPUT TYPE=xxx NAME=foo VALUE=John>
<A HREF=\"xxxx?nnnnn=&homepage=www.myhome.com\">
. . .
A HREF=\"xxxx?nnnnn=&homepage=www.myhome2.com\">
```

The following call returns an array of strings equal to {"John", "www.myhome2.com"} and assigns it to the array SgenRes_001.

```
string SgenRes_001[];
SgenRes_001 = http_find_values("foo", HTTP_FORM_DATA, 1,
"homepage", HTTP_HREF_DATA, 2);
```

See Also http_request http_recv

http_header_info

Emulation Function

Description Gets individual header values from header metadata.

Syntax string **http_header_info** "*header_var_name*"

Syntax Element	Description
header_var_name	A string that is the name of a header metadata field. This string is case-insensitive.

Comments The http_header_info function scans the headers received by http_header_recv to locate lines beginning with the requested attribute, and returns a string containing the value of this attribute. It returns an empty string ("") on error.

If an attribute is listed more than once, only one value is returned.

Example Assume that http_header_recv reads the following header information:

```
HTTP/1.1 200 OK
Date: Mon, 24 Nov 1997 22:57:44 GMT
Server: Apache/1.2.4
Last-Modified: Fri, 21 Nov 1997 20:45:11 GMT
ETag: "7a398-cf1-3475f2d7"
Content-Length: 3313
Accept-Ranges: bytes
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
```

The following call returns 3313:

```
http_header_info ("Content-Length")
```

See Also http_header_recv

http_header_recv

Receive Emulation Command

Description Receives header metadata from a Web server.

Syntax `int` **`http_header_recv`** `[`*`cmd_id`*`]` *`status_code`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].

(Continued)

Syntax Element	Description
status_code	The expected HTTP status code for this response. You can use either the code number or the equivalent text string. The status codes are defined as follows: 100 "Continue" 101 "Switching Protocols" 200 "OK" 201 "Created 202 "Accepted" 203 "Non-Authoritative Information" 204 "No Content" 205 "Reset Content" 206 "Partial Content" 300 "Multiple Choices" 301 "Moved Permanently" 302 "Moved Temporarily" 303 "See Other" 304 "Not Modified" 305 "Use Proxy" 307 "Temporary Redirect" 400 "Bad Request" 401 "Unauthorized" 402 "Payment Required" 403 "Forbidden" 404 "Not Found" 405 "Method Not Allowed" 406 "Not Acceptable" 407 "Proxy Authentication Required" 408 "Request Time-out" 409 "Conflict" 410 "Gone" 411 "Length Required" 412 "Precondition Failed" 413 "Request Entity Too Large" 414 "Request-URI Too Large" 415 "Unsupported Media Type" 500 "Internal Server Error" 501 "Not Implemented" 502 "Bad Gateway" 503 "Service Unavailable" 504 "Gateway Time-out" 505 "HTTP Version not supported"

Comments

If `http_header_recv` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0.

This command occurs in response to an `http_request` command.

The metadata is sent from the Web server when a client requests a page. For example, metadata might contain protocol; type; URL address; size of page; date created, date last modified, and date last updated; as well as an indication of the security status of your connection.

The metadata received is stored in the read-only variable `_response` and is overwritten when you issue other receive emulation commands.

The `http_header_recv` emulation command is affected by the following VU environment variables: `Http_control`, `Timeout_act`, `Timeout_val`, `Timeout_scale`, `Log_level`, `Record_level`, and `Server_connection`.

The `Http_control` environment variable can affect how the `http_header_recv` emulation command interprets the received status. For more information, see ***Http_control*** on page 5-29**.**

Example

This example connects to a Web server, sets the server connection, receives the header information, and then receives a complete page of data (100 percent of the page, as indicated by 100 %%).

```
#include <VU.h>
{
CAPRICORN_WEB_80 = http_request "CAPRICORN-WEB:80",
     HTTP_CONN_DIRECT,
     "GET / HTTP/1.0\r\n"
     "Accept: application/vnd.ms-excel, application/mswo"
     "rd, application/vnd.ms-powerpoint, image/gif, imag"
     "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
     "Accept-Language: en\r\n"
     "UA-pixels: 1152x864\r\n"
     "UA-color: color8\r\n"
     "UA-OS: Windows NT\r\n"
     "UA-CPU: x86\r\n"
    "User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)\r\n"
     "Host: capricorn-web\r\n"
     "Connection: Keep-Alive\r\n\r\n";
set Server_connection = CAPRICORN_WEB_80;
http_header_recv 200;/* OK */
/* more data (4853) than expected >> 100 % */
http_nrecv 100 %% ; /* 4853/4051 bytes */
http_disconnect(CAPRICORN_WEB_80);
}
```

The header information received looks like the following:

```
HTTP/1.1 200 OK
Date: Mon, 24 Nov 1997 22:57:44 GMT
Server: Apache/1.2.4
Last-Modified: Fri, 21 Nov 1997 20:45:11 GMT
ETag: "7a398-cf1-3475f2d7"
Content-Length: 3313
Accept-Ranges: bytes
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
```

See Also `http_request`

http_nrecv

Receive Emulation Command

Description Receives a user-specified number of bytes from a Web server.

Syntax `int` **`http_nrecv`** `[`*`cmd_id`*`] {`*`count`*` | `*`count %%`*`}`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
count	The number of bytes to receive from the connection.
count %%	The number of bytes to receive as a percentage of the size of the last page processed. The size is calculated from the information in the last header processed for the connection.

Comments If `http_nrecv` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0.

The `http_nrecv` emulation command succeeds when it receives *count* bytes from the server. Binary data is translated into embedded hexadecimal strings. See *Unprintable HTTP or Socket Data* on page 3-35.

The `http_nrecv` command sets the “first received” (`_fr_ts`) and “last received” (`_lr_ts`) read-only variables.

The data received is stored in the read-only variable `_response` and is overwritten when you issue another receive emulation command.

If `Timeout_val` (subject to scaling) milliseconds elapses before the `http_nrecv` is satisfied, `http_nrecv` fails and returns 0. Otherwise, `http_nrecv` passes and returns 1.

The http_nrecv emulation command is affected by the following VU environment variables: Timeout_act, Timeout_val, Timeout_scale, Log_level, Record_level, Max_nrecv_saved, and Server_connection. Max_nrecv_saved applies to the actual data received, before any binary data is translated into embedded hexadecimal strings.

Example This example sets the server connection, receives the header metadata, and then receives a complete page of data (100 percent of the page, as indicated by 100 %%).

```
set Server_connection = CONN1;
http_header_recv 200;
http_nrecv 100 %%;
```

See Also http_recv

http_recv

Receive Emulation Command

Description Receives data from a Web server until the specified text string occurs.

Syntax int **http_recv** [*cmd_id*] *recv_str*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
recv_str	A string that marks the point at which to stop retrieving data.

Comments The data received is stored in the read-only variable _response and is overwritten when you issue other receive emulation commands.

If Timeout_val (subject to scaling) milliseconds elapses before the http_nrecv is satisfied, http_recv fails and returns 0. Otherwise, http_nrecv passes and returns 1.

The http_nrecv command sets the “first received” (_fr_ts) and “last received” (_lr_ts) read-only variables.

The http_recv emulation command is affected by the following VU environment variables: Timeout_act, Timeout_val, Timeout_scale, Log_level, Record_level, Max_nrecv_saved, and Server_connection. Max_nrecv_saved applies to the actual data received, before any binary data is translated into embedded hexadecimal strings.

Example

This example reads until the end of the connection or a timeout.

```
http_recv ["cmd003r"]  "$";
```

This example matches as soon as `EXCEL Home Page</title>\r\n` is found anywhere within the response:

```
Set Server_connection = conn1;
http_recv ["cmd001r"]  "EXCEL Home Page</title>\r\n";
```

This example reads until the end of the connection, and passes only if `_response` is exactly equal to `"EXCEL Home Page</title>\r\n"`. This is because the `^` forces the comparison to begin at the start of `_response`, and the `$` forces the comparison to begin at the start of `_response`.

```
http_recv ["cmd002r"]  "^EXCEL Home Page</title>\r\n$";
```

This example matches only if the first 5 characters of `_response =="EXCEL"`. If the first 5 characters do not match, `http_recv` continues to read until the end of the connection or a timeout.

```
http_recv ["cmd003r"]  "^EXCEL";
```

See Also

`http_nrecv`

http_request

Send Emulation Command

Description

Sends an HTTP request to a Web server.

Syntax

```
int http_request [cmd_id] primary_addr [, secondary_addr]
[, flags], text
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
primary_addr	A string expression that contains the host computer name and port number of the Web server to which you are connecting.
secondary_addr	A string expression that contains the host computer name and port number of the Web server. If *flag* is `HTTP_CONN_DIRECT`, this field is not used.
flags	An integer expression that indicates: ▶ The type of connection (`HTTP_CONN_DIRECT`, `HTTP_CONN_PROXY`, `HTTP_CONN_GATEWAY`, `HTTP_CONN_TUNNEL`). `HTTP_CONN_GATEWAY` and `HTTP_CONN_TUNNEL` are currently unused. ▶ Whether or not the connection is secure and the strength of the encryption (`HTTP_CONN_SECURE`, `HTTP_CONN_SECURE_40`, `HTTP_CONN_SECURE_56`, `HTTP_CONN_SECURE_`128) These connection flags are defined in the `VU.h` file.
text	A string that contains the request headers. If you are sending information, this string also contains the request body. For example, if you fill in a form, the information you provide in the form is the request body.

Comments

The `http_request` command returns a connection ID that is used as a reference for subsequent interactions with the Web server until the `http_disconnect` is issued. It returns an integer value: 0 or less for failure, or a unique connection number greater than or equal to 1 for success.

This command emulates all HTTP protocol request primitives: `GET`, `HEAD`, `POST`, `PUT`, `TRACE`, `LINK`, `UNLINK`, `DELETE`, `OPTIONS`, `COPY.`

Binary data is translated into embedded hexadecimal strings. See *Unprintable HTTP or Socket Data* on page 3-35.

The `http_request` command sets the "first connect" (`_fc_ts`), "last connect" (`_lc_ts`), "first sent" (`_fs_ts`), and "last sent" (`_ls_ts`) read-only variables.

The `http_request` command is affected by the following VU environment variables: `Connect_retries`, `Connect_retry_interval`, the think time variables, `Timeout_val`, `Timeout_scale`, `Timeout_act`, `Log_level`, `Record_level`, and `Suspend_check`. The think time is applied before the connect, and suspend checking is done (as normal) after the think time delay.

The `http_request` command automatically parameterizes cookie information during script playback. When dynamic cookie information is available from a server, that cookie value replaces the values in the VU script. Otherwise, the scripted value is used.

Example This example connects to a Web server. The variable `CAPRICORN_WEB_80` holds the returned ID for the connection.

```
#include <VU.h>
{
CAPRICORN_WEB_80 = http_request "CAPRICORN-WEB:80",
     HTTP_CONN_DIRECT,
     "GET / HTTP/1.0\r\n"
     "Accept: application/vnd.ms-excel, application/mswo"
     "rd, application/vnd.ms-powerpoint, image/gif, imag"
     "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
     "Accept-Language: en\r\n"
     "UA-pixels: 1152x864\r\n"
     "UA-color: color8\r\n"
     "UA-OS: Windows NT\r\n"
     "UA-CPU: x86\r\n"
    "User-Agent: Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)\r\n"
     "Host: capricorn-web\r\n"
     "Connection: Keep-Alive\r\n\r\n";
set Server_connection = CAPRICORN_WEB_80;
http_header_recv 200;/* OK */
http_nrecv 100 %% ; /* 4051 bytes */
http_disconnect(CAPRICORN_WEB_80);
}
```

See Also None.

http_url_encode

Emulation Function

Description Prepares a VU string for inclusion in `http_request` data.

Syntax `string` **`http_url_encode`**`(`*`str`*`)`

Syntax Element	Description
str	VU string expression.

Comments

The returned string consists of the original VU string expression with all HTTP special characters in the proper escape sequence format.

If your recording contains HTTP traffic, and datapooling is enabled, then your script contains a call to the `http_url_encode` function for every call to the `datapool_value` function to ensure that the data sent to the Web server is in the correct format.

Example

This example script fragment sends a `POST` request containing datapool values to a previously established connection, and then closes the connection.

```
set Server_connection = bonnie_rational_com_80
http_request ["NewHttp058"] /* Keep-Alive request */
    "POST /cgi-bin/www/prcat.cgi HTTP/1.1\r\n"
    "Accept: application/vnd.ms-excel, application/msword"
    "application/vnd.ms-powerpoint, image/gif, imag"
    "e/x-xbitmap, image/jpeg, image/pjpeg, */*\r\n"
    "Referer: http://www.rational.com/world/press/releases/\r\n"
    "Accept-Language: en-us\r\n"
    "User-Agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows NT) \r\n"
    Host: www.rational.com\r\n"
    Content-Length: 28\r\n"
    "\r\n"
    "financials="
    +http_url_encode(datapool_value(DP1, "financial" )) +
    "&chapter="
    +http_url_encode(datapool_value(DP1, "chapter" )) +
    "";
http_disconnect (bonnie_rational_com_80);
```

See Also

`http_request`   `datapool_value`

if-else

Flow Control Statement

Description Conditionally executes a VU statement.

Syntax

```
if (exp1)
        statement1;
else
        statement2;
```

Syntax Element	Description
exp1	An integer expression to be evaluated.
statement1	A VU language statement that is executed if the value of *exp1* is not 0.
statement2	A VU language statement that is executed if the value of *exp1* is 0.

Comments Multiple statements can appear in braces, such as:

```
if (exp1) {
 statement3;
 statement4;
 statement5;
} else {
 statement6;
 statement7;
 statement8;
}
```

It is advisable to indent statements for readability.

Example This example aborts script execution if the string is ERROR. If the string is not ERROR, the script continues processing and writes a message to the log file:

```
if (string1=="ERROR")
    user_exit(-1, "Fatal Error - Aborting");
else
    log_msg("Emulation proceeding normally");
```

See Also else-if

iiop_bind

Send Emulation Command

Description Binds an interface name to an Object Reference pseudo-object.

Syntax `int` **`iiop_bind`** `[`*`cmd_id`*`]` *`repository_id`*`,` *`instance_id`* `[,`*`ior`*`]`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
repository_id	A string constant specifying the name of the interface to bind to. It is invalid to pass the empty string ("") if *ior* is not specified. The only interface specification format supported is the CORBA IDL RepositoryId format. The *repository_id* consists of three components, separated by colons: ▶ The first component is the format name, "IDL." ▶ The second component is a list of identifiers, separated by "/" characters. These identifiers are arbitrarily long sequences of alphabetic, digit, underscore ("_"), hyphen ("-"), and period (".") characters. Typically, the first identifier is a unique prefix, and the rest are the OMG IDL Identifiers that make up the scoped name of the definition. ▶ The third component is made up of major and minor version numbers, in decimal format, separated by a ".". When two interfaces have *repository_id*s differing only in minor version number, you can assume that the definition with the higher version number is upwardly compatible with the one with the lower minor version number.
instance_id	A string expression identifying a particular instance of an interface implementation. Some ORBs require this string to identify persistent implementations. An empty string ("") means any instance is acceptable.
ior	An optional string expression specifying an IIOP Interoperable Object Reference (IOR) to be used by the IOR bind modus.

Comments If `iiop_bind` completes successfully, it returns a handle to the Object Reference pseudo-object bound to the interface implementation specified by the *`repository_id`*. Otherwise it returns `NULL_HANDLE`.

The `iiop_bind` command binds an interface implementation, identified by *`repository_id`*, to an Object Reference pseudo-object. The result of binding is a handle to an Object Reference pseudo-object which contains (among other things) an IIOP object key used in later IIOP requests to the implementation.

The actual mechanism used by the playback engine to execute the bind is ORB vendor-dependent.

The `iiop_bind` command sets the first sent (`_fs_ts`), last sent (`_ls_ts`), first received (`_fr_ts`), last received (`_ls_ts`), and error information (`_error_type`, `_error`, and `_error_text`) read-only variables.

The `iiop_bind` command is affected by the following VU environment variables: `Timeout_val`, `Timeout_scale`, `Timeout_act`, `Log_level`, `Record_level`, and `Suspend_check`.

Example This example binds an interface name to an Object Reference pseudo-object. Object references are the only way for a client to reach target objects. The `iiop_bind` command takes information about an object and uses it to try and obtain a reference to the object for use in invoking methods on the object.

```
objref = iiop_bind ["bind001"]
"IDL:Bank/BranchManager:1.0", "Branch15", " ";
```

See Also None.

iiop_invoke

Send Emulation Command

Description Initiates a synchronous IIOP request to an interface implementation

Syntax Form 1: initialize and invoke a Request pseudo-object

```
int iiop_invoke [cmd_id] [&request,]
        object_ref, operation,
        [parameter_expr,...]
```

Form 2: reuse a Request pseudo-object

```
int iiop_invoke [cmd_id] request
        [,parameter_expr,...]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *`cmd_id`* has the form [*`string_exp`*].
request	An integer variable for the handle of the created request.

(Continued)

Syntax Element	Description
object_ref	An integer handle to the Object Reference pseudo-object bound to the interface implementation to be invoked. *object_ref* cannot be NULL_HANDLE.
operation	A string expression containing the name of the interface operation to be invoked.
parameter_expr	An optional list of one or more parameter binding expressions for the IN, INOUT, and OUT arguments and return value of the invoked operation.

Comments

The iiop_invoke emulation command has two forms. The first form constructs an IIOP Request message by creating and initializing a new Request pseudo-object. The second form constructs an IIOP Request message by overriding an existing Request pseudo-object with a new set of parameters.

In the first form, specifying the optional request argument causes the handle of the new Request pseudo-object to be stored in the VU integer variable referenced by request. The pseudo-object referenced by the handle persists until it is released by a call to iiop_release. If the request argument is not supplied, then a temporary internal Request pseudo-object is created to store the request context and is automatically released before the command returns.

In the second form, the request argument is the handle to the Request pseudo-object to be reused for storing the request context.

After the message is constructed, it is sent to the interface implementation and the command then awaits its reply. After successful completion, the associated INOUT, OUT, and RETURN parameter variables are loaded with the results of the operation invocation.

This command is equivalent to the CORBA::Object::_create_request() and CORBA::Request::invoke() function pairs.

The iiop_invoke command sets the first sent (_fs_ts), last sent (_ls_ts), first received (_fr_ts), last received (_ls_ts), and error information (_error_type, _error, and _error_text) read-only variables.

The iiop_invoke command is affected by the following VU environment variables: the think time variables, Timeout_val, Timeout_scale, Timeout_act, Log_level, Record_level, and Suspend_check.

Example This example initiates a synchronous IIOP request to an interface implementation. The `iiop_invoke` command is used to invoke methods on an object.

```
/* bind to the Branch15 instance of the BranchManager interface */
bm_ref = iiop_bind ["bind001"]
       "IDL:Bank/BranchManager/1.0", "Branch15";

/* fetch account balance, using global request context */
{ string Balance; }
iiop_invoke ["Balance001"] "Balance", bm_ref,
       "Account":Account, "Balance"::&Balance;

/* log the balance query to the transaction log, preserving
       the request context in a new Request pseudo-object
       referenced by log_req */
iiop_invoke ["LogTransaction001"] &log_req, "Log Transaction", bm_ref,
       "LogTransaction", "Account":Account,
       "TransactionType":"Balance";

/* withdraw all funds from account, again using the global
       request context but re-initializing it */
iiop_invoke ["Withdraw001"] "Withdraw", bm_ref,
       "Account":Account, "Amount":Balance;

/* log the withdraw transaction to the log, reusing the
       previous LogTransaction request context */
iiop_invoke ["LogTransaction002"] log_req,
       "TransactionType":"Withdraw";

/* release log_req Request pseudo-object */
iiop_release(log_req);
```

See Also `iiop_bind`

iiop_release

Emulation Function

Description Releases storage associated with a pseudo-object.

Syntax `int` **`iiop_release`** `(`*`handle`*`[, ...])`

Syntax Element	Description
handle	A list of integer handles to pseudo-objects of any type. At least one handle argument must be supplied.

Comments The `iiop_release` function deletes and releases the storage associated with one or more pseudo-objects. When a handle is released, it becomes invalid and cannot be used again.

Upon success the function returns 1, else it returns 0 indicating an error.

Example This example releases storage associated with a pseudo-object. You can use `iiop_release` to free memory used for storing requests or object references.

```
iiop_release(objref);
```

See Also None.

IndexedField

VU Toolkit Function: File I/O

Description Parses the line read by the `ReadLine` function and returns the field designated by *`index`*.

Syntax

```
#define _PV_FILEIO_FIELD  "delimiter characters"
#include <sme/fileio.h>
string func IndexedField(index)
int index;
```

Syntax Element	Description
delimiter characters	The characters that delimit the fields in the index. The default field delimiter is a vertical bar (|).
index	The number of the field to be retrieved (begins with 1).

Comments The `IndexedField` function parses the data returned by the most recent call to the `ReadLine` function. A null string is returned when *`index`* is greater than the number of fields in the line. Multiple contiguous occurrences of the delimiter are considered a single delimiter.

The `IndexedField` function affects the order of the results returned by `NextField`. Either functions modify the field pointer, which is the starting point for the next invocation of this function.

If `IndexedField` is called before the first call to `ReadLine`, the return value is undefined. The `SHARED_READ` macro uses the `ReadLine` function to read from the file, so it also may be used to retrieve the data to be parsed.

The string variable `Last_Field` contains the value returned by the most recent use of the `IndexedField` or `NextField` function.

The list of characters to be considered as field delimiters is contained in the macro definition `_PV_FILEIO_FIELD`. Define this macro constant (`# define`) before the inclusion of the header file `fileio.h`.

Example This example first frees any previously saved data from the "parts" text buffer. A loop is started to query the database five times. The script then obtains the next record from a file being shared by all virtual users that execute this script. The record is parsed by selection of the first field and direct selection of the third field, skipping the second field. The third field is composed of four or more subfields. Parsing of the third field continues by selection of the first subfield, which provides a count of the number of remaining subfields. One of the remaining subfields is selected at random to form a part of the query. After the query is performed, the returned rows are saved. If this is the first iteration of the loop, the rows are saved to the "parts" text buffer. Subsequent iterations of the loop append the data from the returned rows to the "parts" text buffer.

```
#include <VU.h>
#include <sme/data.h>
#include <sme/fileio.h>

{
   shared int file_tag_lock, file_tag_offset;
   string product_id, part_id, subassm_id;
   string temp_str;
   int subassm_cnt;

   /* This script assumes a connection was made to the database. */

   /* Record layout of "myfile"                                  */
   /* product | part | subassm_cnt ; subassm_1; subassm_2 ; subassm_3;
... */

   /* There will be a minimum of three subassemblies in each record. */

   FreeData("parts");

   /* Perform 5 queries for parts. */

   for (i=0; i<=4; i++)
      {
      SHARED_READ ("myfile", file_tag);

      /* Parse the record. */
      product_id = NextField();

      temp_str = IndexedField(3);
      /* Note: The entire unparsed field is returned but it is not
       used directly. So the returned text string is not used.     */

      subassm_cnt = atoi(NextSubField());
      subassm_id = IndexSubField(uniform(2,subassm_cnt+1));


      /* Query for the part. */
      sqlexec ["test_001"]
         "select part_name from product_db "
         "where product='"+product_id+"' "
         "and subassembly='"+subassm_id+"'";
      sqlnrecv ["test_002"] ALL_ROWS;
```

```
            if i = 0
               SaveData("parts");
            else
               AppendData("parts");
            }
         }
```

See Also IndexedSubField NextField NextSubField ReadLine SHARED_READ

IndexedSubField

VU Toolkit Function: File I/O

Description Parses the field set by the `NextField` or `IndexedField` function and returns the subfield designated by *`index`*.

Syntax

```
#define _PV_FILEIO_SUBFIELD  "delimiter characters"
#include <sme/fileio.h>
string func IndexedSubField(index)
int index;
```

Syntax Element	Description
delimiter characters	The characters that delimit the subfields in the index. The default delimiter is a colon (:). Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter.
index	The number of the field to be retrieved (begins with 1).

Comments The `IndexedSubField` function parses the field returned by the most recent call to the `NextField` or `IndexedField` function. The *`index`* argument, which begins at 1, is the number of the field to be retrieved. A null string is returned when index is greater than the number of fields in the line.

The `IndexedSubField` function affects the order of the results returned by `NextSubField`. Either functions modifies the subfield pointer, which is the starting point for the next invocation of this function.

If `IndexedSubField` is called before the first call to `NextField` or `IndexedField`, the return value is undefined.

The string variable `Last_SubField` contains the value returned by the most recent use of `IndexedSubField` or `NextSubField` function.

The list of characters to be considered as subfield delimiters is contained in the macro definition `_PV_FILEIO_SUBFIELD`. Define this macro constant (# `define`) before the inclusion of the header file *`fileio.h`*.

Example

This example first frees any previously saved data from the "parts" text buffer. A loop is started to query the database five times. The script then obtains the next record from a file being shared by all virtual users that execute this script. The record is parsed by selection of the first field and direct selection of the third field, skipping the second field. The third field is composed of four or more subfields. Parsing of the third field continues by selection of the first subfield, which provides a count of the number of remaining subfields. One of the remaining subfields is selected at random to form a part of the query. After the query is performed, the returned rows are saved. If this is the first iteration of the loop, the rows are saved to the “parts” text buffer. Subsequent iterations of the loop append the data from the returned rows to the “parts” text buffer.

```
#include <VU.h>
#include <sme/data.h>
#include <sme/fileio.h>

{
   shared int file_tag_lock, file_tag_offset;
   string product_id, part_id, subassm_id;
   string temp_str;
   int subassm_cnt;

   /* This script assumes a connection was made to the database. */

  /* Record layout of "myfile"                                   */
  /* product | part | subassm_cnt ; subassm_1; subassm_2 ; subassm_3;
... */

  /* There will be a minimum of three subassemblies in each record. */

   FreeData("parts");

   /* Perform 5 queries for parts. */

   for (i=0; i<=4; i++)
      {
      SHARED_READ ("myfile", file_tag);

      /* Parse the record. */
      product_id = NextField();

      temp_str = IndexedField(3);
      /* Note: The entire unparsed field is returned but it is not
       used directly. So the returned text string is not used.     */

      subassm_cnt = atoi(NextSubField());
      subassm_id = IndexSubField(uniform(2,subassm_cnt+1));

      /* Query for the part. */
      sqlexec ["test_001"]
         "select part_name from product_db "
         "where product='"+product_id+"' "
         "and subassembly='"+subassm_id+"'";
      sqlnrecv ["test_002"] ALL_ROWS;
```

```
        if i = 0
           SaveData("parts");
        else
           AppendData("parts");
        }
    }
```

See Also IndexedField NextField NextSubField ReadLine SHARED_READ

itoa

Library Routine

Description Converts integers to strings.

Syntax `string itoa(int)`

Syntax Element	Description
int	The integer expression to convert to a string.

Comments The `itoa` routine returns a string expression, the ASCII form of the integer. If *int* is negative, then the returned string expression is prefixed with a negative sign.

The `itoa` routine is the converse of `atoi`. It takes an integer argument and returns a string expression made up of digits representing the integer in ASCII.

Example This example returns the string "93":

```
itoa(93);
```

This example returns the string "30":

```
itoa(21 + 9);
```

This example returns the string "23":

```
itoa(atoi("23"));
```

See also atoi

lcindex

Library Routine

Description Returns the position of the last occurrence of a user-supplied character.

Syntax `int lcindex (str, char)`

Syntax Element	Description
str	The string to search.
char	The character to search for within *str*.

Comments The `lcindex` (last character index) routine returns the position within *str* of the last occurrence of the character *char*. If no occurrences are found, `lcindex` returns the integer zero.

The routines `cindex`, `lcindex`, `sindex`, and `lsindex` return positional information about either the first or last occurrence of a specified character or set of characters within a string expression. `strspan` returns distance information about the span length of a set of characters within a string expression.

Example This example returns the integer value 6, which is the position of the last occurrence of the letter `a` in the string `aardvark`:

```
lcindex("aardvark", 'a');
```

See Also `cindex` `sindex` `lsindex` `strspan` `strstr`

log_msg

Library Routine

Description Writes messages to the log file with a standard header format.

Syntax `int log_msg (msg_str)`

Syntax Element	Description
msg_str	A string expression containing the message to write to the log file.

Comments The `log_msg` routine returns an integer expression equal to the value of *T*.

`log_msg` writes *msg_str* to the standard log file, preceded by the following explanatory text:

```
<<< log_msg(): script = script_name, time = T >>>
```

script_name is replaced by the script name (corresponding to the read-only variable `_script`). *T* is replaced by the current time, in milliseconds format. The text of *msg_str* is printed in a manner consistent with other logged information — for example, unprintable characters are replaced by their VU-style escape sequences as described in *How a VU Script Represents Unprintable Data* on page 3-34.

Example In this example, assume the current script's name is `db2`, the value of `trans_no` before the `log_msg` statement is executed is 3, and the current time is 29130:

```
log_msg("Beginning Transaction " + {itoa(++trans_no));
```

The following is message is logged:

```
<<< log_msg(): script = db2, time = 29130 >>>
Beginning Transaction 4
```

See Also None.

lsindex

Library Routine

Description Returns the position of the last occurrence of any character from a specified set.

Syntax `int lsindex (str, char_set)`

Syntax Element	Description
str	The string expression to search.
char_set	The characters to search for within *str*.

Comments The `lsindex` (last string index) routine returns the position within *str* of the last occurrence of any character from *char_set*. If no occurrences are found, `lsindex` returns an integer value of `0`.

The routines `cindex`, `lcindex`, `sindex`, and `lsindex` return positional information about either the first or last occurrence of a specified character or set of characters within a string expression. `strspan` returns distance information about the span length of a set of characters within a string expression.

Example This example returns the integer value 14, because `a` is the last vowel in the string `"moo goo gai pan"` and it is the 14th character.

```
lsindex("moo goo gai pan", "aeiou");
```

See Also `cindex` `lcindex` `sindex` `strspan` `strstr`

match

Library Routine

Description Determines whether a subject string matches a specified pattern.

Syntax `int match (pattern, subject [, &arg ] ...)`

Syntax Element	Description
pattern	A string expression specifying the pattern to match, as expressed in VU regular expression notation. (The section *VU Regular Expressions* on page 3-29 discusses regular expression notation.) To assign the results of the match to &*arg*, place the regular expression portion of the pattern in the format (*regular_exp*)$*n*, where *n* is an integer representing the position of the argument. For example, (*regular_exp*)$0 places the results in *arg1*, (*regular_exp)*$1 places the results in *arg2*, and so on.
subject	A string expression specifying the string to match. *subject* is often the read-only variable `_response`, because you may want to match a certain pattern in your response.
argn	The optional string output variable that contains the results of the match. The number of *argn* variables must be equal to or greater than the number of (*regular_exp*)$*n*, even if some variables are left unassigned.

Comments The `match` routine returns the integer value 1 if the subject string matches *pattern*; Otherwise it returns a value of 0.

In making assignments to *argn* variables, `match` follows these rules:

- Assignments are made unconditionally.
- The value of recursive assignments are undefined.
- If an assignment is not made, the original values of *argn* variables are unchanged.

Example This example uses `match` to check whether the database contains `Smith A.E.`, and, if not, adds his name and relevant data:

```
sqlexec "SELECT * FROM dbo.Student WHERE Studid < 5000";
sqlnrecv ["test001"] ALL_ROWS;
if (!match('Smith *A\.E.\', _response))
{
    sqlexec "INSERT dbo.Student VALUES"
     "1005, 'Smith", "A.E.', "215 Charles St.', '050263", 'M");
}
```

In this example, `match` returns a `1`, `"4"` is assigned to `str1`, and `"def"` is assigned to `str2`:

```
match("abc([0-9]+)$0 ([A-Za-z]+)$1", "abc4 def", &str1, &str2);
```

See Also None.

mixed2hexstring

Library Routine

Description Returns a pure hexadecimal version of a VU string.

Syntax `string` **`mixed2hexstring`**`(`*`str`*`)`

Syntax Element	Description
str	VU string expression.

Comments The returned string consists of a leading grave accent (‘), the hexadecimal representation of the string expression, and a trailing grave accent (‘). Strings used (and returned) by VU with socket and HTTP emulation commands are in mixed ASCII and hexadecimal format.

Example

```
#include <VU.h>
{
    string hexstr;
    calvin_700 = http_request ["cal001"] "calvin:700", "", 2,
      "GET / HTTP/1.0\r\n"
      "Connection: Keep-Alive\r\n"
     "User-Agent: Mozilla/4.03 [en] (X11; I; SunOS 5.5.1 sun4u)\r\n"
      "Pragma: no-cache\r\n"
      "Host: calvin:700\r\n"
      "Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
          */*\r\n"
      "Accept-Language: en\r\n"
      "Accept-Charset: iso-8859-1,*,utf-8\r\n"
      "\r\n";
    set Server_connection = calvin_700;
    http_header_recv ["cal002"]  200;/* OK */
    http_nrecv ["cal003"]  100 %% ; /* 1316 bytes */
    hexstr = mixed2hexstring(_response);
    http_disconnect(calvin_700);
}
```

See Also `hex2mixedstring` `http_request` `http_nrecv` `http_recv`

mkprintable

Library Routine

Description Creates printable versions of strings that contain nonprintable characters.

Syntax `string` **`mkprintable`** `(`*`str`*`)`

Syntax Element	Description
str	A string expression that serves as the subject string.

Comments The `mkprintable` routine returns a printable version of *`str`* by replacing all unprintable characters with their corresponding VU-style escape sequences, as follows:

`\r`	A single character representing a carriage return.
`\f`	A single character representing a formfeed.
`\n`	A single character representing a newline.
`\t`	A single character representing a horizontal tab.
`\b`	A single character representing a backspace.
`\0`	The null character (the character with value 0).
`\`*`ddd`*	A single character representing the character *ddd*.

Example This example returns a string equivalent to the string constant `"\\033"`. Although the strings look similar, they are quite different; the length of the subject string is 1 character and the length of the returned string is 4 characters.

```
mkprintable ("\033");
```

This example returns a string equivalent to the string constant `"\\t\\t\\t"`, escaping each tab character with the two-character combination `\t`.

```
mkprintable("\t\t\t");
```

See Also `print`

negexp

Library Routine

Description

Returns a random integer from a negative exponential distribution with the specified mean.

Syntax

`int negexp (mean_value)`

Syntax Element	Description
mean_value	An integer expression whose value specifies the mean of the negative exponentially distributed random integers returned by `negexp`. The value of *mean_value* must be non-negative.

Comments

The `rand`, `srand`, `uniform`, and `negexp` routines enable the VU language to generate random numbers. The behavior of these random number routines is affected by the way you set the **Seed** and **Seed Flags** options in a LoadTest schedule. By default, the **Seed** generates the same sequence of random numbers but sets unique seeds for each virtual user, so that each virtual user has a different random number sequence. For more information about setting the seed and seed flags in a schedule, see the *Using Rational LoadTest* manual.

`srand` uses the argument *seed* as a seed for a new sequence of random numbers returned by subsequent calls to `negexp`. If `srand` is then called with the same seed value, the sequence of random numbers is repeated. If `negexp` is called before any calls are made to `srand`, the same sequence is generated as when `srand` is first called with a seed value of 1.

Example

In this example, seeds the random number generator with the current time and prints the first 10 random numbers with a mean of 10.

```
srand(time());
for (i = 0; i < 10; i++)
printf("random number (%d): %d\n", i, negexp(10));
```

See Also

`rand` `uniform` `srand`

NextField

VU Toolkit Function: File I/O

Description

Parses the line read by the `ReadLine` function.

Syntax

```
#define _PV_FILEIO_FIELD  "delimiter characters"
#include <sme/fileio.h>
string func NextField()
```

Syntax Element	Description
delimiter character	The characters that delimit the fields in the index. The default delimiter is a vertical bar (|). Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter.

Comments

The `NextField` function retrieves the next available field from the data returned by the most recent call to the `ReadLine` function. The null string is returned when the fields in the line have been exhausted.

The `IndexedField` function affects the order of the results returned by `NextField`. Either function modifies the field pointer, which is the starting point for the next invocation of this function.

If `NextField` is called before the first call to `ReadLine` the return value is undefined. The `SHARED_READ` macro uses the `ReadLine` function to perform the read from the file, so it also may be used to retrieve the data to be parsed.

The string variable `Last_Field` contains the value returned by the most recent use of `IndexedField` or `NextField` function.

The list of characters to be considered as field delimiters is contained in the macro definition `_PV_FILEIO_FIELD`. Define this macro constant (# `define`) before the inclusion of the header file `fileio.h`.

Example

This example first frees any previously saved data from the "parts" text buffer. A loop is started to query the database five times. The script then obtains the next record from a file being shared by all virtual users that execute this script. The record is parsed by selection of the first field and direct selection of the third field, skipping the second field. The third field is composed of four or more subfields. Parsing of the third field continues by selection of the first subfield, which provides a count of the number of remaining subfields. One of the remaining subfields is selected at random to form a part of the query. After the query is performed, the returned rows are saved. If this is the first iteration of the loop, the rows are saved to the "parts" text buffer. Subsequent iterations of the loop append the data from the returned rows to the "parts" text buffer.

```
#include <VU.h>
#include <sme/data.h>
#include <sme/fileio.h>

{
   shared int file_tag_lock, file_tag_offset;
   string product_id, part_id, subassm_id;
   string temp_str;
   int subassm_cnt;

   /* This script assumes a connection was made to the database. */

  /* Record layout of "myfile"                                   */
  /* product | part | subassm_cnt ; subassm_1; subassm_2 ; subassm_3;
... */

  /* There will be a minimum of three subassemblies in each record. */

   FreeData("parts");

   /* Perform 5 queries for parts. */

   for (i=0; i<=4; i++)
      {
      SHARED_READ ("myfile", file_tag);

      /* Parse the record. */
      product_id = NextField();

      temp_str = IndexedField(3);
      /* Note: The entire unparsed field is returned but it is not
       used directly. So the returned text string is not used.     */

      subassm_cnt = atoi(NextSubField());
      subassm_id = IndexSubField(uniform(2,subassm_cnt+1));

      /* Query for the part. */
      sqlexec ["test_001"]
         "select part_name from product_db "
         "where product='"+product_id+"' "
         "and subassembly='"+subassm_id+"'";
      sqlnrecv ["test_002"] ALL_ROWS;

      if i = 0
         SaveData("parts");
      else
         AppendData("parts");
      }
   }
```

See Also

IndexedField IndexedSubField NextSubField ReadLine SHARED_READ

NextSubField

VU Toolkit Function: File I/O

Description Parses the field returned by the most recent call to NextField or IndexedField.

Syntax

```
#define _PV_FILEIO_SUBFIELD "delimiter characters"
string func NextSubField()
```

Syntax Element	Description
delimiters	The characters that delimit the subfields in the index. The default delimiter is a colon (:). Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter.

Comments The NextSubField function retrieves the next available subfield returned by the most recent call to the NextField or IndexedField function. The null string is returned when the subfields within the field have been exhausted.

The IndexedSubField function affects the order of the results returned by NextSubField. Either function modifies the subfield pointer, which is the starting point for the next invocation of this function.

If NextSubField is called before the first call to NextField or IndexedField, the return value is undefined.

The string variable Last_SubField contains the value returned by the most recent use of IndexedSubField or NextSubField function.

The list of characters to be considered as subfield delimiters is contained in the macro definition _PV_FILEIO_SUBFIELD. Define this macro constant (# define) before the inclusion of the header file fileio.h.

Example This example first frees any previously saved data from the "parts" text buffer. A loop is started to query the database five times. The script then obtains the next record from a file being shared by all virtual users that execute this script. The record is parsed by selection of the first field and direct selection of the third field, skipping the second field. The third field is composed of four or more subfields. Parsing of the third field continues by selection of the first subfield, which provides a count of the number of remaining subfields. One of the remaining subfields is selected at random to form a part of the query. After the query is performed, the returned rows are saved. If this is the first iteration of the loop, the rows are saved to the "parts" text buffer. Subsequent iterations of the loop append the data from the returned rows to the "parts" text buffer.

```
#include <VU.h>
#include <sme/data.h>
#include <sme/fileio.h>

{
   shared int file_tag_lock, file_tag_offset;
   string product_id, part_id, subassm_id;
   string temp_str;
   int subassm_cnt;

   /* This script assumes a connection was made to the database. */

  /* Record layout of "myfile"                                   */
  /* product | part | subassm_cnt ; subassm_1; subassm_2 ; subassm_3;
... */

  /* There will be a minimum of three subassemblies in each record. */

   FreeData("parts");

   /* Perform 5 queries for parts. */

   for (i=0; i<=4; i++)
      {
      SHARED_READ ("myfile", file_tag);

      /* Parse the record. */
      product_id = NextField();

      temp_str = IndexedField(3);
      /* Note: The entire unparsed field is returned but it is not
       used directly. So the returned text string is not used.     */

      subassm_cnt = atoi(NextSubField());
      subassm_id = IndexSubField(uniform(2,subassm_cnt+1));

      /* Query for the part. */
      sqlexec ["test_001"]
         "select part_name from product_db "
         "where product='"+product_id+"' "
         "and subassembly='"+subassm_id+"'";
      sqlnrecv ["test_002"] ALL_ROWS;

      if i = 0
         SaveData("parts");
      else
         AppendData("parts");
      }
   }
```

See Also IndexedField IndexedSubField ReadLine NextField SHARED_READ

open

Library Routine

Description Opens a file for reading or writing.

Syntax *int* **open** (*filename*, *mode*)

Syntax Element	Description
filename	A string expression specifying the file to be opened.
mode	A string expression specifying how the file is to open. Valid values: ▶ "r" opens the file for reading. If the file does not exist, a runtime error is generated. ▶ "w" opens the file for writing. If the file exists, its contents are discarded. If it does not exist, it is created. ▶ "a" opens the file for appending. If the file exists, its contents are retained and any new output to the file is appended to what is already in the file. If the file does not exist, it is created. Information already in the file is never overwritten. If multiple processes open the same file for appending, their output is intermixed in the file in the order in which it is written. ▶ "r+" opens the file for update. You can read or write to a file for update. If the file does not exist, a runtime error is generated. If the file does exist and new output is written to it, the new output is written at the beginning of the file, overwriting what is already there. ▶ "w+" opens the file for update and create or truncate. You can read or write to a file for update in this mode. If the file does not exist, it is created. If the file exists, its current contents are discarded. ▶ "a+" opens the file for update and append. You can read or write to a file for update in this mode. If the file does not exist, it is created. If the file does exist, data written to it is appended. ▶ "p" opens the file in persistent mode. "p" can accompany any other mode (the mode string for open() can include a"p" anywhere in the string). A persistent file remains open across scripts in a single run.

Comments If open can successfully open the file, it returns an integer file descriptor. You use this file descriptor to make subsequent references to the file. If open cannot open the file as specified, open generates a runtime error.

The open routine specifies a file to open for reading or writing. A file must be opened before it is used. You do not have to open the standard input, output, error, log, or record files, however, because they are automatically opened by the system.

The VU language open routine corresponds to the C language **fopen** library routine. The options on your computer determine the maximum number of open files. The number of reserved files for VU is seven.

To enable subsequent scripts to access a persistent file without reopening the file, use a persistent integer variable to hold the file descriptor returned from `open`.

Example This example declares the variable `theline` as a string. It then:

- Opens `data_file` for reading and assigns it the file descriptor `file1`.
- Positions the character pointer so that each user reads a different line (file pointer for user1 is 80 (`_uid*80`) bytes from the beginning of the file, file pointer for user 2 is 160 bytes from the beginning of the file, and so on).
- Reads an entire line (anything but a new line followed by a new line) and stores it in `theline`.
- Closes the file after reading 10 lines.

```
string theline;
for (i=0; i<10; I++) {
    file1=open("data_file","r");
    fseek(file1, (_uid*80),0);
    fscanf(file1, "%[^\n]\n", &theline);
}
close(file1);
```

See Also `close`

pop

Environment Control Command

Description Removes the value of a VU environment variable from the top of the stack.

Syntax **pop [***env_var_list***];**

Syntax Element	Description
env_var_list	Use one of the following for *env_var_list*: ▶ A list of one or more environment variables, separated by commas and optionally by white space. If *env_var_list* contains one item, the brackets are optional. If *env_var_list* contains more than one item, `pop` operates on the items from left to right. ▶ `ENV_VARS`. This specifies all the environment variables.

Comments The `pop` command removes and discards the element at the top of the stack of each variable in *env_var_list*. Thus, the next-to-top element of each stack moves to the top of that stack and becomes the current value of that variable. A runtime error occurs if you attempt to `pop` a stack that contains only one element.

Example This example sets the value for `Timeout_val` to 120000 ms, pushes the value of 30000 to the top of the `Timeout_val` stack (so that 30000 is now the current value and 120000 is the second element on the stack), and then removes 30000 from the stack (so that 120000 is the only element left on the stack).

```
/* Set values for Timeout_val and Log_level. */
set [Timeout_val = 120000, Log_level = TIMEOUT];
push Timeout_val = 30000;
pop Log_level;
```

This example disables the normal checking for any queued suspend requests, and encapsulates this disabling within the `push` and `pop` commands:

```
push Suspend_check off;
/* code that performs input emulation commands where you do not want
suspend or step operations to stop */
pop Suspend_check;
```

See Also `push` `eval` `set`

print

Statement

Description Writes to standard output when the formatting capability of `printf` is not required.

Syntax **`print`** *`exp_list`*`;`

Syntax Element	Description
exp_list	One or more expressions separated by commas, and optionally by white space. The expressions can have string or integer values; `print` automatically handles the conversion of integer values to ASCII.

Comments The `print` routine writes the values of each expression to standard output, each followed by a single blank, in the order in which they are specified in *`exp_list`*. Specifically, the `printf` format equivalents for `print` output are `"%d "` for integer expressions and `"%s "` for string expressions. Because it does not return a value, `print` cannot be used as an expression.

Example This example writes the string `The square of 7 is 49 \n` to standard output. The newline is added to the `print` output because it was explicitly requested:

```
print "The square of", 7, "is", 7*7, "\n";
```

This example writes the string `0 1 2 3 4` to standard output. Recall that the `srand` routine always returns the integer value 1.

```
i = 4;
j = 2;
print i<j, j<i, j, srand(i+j) + j, i;
```

See Also `mkprintable` `printf` `fprintf` `sprintf`

printf, fprintf, sprintf

Library Routine

Description Writes specified output to standard output, to a file, or to a string variable.

Syntax

int **printf** (*format_str* [, *arg_list*])
int **fprintf** (*file_des*, *format_str* [, *arg_list*])
int **sprintf** (*location*, *format_str* [, *arg_list*])

Syntax Element	Description
format_str	A string expression that specifies the format in which the output is written.
arg_list	The output to be written. Separate multiple arguments with a comma.
file_des	The integer file descriptor, obtained from `open`, of the file to which the output is written
location	The address of the string variable (`&str1`) to which the output is written. Additional space is allocated if the output exceeds the size of the current string.

Comments If `printf`, `fprintf`, or `sprintf` successfully writes the requested output, it returns the number of characters written. If the routine is unable to write the output as requested, it generates a runtime error.

The `printf`, `fprintf`, and `sprintf` routines are closely related; the difference among them is where they write the specified output: a file, standard output, or a string variable.

format_str and *arg_list* are like the output format and arguments in the C library routines `printf`, `fprintf`, and `sprintf`, with the following exceptions:

- Floating-point conversion characters (e, E, f, F, g, G) are not allowed. They are unnecessary because the VU language does not have floating-point values.
- The use of `*` to specify a field width or precision taken from the corresponding argument is not supported.

- Integer conversion characters (d, o, u, x, X) are automatically prefixed by the character 'l' in keeping with the VU language treatment of all integers as 32 bits. This is transparent; if you explicitly specify the 'l', no change is made.
- *format_str* and *arg_list* are checked at runtime to ensure that their syntax is correct, that every conversion specification has an argument, and that each argument is the correct type for the corresponding conversion specification. As in C, extra arguments are ignored.

Example In this example, assume that the value of the dividend is 3 and the value of the divisor is 9:

```
printf("%d is %d%% of %d",
    dividend, (100*dividend)/divisor, divisor);
```

The following line is printed on standard output:

```
 3 is 33% of 9
```

In this example, assume that the value of `arg1` is 12 and the value of `arg2` is 6:

```
fprintf(outfile_des,
    "%X (HEX) is %s than %d (decimal)", arg1,
    arg1 > arg2 ? "greater" : "equal to or less", arg2);
```

The following line is written to the file whose descriptor is `outfile_des:`

```
C (HEX) is greater than 6 (decimal)
```

If `arg1` is 63 and `arg2` is 64, the line written to the file is:

```
3F (HEX) is equal to or less than 64 (decimal)
```

In this example, if the value of `char_arg` is the character $, then `data_str` is assigned the value \044:

```
sprintf(&data_str, "\%.3o", char_arg);
```

See Also `print mkprintable`

push

Environment Control Command

Description Pushes the value of a VU environment variable to the top of the stack.

Syntax **push** [*env_assign_list*];

Syntax Element	Description
env_assign_list	A list of one or more environment variable assignments, of the form *env_var = expr*, where *env_var* is any VU environment variable and *expr* is an expression separated by commas and optionally by white space. If *env_assign_list* contains one item, the brackets are optional. If *env_assign_list* contains more than one item, push operates on them from left to right.

Comments For each *env_var* in *env_assign_list*, the corresponding value of *expr* is pushed to the top of that *env_var*'s stack. Thus, *expr* becomes the current value of that *env_var* and the previous value becomes the next-to-top element of that *env_var*'s stack.

Example This example disables the normal checking for any queued suspend requests, and encapsulates this disabling within the push and pop commands:

```
push Suspend_check off;
/* code that performs input emulation commands where you do not want
suspend or step operations to stop */
pop Suspend_check;
```

This example shows how to change the values in the stack:

```
/* Set values for Timeout_val and Log_level. */
set [Timeout_val = 120000, Log_level = TIMEOUT];

/* Set the current values of Timeout_val to 60000, and save the value.
The current and saved values of are 60000. */
set Timeout_val = 60000;
save Timeout_val;

/* Push 30000 to the top of the Timeout_val stack, making it the cur-
rent value. 60000 is now the second element on the stack. */
push Timeout_val = 30000;

/* Write values to standard output. */
show [Timeout_val, Log_level];
Timeout_val = 30000
Log_level = TIMEOUT

/* Set the current value of Timeout_val to 20000. The Timeout_val
stack now contains 20000 and 60000. */
set Timeout_val = 20000;

/* Push ALL to the top of the Log_level stack, making it the current
value. TIMEOUT is now the second element on that stack. */
push Log_level = "ALL";
```

See Also pop eval set

putenv

Library Routine

Description Sets the values of Windows NT or UNIX environment variables from within a virtual user script.

Syntax `int putenv (string)`

Syntax Element	Description
string	A string expression of the form *name=value* specifying the environment variable name and value.

Comments The `putenv` routine, like the C routine of the same name, sets the values of Windows NT or UNIX environment variables from within a virtual user script.

If `putenv` completes successfully, it returns a value of 0. Otherwise, it returns a nonzero value.

Example This example sets `LIMIT` to 100:

```
string name;
string value;

name = "LIMIT";
value = "100";

putenv (name + "=" + value);
```

See Also getenv

rand

Library Routine

Description Returns a random integer in the range 0 to 32767.

Syntax `int rand ()`

Comments The `rand` routine is similar to its corresponding C library routine but does a better job of generating random numbers.

The `rand`, `srand`, `uniform`, and `negexp` routines enable the VU language to generate random numbers. The behavior of these random number routines is affected by the way you set the **Seed** and **Seed Flags** options in a LoadTest schedule. By default, the **Seed** generates the same sequence of random numbers but sets unique seeds for each virtual user, so that each virtual user has a different random number sequence. For more information about setting the seed and seed flags in a schedule, see the *Using Rational LoadTest* manual.

`srand` uses the argument *`seed`* as a seed for a new sequence of random numbers to be returned by subsequent calls to the `rand` routine. If `srand` is then called with the same seed value, the sequence of random numbers is repeated. If `rand` is called before any calls are made to `srand`, the same sequence is generated as when `srand` is first called with a seed value of 1.

Example This example sets a random delay. It first defines a maximum delay of 10 seconds, and then uses the `rand` routine to delay a random amount of time from 0 to 10 seconds:

```
#define MaxDelay 10

{
    delay_time = rand() % (MaxDelay + 1);
    delay(delay_time * 1000);
}
```

See Also `uniform` `negexp` `srand`

ReadLine

VU Toolkit Function: File I/O

Description Reads a line from the open file designated by *`file_descriptor`*.

Syntax

```
#define _PV__FILEIO_NOWRAP
#define _PV_FILEIO_COMMENT  "delimiter characters"
#define _PV_FILEIO_WHITESPACE "whitespace characters"
#define _PV_FILEIO_BLANKLINE
#include <sme/fileio.h>
func ReadLine(file_descriptor)
int file_descriptor;
```

Syntax Element	Description
delimiter characters	The characters that delimit comments. The default delimiter is a # . All text following a comment delimiter, up to end of line, is removed. Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter. All text following a comment delimiter, up to end of line, is removed.
whitespace characters	The characters that are considered as white space for trimming the line read. The default is the tab character (\t). Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter.
file_descriptor	The open file that you want to read.

Comments

The `ReadLine` function returns a single line of data from the open file identified by *`file_descriptor`*. In processing the file, the following actions occur:

- Lines beginning with a comment delimiter are skipped.
- Trailing comments are removed from the line.
- All white space is removed from the end of the line (trimming occurs after comments have been removed).
- Blank lines (after trimming comments and white space) are skipped.
- A line consisting only of the tilde character (~) results in a blank line being read.
- `ReadLine` returns 1 if successful, and –1 if no data is read.

By default, `ReadLine` skips any line that is only white space, and wraps back to the top of the file when the end of file is reached. The function returns 1 on success, and –1 on failure. The string variable `Last_Line` contains the line read by the most recent successful invocation of `ReadLine`.

When the macro constant `_PV_FILEIO_NOWRAP` is defined, `ReadLine` returns failure after reaching the end of the file. The default behavior is to wrap back to the top of the file.

The macro constant `_PV_FILEIO_COMMENT` allows you to redefine the characters that are considered as comment delimiters.

The macro constant `_PV_FILEIO_WHITESPACE` defines the characters that are considered as white space for trimming the line read. The default is the tab character (\t).

The macro constant `_PV_FILEIO_BLANKLINE` defines a string that, when read as the only item in a line, returns a blank line. The default string is `"~"`. Setting this string to null (`""`) disables skipping of blank lines, and returns a blank line if the input contains only white space, or white space followed by a comment.

Example This example opens a file and inserts data until the end of the file:

```
#include <VU.h>
#define _PV_FILEIO_NOWRAP     1
#define _PV_FILEIO_FIELD     ","
#include <sme/fileio.h>

#define IDX_STUDENT           1   /* STUDENT is 1st field */
#define IDX_CLASS             2   /* CLASS is 2nd field */
#define IDX_GRADE             3   /* GRADE is 3rd field */

{
  /* open input data file for transaction A */
  transA_fd = open ("transA_input_file", "r");

  /* loop until input data is exhausted */
  while (ReadLine(transA_fd) != -1)
  {
        sqlexec ["Insert A"]
        "INSERT INTO REPORTCARD (STUDENT, CLASS, GRADE) VALUES ("
                + IndexedField(IDX_STUDENT) + ", "
                + IndexedField(IDX_CLASS)   + ", "
                + IndexedField(IDX_GRADE)   + ") ";
  }
}
```

See Also `NextField` `IndexedField` `NextSubField` `IndexedSubField` `SHARED_READ`

reset

Environment Control Command

Description Changes the current value of a VU environment variable to its default value, and discards all other values in the stack.

Syntax **reset [***env_var_list***];**

Syntax Element	Description
env_var_list	Use one of the following for *env_var_list*: ▶ A list of one or more environment variables, separated by commas and optionally by white space. If *env_var_list* contains one item, the brackets are optional. If *env_var_list* contains more than one item, reset operates on them from left to right. ▶ ENV_VARS. This specifies all of the environment variables.

Comments The current value of each variable in *env_var_list* is set to that variable's default value. All other values on that variable's stack are discarded. The default and saved values of the variables in *env_var_list* are unchanged.

Example This example changes the values for Timeout_val and Log_level, clears the stack, and then sets the values to their default values.

```
/* Set values for Timeout_val and Log_level. */
set [Timeout_val = 120000, Log_level = TIMEOUT];

/* Set the current values of Timeout_val to 60000, and save the value.
The current and saved values of are 60000. */
set Timeout_val = 60000;
save Timeout_val;

/* Push 30000 to the top of the Timeout_val stack, making it the cur-
rent value. 60000 is now the second element on the stack. */
push Timeout_val = 30000;

/* Reset the Timeout_val and Log_level */
reset [Timeout_val, Log_level];
show [Timeout_val, Log_level];
Timeout_val = 120000
Log_level = TIMEOUT
```

See Also set

restore

Environment Control Command

Description Makes the saved value of a VU environment variable the current value.

Syntax **restore [***env_var_list***];**

Syntax Element	Description
env_var_list	Use one of the following for *env_var_list*: ▶ A list of one or more environment variables, separated by commas and optionally by white space. If *env_var_list* contains one item, the brackets are optional. If *env_var_list* contains more than one item, restore operates on them from left to right. ▶ ENV_VARS. This specifies all of the environment variables.

Comments The current value of each variable in *env_var_list* is set to that variable's saved value. The saved values of the variables in *env_var_list* are unchanged. This is the inverse of the save command.

Example This example sets Timeout_val to 60000 ms, saves this value to the stack, sets Timeout_val to 30000 ms, and then restores the value to 60000 ms:

```
set Timeout_val = 60000;
save Timeout_val;
set Timeout_val = 30000;
restore Timeout_val;
show Timeout_val;
```

See Also save reset

save

Environment Control Command

Description Saves the value of a VU environment variable.

Syntax **save [***env_var_list***];**

Syntax Element	Description
env_var_list	Use one of the following for *env_var_list*: ▶ A list of one or more environment variables, separated by commas and optionally by white space. If *env_var_list* contains one item, the brackets are optional. If *env_var_list* contains more than one item, save operates on them from left to right. ▶ ENV_VARS. This specifies all of the environment variables.

Comments The saved value of each variable in *env_var_list* is set to that variable's current value. The current values of the variables in *env_var_list* are unchanged. This is the inverse of the restore command.

Example This example sets Timeout_val to 60000 ms, saves this value to the stack, sets Timeout_val to 30000 ms, and then restores the value to 60000 ms:

```
set Timeout_val = 60000;
save Timeout_val;
set Timeout_val = 30000;
restore Timeout_val;
show Timeout_val;
Timeout_val = 60000
```

See Also restore

SaveData

VU Toolkit Function: Data

Description Stores the data returned by the most recent sqlnrecv command into a data set.

Syntax

```
#define _PV_FILEIO_REBUILD
#include <sme/data.h>
proc SaveData(data_name)
string data_name;
```

Syntax Element	Description
data_name	A string that names the data that is saved.

Comments This procedure stores the data retrieved by the most recent sqlnrecv command. Once saved, the data can be referenced using the name given in the string argument *data_name*.

After the data is stored, the column headers are examined to determine the number and size of the columns. This information is stored for use by the functions that parse the data based on rows and columns. Because this is an expensive operation, it is performed only the first time a data set is created using this name, or when the name has been cleared using the FreeData command.

If a data set already exists with the given name, the data is replaced but the field definitions are retained. If the new data does not have the same structure as the original, the results of subsequent attempts to parse the fields are undefined. To avoid this problem, you can create different data sets for different sets of queries, or you can explicitly clear the data set with FreeData before doing the next SaveData.

The stored data sets and their field definitions persist across script boundaries.

The macro constant _PV_DATA_REBUILD, when defined, forces SaveData to re-compute field counts and sizes for every call, even if the data set already exists with this name. While it provides an extra degree of protection from using the same name for different types of data sets, if also increases the amount of processing required in the script.

Because data is stored using only the results of the most recent sqlnrecv command, any VU environment variables that affect the data returned also affect this function. In particular, it assumes that only one table was fetched. If Table_boundaries is set to "OFF" and multiple tables are retrieved, the results of this function and subsequent data commands on the stored data have undefined results.

Example

This example saves the data retrieved in the tmp_results buffer, stores the second field in accessprofile_id, then frees tmp_results.

```
#include <VU.h>
#include <sme/data.h>

{
  string accessprofile_id;

  sqlexec ["test_gr003"]
    "select PASSWORD, ACCESSPROFILEID, INACTIVE, "
    "PW_UPDATE_DT from USERACCOUNT where NAME = ’davidj’";
  sqlnrecv  ["test_gr004"] ALL_ROWS;

  SaveData ("tmp_results");
  accessprofile_id = GetData1("tmp_results", 2);
  FreeData ("tmp_results");

  sqlexec  ["test_gr005"]
     "select LOGONNAME, LOGONPASSWD, EXP_DAYS from "
     "ACCESSPROFILE where ACCESSPROFILEID = "
     + accessprofile_id;
}
```

See Also

AppendData FreeAllData FreeData GetData GetData1

scanf, fscanf, sscanf

Library Routine

Description

Reads specified input from standard input, a file, or a string expression.

Syntax

```
int scanf (control_str [, ptr_list])
int fscanf (file_des, control_str [, ptr_list])
int sscanf (str, control_str [, ptr_list])
```

Syntax Element	Description
control_str	A string expression that specifies how to interpret the input that is read.
ptr_list	Specifies where the input is placed after it is read.
file_des	The integer file descriptor, obtained from `open`, of the file from which the input is read.
str	A string expression from which the input is taken.

Comments

The `scanf`, `fscanf`, and `sscanf` routines return the number of input items successfully read and assigned even if this is less than the requested number. Each returns EOF (as defined in the standard VU header file) if the input ends before the first attempt to match the format control string.

The `scanf`, `fscanf`, and `sscanf` routines are closely related, the difference among them is where they read the specified input.

Specify *control_str* and *ptr_list* like the format control string and pointer arguments in the C library routines `scanf`, `fscanf`, and `sscanf`, with the following exceptions:

- If a maximum field width is not given for a string conversion specification (for example as in `%s` or `%[a-z]`), a width of 100 is inserted. Therefore, if you expect a string exceeding 100 characters, specify an appropriately large field width. Unused space is freed after the assignment is made, so a large field width does not waste space.
- Floating-point conversion characters (e, E, f, F, g, G) are not allowed. They are unnecessary, because the VU language does not have floating-point values.
- Integer conversion characters (d, o, u, x) are transparently changed to uppercase to indicate that their corresponding pointer arguments are addresses of 32-bit (non-shared) integer variables.
- *control_str* and *ptr_list* are checked at runtime to ensure that their syntax is correct, that every conversion specification has a pointer argument, and that each pointer argument is an address of the correct variable type (non-shared integer or string) for the corresponding conversion specification. Pointers to arguments are not allowed. As in C, extra pointer arguments are ignored.

These routines stop reading input if they encounter the end of the file, after they have handled the entire *control_str*, or if input data conflicts with the format control string. The conflicting data is left unread.

Example In this example, if the string abcdefg is supplied on standard input, then the string abc is assigned to part1 and the string defg is assigned to part2.

```
scanf("%3s%s", &part1, &part2);
```

In this example, if the file with file descriptor infile_des contains the characters abcde 12345, then the string abcde is assigned to str1 and num is assigned the integer 12345.

```
fscanf(infile_des, "%[a-zA-Z]%d", &str1, &num);
```

In this example, if the value of the string data_str is \044, then the character $ (or equivalently the decimal value 36) is assigned to char_arg:

```
sscanf(data_str, "\%3o", &char_arg);
```

See Also None.

script_exit

Library Routine

Description Exits from a script.

Syntax *int* **script_exit** (*msg_str*)

Syntax Element	Description
msg_str	A string expression specifying an optional message to be written to the standard error file.

Comments The script_exit routine causes the current script to exit immediately. If *msg_str* is not of zero length, it is written (before exiting the script) to standard error, preceded by the following explanatory line of text:

```
Script script_name exited at user's request with message:
```

script_name is replaced by the appropriate script name (corresponding to the read-only variable _script). Virtual user execution continues with the next scheduled script, just as if the current script had completed normally. Therefore, script_exit never returns, although for syntactical purposes its return value is considered to be an integer.

Example This example causes the current script to exit. No message is written to standard error. Emulation proceeds with the next scheduled script, if any:

```
script_exit("");
```

See Also user_exit

set

Environment Control Command

Description Sets a VU environment variable to the specified expression.

Syntax **set** [*env_assign_list*];

Syntax Element	Description
env_assign_list	A list of one or more environment variable assignments, of the form *env_var = expr*, where *env_var* is any VU environment variable and *expr* is an expression separated by commas and optionally by white space. If *env_assign_list* contains one item, the brackets are optional. If *env_assign_list* contains more than one item, set operates on them from left to right.

Comments The current value of each *env_var* in *env_assign_list* is replaced by the value of the corresponding *expr*.

Example This example sets the Timeout_val and Log_level values and writes them to standard output.

```
set [Timeout_val = 60000, Log_level= ALL];
show [Timeout_val, Log_level];
```

See Also None.

set_cookie

Emulation Function

Description Adds a cookie to the cookie cache.

Syntax **set_cookie**(*name*, *value*, *domain*, *path* [, *secure*])

Syntax Element	Description
name	A string expression that specifies the name of the cookie.
value	A string expression that specifies the value for the cookie.

Syntax Element	Description
domain	A string expression that specifies the domain for which this cookie is valid.
path	A string expression that specifies the path for which this cookie is valid.
secure	An optional string expression that, if given, provides the secure modifier for the cookie. The value of this parameter should be `"secure"`.

Comments The `set_cookie` function creates the named cookie with the given value. If a cookie already exists with this name for the given domain and path then `set_cookie`() sets the value of that cookie to *`value`*.

The expiration date of the cookie is set sufficiently in the future that it will not expire during the run.

Example This example adds a secure cookie named `AA002` for domain `avenuea.com` and path `/`.

```
set_cookie("AA002", "00932743683-
101023411/933952959", ".avenuea.com", "/",
"secure");
```

See Also `COOKIE_CACHE` `expire_cookie`

SHARED_READ

VU Toolkit Function: File I/O

Description Allows multiple users to share a file.

Syntax

```
#define _PV__FILEIO_NOWRAP
#define _PV_FILEIO_COMMENT  "delimiter characters"
#define _PV_FILEIO_WHITESPACE "whitespace characters"
#define _PV_FILEIO_BLANKLINE
#include <sme/fileio.h>
shared prefix_lock, prefix_offset;
SHARED_READ(filename, prefix)
```

Syntax Element	Description
delimiter characters	The characters that delimit comments. The default delimiter is a # . All text following a comment delimiter, up to end of line, is removed. Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter. All text following a comment delimiter, up to end of line, is removed.
whitespace characters	The characters that are considered as white space for trimming the line read. The default is the tab character (\t). Do not separate delimiter characters with white space or any other character. Multiple contiguous occurrences of the delimiter are considered as a single delimiter.
prefix_lock	A variable to ensure that only one user at a time accesses the file.
*prefix*_offset	A variable to keep track of the next location to be read.
filename	The name of the shared file.
prefix	Any string constant (for example, myfile_lock and myfile_offset). *prefix* is not a string constant, but is a tag the precompiler uses to create the actual variable name; do not enclose the prefix tags in quotes.

Comments

SHARED_READ provides coordinated access by multiple users to the file specified by the *filename* argument, such that no two users retrieve the same line of data.

Two shared variables are used to coordinate the reads. These must be defined in your script with the names matching the format *prefix*_lock and *prefix*_offset.

SHARED_READ opens the file and closes it again upon exiting. SHARED_READ uses the ReadLine function to perform the actual file I/O, therefore all of the comments and white space processing described under ReadLine apply to SHARED_READ. The NextField and IndexedField functions can also be used after a SHARED_READ.

The string variable Last_Line contains the line of data returned by the most recent call to SHARED_READ.

When the macro constant _PV_FILEIO_NOWRAP is defined, SHARED_READ returns failure after reaching the end of the file. The default behavior is to wrap back to the top of the file.

The macro constant `_PV_FILEIO_COMMENT` allows you to redefine the characters that are considered as comment delimiters. All text following a comment delimiter, up to end of line, is removed.

The macro constant `_PV_FILEIO_WHITESPACE` defines the characters that are considered as white space for trimming the line read. The default is the tab character (\t).

The macro constant `_PV_FILEIO_BLANKLINE` defines a string that, when read as the only item in a line, returns a blank line. The default string is `"~"`. Setting this string to null (`""`) disables skipping of blank lines, and returns a blank line if the input contains only white space, or white space followed by a comment.

Example

```
#include <VU.h>
#define _PV_FILEIO_NOWRAP     1
#define _PV_FILEIO_FIELD     ","
#include <sme/fileio.h>

#define IDX_STUDENT           1   /* STUDENT is 1st field */
#define IDX_CLASS             2   /* CLASS is 2nd field */
#define IDX_GRADE             3   /* GRADE is 3rd field */
{
  shared transA_lock, transA_offset;

  while (1)
  {
      SHARED_READ("transA_input_file", transA);
      if (Last_line == "")
          break;
      sqlexec [Insert A"]
        "INSERT INTO REPORTCARD (STUDENT, CLASS, GRADE) VALUES ("
                + IndexedField(IDX_STUDENT) + ", "
                + IndexedField(IDX_CLASS)   + ", "
                + IndexedField(IDX_GRADE)   + ") ";
  }
}
```

See Also `IndexedField` `IndexedSubField` `NextField` `ReadLine` `NextSubField`

show

Environment Control Command

Description Writes the current values of the specified variables to standard output.

Syntax **show [***env_var_list***];**

Syntax Element	Description
env_var_list	Use one of the following for *env_var_list*: ▶ A list of one or more environment variables, separated by commas and optionally by white space. If *env_var_list* contains one item, the brackets are optional. If *env_var_list* contains more than one item, `show` operates on them from left to right. ▶ `ENV_VARS`. This specifies all of the environment variables.

Comments The `show` command does not alter any values of environment variables. `show` does not escape unprintable characters when printing string expression values. For bank variables, strings are listed first (enclosed in double quotation marks), followed by integers.

Example This example writes the values of `Timeout_val` and `Log_level` to standard output:

```
show [Timeout_val,Log_level];
Timeout_val = 120000
Log_level = TIMEOUT
```

See Also None.

sindex

Library Routine

Description Returns the position of the first occurrence of any character from a specified set.

Syntax `int` **sindex** (*str*, *char_set*)

Syntax Element	Description
str	The string expression to search.
char_set	The characters to search for within *str*.

Comments The `sindex` (string index) routine returns the ordinal position within *str* of the first occurrence of any character from *char_set*. If no occurrences are found, `sindex` returns an integer value of `0`.

The routines `cindex`, `lcindex`, `sindex`, and `lsindex` return positional information about either the first or last occurrence of a specified character or set of characters within a string expression. `strspan` returns distance information about the span length of a set of characters within a string expression.

Example This example returns the integer value 2, because 2 is the position of the first vowel in the string `"moo goo gai pan"`:

```
sindex("moo goo gai pan", "aeiou");
```

See Also `cindex` `lcindex` `lsindex` `strspan` `strstr`

sock_connect

Emulation Function

Description Opens a socket connection.

Syntax *int* **sock_connect** (*label, address*)

Syntax Element	Description
label	A string expression that identifies the name of the connection.
address	A string expression of the form *host*:*port*. *port* is required. *host* is a symbolic host name or an IP address in dotted-decimal form. Equivalent examples: `"calvin:80"` and `"152.52.110.86:80"` (Assuming calvin's IP address is `152.52.110.86`).

Comments The `sock_connect` function returns an integer value: 0 or less for failure, or a unique connection number greater than or equal to 1 for success. If `sock_connect` fails, an entry is written to `_error` and `error_text`.

The `sock_connect` function makes a connection to the server defined by *address*, and identifies the name of this connection as *label* (for the Trace report output). Supply a descriptive name to make it easier to identify the connection when you examine the outputs.

The `sock_connect` function sets the "first connect" (`_fc_ts`) and "last connect" (`_lc_ts`) read-only variables.

The `sock_connect` function is affected by the following VU environment variables: `Record_level`, `Timeout_val`, `Timeout_scale`, `Timeout_act`, `Connect_retries`, and `Connect_retry_interval`.

Example This example connects to a computer named calvin. The connection number is returned in the variable `conn1`:

```
int conn1
conn1 = sock_connect("calvin", "152.52.110.86:25");
```

See Also `sock_disconnect`

sock_create

Emulation Function

Description Creates a socket to which another process may connect.

Syntax `int sock_create ( [service | port [, type [, backlog]]] )`

Syntax Element	Description
service	A string expression that names the service whose port is to be used.
port	An integer expression specifying the port to use.
type	An integer specifying the type of socket to create The only currently supported type is `SOCK_TYPE_STREAM`, defined in VU.h.
backlog	An integer specifying the maximum number of pending incoming connections. The default is 1.

Comments LoadTest automatically generates the VU code necessary to accept incoming socket connections from a server by inserting the following emulation commands in your socket script: `sock_create`, `sock_fdopen`, `sock_isinput`, and `sock_open.`

The `sock_create` function creates an Internet socket and prepares for incoming connections. It returns the port of the created socket.

The desired port for the created socket may be specified by either a service name or by a port number. If the port is not specified or is given as 0, the socket uses a system-assigned port.

Example This example creates a socket on port 80 and then waits for a connection to be made on that socket:

```
int port, con;

port = sock_create(80);

/* do something here to let other process know that
   socket is ready for connections. */

con = sock_open("sock_open", port);
set Server_connection = con;
sock_nrecv 1;
```

See Also `sock_open` `sock_connect` `sock_fdopen`

sock_disconnect

Emulation Function

Description Disconnects a socket connection.

Syntax *int* **sock_disconnect** (*connection*)

Syntax Element	Description
connection	An integer expression specifying a connection number that has been returned by `sock_connect` and has not been disconnected. If *connection* is invalid, `sock_disconnect` generates a fatal runtime error.

Comments The `sock_disconnect` function returns 1 for success and 0 for failure.

Example This example disconnects the connection `conn1`:

```
sock_disconnect(conn1);
```

See Also `sock_connect`

sock_fdopen

Emulation Function

Description Associates a file descriptor with a socket connection.

Syntax `int` **`sock_fdopen`** `(`*`label`*`,` *`fd`*`)`

Syntax Element	Description
label	A string expression that identifies the name of the connection.
fd	An integer expression that identifies the file descriptor of a socket created by external C code.

Comments LoadTest automatically generates the VU code necessary to accept incoming socket connections from a server by inserting the following emulation commands in your socket script: `sock_create`, `sock_fdopen`, `sock_isinput`, and `sock_open`.

The `sock_fdopen` function returns an integer value: 0 or less for failure, or a unique connection number greater than or equal to 1 for success. The `sock_fdopen` function assigns the given file descriptor to a connection and identifies the name of this connection as label (for the Trace report output). The *`fd`* parameter must be a file descriptor for a socket connection created by an external C function.

The `sock_fdopen` function is affected by the `Record_level` VU environment variable.

Example This example creates a specialized socket via the external C function and then uses that socket as the current `Server_connection`.

```
external_C int func make_socket()
{}

int fd, con;

fd = make_socket();

con = sock_fdopen(“sock_fdopen”, fd);

set Server_connection = con;

sock_nrecv 1;
```

See Also `sock_create` `sock_connect` `sock_open`

sock_isinput

Emulation Function

Description Checks for available input on a socket connection.

Syntax *`int`* **`sock_isinput`** `()`

Comments LoadTest automatically generates the VU code necessary to accept incoming socket connections from a server by inserting the following emulation commands in your socket script: `sock_create`, `sock_fdopen`, `sock_isinput`, and `sock_open`.

The `sock_isinput` function returns an integer value equal to the number of characters currently available on the socket connection that have not been read by any of the socket receive commands. This function does not process the incoming data. Incoming data is still available for processing by a socket receive emulation command.

The `sock_isinput` function is affected by the `Server_connection` VU environment variable.

Example This example conditionally reads the data from the socket until no more data exists. This example is useful as a substitute for a `sock_nrecv [cmd_id]$` command. Although the `$` tells LoadTest to read until the end of file, the command does not terminate if the socket is not closed by the server.

```
Set Server_connection = conn1;
if (n = sock_isinput())
    sock_nrecv n;
```

See Also `sock_nrecv`

sock_nrecv

Receive Emulation Command

Description Receives *n* bytes from the server.

Syntax `int` **`sock_nrecv`** `[`*`cmd_id`*`]` *`n_bytes`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
n_bytes	An integer expression, specifying the number of bytes to read from the connection identified by `Server_connection`.

Comments The `sock_nrecv` command receives *n_bytes* from the server specified by the VU environment variable `Server_connection`. Binary data is translated into embedded hexadecimal strings. See *Unprintable HTTP or Socket Data* on page 3-35.

If `Timeout_val` (subject to scaling) milliseconds elapses before `sock_nrecv` is satisfied, it fails and returns 0. Otherwise, it passes and returns 1.

The sock_nrecv command is affected by the following VU environment variables: Timeout_act, Timeout_val, Timeout_scale, Log_level, Record_level, Max_nrecv_saved, and Server_connection.

Max_nrecv_saved applies to the actual data received, before expanding any binary data into embedded hexadecimal strings.

Example This example receives 1355 bytes from the server conn1:

```
Set Server_connection = conn1;
sock_nrecv ["cmd001"] 1355;
```

See Also sock_recv sock_send sock_isinput

sock_open

Emulation Function

Description Waits for a socket connection from another process.

Syntax int **sock_open** (*label*, *port*)

Syntax Element	Description
label	A string expression that identifies the name of the connection
port	An integer expression that identifies the port of a socket created by sock_create.

Comments LoadTest automatically generates the VU code necessary to accept incoming socket connections from a server by inserting the following emulation commands in your socket script: sock_create, sock_fdopen, sock_isinput, and sock_open.

The sock_open function returns an integer value: 0 or less for failure, or a unique connection number greater than or equal to 1 for success. If sock_open fails, an entry is written to _error and _error_text.

The sock_open function waits for a connection from another process and identifies the name of this connection as *label* (for the Trace report output). The port parameter must be a port returned by sock_create.

The sock_open function sets the “first connect” (_fc_ts) and “last connect” (_lc_ts) read-only variables.

The sock_open function is affected by the following VU environment variables: Record_level, Timeout_val, Timeout_scale, and Timeout_act.

Example This example creates a socket on port 80 and then waits for a connection to be made on that socket:

```
int port, con;

port = sock_create(80);
/* do something here to let other process know that
   socket is ready for connections */

con = sock_open("sock_open", port);
set Server_connection = con;
sock_nrecv 1;
```

See Also sock_create sock_connect sock_fdopen

sock_recv

Receive Emulation Command

Description Receives data until the specified delimiter string is found.

Syntax int **sock_recv** [*cmd_id*] *reply*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
reply	A string expression specifying the desired reply from the server. Data is received from the connection identified by Server_connection until *reply* is encountered. *reply* can contain the following special characters: ▶ ^ (carat). As the first character in *reply*, the carat signifies binding to the beginning of the response, such as that used in VU regular expressions for the match() built-in function. It is considered an error if no characters follow the ^. ▶ $ (dollar sign). As the last character in *reply*, the dollar sign signifies binding to the end of the response (for example, the end of the connection) such as that used in VU regular expressions for the match() built-in function. If no characters precede the $, sock_recv reads until the end of connection, thus matching any combination of 0 or more received characters. To override the special meaning of ^ and $, escape them with a backslash or use embedded hex string notation (5e for the carat and 24 for the dollar sign). When used anywhere else within *reply*, the carat and dollar sign have no special meaning.

Comments This command returns data until the specified pattern appears. Binary data is translated into embedded hexadecimal strings. See *Unprintable HTTP or Socket Data* on page 3-35.

If `Timeout_val` (subject to scaling) milliseconds elapses before `sock_recv` is satisfied, it fails and return 0. Otherwise, it passes and returns 1.

The `sock_recv` command is affected by the following VU environment variables: `Timeout_act`, `Timeout_val`, `Timeout_scale`, `Log_level`, `Record_level`, `Max_nrecv_saved`, and `Server_connection`.

`Max_nrecv_saved` applies to the actual data received, before expanding any binary data into embedded hexadecimal strings.

Example This example matches as soon as the string `"This is an extremely small file\r\n"` is found anywhere within the response:

```
sock_recv ["cmd001r"]  "This is an extremely small file\r\n";
```

This example reads until the end of the connection, and passes only if `_response` ends with `"This is an extremely small file\r\n"`:

```
sock_recv ["cmd002r"]  "This is an extremely small file\r\n$";
```

This example matches only if the first 20 characters of `_response =="This is an extremely"`. If the first 20 characters do not match, `sock_recv` continues to read until the end of the connection or a timeout.

```
sock_recv ["cmd003r"]  "^This is an extremely";
```

This example reads until the end of the connection. It fails only if `Timeout_val` (subject to scaling) milliseconds expires before reaching the end of the connection:

```
sock_recv ["cmd003r"]  "$";
```

See Also `sock_nrecv` `sock_recv`

sock_send

Send Emulation Command

Description Sends data to the server.

Syntax `int` **`sock_send`** `[`*`cmd_id`*`]` *`data`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
data	A string expression that is parsed for embedded hexadecimal strings delimited by grave accent (‘) characters. See *Unprintable HTTP or Socket Data* on page 3-35.

Comments The `sock_send` command sends data to the connection specified by the VU environment variable `Server_connection`. The `sock_send` command returns an integer value — 0 for failure, and 1 for success.

The `sock_send` command is affected by the following VU environment variables: the think time variables, `Log_level`, `Record_level`, `Server_connection`, `Suspend_check`, `Timeout_val`, and `Timeout_scale`.

Example This example sends `"data to send"` to the server `conn1`:

```
set Server_connection = conn1;
set Think_avg = 27;
sock_send ["cmd001"] "data to send";
```

See Also `sock_nrecv` `sock_recv`

sqlalloc_cursor

Emulation Function

Description Allocates a cursor for use in cursor oriented SQL emulation commands and functions.

Syntax `int` **`sqlalloc_cursor`**`()`

Comments The `sqlalloc_cursor` function allocates a cursor for use by `sqldeclare_cursor`, `sqlopen_cursor`, `sqlcursor_setoption`, or `sqlsysteminfo`. The returned cursor ID is placed in the read-only variable`_cursor_id`.

Example This example allocates a cursor with `sqlalloc_cursor` and then uses that cursor to execute a query.

```
stmt_2_1_id = sqlalloc_cursor();

sqlcursor_setoption(stmt_2_1_id, ODBC_CURSOR_TYPE,
            ODBC_CURSOR_KEYSET_DRIVEN);
sqlcursor_setoption(stmt_2_1_id, ODBC_CONCURRENCY,
            ODBC_CONCUR_VALUES);

set Cursor_id = stmt_2_1_id;
sqlopen_cursor ["val_6001"] "", "select @@servername";

push CS_blocksize = 100;

sqlfetch_cursor ["val_6002"] stmt_2_1_id, ALL_ROWS;
set Cursor_id = 0;

sqlfree_cursor( stmt_2_1_id );
```

See Also `sqlfree_cursor` `sqlopen_cursor` `sqldeclare_cursor` `sqlcursor_setoption`

sqlalloc_statement

Emulation Function

Description Allocates a cursor data area for Oracle playback.

Syntax `int sqlalloc_statement ();`

Comments The `sqlalloc_statement function` allocates a cursor data area (CDA) for Oracle playback. The returned statement ID is placed in the read-only variable `_statement_id`.

Example This example does a `select` on `stmtid_1` and fetches one row, then it does a `select` on `stmtid_2` and fetches all rows. It then returns to `stmtid_1` and fetches the remaining rows.

```
stmtid_1=sqlalloc_statement();
set Statement_id = stmtid_1;
sqlprepare "select * from customers";

sqlexec stmtid_1;
sqlnrecv 1;
stmtid_2=sqlalloc_statement();
set Statement_id = stmtid_2;
sqlprepare "select distinct composer from products";
sqlexec stmtid_2;
sqlnrecv ALL_ROWS;
set Statement_id=stmtid_1;
sqlnrecv ALL_ROWS;
```

See Also `sqlfree_statement`

sqlclose_cursor

Send Emulation Command

Description Closes the indicated cursor.

Syntax

```
int sqlclose_cursor [ cmd_id ]
[ EXPECT_ERROR ary, ] [ EXPECT_ROWS n, ] csr_id
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. `cmd_id` has the form [`string_exp`].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in `ary`, the response is unexpected.
n	An integer that assigns the of rows this command affects. The default is -1, which indicates any number of rows. If `n` is > = 0, and the number of rows the SQL command processes does not equal `n`, the response is unexpected.
csr_id	The integer cursor identifier of an opened cursor.

Comments If the cursor ID is not valid for the connection indicated by the value of `Server_connection` or if the cursor is not open, an error is reported to both the error file and the log file.

After a cursor is closed, all cursor commands will fail except for `sqlopen_cursor` and `sqlfree_cursor`. The cursor is reopened by `sqlopen_cursor`.

`sqlclose_cursor` is affected by the VU environment variable `Server_connection`.

Example This example declares and opens the cursor, manipulates the rows in the table, and then closes the cursor:

```
/* sqlopen_cursor implicitly declares and then opens the cursor */
cursor_65537 = sqlopen_cursor [ "hand002" ] "cur",
    "SELECT * FROM Room \tFOR UPDATE OF Roomnum, Type, Capacity"
    UPDATE_CURSOR;

/* CS_blocksize is set to 1 to control the fetch api calls */
set CS_blocksize = 1;

/* 4 TDS_CURFETCH NEXT packets of 1 row each are combined
 *  into a single sqlfetch_cursor command. */
sqlfetch_cursor [ "hand003" ] cursor_65537 FETCH_NEXT, 4;
```

```
sqldelete_cursor [ "hand004" ] cursor_65537, "Room",
   "Roomnum='2017  ' Type='OFF ' Capacity='2'";
sqlfetch_cursor [ "hand005" ] cursor_65537 FETCH_NEXT;
sqlupdate_cursor [ "hand006" ] cursor_65537, "Room",
   "UPDATE Room Set Roomnum = @sql0_num , Type = @slq1_type ,"
   " Capacity = @sql2_cap ","Roomnum='2065  ' Type='OFF '"
   "Capacity='2'","2056", "lab", 4;
sqlfetch_cursor [ "hand007" ] cursor_65537 FETCH_NEXT;
sqldelete_cursor [ "hand008" ] cursor_65537, "Room",
   "Roomnum='2111  ' Type='OFF ' Capacity='3'";
sqlfetch_cursor [ "hand009" ] cursor_65537 FETCH_NEXT;
sqlupdate_cursor [ "hand010" ] cursor_65537, "Room",
   "UPDATE Room Set Roomnum = @sql0_num , Type = @slq1_type ,"
   "Capacity = @sql2_cap ","Roomnum='2220  ' Type='OFF '"
   "Capacity='3'","1111", "off", 3;
sqlfetch_cursor [ "hand011" ] cursor_65537 FETCH_NEXT, 2;
sqlclose_cursor [ "hand012" ] cursor_65537;
```

See Also sqlopen_cursor

sqlcommit

Emulation Function

Description Commits the current transaction.

Syntax int **sqlcommit**()

Comments The sqlcommit function is not supported for Sybase and Microsoft SQL Server databases. For Sybase and Microsoft SQL Server databases, use:

```
sqlexec "commit transaction";
```

Using sqlcommit on Sybase or Microsoft SQL Server database produces a fatal runtime error.

sqlcommit is affected by the VU environment variable Server_connection.

Example In this example, a connection is made to the t:calvin:PAC server. The sqlexec expects commands to modify data in an Oracle database. The data is committed to the database and, then the connection is disconnected.

```
#include <VU.h>
{
t_calvin_PAC = sqlconnect("t_calvin_PAC", "scott", "tiger",
     "t:calvin:PAC", "oracle7.3");
set Server_connection = t_calvin_PAC;
sqlexec  ["school001"] "alter session set nls_language= 'AMERICAN' "
     "nls_te"rritory= 'AMERICA'";
sqlexec  ["school002"] "select * from student";
sqlnrecv ["school003"] ALL_ROWS;

sqlexec  ["school004"] "insert into student VALUES (1,'LAURA', "
     "'L.L.R.', '63 Greenwood Drive, TORONTO ONT', "
     "'12-Jun-95', 'F')";
sqlcommit();
sqldisconnect(t_calvin_PAC);
}
```

See Also sqlrollback

sqlconnect

Emulation Function

Description Logs on to a SQL database server.

Syntax

```
int sqlconnect (label,database_login,pwd,
             server,server_info [, connection_opts ] )
```

Syntax Element	Description
label	A string expression that is used as the label for this server connection in LoadTest report output. If *label* has the value *""*, *database_login* and *server* arguments are combined into the default label *"database_login@server"*.
database_login	A string expression that specifies the database login ID for the connection.
pwd	A string expression that specifies the password of the database login ID.
server	A string expression that specifies the server.
server_info	A string expression that specifies a product ID that is used to locate the correct API library for playback.
connection_opts	An optional string expression that contains one or more name= 'value' pairs which give vendor-specific connection-oriented options. All *connection_opts* in automatically generated scripts are taken from the recorded session. The supported names are described below.

Comments The sqlconnect function connects *database_login* to *server* with password *pwd*. If the connection is successful, sqlconnect returns a connection ID, which is an integer for use with the Server_connection environment variable. If the connection is not successful, sqlconnect returns 0 and sets _error and _error_text.

Supported connection options are as follows:

Name	Value
TDS_VERSION	(‘n.n.n.n.‘). For Sybase and Microsoft SQL Server databases only, a sequence of integer digits that indicate the TDS version used to communicate with the server. The default is 5.0.0.0. If the server cannot support the requested TDS version, a lower version is negotiated.
APP_NAME	(’a.b.c.d.e.f.’). For Sybase and Microsoft SQL Server databases only, an optional string that indicates the application name. The value of `APP_NAME` is taken from the client login request, if present in the session. Otherwise, it does not appear in the connection option string.
PACKET_SIZE	(’x’). For Sybase only, an optional integer that indicates the size of the network packet used to communicate with the server.
DRIVER_INFO	(’value’). For ODBC only, a string that contains various ODBC related information such as 'UID= DEFAULT; PWD= DEFAULT' which causes the connect box to use the default username and password that were set up with the ODBC driver. To use the database login and password instead, remove the UID and PWD from the DRIVER_INFO value.
SQL_ODBC_CURSORS	(’value’). For ODBC only, controls what type of cursors to use for playback. The value can be set to any of the following: SQL_CUR_USE_IF_NEEDED SQL_CUR_USE_ODBC SQL_CUR_USE_DRIVER

The `sqlconnect` function is affected by the VU environment variables `Timeout_val`, `Timeout_scale`, and `Record_level`.

Example This example connects to a Sybase server, sets the server connection, and then disconnects:

```
SYBASE=sqlconnect("SERVER","ron","rondo","SYBASEC","sybase",
          "TDS_VERSION='5.0.0.0' APP_NAME='Sample App'");
set Server_connection = SYBASE;
/* emulation functions */
sqldisconnect (SYBASE);
```

See Also `sqldisconnect`

sqlcursor_rowtag

Emulation Function

Description Returns the tag of the last row fetched.

Syntax `string **sqlcursor_rowtag**(*csr_id*)`

Syntax Element	Description
csr_id	The integer cursor identifier of an opened cursor.

Comments The `sqlcursor_rowtag` function returns a string that contains a tag, or bookmark, for the last row fetched from a cursor. In custom scripts, you can use this tag later in `sqlcursor_update` and `sqlcursor_delete` statements to update or delete the specific row identified by the tag value.

The returned string is used as an argument to the emulation commands `sqldelete_cursor` and `sqlupdate_cursor`.

If you capture a SQL Server application that uses embedded SQL cursors, your script includes the `sqlcursor_rowtag` emulation function.

If you capture a Sybase application session that uses SQL cursors, this emulation function is not included in generated scripts. This is because the current row tag is always the last row fetched. Any updates or deletes are always applied to the last row fetched.

If an error occurs, `sqlcursor_rowtag` returns an empty string.

Example In this example, a cursor is opened, five rows are fetched, the current row position is saved in the `rowtag_cursor_a_id` string. The next three rows are fetched, and then the row identified by the `rowtag_cursor_a_id` value is updated.

```
#include <VU.h>
{
SYBASE = sqlconnect("SYBASE", "myuserid", "mypassword",
    "SYBASE_SERVER", "sybase11", "TDS_VERSION='5.0.0.0',
    APP_NAME='csr_disp'");

set Server_connection = SYBASE;

sqlexec  ["csrforu001"] "use pubs2";

push CS_blocksize = 5;

cursor_a_id = sqlopen_cursor ["csr002"] "cursor_a", "select * from "
  "titles where title_id in ( 'TC7777', "
```

```
  'TC3218','TC4203')",UPDATE_CURSOR;

sqlfetch_cursor ["csr003"] cursor_a_id,  5;

{string rowtag_cursor_a_id;}
rowtag_cursor_a_id = sqlcursor_rowtag(cursor_a_id);

sqlfetch_cursor ["csr003"] cursor_a_id,  3;

sqlcursor_update ["csr004"] cursor_a_id, "titles","update "
    "titles set total_sales = 9999", rowtag_cursor_a_id;

sqlfree_cursor( cursor_a_id );

sqldisconnect(SYBASE);

pop CS_blocksize;
}
```

See Also `sqldelete_cursor` `sqlupdate_cursor`

sqlcursor_setoption

Emulation Function

Description Sets a SQL cursor option.

Syntax `int` **`sqlcursor_setoption`**`(`*`csr_id`*`,` *`optioncode`* `[,` *`optarg`* `…])`

Syntax Element	Description
csr_id	The integer cursor identifier of an opened cursor.
optioncode	The integer that indicates the cursor option you want to set. The values for *optioncode* are vendor-specific. The recognized values for *optioncode* and any symbolic constants for optarg are defined in the file VU.h. Comments accompany each *optioncode*, giving the number and type of *optargs* expected.
optarg	The value that you want to supply to the cursor option. The number and type of *optargs* depend on the value of *optioncode*. The number and type of *optargs* are checked at runtime; mismatches result in a fatal runtime error.

Comments The `sqlcursor_setoption` function returns 1 for success and 0 for failure. The function sets `_error` and `_error_text`, and prints an appropriate message to standard error when `_error` is nonzero.

The `sqlcursor_setoption` function is affected by the VU environment variable `Server_connection`.

If the cursor ID is not valid for the connection indicated by the value of `Server_connection`, an error is reported to both the error file and the log file.

Example This example allocates a cursor with `sqlalloc_cursor` and then uses `sqlcursor_setoption` to set two ODBC cursor attributes before using that cursor to execute a query.

```
stmt_2_1_id = sqlalloc_cursor();

    sqlcursor_setoption(stmt_2_1_id, ODBC_CURSOR_TYPE,
                ODBC_CURSOR_KEYSET_DRIVEN);
    sqlcursor_setoption(stmt_2_1_id, ODBC_CONCURRENCY,
                ODBC_CONCUR_VALUES);

    set Cursor_id = stmt_2_1_id;
    sqlopen_cursor ["val_6001"] "", "select @@servername";

    push CS_blocksize = 100;

    sqlfetch_cursor ["val_6002"] stmt_2_1_id, ALL_ROWS;
    set Cursor_id = 0;

    sqlfree_cursor( stmt_2_1_id );
```

See Also None.

sqldeclare_cursor

Send Emulation Command

Description Associates a SQL statement with a cursor ID, which is required to open the cursor.

Syntax

```
int sqldeclare_cursor [ cmd_id ] [ EXPECT_ERROR ary, ]
csr_name, sqlstmt
[READ_ONLY_CURSOR | UPDATE_CURSOR [col_ary] ]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
csr_name	A string expression giving the name of the cursor.
sqlstmt	A previously prepared statement ID or a SQL statement string expression associated with the cursor.
col_ary	An array of strings whose values are the updatable column names. The default is all columns are updatable.

Comments The `sqldeclare_cursor` command returns an integer cursor ID for future reference by other `sql*_cursor` commands and functions. The returned cursor ID is placed in the read-only variable `_cursor_id`.

The `READ_ONLY_CURSOR` keyword indicates that the cursor is read-only. The `UPDATE_CURSOR` keyword indicates that the cursor is updatable. If neither type of cursor is specified, the text of *`sqlstmt`* determines whether the cursor is updatable.

The `sqldeclare_cursor` command is affected by the VU environment variables `Cursor_id` and `Server_connection`.

Example In this example, a connection is made to the Sybase database and a SQL statement is prepared for a SQL execution command. A cursor is then declared for the prepared SQL statement.

```
SYBASE = sqlconnect("SYBASE", "prevue", "prevue", "SYBASEC",
     "sybase", "TDS_VERSION='5.0.0.0'");
set Server_connection = SYBASE;
sqlexec  ["csrdyne001"] "USE pubs2";
stmt = sqlprepare ["csrdyne002"] "SELECT\tau_id, au_lname, au_fname,"
     "\t\t\tphone, address, city, state, \t\t\tpostalcode\t\tFROM
     \tauthors";
authors_id = sqldeclare_cursor["csrdyne003"] "authors", stmt;
sqlopen_cursor ["csrdyne004"] authors_id;
sqlfetch_cursor ["csrdyne005"] EXPECT_ROWS 5, authors_id FETCH_NEXT,
5;
```

See Also `sqlopen_cursor`

sqldelete_cursor

Send Emulation Command

Description Deletes the a row using the indicated cursor.

Syntax `int` **`sqldelete_cursor`** `[` *`cmd_id`* `] [ EXPECT_ERROR` *`ary`*`, ] [ EXPECT_ROWS` *`n`*`, ]` *`csr_id`*`,` *`tbl_name`*`,` *`rowtag`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *`cmd_id`* has the form [*`string_exp`*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *`ary`*, the response is unexpected.
n	An integer that gives the number of rows this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.

(Continued)

Syntax Element	Description
csr_id	The integer cursor identifier of an opened cursor.
tbl_name	A string expression containing the name of the table from which to delete.
rowtag	A string expression identifying the row to delete. The format of the string is SQL database vendor-specific. A valid *rowtag* can be obtained by calling `sqlcursor_rowtag`(). If *rowtag* is *""*, no row identification is used and the current row is deleted.

Comments If the cursor ID is not valid for the connection indicated by the value of `Server_connection`, an error is reported to both the error file and the log file.

The `sqldelete_cursor` command is affected by the VU environment variable `Server_connection`.

Example This example opens and fetches 4 rows from a cursor, and then deletes a row and closes the cursor:

```
/* sqlopen_cursor implicitly declares and then opens the cursor */
cursor_65537 = sqlopen_cursor [ "hand002" ] "cur",
    "SELECT * FROM Room \tFOR UPDATE OF Roomnum, Type, Capacity"
    UPDATE_CURSOR;

/* CS_blocksize is set to 1 to control the fetch api calls */
set CS_blocksize = 1;

/* 4 TDS_CURFETCH NEXT packets of 1 row each are combined
 *  into a single sqlfetch_cursor command. */
sqlfetch_cursor [ "hand003" ] cursor_65537 FETCH_NEXT, 4;
sqldelete_cursor [ "hand004" ] cursor_65537, "Room",
    "Roomnum='2017  ' Type='OFF ' Capacity='2'";
sqlclose_cursor [ "hand012" ] cursor_65537;
```

See Also `sqlcursor_rowtag`

sqldisconnect

Emulation Function

Description Closes the specified connection.

Syntax `int` **`sqldisconnect`** `(`*`connection_id)`*

Syntax Element	Description
connection_id	An integer expression, returned by `sqlconnect`, which specifies the connection to close.

Comments The `sqldisconnect` function returns `1` upon success, and `0` upon failure. The `sqldisconnect` function sets `_error` and `_error_text`.

The `sqldisconnect` function is affected by the VU environment variable `Record_level`.

Example This example connects to a Sybase server, sets the server connection, and then disconnects:

```
SYBASE=sqlconnect("SERVER","ron","rondo","SYBASEC","sybase11",
          "TDS_VERSION='5.0.0.0' APP_NAME='Sample App'");
set Server_connection = SYBASE;
/* emulation functions */
sqldisconnect (SYBASE);
```

See Also `sqlconnect`

sqlexec

Send Emulation Command

Description Executes SQL statements.

Syntax `int` **`sqlexec`** [*`cmd_id`*] [`EXPECT_ERROR` *`ary`*,] [`EXPECT_ROWS` *`n`*,] *`stmt`*, *`arg_spec1`*, *`arg_spec2`*...

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *`cmd_id`* has the form [*`string_exp`*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *`ary`*, the response is unexpected.
n	An integer that gives the number of rows this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.
stmt	A string expression containing a SQL statement or an integer expression indicating a prepared statement ID.
arg_specN	One or more optional argument specifications used when executing *`stmt`*. Use these argument specifications for dynamic SQL placeholders (?'s) or stored procedure arguments.

Format for Specifying sqlexec Arguments

An argument specification has the form:

```
expr [ : &VUvar [ : &VUind ] ]
```

expr is required and is either a string or an integer expression.

If *expr* is a string expression, its value is interpreted at runtime as:

```
name='value' <type:(p,s)[c]: I | O | IO >
```

The syntax has these elements:

- *name*= indicates the name of the argument as it occurs in the SQL statement that is executed.

  name= is required for Oracle and is optional for Sybase and SQLServer. With Sybase and SQLServer, if the name is omitted, the argument is associated with the next SQL placeholder from the beginning of the SQL statement.

- *value* is the string representation of the argument value. If *name*= indicates a scalar argument, enclose the *value* portion of the string in single quotation marks for clarity. These quotation marks are not part of the argument value.

  If *name*= indicates an array argument, the value portion of the string has the form:

  ```
  { 'v1', 'v2', ... 'vN' }
  ```

  where `'v1'` through `'vN'` are string values for the array elements. You can specify a NULL array element as `SQL_NULL` as in:

  ```
  { 'v1', 'v2', SQL_NULL, 'v4' }
  ```

- *type* is the optional VU language database type of the argument. The default type is `varchar`.

- (*p*,*s*) are optional integer constants that represent the precision and scale. Generally, precision indicates the length (in bytes) of the internal format of the data. If present, this information is used in the conversion to the SQL database vendor-specific SQL database type as appropriate for that type.

  The *value* portion of a binary, varbinary, or longbinary argument is represented as pairs of hexadecimal characters.

  For Oracle, the presence of a scale value for a character data type (char or varchar) indicates a null conversion character. Any character equal to the scale is converted to a null (\0) character internally before transmission to the SQL database server.

- [*c*] specifies the number of elements in an array argument. [*c*] is not specified for scalar arguments.

  For output array arguments, the array size is required.

  For input array arguments, the array size is optional, for example, you can specify empty []. If not specified, the number of elements in the array value is transmitted. If specified, the number of elements transmitted is:

  MAX(*actual values*, *c*)

  Example of array arguments:

  ```
  sqlexec "proc(:a, :b, :c)",
      ":a=4<numeric(21):I>",
      ":b= {1, 2, 3, 4} <numeric(21)[]:I>",
      ":c= {’one’, ’two’, SQL_NULL, ’four’}
         <varchar(10)[]:I>”;
  ```

  In the example:

  – :a is an input scalar argument, type numeric, value 4 with precision length of 21.

  – :b is an input array of 4 numerics, values 1, 2, 3, and 4 with precision length of 21.

  – :c is an input array of 4 varchars (maximum length 10 characters each), the third of which is SQL_NULL.

- I, O, or IO indicates whether the argument is input (default), output, or input/output.

If an argument is output (O) or input/output (IO), the output parameter value is not valid until the next receive emulation command is executed.

White space characters within a string expression are optional, surrounding each portion of the string and between the name and = .

The following are some names, data types, and values obtained from Oracle arguments:

String	Name	Type	Value
":spid=50<int4>"	:spid	O_VARNUM	50
":logname=’george’"	:logname	O_VARCHAR2	"george"
":c1=’random=text’"	:c1	O_VARCHAR2	"random=text"
":c2=’01/17/96’<date>"	:c2	O_DATE	"01/17/96"

String	Name	Type	Value
":foo='hi\377pat'<char(6,0377):I>"	:foo	O_VARCHAR2	"hi\0pat"
":bin='00010203'<binary(4):I>"	:bin	O_BINARY	"\000\001\002\003"

The following are some names, data types, and values obtained from Sybase and SQL Server arguments:

String	Name	Type	Value
"@spid=50<int4>"	@spid	**CS_INT_TYPE**	50
"@logname='george'"	@logname	**CS_CHAR_TYPE**	"george"
"'random=text'"		**CS_CHAR_TYPE**	"random=text"
"01/17/96'<datetime4>"		**CS_DATETIME4_TYPE**	"01/17/96"

If *expr* is an integer, its value is the value of the integer. It has no name and it represents an input argument with the VU language database type is `int4`. Note that Oracle expressions require a name.

You get a syntax error if you use a type specification with an integer expression. To specify a type for an integer expression, use a string expression containing the value and type. For example:

```
sqlexec [ "exec001" ] stmt_id, "50 <int1>";
```

The following list shows the data type conversions performed by the VU playback libraries for each VU language data type. The SQL database server could perform further conversions.

VU	Sybase, SQL Server (ct-lib)	Oracle	ODBC
default	CS_CHAR_TYPE	O_VARCHAR2	SQL_C_CHAR
binary	CS_BINARY_TYPE	O_BINARY	SQL_C_BINARY
bit	CS_BIT_TYPE	O_VARCHAR2	SQL_C_CHAR
char	CS_CHAR_TYPE	O_VARCHAR2	SQL_C_CHAR
datetime4	CS_DATETIME4_TYPE	O_DATE	SQL_C_CHAR
datetime8	CS_DATETIME_TYPE	O_DATE	SQL_C_TIMESTAMP
decimal	CS_DECIMAL_TYPE	O_VARNUM	SQL_C_CHAR
float4	CS_REAL_TYPE	O_FLOAT	SQL_C_CHAR

(Continued)

VU	Sybase, SQL Server (ct-lib)	Oracle	ODBC
float8	CS_FLOAT_TYPE	O_FLOAT	SQL_C_CHAR
int1	CS_TINYINT_TYPE	O_VARNUM	SQL_C_SLONG
int2	CS_SMALLINT_TYPE	O_VARNUM	SQL_C_SLONG
int4	CS_INT_TYPE	O_VARNUM	SQL_C_SLONG
money4	CS_MONEY4_TYPE	O_VARCHAR2	SQL_C_CHAR
money8	CS_MONEY_TYPE	O_VARCHAR2	SQL_C_CHAR
numeric	CS_NUMERIC_TYPE	O_VARNUM	SQL_C_CHAR
varchar	CS_VARCHAR_TYPE	O_VARCHAR2	SQL_C_CHAR
text	CS_TEXT_TYPE	O_VARCHAR2	SQL_C_CHAR
image	CS_IMAGE_TYPE	O_VARCHAR2	SQL_C_CHAR
void	not supported	O_VARCHAR2	SQL_C_CHAR
varbinary	CS_VARBINARY_TYPE	O_BINARY	SQL_C_BINARY
longbinary	not supported	O_LONGBIN	SQL_C_BINARY
longchar	not supported	O_LONG	SQL_C_CHAR
sensitivity	not supported	O_VARCHAR2	SQL_C_CHAR
boundary	not supported	O_VARCHAR2	SQL_C_CHAR
date	not supported	O_DATE	SQL_C_DATE

You can specify any numeric argument as a string. Non-integer numeric arguments (such as floating point) must be specified as strings.

The `sqlexec` command accepts both named and positional arguments in the same command, and passes them on to the server. Any restrictions regarding mixing of named and positional arguments are enforced by the SQL server.

:&*`VUvar`* and :&*`VUind`* indicate VU language variable bindings. When *`VUvar`* and *`VUind`* are arrays, the & is not required. If present, a warning is generated.

The optional *`VUvar`* is a string, integer, array variable, or array element that indicates that the corresponding SQL argument is bound to this VU variable. If the SQL argument is a scalar, the VU variable must be a scalar. If the SQL argument is an array, the VU variable must be an array.

These bindings are interpreted as in the following table, depending on whether the SQL argument is input, output, or input/output:

SQL Argument	**How *VUvar* Is Bound**
input	If *expr* has no value component, the value of *VUvar* is used as the input value. If *VUvar* is not set, a runtime error occurs (unless *VUind* is present and has value -1). If *expr* has a value component, the value of *VUvar* is ignored.
output	*VUvar* receives the value of the SQL arguments after execution of the SQL statement. If *VUvar* is omitted, the SQL result is returned into an internal temporary space and discarded.
input/output	Same as input and output, above.

The optional *VUind* is an integer VU variable for scalar arguments and an array of integers for array arguments. *VUind* represents the `SQL NULL` indicator or array of `SQL NULL` indicators, as follows:

SQL Argument	**How *VUind* Is Bound**
input	If *expr* has no value component, the value of *VUind* has the following meaning: ▶ -1. The input value used is `SQL_NULL` ▶ > = 0. The input value is the value of *VUvar* If *VUind* is unset, it is a runtime error.
output	*VUind* receives the value assigned by the SQL server. Possible values for *VUind* are: ▶ -2. The return value (in *VUvar*) has been truncated and the actual length is greater than 65535. ▶ -1. The return value is SQL_NULL (*VUvar* is unchanged). ▶ 0. The return value is intact and stored in *VUvar*. ▶ > 0. The return value has been truncated and *VUind* contains the length before truncation.
input/output	Same as input and output, above.

To specify a SQL NULL input value, use any of the following formats:

- SQL_NULL
- "SQL_NULL"
- "*name*=SQL_NULL<*type*:I>"
- "*name*= <*type*:I> " : &*VUvar* : &*VUind* /* where *VUind* = = -1 */

How sqlexec Processes Statements

The sqlexec command executes any SQL statement. It does not return until the SQL statement has completed, or until Timeout_val elapses. sqlexec returns 1 indicating success, and returns 0 indicating an error. When sqlexec returns 0, _error and _error_text are set appropriately. If *stmt* is a prepared statement ID that is invalid for the current value of Server_connection, sqlexec fails. Zero is never a valid statement ID. The values of *arg_spec1* ... *arg_specN* are passed to the statement (*stmt*), prepared or not, as values for placeholders (?'s) or stored procedure arguments.

The sqlexec command can be used to execute statements using Oracle's array interface. If sqlsetoption() is used to set ORA_EXECCOUNT to a value greater than 1, then each input parameter to sqlexec must be an array containing the same number of elements as the value of ORA_EXECCOUNT. The sqlexec command then executes the statement using the array interface which performs the specified SQL statement multiple times with a single call to the SQL database server.

The sqlexec command delays execution of the SQL statement for the duration of a think time interval controlled by the think time variables. For more information, see *Think Time Variables* on page 5-42.

The read-only variable _fs_ts is set to the time the SQL statement is submitted to the server. The read-only variables _ls_ts, _fr_ts, and _lr_ts are set to the time the server has completed execution of the SQL statement.

The sqlexec command is affected by the following VU environment variables: Log_level, Record_level, Server_connection, Sqlexec_control_oracle, Sqlexec_control_sybase, Sqlexec_control_sqlserver, Statement_id, the think time variables, Timeout_act, Timeout_val, Timeout_scale, and Suspend_check.

Sqlexec_control_* controls precisely how sqlexec executes the SQL statement. See *Client/Server Environment Variables* on page 5-21.

Example

In this example, assume two SQL database servers: SYBORG (a Sybase 11.0 server) and ORCA (an Oracle 7.3 server). The following script accesses both servers and generates a log file (shown on page 5-37).

```
#include <VU.h>
{
   /* connection variables */
   int syborg, syberspace, orca;

   int deptno[] = { 50, 60, 70 };
   string deptname[] = { "testing", "shipping", "receiving" };
   string deptloc[] = { "Raleigh", "Durham", "Chapel Hill" };
   set Log_level = "ALL";

   /* connect to both servers */

   /* sybase connection, use all defaults */
   syborg = sqlconnect("", "hugh", "3ofFive", "sybserver",
       "sybase11");

   /* oracle connection, override defaults */
   orca = sqlconnect("", "willy", "wonka", "SEA.world", "oracle7.3");

   /* access syborg */
   set Server_connection = syborg;
   sqlexec [ "school" ] "use school";

   sqlexec"select Empnum, Empname, Roomnum from Employee where
       Rank='TUTOR'";

   set CS_blocksize = 3;
   while (_error == 0)
       sqlnrecv [ "Tutors" ] 10;

   /* switch to orca */
   set Server_connection = orca;

   sqlsetoption(ORA_AUTOCOMMIT, 1);

   sqlexec "select * from Dept";
   sqlnrecv [ "dept (a)" ] ALL_ROWS;
   /* insert some rows */
   sqlprepare [ "prep insert" ]
      "insert into Dept values (:no, :name, :place)";

   for (i = 0; i <= limitof deptno; i++)
      sqlexec _statement_id, ":no="+itoa(deptno[i]),
        ":name="+deptname[i], ":place="+deptloc[i];

   sqlexec "select * from Dept";
   sqlnrecv [ "dept (b)" ] ALL_ROWS;

   /* now delete rows */
   sqlexec "delete from Dept where deptno >= "+itoa(deptno[0]);

   sqlexec "select * from Dept";
   sqlnrecv [ "dept (c)" ] ALL_ROWS;

   /* done with orca */
   sqldisconnect(orca);

   /* done with syborg */
   sqldisconnect(syborg);
}
```

See Also None.

sqlfetch_cursor

Receive Emulation Command

Description Fetches the requested rows from the specified cursor.

Syntax

```
int sqlfetch_cursor [ cmd_id ]
[ EXPECT_ERROR ary, ] [ EXPECT_ROWS n, ]
csr_id [ row ]  [, count ]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
n	An integer that gives the number of rows this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.
csr_id	The cursor identifier returned by `sqldeclare_cursor` (or `sqlopen_cursor`) and opened by `sqlopen_cursor`.
row	Can be `FETCH_NEXT` (default), `FETCH_FIRST`, `FETCH_LAST`, `FETCH_PREV`, `FETCH_RELATIVE` *x*, or `FETCH_ABSOLUTE` *x*, where *x* is an integer that specifies the row to fetch.
count	Specifies the number of rows to fetch or `ALL_ROWS`. The default is 1.

Comments The first call to `sqlfetch_cursor` retrieves the column header information if `Column_headers` is `"ON."` The column headers are stored in the read-only variable `_column_headers` in two lines.

The rows returned by the SQL database server are stored in the read-only variable `_response`. A maximum of `Max_nrecv_saved` rows are stored. If more than `Max_nrecv_saved` rows are requested, the excess rows are fetched but not returned in `_response` and not logged.

If the cursor ID is not valid for the connection indicated by the value of `Server_connection` or if the cursor is not open, an error is reported to both the error file and the log file.

Rows are fetched in groups of `CS_blocksize` until the requested number of rows is returned or the end of the results is encountered. If `ALL_ROWS` are requested, then rows are fetched until the end of the result set (or table if `Table_boundaries` is `"ON"`) is reached. If fewer than *`count`* rows are retrieved, an error is logged.

The `sqlfetch_cursor` command is affected by the following VU environment variables: `CS_blocksize`, `Max_nrecv_saved`, `Column_headers`, `Table_boundaries`, `Server_connection`, and `Sqlnrecv_long`.

Example

This example prepares a statement, declares and opens a cursor on the prepared statement, and fetches five rows from the cursor result set. The last row fetched is updated using a parameterized update statement, and the next four rows from the cursor set are fetched for a total of nine rows fetched:

```
#include <VU.h>
{

SYBASE = sqlconnect("SYBASE", "prevue", "prevue", "PROXYC",
"sybase11sybase11", "TDS_VERSION='5.0.0.0'");

set Server_connection = SYBASE;

sqlexec  ["csrdyne001"] "USE pubs2";
stmt = sqlprepare ["csrdyne002"] "SELECT au_id, au_lname, au_fname,"
   "phone, address, city, state, postalcode FROM authors";

authors_id = sqldeclare_cursor["csrdyne003"] "authors", stmt;

sqlopen_cursor ["csr004"] authors_id;

sqlfetch_cursor ["csr005"] EXPECT_ROWS 5, authors_id FETCH_NEXT, 5;
sqlupdate_cursor ["csr006"] EXPECT_ROWS 1, authors_id, "authors",
    "UPDATE "
    "authors SET au_lname = @sql0_m_au_lname , au_fname = "
    "@sql1_m_au_fname , phone = @sql2_m_phone , "
    "address = @sql3_m_address , city = @sql4_m_city ,"
    " state = @sql5_m_state , postalcode = "
    "@sql6_m_zip ", "",
    "'Smith                          '",
    "'Meander                        '",
    "913 843-0462",
    "'10 Mississippi Dr.             '",
    "'Lawrence                       '"
    "KS", "'66044                    '";


sqlfetch_cursor ["csr007"] EXPECT_ROWS 9, authors_id FETCH_NEXT, 4;

sqlclose_cursor ["csr008"]  authors_id ;

sqldisconnect(SYBASE);
}
```

See Also

`sqlconnect`

sqlfree_cursor

Emulation Function

Description Frees a cursor.

Syntax `int` **`sqlfree_cursor`**`(`*`csr_id`*`)`

Syntax Element	Description
csr_id	The identifier of the cursor to free. If *csr_id* is not declared by either `sqldeclare_cursor` or `sqlopen_cursor`, or allocated by `sqlalloc_cursor`, a nonfatal error is reported in the error file.

Comments After a cursor ID is freed, any cursor emulation command or function that attempts to use that cursor ID produces a nonfatal error, which is reported in the error file.

If you are emulating a Sybase, ODBC, or Microsoft SQL Server application that uses embedded SQL cursors, your script includes the `sqlfree_cursor` emulation function. This function closes (if necessary), then deallocates the cursor ID declared with the emulation commands `sqldeclare_cursor` or `sqlopen_cursor`.

Example In this example, a cursor is opened, some cursor rows are fetched, and the cursor is freed.

```
#include <VU.h>
{
SYBASE = sqlconnect("SYBASE", "myuid", "mypasswrd","SYBASE_SERVER",
     "sybase11", "TDS_VERSION='5.0.0.0', APP_NAME='csr_disp'");

set Server_connection = SYBASE;

sqlexec  ["csr_upd001"]  "use pubs2";

push CS_blocksize = 5;

cursor_a_id = sqldeclare_cursor ["csr_upd002"]  "cursor_a",
   "select * from titles" UPDATE_CURSOR{"total_sales","type"};
sqlopen_cursor cursor_a_id;

sqlfetch_cursor ["csr_upd003"] cursor_a_id FETCH_NEXT, 1;

sqlfree_cursor( cursor_a_id );

sqldisconnect(SYBASE);

pop CS_blocksize;
}
```

See Also `sqldeclare_cursor`   `sqlopen_cursor`   `sqlopen_cursor`

sqlfree_statement

Emulation Function

Description Frees all of the client and server resources for a prepared statement.

Syntax `int` **`sqlfree_statement`**`(`*`stmt_id`*`)`

Syntax Element	Description
stmt_id	An integer value returned by the `sqlprepare` emulation command. If *`stmt_id`* is not the result of the `sqlprepare` emulation command or *`stmt_id`* has already been freed by `sqlfree_statement`, an error message is printed and `_error` and `_error_text` are set.

Comments The `sqlfree_statement` function is affected by the VU environment variable `Server_connection`.

Example In this example, a SQL `SELECT` statement is prepared, for which the statement ID `stmt` is returned. A cursor is declared for `stmt`, and the cursor is opened on the prepared statement with an argument of `2`. The server processes the prepared statement and returns a cursor result set. The cursor rows are fetched, and the prepared statement is freed.

```
#include <VU.h>
{
SYBASE = sqlconnect("SYBASE", "myuserid", "mypassword",
     "SYBASE_SERVER", "sybase11", "TDS_VERSION='5.0.0.0'");
set Server_connection = SYBASE;

sqlexec  ["csrsimp001"] "USE pubs2";

stmt = sqlprepare ["csrsimp002"] "SELECT * FROM mytable where id = ?";

simple_id = sqldeclare_cursor["csrsimp003"] "simple", stmt;

sqlopen_cursor ["csrsimp004"] simple_id, 2;

sqlfetch_cursor ["csrsimp005"] simple_id FETCH_NEXT, 1;

sqlfree_statement(stmt);

sqlclose_cursor ["csrsimp008"]  simple_id ;

sqldisconnect(SYBASE);
}
```

See Also None.

sqlinsert_cursor

Send Emulation Command

Description Inserts rows via a cursor.

Syntax

```
int sqlinsert_cursor [ cmd_id ] [ EXPECT_ERROR ary, ]
[EXPECT_ROWS n, ] [ CURSOR_LOCK | CURSOR_UNLOCK , ] csr_id,
tbl_name, rowtag [ , values]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in `ary`, the response is unexpected.
n	An integer that gives the number of rows this command should affect. The default is –1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, then response is unexpected.
csr_id	The integer cursor identifier of an opened cursor.
tbl_name	A string expression containing the name of the table affected by the insert.
rowtag	A string expression identifying the row to position the cursor. The format of the string is SQL database vendor-specific. A valid rowtag can be obtained by calling `sqlcursor_rowtag`().
values	A list of string values, integer values, or both to insert into the table via the cursor. Values may include type specifiers. Each value is the string representation of the argument value as described for the `sqlexec` emulation command.

Comments If the cursor ID is not valid for the connection indicated by the value of `Server_connection`, an error is reported to both the error file and the log file.

If `CURSOR_LOCK` is specified, the `sqlinsert_cursor` command locks the inserted rows. If `CURSOR_UNLOCK` is specified, `sqlinsert_cursor` unlocks the inserted rows.

The `sqlinsert_cursor` command is affected by the VU environment variable `Server_connection`.

Example This example inserts the row `Dodsworth, Anne` into the employees table.

```
stmt_2_1_id=sqlalloc_cursor();

set Cursor_id = stmt_2_1_id;
sqlopen_cursor "C1", "select lastname, firstname from employees";

sqlfetch_cursor stmt_2_1_id, 8;

sqlinsert_cursor stmt_2_1_id, "", "1", "'Dodsworth'<varchar(21):I>",
"'Anne'<varchar(16):I>";

sqlfree_cursor( stmt_2_1_id );
```

See Also `sqlexec` `sqlcursor_rowtag`

sqllongrecv

Receive Emulation Command

Description Retrieves longbinary and longchar results.

Syntax `int` **`sqllongrecv`** `[` *`cmd_id`* `] [ EXPECT_ERROR` *`ary`*`, ]` *`column`*`,` *`offset`*`,` *`size`*`,` *`count`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
column	An integer expression indicating the column that contains the long data type. The first column in the row is 1.
offset	An integer expression indicating the beginning offset within the column.
size	An integer expression indicating the number of bytes to retrieve from the column at one time.
count	An integer expression indicating the number of blocks of *size* bytes to retrieve.

Comments The `sqllongrecv` command retrieves *count* * *size* bytes from a column of type longbinary or longchar. If fewer than *count* * *size* bytes are retrieved, `_error` and `_error_text` are set to indicate the reason.

The sqllongrecv command operates on the last row retrieved by sqlnrecv or sqlfetch_cursor, and thus can be called after sqlnrecv or sqlfetch_cursor was called.

The sqllongrecv command is affected by the following VU environment variables: Timeout_val, Timeout_scale, Timeout_act, Log_level, Record_level, Max_nrecv_saved, and Server_connection.

The sqllongrecv command is also affected by Statement_id if Statement_id is not zero. Otherwise sqllongrecv operates on the last sqlexec command.

Example In this example, sqlnrecv fetches the first 100 bytes of column 3. The next sqllongrecv fetches 3 blocks, each 65536 bytes in size, of column 3. The last sqllongrecv fetches the last 3392 bytes of column 3, starting at offset 199608.

```
sqlprepare "select msg_id, msg_len, msg from voicemail"
    "where msg_id=100";
push CS_blocksize = 1;
set sqlnrecv_long=100;
sqlnrecv 1;
sqllongrecv 3, 65536, 3;
sqllongrecv 3, 196608, 3392, 1;
```

See Also None.

sqlnrecv

Receive Emulation Command

Description Retrieves row results after sqlexec is executed.

Syntax

```
int sqlnrecv [ cmd_id ]
[ EXPECT_ERROR ary, ] [ EXPECT_ROWS n, ] m
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets _error to a value not in *ary*, the response is unexpected.
n	An integer that gives the number of rows that this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.

(Continued)

Syntax Element	Description
m	An integer that gives the number of rows requested or ALL_ROWS, which receives all remaining rows. The default is 1.

Comments

The sqlnrecv command retrieves *m* rows from the last command processed by sqlexec. sqlnrecv repeatedly requests CS_blocksize rows from the SQL database server until *m* rows have been retrieved, an error occurs, or it reaches the end of the table and Table_boundaries is ON.

If fewer than *m* rows are retrieved, _error is set to indicate the reason. If *m* is not ALL_ROWS, and if the end of the row results (or the end of the table) is reached, _error and _error_text are set to indicate the condition that terminated the command. If there are no more row results, sqlnrecv returns immediately, setting _error and _error_text appropriately.

The sqlnrecv command processes the first Sqlnrecv_long bytes of columns of type longbinary or longchar. Any remaining data in these columns must be processed by sqllongrecv.

The sqlnrecv command is affected by the following VU environment variables: CS_blocksize, Column_headers, Timeout_val, Timeout_scale, Log_level, Record_level, Max_nrecv_saved, Server_connection, Timeout_act, Table_boundaries, Sqlnrecv_long. It is also affected by Statement_id if Statement_id is not zero. Otherwise sqlnrecv operates on the last sqlexec command.

Example This example issues a `select` query. The `sqlnrecv` fetches and processes all rows returned by the query. The same `select` query is issued, and the first twenty-five rows are fetched and process. The next `sqlnrecv` fetches and processes the remaining rows held in the `fetch` buffer.

```
#include <VU.h>
{
SERVER = sqlconnect("SERVER", "myuserid", "mypassword",
    "NTSQL_SERVER","sqlserver", "TDS_VERSION='4.2.0.0',"
     "APP_NAME='isql'");
set Server_connection = SERVER;

sqlexec  ["sql_1001"] "use school";

sqlexec  ["sql_1002"] "select * from Assignment";

/* Get all rows returned */
sqlnrecv ["sql_1003"] EXPECT_ROWS 50, ALL_ROWS;
sqlexec  ["sql_1004"] "select * from Assignment";

/* Get first twenty-five rows returned */
sqlnrecv ["sql_1005"] EXPECT_ROWS 25, 25;

/* Get rest of rows returned */
sqlnrecv ["sql_1005"] EXPECT_ROWS 25, ALL_ROWS;

sqldisconnect(SERVER);
}
```

See Also `sqllongrecv`

sqlopen_cursor

Send Emulation Command

Description Opens the specified cursor.

```
int sqlopen_cursor [ cmd_id ]
[ EXPECT_ERROR ary, ] [ EXPECT_ROWS n, ]
 csr_spec [, values ]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
n	An integer that gives the number of rows that this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.

(Continued)

Syntax Element	Description
csr_spec	Choose one of the following: ▶ A cursor ID returned by `sqldeclare_cursor` ▶ *csr_name*, *sqlstmt* [,{ READ_ONLY_CURSOR | UPDATE_CURSOR [*col_ary*] }] *csr_name* is a string expression giving the name of the cursor. *sqlstmt* is either a previously prepared statement ID or a SQL statement string expression associated with the cursor. `sqlopen_cursor` implicitly declares a cursor for that statement and then opens that cursor. READ_ONLY_CURSOR indicates that the cursor is read-only. UPDATE_CURSOR indicates that the cursor is updatable. If neither type of cursor is specified, the text of *sqlstmt* determines whether the cursor is updatable. *col_ary* is an array of strings whose values are the updatable column names. The default is all columns are updatable.

values	A list of string values, integer values, or both to use for opening the cursor. *values* could include type specifiers.
	Each *value* is the string representation of the argument value. If *name*= indicates a scalar argument, enclose the value portion of the string in single quotation marks for clarity. These quotation marks are not part of the argument value. If *name*= indicates an array argument, the value portion of the string has the form:
	`{ 'v1', 'v2', ... 'vN''}`
	where *'v1'* through *'vN'* are string values for the array elements. You can specify a NULL array element as `SQL_NULL`, as in:
	`{ 'v1', 'v2', SQL_NULL, 'v4' }`

Comments The `sqlopen_cursor` command returns an integer cursor ID for future reference by other `sql*_cursor` command and functions. The returned cursor ID is placed in the read-only variable `_cursor_id`.

If *`csr_spec`* is a cursor ID and is not a valid declared cursor (with `sqldeclare_cursor`) for the connection indicated by the value of `Server_connection`, then an error is reported to both the error file and the log file.

The `sqlopen_cursor` command is affected by the VU environment variables `Cursor_id`, `Sqlexec_control_*`, and `Server_connection`.

Example This example opens a cursor, fetches the results, and closes the cursor. Note that the cursor was not freed and deallocated. The cursor is reopened at a later point in the script without redeclaring it.

```
#include <VU.h>
{
SYBASE = sqlconnect("SYBASE", "myuserid", "mypassword",
     "SYBASE_SERVER","sybase11", "TDS_VERSION='5.0.0.0',
APP_NAME='csr_disp'");
set Server_connection = SYBASE;
sqlexec  ["csr_upd001"]  "use pubs2";
push CS_blocksize = 5;
cursor_a_id = sqldeclare_cursor ["csr_upd002"]  "cursor_a",
   "select * from titles" UPDATE_CURSOR {"total_sales","type"};
sqlopen_cursor cursor_a_id;
sqlfetch_cursor ["csr_upd003"] cursor_a_id FETCH_NEXT, 1;
sqlclose_cursor( cursor_a_id );
sqlexec ["csr_upd004"] "select * from authors";
sqlopen_cursor cursor_a_id;
sqlfetch_cursor ["csr_upd003"] cursor_a_id FETCH_NEXT, 1;
sqlclose_cursor( cursor_a_id );
sqlfree_cursor( cursor_a_id );
sqldisconnect(SYBASE);
pop CS_blocksize;
}
```

See Also `sqlclose_cursor` `sqlexec` `sqldeclare_cursor` `sqlfree_cursor`

sqlposition_cursor

Send Emulation Command

Description Positions a cursor within a result set.

Syntax

`int` **`sqlposition_cursor`** `[` *`cmd_id`* `] [ EXPECT_ERROR` *`ary`*`, ]`
`[ CURSOR_LOCK | CURSOR_UNLOCK , ]` *`csr_id`*`,` *`rowtag`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
csr_id	The integer cursor identifier of an opened cursor.
rowtag	A string expression identifying the row to position the cursor. The format of the string is SQL database vendor-specific. A valid rowtag can be obtained by calling `sqlcursor_rowtag`().

Comments

If the cursor ID is not valid for the connection indicated by the value of `Server_connection`, an error is reported to both the error file and the log file.

If `CURSOR_LOCK` is specified, the `sqlposition_cursor` command locks the inserted rows. If `CURSOR_UNLOCK` is specified, `sqlposition_cursor` unlocks the inserted rows.

The `sqlposition_cursor` command is affected by the VU environment variable `Server_connection`.

Example

This example sets the current row position to row 1 in the result set.

```
sqlopen_cursor "C1", "select lastname, firstname from employees";
sqlfetch_cursor stmt_2_1_id, 8;
sqlposition_cursor stmt_2_1_id, "1";
```

See Also

`sqlcursor_rowtag`

sqlprepare

Send Emulation Command

Description

Prepares a SQL statement for execution.

Syntax `int sqlprepare [ cmd_id ] [ EXPECT_ERROR ary, ] stmt`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
stmt	A string expression containing a SQL statement.

Comments The `sqlprepare` command prepares SQL statements. It does not return until the server has parsed the SQL statement, or until `Timeout_val` elapses. Upon success, `sqlprepare` returns the value assigned as the prepared statement ID, and sets `_statement_id` to the value. Upon failure, `sqlprepare` sets `_statement_id` to a negative value, returns the value of `_statement_id`, and sets `_error` and `_error_text`. The `sqlprepare` command associates the statement ID with the connection indicated by `Server_connection`. Because `sqlprepare` sets and returns the value of `_statement_id`, the statement ID is saved in an integer variable, either by:

```
stmt_id = sqlprepare ...
```

or

```
sqlprepare ...
stmt_id = _statement_id;
```

The `sqlprepare` command delays submitting the SQL statement to the server for the duration of a think time interval controlled by the think time environment variables.

The read-only variable `_fs_ts` is set to the time the SQL statement is submitted to the server. The read-only variables `_ls_ts`, `_fr_ts`, and `_lr_ts` are set to the time the server has completed parsing the SQL statement.

The `sqlprepare` command is affected by the following VU environment variables: the think time variables, `Timeout_val`, `Timeout_scale`, `Log_level`, `Record_level`, `Server_connection`, `Statement_id`, and `Suspend_check`.

Example

This example shows a script that prepares a `select` statement and assigns the statement ID to `stmtid_1`. The prepared statement `stmtid_1` is executed with a runtime parameter of `:id='12345'`. Any rows returned are fetched and processed. Statement `stmtid_1` is freed and deallocated. The same variable `stmtid_1` is reused for another `sqlprepare` on a different `select` statement. The prepared statement is executed and any rows returned are fetched and processed. The statement ID stopped in `stmtid_1` is freed and deallocated.

```
#include <VU.h>
{
t_calvin_PAC = sqlconnect("t_calvin_PAC", "oracle", "oracle",
    "t:calvin:PAC", "oracle7.3");

push Sqlexec_control_oracle = "STATIC_BIND";
set Server_connection = t_calvin_PAC;
stmtid_1 = sqlprepare ["oraclee016"] "select * from Student where id"
    "= :id";
sqlexec  ["oraclee017"] stmtid_1,":id='12345'";
sqlnrecv ["oraclee018"] EXPECT_ROWS 1, ALL_ROWS;
sqlfree_statement(stmtid_1);
stmtid_1 = sqlprepare ["oraclee019"] "select * from Course";
sqlexec  ["oraclee020"] stmtid_1;
sqlnrecv ["oraclee021"] EXPECT_ROWS 14, ALL_ROWS;
sqlfree_statement(stmtid_1);
sqldisconnect(t_calvin_PAC);
pop CS_blocksize;
}
```

See Also

`sqlexec`

sqlrefresh_cursor

Send Emulation Command

Description

Refreshes the result set of a cursor.

Syntax

int **sqlrefresh_cursor** [*cmd_id*] [EXPECT_ERROR *ary*,] [EXPECT_ROWS *n* ,][CURSOR_LOCK | CURSOR_UNLOCK ,] *csr_id*, *rowtag*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.

Syntax Element	Description
n	An integer that gives the number of rows this command should affect. The default is –1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, then response is unexpected.
csr_id	The integer cursor identifier of an opened cursor.
rowtag	A string expression identifying the row to position the cursor. The format of the string is SQL database vendor-specific. A valid rowtag can be obtained by calling `sqlcursor_rowtag`().

Comments If the cursor ID is not valid for the connection indicated by the value of `Server_connection`, an error is reported to both the error file and the log file.

If `CURSOR_LOCK` is specified, the `sqlrefresh_cursor` command locks the inserted rows. If `CURSOR_UNLOCK` is specified, `sqlrefresh_cursor` unlocks the inserted rows.

The `sqlrefresh_cursor` command is affected by the VU environment variable `Server_connection`.

Example This example refreshes row 2 in the rowset. This is done, because the update on row 2 invalidated the row currently stored in the rowset.

```
stmt_2_1_id=sqlalloc_cursor();

set Cursor_id = stmt_2_1_id;
sqlopen_cursor "C1", "select lastname, firstname from employees";

sqlfetch_cursor stmt_2_1_id, 8;

sqlupdate_cursor stmt_2_1_id, "", "", "2", "'Buchanan'<var-
char(21):I>",
"'Anne'<varchar(16):I>";

sqlrefresh_cursor stmt_2_1_id, "2";

sqlfree_cursor( stmt_2_1_id );
```

See Also `sqlcursor_rowtag`

sqlrollback

Emulation Function

Description Rolls back the current transaction.

Syntax `int sqlrollback()`

Comments The `sqlrollback` function is not supported for Sybase and Microsoft SQL server, and produces a fatal runtime error. For Sybase and Microsoft SQL server databases, use the following:

```
sqlexec "rollback transaction";
```

The `sqlrollback` function is affected by the VU environment variable `Server_connection`.

Example In this example, an update statement is sent to the server. The `sqlrollback` function restores the affected rows of the updated table to their original value.

```
#include <VU.h>
{
t_calvin_PAC = sqlconnect("t_calvin_PAC", "oracle", "oracle",
    "t:calvin:PAC", "oracle7.3");

set Server_connection = t_calvin_PAC;

sqlexec ["oracle003"] "INSERT INTO voice_mail (msg_id, msg_len, msg)"
"VALUES (100, 5, Hello";

sqlrollback();

sqldisconnect(t_calvin_PAC);

pop CS_blocksize;
}
```

See Also `sqlcommit`

sqlsetoption

Emulation Function

Description Sets a SQL database server option.

Syntax `int sqlsetoption(optioncode [, optarg ...])`

Syntax Element	Description
optioncode	The integer that indicates the server option you want to set. The values for *optioncode* are vendor-specific. The recognized values for *optioncode* and any symbolic constants for *optarg* are defined in the file VU.h. Comments accompany each *optioncode*, giving the number and types of *optarg*'s expected. All definitions for Sybase options are prefixed by SYB_; all definitions for Oracle options are prefixed by ORA_.
optarg	The value that you want to supply to the server option. All options require at least one *optarg*. The number and type of *optarg*'s depends on the value of *optioncode*. The number and type of *optarg*'s are checked at runtime; mismatches result in a fatal runtime error.

Comments The sqlsetoption function returns 1 for success and 0 for failure. sqlsetoption sets _error and _error_text, and prints an appropriate message to standard error when _error is nonzero.

The sqlsetoption function sets the server option indicated by the integer *optioncode* to the value given by *optarg* for the server indicated by the current value of Server_connection.

The sqlsetoption function is affected by the VU environment variable Server_connection.

Example This example sets options for a Sybase server:

```
SYBASE = sqlconnect("", "sybase", "sybase", "", "sybase11");
set Server_connection = SYBASE;
/* assorted options */
sqlsetoption(SYB_OPT_ANSINULL, 1);
sqlsetoption(SYB_OPT_STR_RTRUNC, 1);
sqlsetoption(SYB_OPT_ARITHABORT, 0);
sqlsetoption(SYB_OPT_TRUNCIGNORE, 1);
sqlsetoption(SYB_OPT_ARITHIGNORE, 0);
sqlsetoption(SYB_OPT_ISOLATION, SYB_OPT_LEVEL3);
sqlsetoption(SYB_OPT_CHAINXACTS, 1);
sqlsetoption(SYB_OPT_CURCLOSEONXACT, 1);
sqlsetoption(SYB_OPT_QUOTED_IDENT, 1);
```

See Also None.

sqlsysteminfo

Send Emulation Command

Description Queries the server for various types of system information.

Syntax

```
sqlsysteminfo [ cmd_id ] [ EXPECT_ERROR ary , ]
              [ EXPECT_ROWS n , ] operation , arglist ...
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. `cmd_id` has the form [*string_exp*].
operation	A string expression specifying what type of information to retrieve.
arglist	A comma-seperated list of string or integer expressions. The interpretation of each argument depends on the value of *operation*.

Comments The `sqlsysteminfo` command performs any of several specific system information requests depending on the value of *operation*.

List of Operations

The valid values for *operation* and their purpose are shown in the following table:

Operation	Purpose
`Tables`	Retrieves a list of table names stored in a specific data source's system catalog.
`TablePrivileges`	Retrieves a list of table names stored and privileges associated with them.
`Columns`	Retrieves a list of column names associated with a specified table.
`ColumnPrivileges`	Retrieves a list of column names and privileges for a specified table.
`SpecialColumns`	Retrieves a unique row ID for a specified table.
`Statistics`	Retrieves statistical information about a specified table and its associated indexes.
`PrimaryKeys`	Retrieves the list of column names that make up the primary key for a specified table.

(Continued)

Operation	Purpose
ForeignKeys	Retrieves information about the foreign keys defined for a specified table and what primary keys in other tables they access.
Procedures	Retrieves a list of stored procedure names that have been registered in a specified data source.
ProcedureColumns	Retrieves a list of I/O parameters to a stored procedure.

List of Operation Arguments

The valid values for *arglist* for each operation are shown in the following table. All arguments are strings unless marked with a (*).

Operation	arglist
Tables	catalogName, schemaName, tableName, tableType
TablePrivileges	catalogName, schemaName, tableName
Columns	catalogName, schemaName, tableName, columnName
ColumnPrivileges	catalogName, schemaName, tableName, columnName
SpecialColumns	rowid(*), catalogName, schemaName, tableName, columnName, scope(*), nullable(*)
Statistics	catalogName, schemaName, tableName, indexType(*), accuracy(*)
PrimaryKeys	catalogName, schemaName, tableName
ForeignKeys	PKcatalogName, PKschemaName, PKtableName, FKcatalogName, FKschemaName, FKtableName (PK = primary key, FK = foreign key)
Procedures	catalogName, schemaName, procedureName
ProcedureColumns	catalogName, schemaName, procedureName columnName

If `Cursor_id` is non-zero, `sqlsysteminfo` will perform the operation using the cursor specified by `Cursor_id`. Otherwise, `sqlsysteminfo` will allocate a new cursor (and set `_cursor_id`) for the operation. `sqlsysteminfo` returns the cursor ID used for the operation.

The `sqlsysteminfo` command is affected by the VU environment variables `Cursor_id`, `Server_connection`, the think time variables, `Timeout_val`, `Timeout_scale`, `Timeout_act`, `Log_level`, `Record_level`, and `Suspend_check`.

Example

```
x = sqlalloc_cursor();
set Cursor_id = x;

sqlsysteminfo [ "info001" ] "Tables", "catalog_1",
    "schema_1", "Cities", "user";

    sqlfetch_cursor x, ALL_ROWS;
```

sqlupdate_cursor

Send Emulation Command

Description Updates the current row of the indicated cursor.

Syntax

```
int sqlupdate_cursor [ cmd_id ] [ EXPECT_ERROR ary, ]
[ EXPECT_ROWS n, ] [ CURSOR_LOCK | CURSOR_UNLOCK ]
csr_id, tbl_name, set_clause, rowtag [, values ]
```

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
ary	An array of integers that contains all acceptable error numbers for this SQL command. The default value is {0}, which indicates that no error is acceptable. If a SQL command sets `_error` to a value not in *ary*, the response is unexpected.
n	An integer that gives the number of rows this command affects. The default is -1, which indicates any number of rows. If *n* is > = 0, and the number of rows the SQL command processes does not equal *n*, the response is unexpected.
csr_id	The integer cursor identifier of an opened cursor.
tbl_name	A string expression containing the name of the table to update.
set_clause	A string expression containing the SET clause of that SQL update statement.
rowtag	A string expression identifying the row to update and which is obtained by calling `sqlcursor_rowtag`(). The format of the string is vendor-specific. If *rowtag* is `""`, no row identification is used and the current row is updated.

(Continued)

Syntax Element	Description

values	A list of string values, integer values, or both to use for updating the current row of the cursor. *values* may include type specifiers.
	Each *value* is the string representation of the argument value. If *name*= indicates a scalar argument, enclose the value portion of the string in single quotation marks for clarity. These quotation marks are not part of the argument value. If *name*= indicates an array argument, the value portion of the string has the form:
	`{ 'v1', 'v2', ... 'vN'}`
	where *'v1'* through *'vN'* are string values for the array elements. You can specify a NULL array as `SQL_NULL` as in:
	`{ 'v1', 'v2', SQL_NULL, 'v4' }`

Comments If the cursor ID is not valid for the connection indicated by the value of `Server_connection` or if the cursor is not open, an error is reported to both the error file and the log file.

If `CURSOR_LOCK` is specified, the `sqlupdate_cursor` command locks the updated rows. If `CURSOR_UNLOCK` is specified `sqlupdate_cursor` unlocks the updated rows.

The `sqlupdate_cursor` command is affected by the VU environment variable `Server_connection`.

Example This example positions the cursor at the next row and updates that row:

```
sqlfetch_cursor [ "hand009" ] cursor_65537 FETCH_NEXT;
sqlupdate_cursor [ "hand010" ] cursor_65537, "Room",
    "UPDATE Room Set Roomnum = @sql0_num , Type = @slq1_type ,"
    "Capacity = @sql2_cap ","Roomnum='2220  ' Type='OFF '"
    "Capacity='3'","1111", "off", 3;
```

See Also `sqlcursor_rowtag`

sqtrans

Library Routine

Description Creates string expressions based on character translations of string expressions, squeezing out any repeated characters.

Syntax string **sqtrans** (*str*, *in_str*, *out_str*)

Syntax Element	Description
str	The subject string expression.
in_str	A string expression that specifies the set of characters within *str* that is translated or deleted.
out_str	A string expression that specifies the corresponding set of characters to which the characters in *in_str* are translated.

Comments The sqtrans routine returns a translated version of *str* by substituting or deleting selected characters and then squeezing all strings of repeated characters in the returned string that occur in *out_str* to single characters. Any character in *str* not found in *in_str* is copied unmodified to the returned string. Characters found in *in_str* are substituted by the corresponding character in *out_str* (based on character position). If there is not a corresponding character in *out_str*, the character is deleted (not copied to the returned string).

A special convention is useful for padding *out_str*. If *out_str* has at least two characters and ends in an asterisk (*), *out_str* is automatically padded with the character preceding the * until the length of *out_str* is the same as the length of *in_str*. For example, if *out_str* is *"abc*"* and the length of *in_str* is 10, *out_str* is converted to *abcccccccc* before the translation begins. If this action is undesirable, the ordering of the characters in *in_str* and *out_str* must be changed such that *out_str* does not end in *.

The trans routine also translates string expressions, except that it does not perform the “squeeze” translation.

Example This example removes each tab in the input string and replaces it with a space, and then squeezes the repeated spaces so that the result has only one space around each word:

```
sqtrans("\t\tHello,\t\tworld\t\t" "\t", " ");
```

See Also trans

srand

Library Routine

Description

Reseeds the random number generator, essentially resetting it to a specific starting place.

Syntax

`int` **`srand`** `(`*`seed`*`)`

Syntax Element	Description
seed	The integer expression used to seed the random number generator. Its value must be non-negative.

Comments

The `srand` routine is similar to its corresponding C library routine but generates random numbers with better "randomness."

The `rand`, `srand`, `uniform`, and `negexp` routines enable the VU language to generate random numbers. The behavior of these random number routines is affected by the way you set the **Seed** and **Seed Flags** options in a LoadTest schedule. By default, the **Seed** generates the same sequence of random numbers but sets unique seeds for each virtual user, so that each virtual user has a different random number sequence. For more information about setting the seed and seed flags in a schedule, see the *Using Rational LoadTest* manual.

The `srand` routine uses the argument *`seed`* as a seed for a new sequence of random numbers to be returned by subsequent calls to the `rand` routine. If `srand` is then called with the same seed value, the sequence of random numbers is repeated. If `rand` is called before any calls are made to `srand`, the same sequence is generated as when `srand` is first called with a seed value of 1.

Example

This example seeds the random number generator with the current time and then prints the first 10 random numbers:

```
srand(time());
for (i = 0; i < 10; i++)
printf("random number (%d): %d\n", i, rand());
```

See Also

`rand` `uniform` `negexp`

start_time

Emulation Command

Description Marks the start of a block of actions to be timed.

Syntax

```
int start_time;
int start_time time;
int start_time [time_id];
int start_time [time_id] time;
```

Syntax Element	Description
time	An integer expression specifying a timestamp that overrides the current time.
time_id	An optional ID, similar to a command ID, that has the form [*string_exp*]. If *time_id* is not specified, the starting timestamp is saved internally.

Comments The start_time command associates a starting timestamp with *time_id* for later reference by stop_time, and returns an integer expression equal to the starting timestamp.

VU automatically timestamps the time that any send emulation command is sent to the SQL database server as _fs_ts, and the time that the command returns as _ls_ts. VU also timestamps the time of the first and last results received by any receive emulation command, allowing six possible "response time" definition choices with LoadTest reports. If these are not sufficient, use start_time and stop_time when generating report output.

The start_time and stop_time commands can span multiple emulation commands in the same script, such as the elapsed time for a logical transaction that consists of several commands.

Example This example shows how IDs are used with start_time to measure nested transactions. The ID T2.x on the second start_time is not necessary, but it is recommended for clarity:

```
start_time ["T2"];/* beginning of entire T2 */
...
start_time ["T2.x"];/* beginning of subset of T2 */
...
stop_time ["T2.x"];/* ending of subset of T2 */
...
stop_time ["T2"];/* ending of entire T2 */
```

This example shows how IDs can be used with `start_time` to measure overlapping transactions:

```
start_time ["T3"];/* beginning of T3 */
...
start_time ["T4"];/* beginning of T4 */
...
stop_time ["T3"];/* ending of transaction T3 */
...
stop_time ["T4"];/* ending of transaction T4 */
```

This example shows how transactions can easily share the same starting time. The example would not work correctly if a previous `start_time` in the script had been given an ID `T1`, `T2`, or `T3`, because `stop_time` selects *`prev_time`* as the starting time only if a matching ID is not found:

```
start_time;/* beginning of T1, T2 & T3*/
...
stop_time ["T1"];/* ending of transaction T1 */
...
stop_time ["T2"];/* ending of transaction T2 */
...
stop_time ["T3"];/* ending of transaction T3 */
```

This alternative example removes the potential problem by providing separately labeled start times for `T1`, `T2`, and `T3`, all using a common starting timestamp.

```
beg = start_time ["T1"];/* beginning of T1, T2 & T3*/
start_time ["T2"] beg;/* associate time with ID T2 */
start_time ["T3"] beg;/* associate this with ID T3 */
...
stop_time ["T1"];/* ending of transaction T1 */
...
stop_time ["T2"];/* ending of transaction T2 */
...
stop_time ["T3"];/* ending of transaction T3 */
```

Because the starting timestamps for `T2` and `T3` were user-defined, their associated `start_time` commands could have been executed at any time before their respective `stop_time` command. However, because the Trace report output displays all emulation commands in order of execution, you execute the `start_time` as close to the actual starting time as possible, as shown in the previous example.

With the creative use of `start_time` and `stop_time`, emulation commands, and the read-only timestamp variables `_fs_ts`, `_ls_ts`, `_fr_ts`, and `_lr_ts`, you can measure a complex transaction using any statement submitted to the server or data received from the server as end points. Avoid measuring very short transactions; your operating system could restrict timing resolution.

This example splits a response into arbitrary units, each measured as separate transactions.

> **NOTE:** The use of multiple `sqlnrecv` commands per `sqlexec` lets Performance reports automatically calculate separate response times for individual parts of a response. However, each `sqlnrecv` command's response time must share the same starting time, namely that of the common `sqlexec` command. This restriction does not apply to `start_time`/`stop_time.`

```
sqlexec "select * from Student";

start_time ["p1_wait"] _lr_ts;

sqlnrecv 10/* fetch the first 10 rows */

/* wait for phase 1 ends and output for phase 1 begins*/
stop_time ["p1_wait"] _fr_ts;
start_time ["p1_out"] _fr_ts;

/* output for phase 1 ends and wait for phase 2 begins*/
stop_time ["p1_out"] _lr_ts;
start_time ["p2_wait"] _lr_ts;

sqlnrecv ALL_ROWS/* fetch rest of results */

/* wait for phase2 ends; output for phase2 begins*/
stop_time ["p2_wait"] _fr_ts;
start_time ["p2_out"] _fr_ts;

/* output for phase 2 ends: */
stop_time ["p2_out"] _lr_ts;
```

`time_ids` are truncated to 40 characters during command recording.

See Also `stop_time`

stoc

Library Routine

Description Returns a selected character from a string argument.

Syntax `int` **`stoc`** `(`*`str`*`, `*`n`*`)`

Syntax Element	Description
str	The string expression to search.
n	An integer expression used to specify the position of one character to extract.

Comments The `stoc` routine returns the *n*th character (as an integer) of the string *`str`*. If *n* is less than 1 or exceeds the length of *`str`*, `stoc` returns the integer 0.

The `ctos` routine is the converse of `stoc`; `ctos` converts characters to strings.

Example This example returns the character `'n'`:

```
stoc("manual", 3);
```

These examples both return the character `'\0'` (zero):

```
stoc("guide", 6);
stoc("guide", 0);
```

See Also `ctos`

stop_time

Emulation Command

Description Marks the end of a block of actions being timed.

Syntax

```
int stop_time time_id ;
int stop_time time_id time;
```

Syntax Element	Description
time_id	A required ID, similar to a command ID, that has the form [*string_exp*]. If *time_id* has not been specified in a previous `start_time` in the current script, the most recent start time without a label is used instead.
time	An integer expression specifying a timestamp that overrides the current time. If *time* is not specified, the current time is used.

Comments The `stop_time` command returns an integer expression equal to the ending timestamp.

The `stop_time` command associates an ending timestamp with the *time_id*, and records both the starting time and ending time for use by LoadTest reports.

One `stop_time` command is normally used with each `start_time` command. However, multiple `stop_time` commands per `start_time` command are allowed.

Example This example shows a simple use of `start_time` and `stop_time`:

```
start_time;            /* beginning of T1 */
. . .                  /* T1 commands & responses */
stop_time ["T1"];      /* ending of transaction T1 */
```

See Also `start_time`

strlen

Library Routine

Description Returns the length of a string expression.

Syntax `int` **`strlen`** `(`*`str`*`)`

Syntax Element	Description
str	The string expression whose length you want to obtain.

Comments The `strlen` routine, equivalent to the C library routine of the same name, returns an integer specifying the number of characters in its argument.

Example In this example, the integer returned has the value 26; note that `'\n'` is a single character.

```
strlen("A string of 26 characters\n");
```

In this example, `strlen` returns the number of characters in the read-only variable `_response` and assigns them to `var`.

```
var = strlen(_response);
```

See Also `strneg` `strspan`

strneg

Library Routine

Description Creates a string expression based on character set negation (complements).

Syntax `string` **`strneg`** `(`*`str`*`)`

Syntax Element	Description
str	The string expression to negate.

Comments The `strneg` routine returns a string consisting of the negation of string *`str`* with respect to the 255-character native character set on the computer on which LoadTest is installed. Every character, numerical values 1–255, *not* occurring in *`str`* is included *once* in the returned string, sorted numerically. This routine is useful with several others, such as `strspan` and `strlen`.

The `strrep`, `strset`, and `strneg` routines create string expressions based on character repetition, character sets, or character negation.

Example In this example, the integer value 8 is assigned to `unique`, equivalent to the number of unique characters in `polyethylene`:

```
unique = 255 - strlen(strneg("polyethylene"));
```

In this example, `strneg` returns the string `abcd`, which lists each of the unique characters in `ddccbbaa` in alphabetical order:

```
strneg(strneg("ddccbbaa"));
```

In this example, `strspan` returns `22` (the number of consecutive nondigit characters beginning with the first character of the string `"up to the first digit 0 - 9"`).

```
strspan("up to the first digit 0 - 9", strneg(strset('0','9')), 1);
```

In this example, `strneg` returns the string `""`.

```
strneg(strset('\1', '\377'));
```

See Also `strspan` `strlen` `strset`

strrep

Library Routine

Description Creates a string expression based on character repetition.

Syntax string **strrep** (*rep_char*, *len*)

Syntax Element	Description
rep_char	An integer expression specifying the character to repeat.
len	An integer expression specifying the desired length.

Comments The `strrep` routine returns a string of length *len* consisting of *len* repetitions of the character *rep_char*. If *rep_char* or *len* is less than 1, or if *rep_char* is greater than 255 (`'\377'`), `strrep` returns a string of length zero (`""`).

The `strrep`, `strset`, and `strneg` routines create string expressions based on character repetition, character sets, or character negation.

Example This example returns the string `"aaaaa"`:

```
strrep('a', 5);
```

These examples both return the string "":

```
strrep('a', 0);
strrep(256, 5);
```

See Also strset strneg

strset

Library Routine

Description Creates a string expression based on user-supplied characters.

Syntax string **strset** (*beg_char*, *end_char*)

Syntax Element	Description
beg_char	An integer expression (interpreted as a character) that indicates the first character in the expression. If *beg_char* is less than 1 or exceeds the value of *end_char*, strset returns a string of length zero ("").
end_char	An integer expression (interpreted as a character) that indicates the last character in the expression. If *end_char* is greater than 255 ('\377'), its value is silently changed to 255.

Comments The strset routine returns a string consisting of the set of characters between (and including) the characters *beg_char* and *end_char*.

The strrep, strset, and strneg routines create string expressions based on character repetition, character sets, or character negation.

Example This example returns the string "abcdefghijklmnopqrstuvwxyz":

```
strset('a', 'z');
```

This example returns the string "":

```
strset('B', 'A');
```

This example returns the set of characters between temp1 and temp2, and stores the returned string in var:

```
var = strset(temp1, temp2);
```

See Also strrep strneg

strspan

Library Routine

Description

Returns the length of the initial segment within a string expression, beginning at the specified position.

Syntax

`int` **`strspan`** `(`*`str`*`,` *`char_set`*`,` *`pos`*`)`

Syntax Element	Description
str	The string to search.
char_set	A set of characters to search for within *str*.
pos	An integer expression that specifies the position within *str* where the search should begin.

Comments

The `strspan` routine returns distance information about the span length of a set of characters within a string expression. Specifically, it returns the length of the initial segment within *str*, beginning at the ordinal position *pos*, which consists entirely of characters from *char_set*. If *pos* is less than 1 or exceeds the length of *str*, `strspan` returns an integer value of 0.

The `cindex`, `lcindex`, `sindex`, and `lsindex` routines return positional information about either the first or last occurrence of a specified character or set of characters within a string expression.

Example

This example returns the fifth field in the read-only variable `_response` and stores the value in `var`:

```
var= strspan(_response ",", 5);
```

This example returns the integer value 2:

```
strspan("moo goo gai pan", "aeiou", 2);
```

This example returns the integer value 3:

```
strspan("aeiou", "eieio", 3);
```

This example returns the integer value 0:

```
strspan("had a farm", "eieio", 11);
```

In this example, `strspan` returns `22` (the number of consecutive nondigit characters beginning with the first character of the string `"up to the first digit 0 - 9"`).

```
strspan("up to the first digit 0 - 9", strneg(strset('0','9')), 1);
```

See Also

`cindex` `lcindex` `sindex` `lsindex` `strstr`

strstr

Library Routine

Description

Searches for one string within another.

Syntax

`int` **`strstr`**`(`*`str1`*`, `*`str2`*`)`

Syntax Element	Description
str1	The string expression to search.
str2	The string expression to find.

Comments

The `strstr()` function returns the ordinal position within *`str1`* of the first occurrence of *`str2`*. If *`str2`* is not found in str1, `strstr()` returns 0. This function is equivalent to the standard C library function of the same name.

Example

This example uses `strstr()` to find the base64–encoded login ID and password contained in the given request text.

```
string auth_str, key, log_pass, request_text;
int start, end;

key = "Authorization:Basic";
start = strstr(request_text, key);
start += strlen(key);
auth_str = substr(request_text, start, 10000);
end = strstr(auth_str, "\r\n");
auth_str = substr(auth_str, 1, end – 1);
```

See Also

`cindex` `lcindex` `lsindex` `sindex` `strspan`

subfield

Library Routine

Description

Extracts substrings from string expressions based on field position.

Syntax

`string` **`subfield`** `(`*`str`*`, `*`field_sep`*`, `*`n`*`)`

Syntax Element	Description
str	The string to search.
field_sep	A string expression containing a set of field separator characters.
n	An integer expression indicating the desired field to search within *`str`*.

Comments The `subfield` routine returns a string representing the *n*th field within the string *str*, where fields are delimited within *str* by one or more consecutive separator characters contained in the string *field_sep*. If *n* is less than 1, or if *str* contains fewer than *n* fields, or if *n* equals 1 *and* *str* begins with a separator character, `subfield` returns a string of zero length (`""`).

Example This example returns the fifth field in the read-only variable `_response` and stores the value in `var`:

```
var= subfield(_response ",", 5);
```

This example returns the string "b":

```
subfield("a,b,c,d", ",", 2);
```

This example returns the string `"104"`:

```
subfield("104.13", ".", 1);
```

This example returns the string `"9"`:

```
subfield("1,000.9", ",.", 3);
```

This example returns the string (`""`):

```
subfield("xxyzxxx", "xyz", 1);
```

This example returns the string `"3"`:

```
subfield(",1,2,3"", ",", 4);
```

See Also `substr`

substr

Library Routine

Description Extracts substrings from string expressions based on character position.

Syntax `string` **`substr`** `(`*`str`*`, `*`pos`*`, `*`len`*`)`

Syntax Element	Description
str	The string to search.
pos	An integer expression specifying the position of the first character of the substring.
len	An integer expression specifying the maximum length of the returned substring.

Comments The `substr` routine returns the substring within the string *str*, beginning at the ordinal position *pos* with (maximum) length *len*. If either *len* or *pos* is less than 1 or if pos exceeds the length of *str*, `substr` returns a string of zero length ("").

Example This example returns the first five characters in the read-only variable `_response` and stores the value in `var`:

```
var = substr(_response 1, 5);
```

This example returns the string "knack":

```
substr("knackwurst", 1, 5);
```

This example returns the string "wurst":

```
substr("knackwurst", 6, 100);
```

This example returns the string (""):

```
substr("knackwurst", 11, 1);
```

See Also `subfield`

sync_point

Statement

Description Waits for users in a LoadTest schedule to synchronize.

Syntax **sync_point** *sync_point_name*;

Syntax Element	Description
sync_point_name	A string constant that names the synchronization point. The name can have from 1 to 40 characters.

Comments A script pauses at a synchronization point until the release criteria specified by the schedule have been met. At that time, the script delays a random time specified in the schedule, and then resumes execution.

Typically, you will want to insert synchronization points into a LoadTest schedule rather than inserting the `sync_point` command into a script.

If you insert a synchronization point through a schedule, synchronization occurs at the beginning of the script. If you insert a synchronization point into a script through the `sync_point` command, synchronization occurs at that point in the script where you inserted the command. You can insert the command anywhere in the script.

For more information about inserting synchronization points in a schedule, see the *Using Rational LoadTest* manual.

Example In this example, a user makes a database connection and then synchronizes with other users before proceeding.

```
t_calvin_PAC = sqlconnect("t_calvin_PAC", "scott", "tiger",
     "t:calvin:PAC", "oracle7.3");
set Server_connection = t_calvin_PAC;
sync_point "logon";
sqlexec  ["school001"] "alter session set nls_language= 'AMERICAN' "
     "nls_te"rritory= 'AMERICA'";
sqlexec  ["school002"] "select * from student";
sqlnrecv ["school003"] ALL_ROWS;
```

See Also `wait`

system

Library Routine

Description Allows an escape mechanism to the UNIX shell from within a virtual user script running on a UNIX system.

Syntax **`system`** `(`*`cmd_str`*`)`

Syntax Element	Description
cmd_str	A string expression specifying the UNIX command to execute.

Comments The `system` routine behaves like the C routine of the same name.

`system` causes *`cmd_str`* given to the UNIX shell **`/bin/sh(1)`** as input, as if the string had been typed as a command at a terminal. `system` waits until the shell has completed execution of *`cmd_str`*, and then returns the exit status of the shell (as an integer expression). *`cmd_str`* must be accessible from the PATH environment variable and must have execute permissions set. The standard input, standard output, and standard error files used by the shell correspond to the same files used by VU. If standard output, or any other user-specified file opened for writing, is accessed by both the virtual user script and the invoked `system` command, all previous buffered output by VU is written out with `fflush` before the call to `system` to ensure correct file I/O operation.

The UNIX process environment available to *`cmd_str`* is identical to the environment of the virtual user, as described under *getenv* on on page 6-42. Therefore, if *`cmd_str`* requires values of certain predetermined environment variables to be different from those in the virtual users environment, they should be explicitly mentioned on the `system` command line, as shown in the second example below.

Example

In this example, if the virtual user's ID has the value 1, then the current working directory is output to the file `dir1`, and `system` returns an integer expression equal to the shell's exit status. After completion of `system`, the VU I/O library routines are used to access `dir1`, and then used to incorporate the result of the `pwd` command in further processing.

```
system("pwd > dir" + itoa(_uid));
```

This example defines the environment variables `HOME` and `MAIL` to the script `read_my_mail`; executes `read_my_mail`; and then returns its exit status.

```
system("HOME=/u/tester1 MAIL=/u/tester1/mail read_my_mail");
```

See Also

None.

tempnam

Library Routine

Description

Generates unique temporary file names.

Syntax

```
string tempnam (dir, prefix)
```

Syntax Element	Description
dir	A string expression that qualifies the pathname. The directory part of the pathname is chosen as the first accessible directory name from the following four sources (in the order shown): ▶ The Windows NT or UNIX environment variable `TMPDIR` (the `getenv` library routine discusses the UNIX process environment available to virtual user scripts) ▶ *dir* ▶ `P_tmpdir` as defined in `<stdio.h>` ▶ `/tmp`
prefix	A string expression that indicates the prefix added to the temporary file name.

Comments

The `unlink` routine, which deletes files, and `tempnam` are often used together because temporary files are removed as soon as their usefulness has expired.

Example

If the Windows NT or UNIX environment variable `TMPDIR` is undefined, `tempnam` returns a temporary file name in the current (.) directory, such as `./AAAa02179`. The actual file name of the temporary file returned by `tempnam` will vary.

```
tempnam(".", "");
```

If the Windows NT or UNIX environment variable `TMPDIR` has the value `/tmp`, `tempnam` returns a temporary file name in the `/tmp` directory, prefixed by `mine`, such as `/tmp/mineBAAa02179`:

```
tempnam(".", "mine");
```

If the Windows NT or UNIX environment variable TMPDIR is undefined, and `P_tmpdir` is defined in < `stdio.h`> to have the value `/usr/tmp`, `tempnam` returns a temporary file name in the `/usr/tmp` directory, such as `/usr/tmp/CAAa02179`. After the file has been opened, processed, and closed, `unlink` removes it:

```
string temp_filename;

temp_filename = tempnam("", "");
tmpfile_des = open(temp_filename, "w");

/* do file processing on the temporary file */

close(tmpfile_des);
unlink(temp_filename);
```

See Also `unlink` `getenv`

testcase

Emulation Command

Description Checks a response for specific results, and reports and logs them.

Syntax `int` **`testcase`** `[`*`cmd_id`*`]` *`condition`* `[,` *`log_string`* `[,` *`fail_string`*`]]`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
condition	An integer expression. If the value of *condition* is > 0, the `testcase` command passes; otherwise, it fails. `testcase` returns the value of *condition*.
log_string	An optional string expression used when logging a passed `testcase` (or a failed `testcase` if *fail_string* is not specified). If *log_string* is not specified, no log entry is generated for `testcase`.
fail_string	An optional string expression used when logging a failed `testcase`. If *fail_string* is not specified, *log_string* is used for both pass and fail cases.

Comments The `testcase` command enables you to check a response for specific results, and to record or log a pass or fail status based on conditions that you specify.

Like `emulate`, the arguments (*condition*, *log_string*, and *fail_string*) are not evaluated before calling the command. Instead, `testcase` operates much like the conditional operator (`?:`). *condition* is evaluated, and based on the result of *condition*, either *log_string* or *fail_string* is evaluated.

Another difference between `testcase` and most other emulation commands is that `testcase` does not "think" before evaluating the condition.

The `testcase` command is affected by the following VU environment variables: `Log_level` and `Record_level`.

Example In this example, `test001` is not logged, but `test002` and `test003` are logged, depending on the value of `Log_level`.

```
testcase ["test001"] match ("XYZ", _response);
testcase ["test002"] match ("XYZ", _response), "XYZ test";
testcase ["test003"] match ("XYZ", _response), "Found XYZ",
"Could not find XYZ";
```

See Also `emulate`

time

Library Routine

Description Returns the current time in integer format.

Syntax `int time ()`

Comments The `time` routine returns an integer representing the current time in milliseconds. `time` uses the same time source and format used by the emulation commands when timestamping input and output. This time source is reset to zero during initialization.

A related routine, `tod`, returns the current time in string format.

Example This example prints the current time and then prints the time that has elapsed. The `_lr_ts` read-only variable contains the timestamp of the last received data.

```
printf ("The time of day is %s.", tod());
printf ("%d milliseconds have elapsed since the \
last rows received from the server",
time() - _lr_ts);
```

See Also `tod`

tod

Library Routine

Description Returns the current time in string format.

Syntax string **tod** ()

Comments The `tod` routine returns a 24-character string representing the current time in time-of-day format (such as "`Fri Apr 11 15:29:02 1997`").

A related routine, `time`, returns the current time in integer format.

Example This example prints the current time and then prints the time that has elapsed. The `_lr_ts` read-only variable contains the timestamp of the last received data.

```
printf ("The time of day is %s.", tod());
printf ("%d milliseconds have elapsed since the \
last rows received from the server",
time() - _lr_ts);
```

See Also time

trans

Library Routine

Description Substitutes or deletes selected characters in a string expression.

Syntax string **trans** (*str*, *in_str*, *out_str*)

Syntax Element	Description
str	The subject string expression.
in_str	A string expression that specifies the set of characters within *str* that should be translated or deleted.
out_str	A string expression that specifies the set of characters to which the characters in *in_str* are translated.

Comments The `trans` routine returns a translated version of *str* by substituting or deleting selected characters. Any character in *str* not found in *in_str* is copied unmodified to the returned string. Characters found in *in_str* are substituted by the corresponding character in *out_str* (based on character position). If there is not a corresponding character in *out_str*, the character is deleted (not copied to the returned string).

A special abbreviated convention is useful for padding *out_str*. If *out_str* has at least two characters and ends in an asterisk (*), *out_str* is automatically padded with the character preceding the asterisk until the length of *out_str* is the same as the length of *in_str*. For example, if *out_str* is "abc*" and the length of *in_str* is 10, *out_str* is converted to abcccccccc before the translation begins. If this action is undesirable, change the order of the characters in *in_str* and *out_str* so that *out_str* does not end in an asterisk.

The sqtrans routine is the same as trans, except that it "squeezes" all strings of repeated characters in the returned string that occur in *out_str* to single characters.

Example This example takes the string rational and translates each letter into uppercase. The strset routine specifies a range of letters.

```
trans("rational", strset('a','z'), strset('A','Z'));
```

This example produces the string "Spanish." When trans finds the letter g, it substitutes a; when it finds the letter l it substitutes n, and so on:

```
trans("English", "glnE", "anpS");
```

This example produces the string "rmv my vwls." When trans finds the letter a, e, i, o, or u, it deletes it (substitutes nothing).

```
trans("remove my vowels", "aeiou", "");
```

These two examples are equivalent and produce the string "$XXX.XX":

```
trans("$141.19", strset('0','9'), "X*");
trans("$141.19", "0123456789", "XXXXXXXX");
```

This example, without the asterisk, produces the string "$.":

```
trans("$141.19", strset('0','9'), "X");
trans("$141.19", "0123456789", "X");
```

This example removes each tab in the input string and replaces it with a space, so two spaces surround each word:

```
trans("\t\tHello,\t\tworld\t\t" "\t", " ");
```

See Also sqtrans

tux_allocbuf

Emulation Function

Description Allocates a free buffer.

Syntax `int` **`tux_allocbuf`** `(`*`buftype`*`)`

Syntax Element	Description
buftype	Must be one of the following buffer types: `BUFTYP_CLIENTID`, `BUFTYP_REVENT`, `BUFTYP_SUBTYPE`, `BUFTYP_TPEVCTL`, `BUFTYP_TPQCTL`, `BUFTYP_TPTRANID`, `BUFTYP_TYPE`.

Comments Buffers allocated by `tux_allocbuf` are freed with `tux_freebuf`.

If `tux_allocbuf` completes successfully, it returns a buffer handle. Otherwise, it returns a value of `NUM_BUF` and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example allocates a buffer of type TPQCTL (queue control) and sets an integer field.

```
tpqctl = tux_allocbuf(BUFTYP_TPQCTL);
tux_setbuf_int(tpqctl, "flags", TPQCORRID | TPQFAILUREQ | TPQREPLYQ |
    TPQGETBYCORRID | TPQMSGID);
```

See Also `tux_freebuf`

tux_allocbuf_typed

Emulation Function

Description Allocates a TUXEDO-typed buffer.

Syntax `int` **`tux_allocbuf_typed`** `(`*`buftype`*`,` *`subtype`*`,` *`size`*`)`

Syntax Element	Description
buftype	Must be one of the following buffer types: `BUFTYP_CARRAY`, `BUFTYP_FML`, `BUFTYP_FML32`, `BUFTYP_STRING`, `BUFTYP_TPINIT`, `BUFTYP_X_OCTET`, `BUFTYP_VIEW`, `BUFTYP_VIEW32`, `BUFTYP_X_C_TYPE`, or `BUFTYP_X_COMMON`.
subtype	A string expression that identifies the user-defined structure contained within the `VIEW`, `VIEW32`, `X_C_TYPE`, or `X_COMMON` typed buffer. You must have defined the UNIX environment variables `VIEWFILES` and `VIEWDIR`. Otherwise, *`subtype`* is an empty string.
size	The requested buffer size, in bytes.

Comments If `tux_allocbuf_typed` completes successfully, it returns a buffer handle. Otherwise, it returns a value of `NULL_BUF` and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

This function is equivalent to the function `tux_tpalloc`. When you record TUXEDO traffic, the resulting script contains `tux_tpalloc`, not `tux_allocbuf_typed`.

Example This example allocates string-typed buffer of 30 bytes and then sets the string "Jake Brake" to the buffer.

```
name = tux_allocbuf_typed(BUFTYP_STRING, "", 30);
tux_setbuf_string(name, "", "Jake Brake");
```

See Also `tux_tpalloc`   `tux_freebuf`

tux_bq

Send Emulation Command

Description Queues a UNIX command for background processing.

Syntax `int` **`tux_bq`** `[` *`cmd_id`* `]` *`cmd`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
cmd	A string expression that contains the UNIX command executed.

Comments If `tux_bq` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

`tux_bq` is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example queues a UNIX command for background printing of a file.

```
tux_bq ["tbq_001"] "lp -d hp5mp /home/tuxedo/tux.env";
```

See Also None.

tux_freebuf
Emulation Function

Description Deallocates a free buffer.

Syntax

```
int tux_freebuf (bufhnd)
```

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.

Comments If `tux_freebuf` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example deallocates the buffer `tpqctl`.

```
/* tux_allocbuf ... */

tux_freebuf(tpqctl);
```

See Also `tux_allocbuf` `tux_allocbuf_typed`

tux_getbuf_ascii
Emulation Function

Description Gets a free buffer or buffer member and converts it to a string.

Syntax

```
string tux_getbuf_ascii (bufhnd, mbrspec)
```

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.
mbrspec	A buffer member specification.

Comments If `tux_getbuf_ascii` completes successfully, it returns a string representation of the buffer or buffer member. Nonprintable characters are converted to hex or backslash format. (See *How a VU Script Represents Unprintable Data* on page 3-34.) Otherwise, `tux_getbuf_ascii` returns an empty string and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

You should check `_error` explicitly after every call to `tux_getbuf_ascii`.

Example This example gets the buffer `odata` and returns an ASCII representation.

```
idata = tux_tpalloc("CARRAY", "", 16);
tux_setbuf_ascii(idata, "", "@S8‘b42fff48ba‘@R‘13e2228114‘E");
odata = tux_tpalloc("CARRAY", "", 8);
tux_tpcall ["k1_cnx020"] "math::mul", idata, odata, (TPSIGRSTRT);

{ string asciified_result; }
asciified_result = tux_getbuf_ascii(odata, "");
if (_error)
    ...    /* asciified_result is invalid */
```

See Also None.

tux_getbuf_int

Emulation Function

Description Gets a free buffer or buffer member and converts it to a VU integer.

Syntax `int` **`tux_getbuf_int`** `(`*`bufhnd`*`, `*`mbrspec`*`)`

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.
mbrspec	A buffer member specification.

Comments If `tux_getbuf_int` completes successfully, it returns an integer representation of the buffer or buffer member. Otherwise, it returns a 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

You must check `_error` explicitly after every call to `tux_getbuf_int`.

Example This example gets the buffer `result_buf` and returns an integer representation.

```
args_buf = tux_tpalloc("FML32", "", 0);
tux_setbuf_int(args_buf, ".FLD_LONG:0", 123);
tux_setbuf_int(args_buf, ".FLD_LONG:1", 456);
tux_tpcall "Add", args_buf, result_buf, TPNOFLAGS;

result = tux_getbuf_int(result_buf, ".FLD_LONG:2");
if (_error)
    ...    /* result is invalid */
```

See Also `tux_setbuf_int`

tux_getbuf_string

Emulation Function

Description Gets a free buffer or buffer member and converts it to a string without converting nonprintable characters.

Syntax `string` **`tux_getbuf_string`** `(`*`bufhnd`*`, `*`mbrspec`*`)`

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.
mbrspec	A buffer member specification.

Comments If `tux_getbuf_string` completes successfully, it returns a string representation of the buffer or buffer member. Otherwise, it returns an empty string and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

You must check `_error` explicitly after every call to `tux_getbuf_string`.

Example This example gets the buffer `result_buf` and returns a string representation.

```
args_buf = tux_tpalloc("FML32", "", 0);
tux_setbuf_int(args_buf, ".FLD_LONG:0", 123);
tux_setbuf_int(args_buf, ".FLD_LONG:1", 456);
tux_tpcall "Add", args_buf, result_buf, TPNOFLAGS;

{ string result_str; }
result_str = tux_getbuf_string(result_buf, ".FLD_LONG:2");
if (_error)
    ...    /* result_str is invalid */
```

See Also `tux_setbuf_string`

tux_reallocbuf

Emulation Function

Description Resizes a free buffer.

Syntax *int* **tux_reallocbuf** (*bufhnd*, *size*)

Syntax Element	Description
bufhnd	A buffer allocated with tux_allocbuf, tux_allocbuf_typed, or tux_tpalloc.
size	The requested buffer size, in bytes.

Comments If tux_reallocbuf completes successfully, it returns a buffer handle. Otherwise, it returns a value of NULL_BUF and sets _error, _error_type, and _error_text to indicate the error condition.

Example This example allocates the string-type buffer msgbuf, checks the length of a message string, and then resizes msgbuf to the length of msglen.

```
msgbuf = tux_allocbuf_typed(BUFTYP_STRING, "", 0);

/* ... */

msglen = strlen(message) + 1;
if (tux_sizeofbuf(msgbuf) < msglen)
    msgbuf = tux_reallocbuf(msgbuf, msglen);
```

See Also tux_allocbuf

tux_setbuf_ascii

Emulation Function

Description Writes a string value into a buffer or buffer member.

Syntax int **tux_setbuf_ascii** (*bufhnd*, *mbrspec*, *ascval*)

Syntax Element	Description
bufhnd	A buffer allocated with tux_allocbuf, tux_allocbuf_typed, or tux_tpalloc.
mbrspec	A buffer member specification.
ascval	A string expression with nonprintable characters in hexadecimal format or backslash format. (See *How a VU Script Represents Unprintable Data* on page 3-34.)

Comments If tux_setbuf_ascii completes successfully, it returns a value of 1. Otherwise it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

Example This example allocates the buffer `idata` and then writes a string value to the buffer.

```
idata = tux_tpalloc("CARRAY", "", 16);
tux_setbuf_ascii(idata, "", "@S8‘b42fff48ba‘@R‘13e2228114‘E");
```

See Also `tux_getbuf_ascii`

tux_setbuf_int

Emulation Function

Description Sets a free buffer or buffer member with a VU integer value.

Syntax `int` **`tux_setbuf_int`** `(`*`bufhnd`*`, `*`mbrspec`*`, `*`intval`*`)`

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.
mbrspec	A buffer member specification.
ascval	An integer expression.

Comments If `tux_setbuf_int` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example allocates the buffer `data` and then writes an integer value to the buffer.

```
data = tux_tpalloc("FML", "", 0);
tux_setbuf_int(data, "XA_TYPE", 5);
```

See Also `tux_getbuf_int`

tux_setbuf_string

Emulation Function

Description Sets a free buffer or buffer member with a VU string value, without converting nonprintable characters.

Syntax `int` **`tux_setbuf_string`** (*`bufhnd`*, *`mbrspec`*, *`strval`*)

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.
mbrspec	A buffer member specification.
strval	A string expression. Do not convert nonprintable characters into hexadecimal or backslash format. If you do, they are loaded into *bufhnd* unmodified.

Comments If `tux_setbuf_string` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example allocates the buffer `tpqctl` and then writes a string value to the buffer.

```
tpqctl = tux_allocbuf(BUFTYP_TPQCTL);
tux_setbuf_string(tpqctl, "corrid", "req302");
```

See Also `tux_getbuf_string`

tux_sizeofbuf

Emulation Function

Description Returns the size of a buffer.

Syntax `int` **`tux_sizeofbuf`** (*`bufhnd`*)

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.

Comments If `tux_sizeofbuf` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example allocates the sting-type buffer `msgbuf`, checks the length of a message string, and then resizes `msgbuf` if the size of `msglen` is greater than `msgbuf`.

```
msgbuf = tux_allocbuf_typed(BUFTYP_STRING, "", 0);

/* ... */

msglen = strlen(message) + 1;
if (tux_sizeofbuf(msgbuf) < msglen)
    msgbuf = tux_reallocbuf(msgbuf, msglen);
```

See Also None.

tux_tpabort

Send Emulation Command

Description Aborts the current transaction.

Syntax `int` **`tux_tpabort`** `[` *`cmd_id`* `]` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
flags	An integer expression whose value must be `TPNOFLAGS`. The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpabort` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpabort` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example aborts a TUXEDO transaction in progress:

```
/* begin transaction, 180-sec timeout */
tux_tpbegin (180, TPNOFLAGS);

/* abort current transaction */
tux_tpabort ["tabo013"] TPNOFLAGS;
```

See Also `tux_tpbegin`

tux_tpacall

Send Emulation Command

Description Sends a service request.

Syntax `int` **`tux_tpacall`** `[` *`cmd_id`* `]` *`svc`*`,` *`data`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
svc	A string expression that identifies the service.
data	A string expression that must reference a buffer allocated by `tux_tpalloc`().
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOREPLY`, `TPNOTIME`, `TPNOTRAN`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpacall` completes successfully, it returns a value of 1. Otherwise it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpacall` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example allocates the buffer `data`, populates the buffer with transaction information, and then sends a service request to the `OPEN_ACCT` service.

```
data = tux_tpalloc("FML", "", 0);
tux_setbuf_int(data, "XA_TYPE", 5);
tux_setbuf_int(data, "8194", 41162);
tux_setbuf_int(data, "8195", 0);
tux_setbuf_int(data, "BRANCH_ID", 1);
tux_setbuf_ascii(data, "ACCT_TYPE", "C");
tux_setbuf_ascii(data, "MID_INIT", "Q");
tux_setbuf_string(data, "40964", "F11");
tux_setbuf_string(data, "40966", "OPEN");
tux_setbuf_string(data, "40968", "OPEN_ACCT");
tux_setbuf_string(data, "PHONE", "919-870-8800");
tux_setbuf_string(data, "ADDRESS", "100 Happy Trail");
tux_setbuf_string(data, "SSN", "123-45-6789");
tux_setbuf_string(data, "LAST_NAME", "John");
tux_setbuf_string(data, "FIRST_NAME", "Customer");
tux_setbuf_string(data, "SAMOUNT", "1000");
tux_setbuf_ascii(data, "49170",
    "‘a0719108000000000000091e8a072910800000000000091e8‘@s‘91080000000000009"
    "1e8a06f9108000000000000091e8a06d9108000000000000091e8a06c910800000000000"
    "091e8‘ h‘9108000000000000091e8a0ca9108000000000000091e8‘"
);
call_1 = tux_tpacall ["bankap002"] "OPEN_ACCT", data, (TPNOBLOCK |
```

```
TPSIGRSTRT);
call_1_fs_ts = _fs_ts;
tux_tpfree(data);
```

See Also tux_tpgetrply

tux_tpalloc

Emulation Function

Description Allocates TUXEDO-typed buffers.

Syntax int **tux_tpalloc** (*type*, *subtype*, *size*)

Syntax Element	Description
type	A string expression that evaluates to CARRAY, FML, FML32, STRING, TPINIT, X_OCTET, VIEW, VIEW32, X_C_TYPE, or X_COMMON.
subtype	A string expression that identifies the user-defined structure contained within the VIEW, VIEW32, X_C_TYPE, or X_COMMON typed buffer. You must have defined the UNIX environment variables VIEWFILES and VIEWDIR. Otherwise, *subtype* is an empty string.
size	The requested buffer size, in bytes.

Comments If tux_tpalloc completes successfully, it returns a buffer handle. Otherwise, it returns a value of NULL_BUF and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpalloc function is equivalent to the function tux_tpalloc, except that it is an ATMI call.

Example This example allocates a buffer of 9 bytes that evaluates to STRING.

```
data = tux_tpalloc("STRING", "", 9);
tux_tpgetrply ["tget006"] call_6, data, TPNOFLAGS;
```

See Also tux_tpfree

tux_tpbegin

Emulation Function

Description Begins a transaction.

Syntax `int` **`tux_tpbegin`** `(`*`timeout`*`, `*`flags`*`)`

Syntax Element	Description
timeout	The transaction timeout threshold, in seconds.
flags	An integer expression whose value must be `TPNOFLAGS`. The values of *`flags`* are defined in the TUXEDO header file.

Comments If `tux_tpbegin` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example begins a TUXEDO transaction with a 60-second timeout.

```
tux_tpbegin(60, TPNOFLAGS);
```

See Also `tux_tpabort` `tux_tpcommit`

tux_tpbroadcast

Send Emulation Command

Description Broadcasts notification by name.

Syntax `int` **`tux_tpbroadcast`** `[` *`cmd_id`* `]` *`lmid`*, *`usrname`*, *`cltname`*, *`data`*, *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
lmid	A string expression that evaluates to a logical computer ID.
usrname	A string expression that selects the user name.
cltname	A string expression that selects the target client set.
data	Typed buffer data that must reference a buffer allocated by `tux_tpalloc()`
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOTIME`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpbroadcast` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpbroadcast` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example allocated the buffer `data`, sets the string "Wake Up" in the buffer, and then broadcasts the string to Jack on `SERVER3`.

```
data = tux_tpalloc("STRING", "", 0);
tux_setbuf_string(data, "", "Wake Up!");
tux_tpbroadcast ["tbro002"] "SERVER3", "Jack", "PCAE05", data,
    TPNOFLAGS;
tux_tpfree(data);
```

See Also None.

tux_tpcall

Send Emulation Command

Description Sends a service request and awaits its reply.

Syntax `int` **`tux_tpcall`** `[` *`cmd_id`* `]` *`svc`*`,` *`idata`*`,` *`odata`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
svc	A string expression that identifies the service.
idata	A buffer handle that must reference a buffer allocated by `tux_tpalloc`().
odata	A buffer handle that must reference a buffer allocated by `tux_tpalloc`().
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOCHANGE`, `TPNOTIME`, `TPNOTRAN`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpcall` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpcall` command updates `_tux_tpurcode`.

The `tux_tpcall` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example allocates the buffers `idata` and `odata`, and then sends a service request to the "math::exp" service.

```
idata = tux_tpalloc("CARRAY", "", 16);
tux_setbuf_ascii(idata, "", "@S8‘b42fff48ba‘@R‘13e2228114‘E");
odata = tux_tpalloc("CARRAY", "", 8);
set Think_avg = 12;
tux_tpcall ["k1_cnx020"] "math::exp", idata, odata, (TPSIGRSTRT);
tux_tpfree(idata);
tux_tpfree(odata);
```

See Also None.

tux_tpcancel

Emulation Function

Description Cancels a call descriptor for an outstanding reply.

Syntax `int` **`tux_tpcancel`** `(`*`cd`*`)`

Syntax Element	Description
cd	The canceled call descriptor.

Comments If `tux_tpcancel` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example cancels the `tux_tpacall` represented by `call_23`.

```
call_23 = tux_tpacall "EDI-SENDJOB", jobdesc, TPNOFLAGS;

/* ... */

tux_tpcancel(call_23);
```

See Also `tux_tpacall`

tux_tpchkauth

Emulation Function

Description Checks whether authentication is required to join an application.

Syntax `int` **`tux_tpchkauth`** `( )`

Comments If `tux_tpchkauth` completes successfully, it returns a valid authorization level. Otherwise, it returns a value of -1 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example checks if authentication is required, and if so, prints a message indicating the script requires authentication.

```
if (tux_tpchkauth() != TPNOAUTH)
    print "Script requires authentication info!";
```

See Also None.

tux_tpcommit

Send Emulation Command

Description Commits the current transaction.

Syntax int **tux_tpcommit** [*cmd_id*] *flags*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
flags	An integer expression whose value must be TPNOFLAGS. The values of *flags* are defined in the TUXEDO header file.

Comments If tux_tpcommit completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpcommit command is affected by the think time, Log_level, and Record_level VU environment variables.

Example This example commits the current transaction.

```
/* tux_tpbegin ... */

tux_tpcommit ["tcom007"] TPNOFLAGS;
```

See Also tux_tpbegin

tux_tpconnect

Send Emulation Command

Description Establishes a conversational service connection.

Syntax `int tux_tpconnect [ cmd_id ] svc, data, flags`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
svc	A string expression that identifies the service.
data	Must reference a buffer allocated by `tux_tpalloc`().
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOTIME`, `TPNOTRAN`, `TPRECVONLY`, `TPSENDONLY`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpconnect` completes successfully, it returns a connection descriptor. Otherwise, it returns a value of -1 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpconnect` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example establishes a conversational connection with the service AUDITC.

```
conn_1 = tux_tpconnect ["demo1.002"] "AUDITC", NULL_BUF, TPSENDONLY;
```

See Also `tux_tpdiscon`

tux_tpdequeue

Send Emulation Command

Description Removes a message from a queue.

Syntax

int tux_tpdequeue [*cmd_id*] *qspace*, *qname*, *ctl*, *data*, *flags*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
qspace	A string expression that identifies the queue space.
qname	A string expression that identifies the queue.
ctl	Must reference a buffer of type BUFTYP_TPQCTL or BUFTYP_NULL.
data	Must reference a buffer allocated by tux_tpalloc().
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOCHANGE, TPNOTIME, TPNOTRAN, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments

If tux_tpdequeue completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpdequeue command is affected by the think time, Log_level, and Record_level VU environment variables.

Example

This example removes the message represented by the buffer tpqctl from the queue space TMQUEUE.

```
tpqctl = tux_allocbuf(BUFTYP_TPQCTL);
tux_setbuf_int(tpqctl, "flags", TPQCORRID | TPQFAILUREQ | TPQREPLYQ |
    TPQGETBYCORRID | TPQMSGID);
tux_setbuf_string(tpqctl, "corrid", "req302");
odata = tux_tpalloc("STRING", "", 9);
tux_tpdequeue ["yang003"] "TMQUEUE", "APP_REPLY", tpqctl, odata,
    TPNOFLAGS;
tux_freebuf(tpqctl);
tux_tpfree(odata);
```

See Also

tux_tpenqueue

tux_tpdiscon

Send Emulation Command

Description Takes down a conversational service connection.

Syntax `int` **`tux_tpdiscon`** `[` *`cmd_id`* `]` *`cd`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
cd	A call descriptor indicating the connection taken down. It must be returned by `tux_tpconnect()`.

Comments If `tux_tpdiscon` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpdiscon` command is affected by the VU environment variables think time, `Log_level`, and `Record_level`.

Example This example takes down the service connection `conn_1`.

```
/* tux_tpconnect ... */

tux_tpdiscon ["demo1.002"] conn_1;
```

See Also `tux_tpconnect`

tux_tpenqueue

Send Emulation Command

Description Queues a message.

Syntax

`int` **`tux_tpenqueue`** `[` *`cmd_id`* `]` *`qspace`*`,` *`qname`*`,` *`ctl`*`,` *`data`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
qspace	A string expression that identifies the queue space.
qname	A string expression that identifies the queue.
ctl	Must reference a buffer of type BUFTYP_TPQCTL or BUFTYP_NULL.
data	Must reference a buffer allocated by tux_tpalloc().
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOCHANGE, TPNOTIME, TPNOTRAN, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments

If tux_tpenqueue completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpenqueue command is affected by the VU environment variables think time, Log_level, and Record_level.

Example

This example queues the message represented by tpqctl (queue control) to the queue space TMQUEUE.

```
tpqctl = tux_allocbuf(BUFTYP_TPQCTL);
tux_setbuf_int(tpqctl, "flags", TPQCORRID | TPQFAILUREQ | TPQREPLYQ |
    TPQMSGID);
tux_setbuf_string(tpqctl, "corrid", "req302");
tux_setbuf_string(tpqctl, "failurequeue", "APP_FAILURE");
tux_setbuf_string(tpqctl, "replyqueue", "APP_REPLY");
data = tux_tpalloc("STRING", "", 8);
tux_setbuf_string(data, "", "NC WAKE 302.82");
tux_tpenqueue ["yin002"] "TMQUEUE", "CalcSalesTax", tpqctl, data,
    TPNOFLAGS;
tux_freebuf(tpqctl);
tux_tpfree(data);
```

See Also

tux_tpdequeue

tux_tpfree

Emulation Function

Description Frees a typed buffer.

Syntax `int` **`tux_tpfree`** `(`*`ptr`*`)`

Syntax Element	Description
ptr	A buffer handle allocated with `tux_tpalloc`.

Comments If `tux_freebuf` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example frees the buffer allocated as `astring`.

```
astring = tux_tpalloc("STRING", "", 0);
 /* ... */
tux_tpfree(astring);
```

See Also `tux_tpalloc`

tux_tpgetrply

Send Emulation Command

Description Gets a reply from a previous request.

Syntax `int` **`tux_tpgetrply`** `[` *`cmd_id`* `]` *`cd`*`,` *`data`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
cd	A call descriptor returned by `tux_tpacall()`.
data	Must reference a buffer allocated by `tux_tpalloc`().
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOCHANGE`, `TPNOTIME`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments

If `tux_tpgetrply` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpgetrply` command updates `_tux_tpurcode`.

Unlike the other emulation commands, the order of the `tux_tpgetrply` emulation commands in your VU script could differ from the TUXEDO `tpgetrply` calls in your original client program. This is due to limitations of TUXEDO workstation protocol decoding. Although the order of the commands are different, they are scripted in a manner consistent with how `tpgetrply` is used by the original client program based on information recorded during the capture.

In addition, a scripted `tux_tpgetrply` blocks waiting for specific asynchronous request responses — for example, specific call descriptors — regardless of how asynchronous responses were gathered by the original client program. It is possible that reported response times for asynchronous calls are skewed when more than one is outstanding.

The `tux_tpgetrply` command is affected by the VU environment variables think time, `Log_level`, and `Record_level`.

Example

This example gets the reply from a previous `tux_tpacall` represented by `call_6`.

```
/* tux_tpacall ... */

data = tux_tpalloc("STRING", "", 9);
tux_tpgetrply ["tget006"] call_6, data, TPNOFLAGS;
start_time ["t15003"] call_6_fs_ts;
stop_time ["t15003"] _lr_ts;
tux_tpfree(data);
```

See Also

`tux_tpacall`

tux_tpinit

Send Emulation Command

Description

Joins an application.

Syntax

`int` **`tux_tpinit`** `[` *`cmd_id`* `]` *`tpinfo`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
tpinfo	Must reference a buffer of type `TPINIT` allocated by `tux_tpalloc`().

Comments In order for tux_tpinit to operate correctly, a TUXEDO-defined system environment variable named WSNADDR must be present. This variable is used by the TUXEDO client library to determine which TUXEDO Workstation Listener (WSL) to connect to.

The WSLHOST and WSLPORT system environment variables are optional. If they are defined, they will be used by tux_tpinit to generate a valid WSNADDR. If they are not defined, then tux_tpinit uses the value of WSNADDR. If WSNADDR is not defined, then tux_tpinit fails, reporting a playback error message indicating that none of the three variables were set.

If WSLHOST and WSLPORT are set, the resulting WSNADDR value overrides any previous WSNADDR value.

WSLHOST and WSLPORT can be set in the script, which is the default recorded script action, or they may be set in a LoadTest schedule. If they are set in a script and a schedule, the script values override the schedule values.

If tux_tpinit completes successfully, it returns a value of 1. Otherwise it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpinit command is affected by the think time, Log_level, and Record_level VU environment variables.

Example This example connects to the TUXEDO Workstation Listener in the environment variables WSLHOST and WSLPORT using the data set in the buffer tpinfo.

```
putenv("WSLHOST=hp715.nc.rational.com");
putenv("WSLPORT=36001");
tpinfo = tux_tpalloc("TPINIT", "", TPINITNEED(10));
tux_setbuf_string(tpinfo, "usrname", "dhinson");
tux_setbuf_string(tpinfo, "cltname", "rocinante");
tux_setbuf_int(tpinfo, "flags", TPNOFLAGS);
tux_setbuf_int(tpinfo, "datalen", 10);
tux_setbuf_ascii(tpinfo, "data", "GL‘0201‘AL‘0102‘NP");
tux_tpinit ["cx1001"] tpinfo;
tux_tpfree(tpinfo);

/* or */

tux_tpinit ["cx1001"] NULL_BUF;
```

See Also tux_tpterm

tux_tpnotify

Send Emulation Command

Description Sends notification by client identifier.

Syntax

`int` **`tux_tpnotify`** `[` *`cmd_id`* `]` *`clientid`*`,` *`data`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
clientid	Must reference a buffer of type BUFTYP_CLIENTID.
data	Must reference a buffer allocated by tux_tpalloc().
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOTIME, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments

If tux_tpnotify completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpnotify command is affected by the think time, Log_level, and Record_level VU environment variables.

Example

This example sends the notification represented in the clientid_ typed-buffer.

```
clientid_ = tux_allocbuf(BUFTYP_CLIENTID);
tux_setbuf_ascii(clientid_, "",
"‘3383‘F&‘0000000000000001c00000000‘");
set Think_avg = 1;
tux_tpnotify ["tnot006"] clientid_, NULL_BUF, TPNOFLAGS;
tux_freebuf(clientid_);
```

See Also

None.

tux_tppost

Send Emulation Command

Description

Posts an event.

Syntax

`int` **`tux_tppost`** `[` *`cmd_id`* `]` *`eventname`*`,` *`data`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
eventname	A string expression that identifies the name of the event.
data	Must reference a buffer allocated by tux_tpalloc().

(Continued)

Syntax Element	Description
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOREPLY, TPNOTIME, TPNOTRAN, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If tux_tppost completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tppost command is affected by the think time, Log_level, and Record_level VU environment variables.

Example This example posts "Switch Power Failure" to an event previously subscribed to by tux_tpsubscribe.

```
data = tux_tpalloc("STRING", "", 7);
tux_setbuf_string(data, "", "03-019");
tux_tppost ["swmon023"] "Switch_Power_Failure", data, TPNOFLAGS;
tux_tpfree(data);
```

See Also tux_tpsubscribe   tux_tpunsubscribe

tux_tprealloc

Emulation Function

Description Changes the size of a typed buffer.

Syntax int **tux_tprealloc** (*ptr*, *size*)

Syntax Element	Description
ptr	Must be a buffer handle allocated by tux_tpalloc().
size	The requested buffer size, in bytes.

Comments If tux_tprealloc completes successfully, it returns a buffer handle. Otherwise, it returns a value of NULL_BUF and sets _error, _error_type, and _error_text to indicate the error condition.

Example This example allocates the string-type buffer idata, checks the length of a message string, and then resizes idata to the length of msglen.

```
idata = tux_tpalloc("STRING", "", 0);

/* ... */

msglen = strlen(message) + 1;
if (tux_tptypes(idata, NULL_BUF, NULL_BUF) < msglen)
    idata = tux_tprealloc(idata, msglen);
```

See Also tux_tpalloc

tux_tprecv

Send Emulation Command

Description Receives a message in a conversational service connection.

Syntax int **tux_tprecv** [*cmd_id*] *cd*, *data*, *flags*, *revent*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
cd	A call descriptor indicating the conversation in which to receive data. It must be returned by tux_tpconnect().
data	Must reference a buffer allocated by tux_tpalloc().
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOCHANGE, TPNOTIME, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.
revent	Must reference a buffer of type BUFTYP_REVENT.

Comments If tux_tprecv completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tprecv command updates _tux_tpurcode.

The tux_tprecv command is affected by the think time, Log_level, and Record_level VU environment variables.

Example This example receives a message from the previously established conversational service connection conn_1.

```
revent_ = tux_allocbuf(BUFTYP_REVENT);
data = tux_tpalloc("STRING", "", 47);
set Think_avg = 1;
tux_tprecv ["bankap004"] conn_1, data, (TPNOCHANGE), revent_;
tux_freebuf(revent_);
tux_tpfree(data);
```

See Also tux_tpconnect

tux_tpresume

Send Emulation Command

Description Resumes a global transaction.

Syntax int **tux_tpresume** [*cmd_id*] *tranid*, *flags*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
tranid	Must reference a buffer of type BUFTYP_TRANID that was suspended by tux_tpsuspend().
flags	An integer expression whose value must be TPNOFLAGS. The values of *flags* are defined in the TUXEDO header file.

Comments If tux_tpresume completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpresume command is affected by the think time, Log_level, and Record_level VU environment variables.

tux_tpresume resumes the currently suspended transaction. It must be preceded by tux_tpbegin, 0 or more transacaction suboperations, and tux_tpsuspend. The data argument to tux_tpresume must be created using tux_allocbuf, and it must have been used in the call to tux_tpsuspend.

Example This example resumes a suspended transaction represented as tranid_40.

```
/* tux_tpsuspend ... */
set Think_avg = 3;
tux_tpresume tranid_40, TPNOFLAGS;
tux_freebuf(tranid_40);
```

See Also tux_tpsuspend tux_tpbegin

tux_tpscmt

Emulation Function

Description Sets when `tux_tpcommit()` returns.

Syntax `int` **`tux_tpscmt`** `(`*`flags`*`)`

Syntax Element	Description
flags	An integer expression with one of the following values: TP_CMT_LOGGED or TP_CMT_COMPLETE. The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpscmt` completes successfully, it returns the previous value of TP__COMMIT_CONTROL. Otherwise, it returns a value of -1 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example sets the return instance for the following `tux_tpcommit`.

```
tux_tpscmt(TP_CMT_COMPLETE);
/* tux_tpcommit ... */
```

See Also `tux_tpcommit`

tux_tpsend

Send Emulation Command

Description Sends a message in a conversational service connection.

Syntax `int` **`tux_tpsend`** `[` *`cmd_id`* `]` *`cd`*`,` *`data`*`,` *`flags`*`,` *`revent`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
cd	A call descriptor indicating the conversation in which to send data. It must be returned by `tux_tpconnect`().
data	Must reference a buffer allocated by `tux_tpalloc`().
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOTIME, TPRECVONLY, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.
revent	Must reference a buffer of type BUFTYP_REVENT.

Comments If `tux_tpsend` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpsend` command updates `_tux_tpurcode`.

The `tux_tpsend` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example sends message to the previously established conversational service connection `conn_1`.

```
/* Must be preceded by tux_tpconnect to start the conversation.*/
revent_ = tux_allocbuf(BUFTYP_REVENT);
data = tux_tpalloc("STRING", "", 2);
tux_setbuf_string(data, "", "t");
set Think_avg = 5043;
tux_tpsend ["bankap003"] conn_1, data, (TPRECVONLY), revent_;
tux_freebuf(revent_);
tux_tpfree(data);
/* Part of conversation between client and server in Bankapp appli-
cation. Send a message during conversation. */
tux_tpsend ["tsen.003"] conn_1, data_, (TPRECVONLY), revent_;
tux_freebuf(revent_);
tux_tpfree(data);
```

See Also `tux_tpconnect`

tux_tpsprio

Emulation Function

Description Sets the service request priority.

Syntax `int` **`tux_tpsprio`** `(`*`prio`*`, `*`flags`*`)`

Syntax Element	Description
prio	An integer expression that increments or decrements the service request priority.
flags	An integer expression with one of the following values: `TPABSOLUTE` or `TPNOFLAGS`. The values of *`flags`* are defined in the TUXEDO header file.

Comments If `tux_tpsprio` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example sets the service request priority for the following `tux_tpcall`.

```
tux_tpsprio(99, TPABSOLUTE);
/* tux_tpcall ... */
```

See Also tux_tpacall tux_tpcall

tux_tpsubscribe

Send Emulation Command

Description Subscribes to an event.

Syntax int **tux_tpsubscribe** [*cmd_id*] *eventexpr*, *filter*, *ctl*, *flags*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
eventexpr	A string expression that identifies the event the caller wants to subscribe to.
filter	A string expression that contains the Boolean file rule associated with *eventexpr*.
ctl	Must reference a buffer of type BUFTYP_TPEVCTL or BUFTYP_NULL.
flags	An integer expression with one of the following values: TPNOFLAGS, TPNOBLOCK, TPNOTIME, or TPSIGRSTRT (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If tux_tpsubscribe completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpsubscribe command is affected by the think time, Log_level, and Record_level VU environment variables.

Example This example subscribes to the event "Switch_Power_Failure".

```
tpevctl_ = tux_allocbuf(BUFTYP_TPEVCTL);
tux_setbuf_int(tpevctl_, "flags", TPEVSERVICE);
tux_setbuf_string(tpevctl_, "name1", "Panic");
subs_1 = tux_tpsubscribe ["tsub001"] "Switch_Power_Failure", "",
    tpevctl_, TPNOFLAGS;
tux_freebuf(tpevctl_);
```

See Also tux_tpunsubscribe

tux_tpsuspend

Send Emulation Command

Description Suspends a global transaction.

Syntax `int` **`tux_tpsuspend`** `[` *`cmd_id`* `]` *`tranid`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
tranid	Must reference a buffer of type BUFTYP_TRANID.
flags	An integer expression whose value must be TPNOFLAGS. The values of *flags* are defined in the TUXEDO header file.

Comments If tux_tpsuspend completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets _error, _error_type, and _error_text to indicate the error condition.

The tux_tpsuspend command is affected by the think time, Log_level, and Record_level VU environment variables.

tux_tpsuspend suspends the current transaction. It must be preceded by a call to tux_tpbegin, which began the transaction.

Example This example suspends the previously established transaction tranid_40.

```
tranid_40 = tux_allocbuf(BUFTYP_TPTRANID);
set Think_avg = 11;
tux_tpsuspend tranid_40, TPNOFLAGS;

/* tux_tpresume ... */
```

See Also tux_tpbegin tux_tpresume

tux_tpterm

Send Emulation Command

Description Leaves an application.

Syntax `int` **`tux_tpterm`** `[` *`cmd_id`* `]`

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].

Comments If `tux_tpterm` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpterm` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This example exits the application represented by command ID `tter002`.

```
/* tux_tpinit ... */

tux_tpterm ["tter002"];}
```

See Also `tux_tpinit`

tux_tptypes

Emulation Function

Description Provides information about a typed buffer.

Syntax `int tux_tptypes (ptr, type, subtype)`

Syntax Element	Description
ptr	A buffer allocated with `tux_tpalloc`.
type	Must reference a buffer of type BUFTYP_TYPE.
subtype	Must reference a buffer of type BUFTYP_SUBTYPE.

Comments If `tux_tptypes` completes successfully, it returns the buffer size. Otherwise, it returns a value of -1, and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example gets information about the typed buffer `odata` and checks if is a string-typed buffer.

```
/* tpcall ... */

type = tux_allocbuf(BUFTYP_TYPE);
tux_tptypes(odata, type, NULL_BUF);
{ string type_str; }
type_str = tux_getbuf_string(type, "");
if (type_str != "FML")
    print "Invalid odata buffer type!";
```

See Also None.

tux_tpunsubscribe

Send Emulation Command

Description Unsubscribes to an event.

Syntax `int` **`tux_tpunsubscribe`** `[` *`cmd_id`* `]` *`subscription`*`,` *`flags`*

Syntax Element	Description
cmd_id	The optional command ID available in all emulation commands. *cmd_id* has the form [*string_exp*].
subscription	An event subscription handle returned by `tux_tpsubscribe`.
flags	An integer expression with one of the following values: `TPNOFLAGS`, `TPNOBLOCK`, `TPNOTIME`, or `TPSIGRSTRT` (ignored). The values of *flags* are defined in the TUXEDO header file.

Comments If `tux_tpunsubscribe` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

The `tux_tpunsubscribe` command is affected by the think time, `Log_level`, and `Record_level` VU environment variables.

Example This examples unsubscribes to previously subscribed to event services.

```
/* tux_tpsubscribe ... */

tux_tpunsubscribe ["tuns001"] -1, TPNOFLAGS;
```

See Also `tux_tpsubscribe`

tux_typeofbuf

Emulation Function

Description Returns the type of a buffer.

Syntax `int` **`tux_typeofbuf`** `(`*`bufhnd`*`)`

Syntax Element	Description
bufhnd	A buffer allocated with `tux_allocbuf`, `tux_allocbuf_typed`, or `tux_tpalloc`.

Comments If `tux_typeofbuf` completes successfully, it returns a valid buffer type. Otherwise, it returns a value of -1 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example check if the `odata` buffer is of type `BUFTYP_FML`.

```
/* tpcall ... */

if (tux_typeofbuf(odata) != BUFTYP_FML)
    print "Invalid odata buffer type!";
```

See Also None.

tux_userlog

Emulation Function

Description Writes a message to the TUXEDO central event log.

`int` **`tux_userlog`** `(`*`message`*`)`

Syntax Element	Description
message	The string you want to write.

Comments If `tux_userlog` completes successfully, it returns a value of 1. Otherwise, it returns a value of 0 and sets `_error`, `_error_type`, and `_error_text` to indicate the error condition.

Example This example writes the `User...completed` message to the TUXEDO central event log.

```
tux_userlog("User " + itoa(_uid) + " completed run.");
```

See Also None.

ungetc

Library Routine

Description

Provides unformatted character input capability.

Syntax

`int ungetc (ret_char, file_des)`

Syntax Element	Description
ret_char	An integer expression (interpreted as a character) that specifies the character to be returned to the input buffer.
file_des	The integer file descriptor, obtained from `open`, of the file associated with the input buffer.

Comments

The `ungetc` routine replaces the character *ret_char* in the input buffer associated with the named file, thus providing an "undo" mechanism for `fgetc`. This character is returned by the next `fgetc` (or other file input) call. The file contents remain unchanged.

The `ungetc` routine returns EOF (as defined in the standard VU header file) if it cannot return the character — for example, if:

- *ret_char* equals EOF
- No input has yet been read from the named file
- More than one character of push back is attempted (via successive calls to `ungetc` with no intervening file input routine call)

Example

In this example, if the file with the descriptor `infile_des` contains the characters `ABZ14`, then the characters `ABZ` are written to the file whose descriptor is `outfile_des`, and the character 1 is returned to the input buffer associated with `infile_des`.

```
#include <VU.h>
while ((c = fgetc(infile_des)) != EOF)
if (c >= 'A' && c <= 'Z')
    fputc(c, outfile_des);
else
{
    ungetc(c, infile_des);
    break;
}
```

See Also

`fgetc`

uniform

Library Routine

Description Returns a random integer uniformly distributed in the specified range.

Syntax `int **uniform** (*min_value*, *max_value*)`

Syntax Element	Description
min_value	An integer expression whose value generally specifies the minimum random integer to be returned.
max_value	An integer expression whose value generally specifies the maximum random integer to be returned.

Comments The `uniform` routine returns a random integer uniformly distributed in the specified range.

The values of *min_value* and *max_value* can be negative as well as positive. Although unconventional, *min_value* can exceed *max_value*. However, the absolute value of the difference *min_value* - *max_value* must be less than 2147483647.

The `rand`, `srand`, `uniform`, and `negexp` routines enable the VU language to generate random numbers. The behavior of these random number routines is affected by the way you set the **Seed** and **Seed Flags** options in a LoadTest schedule. By default, the **Seed** generates the same sequence of random numbers but sets unique seeds for each virtual user, so that each virtual user has a different random number sequence. For more information about setting the seed and seed flags in a schedule, see the *Using Rational LoadTest* manual.

The `srand` routine uses the argument *seed* as a seed for a new sequence of random numbers returned by subsequent calls to the function `uniform`. If `srand` is then called with the same seed value, the sequence of random numbers is repeated. If `uniform` is called before any calls are made to `srand`, the same sequence is generated as when `srand` is first called with a seed value of 1.

Example In this example, `srand` seeds the random number generator with the current time and then prints the first 10 random numbers between -10 and 10.

```
srand(time());
for (i = 0; i < 10; i++)
printf("random number (%d): %d\n", i, uniform(-10, 10));
```

See Also `rand` `negexp` `srand`

unlink

Library Routine

Description Removes files.

Syntax `int unlink (filename)`

Syntax Element	Description
filename	A string expression specifying the name of the file to be removed.

Comments The `unlink` routine removes (unlinks) the directory entry named by *`filename`*. When all links to a file have been removed, space occupied by the file is freed and the file ceases to exist; however, this action is postponed if one or more processes still have the file opened until all references to the file have been closed. `unlink` returns 0 upon successful completion; otherwise, a VU runtime error is generated.

The `tempnam` and `unlink` routines are often used together because you should remove temporary files as soon as their usefulness has expired.

Example If the Windows NT or UNIX environment variable TMPDIR is undefined, and `P_tmpdir` is defined in <`stdio.h`> to have the value `/usr/tmp`, `tempnam` returns a temporary file name in the `/usr/tmp` directory, such as `/usr/tmp/CAAa02179`. After the file has been opened, processed, and closed, `unlink` removes it.

```
string temp_filename;

temp_filename = tempnam("", "");
tmpfile_des = open(temp_filename, "w");

/* do file processing on the temporary file */

close(tmpfile_des);
unlink(temp_filename);
```

See Also `tempnam`

user_exit

Library Routine

Description Exits an entire virtual user emulation from within any point in a virtual user script.

Syntax int **user_exit** (*status*, *msg_str*)

Syntax Element	Description
status	An integer expression specifying the target virtual user's exit status.
msg_str	A string expression specifying an optional message to be written to the standard error file.

Comments The user_exit routine causes the current script to exit immediately followed by one of three user termination sequences (see the following example). Although user_exit never returns, its return value is considered an integer type for syntactical purposes. If *msg_str* is not of zero length, it is written (before exiting the script) to standard error, preceded by the following explanatory line of text:

```
User exited from script script_name with status=N and message:
```

script_name is replaced by the appropriate script name (corresponding to the read-only variable _script), and *N* is replaced by the value of *status*. After termination of the current script, user termination is controlled according to the value of *status*.

- If *status* is greater than 0, no escape or logout sequences are executed, and the user exit status reported to LoadTest is Normal.
- If *status* is equal to 0, any logout sequences are executed, and the user exit status reported to LoadTest is Normal.
- If *status* is less than 0, any escape and logout sequences if any are executed, and the user exit status reported to LoadTest is Abnormal.

Example In this example, assume that the script's name is database4. If the value of string1 is error, the script is exited; the error message is written to standard error; all defined escape or logout sequences are executed, and the user terminates the emulation session with an Abnormal exit status:

```
if (string1 = "ERROR")
    user_exit(-1, "Fatal Error - Aborting");
```

See Also script_exit

usergroup_member

Library Routine

Description Returns the position of a virtual user within a user group.

Syntax `int` **`usergroup_member`**`(`*`group_name`*`)`

Syntax Element	Description
group_name	A string expression whose value is the name of the user group.

Comments The `usergroup_member` routine returns the position of a virtual user within a user group. The first position is 1.

Example In this example, five user groups are defined. The example prints out the position of each virtual user in the group.

```
#define MAX_GROUPS 5
{
    string groups[MAX_GROUPS] = {"Accountants", "Engineers",
       "DB Entry","Administration", "Operations"};
    int index, size;

    for (i = 0; i < MAX_GROUPS; i++)
    {
       index = usergroup_member(groups[i]);
       if (index)
       {
           size = usergroup_size(groups[i]);
          printf ("I am user number: %d in group: %s which has %d
               users", index, groups[i], size);
           break;
       }
    }
}
```

See Also `usergroup_size`

usergroup_size

Library Routine

Description Returns the number of members in a user group.

Syntax `int` **`usergroup_size`**`(`*`group_name`*`)`

Syntax Element	Description
group_name	A string expression whose value is the name of the user group.

Comments The `usergroup_size` routine returns the number of members in a user group.

Example

In this example, five user groups are defined. The example prints out the number of members in each group.

```
#define MAX_GROUPS 5
{
    string groups[MAX_GROUPS] = {"Accountants", "Engineers",
       "DB Entry","Administration", "Operations"};
    int index, size;

    for (i = 0; i < MAX_GROUPS; i++)
    {
       index = usergroup_member(groups[i]);
       if (index)
       {
           size = usergroup_size(groups[i]);
          printf ("I am user number: %d in group: %s which has %d
               users", index, groups[i], size);,
index, groups[i], size);
           break;
       }
    }
}
```

See Also

usergroup_member

wait

Library Routine

Description

Blocks a virtual user from further execution until a user-defined global event occurs.

Syntax

int **wait** (&*sv*, *min* [, *max*, *adj*, *tmout*, &*retval*])

Syntax Element	Description
sv	A shared variable. wait considers an event to have occurred if the value of *sv* is greater than or equal to *min* and less than or equal to *max*. If *max* is not specified, *max* is assumed to equal *min*.
min	An integer expression that specifies the minimum value that the shared variable can have.
max	An integer expression. If omitted, it is assumed to equal *min*.
adj	An integer expression. The value of *adj* is added to the value of *sv*, if and when the event occurs. The adjustment is performed with the "unblocking" of the associated virtual user as a single atomic event. If you do not require an adjustment, but do need a placeholder argument because additional arguments need to be specified, set adj to 0.

(Continued)

Syntax Element	Description
tmout	An integer expression that controls the number of milliseconds `wait` waits for the event to occur. By default, `wait` does not return until the event occurs. If *tmout* equals zero, `wait` is nonblocking, and returns the value zero immediately if the event is false. If *tmout* is greater than zero, `wait` enforces a time out of *tmout* milliseconds, at which time if the event has not occurred, wait returns zero. If no time-out is desired, but *tmout* is required as a placeholder, set *tmout* to a negative value.
retval	A non-shared integer variable. If *retval* is specified, `wait` sets *retval* to the value of *sv* as follows: if `wait` returns 1, *retval* is set to the value of *sv* before the optional adjustment; if `wait` returns 0, *retval* is set to the value of *sv* when the timeout occurs.

Comments

The `wait` routine is an efficient method of blocking a virtual user until a user-defined global event occurs. `wait` returns 1 when the event has occurred; it returns 0 if the event has not yet occurred when the time specified by *tmout* has expired.

If virtual users are blocked on an event utilizing the same shared variable, and if the value of that shared variable is set to `TRUE` simultaneously, VU guarantees that the users are unblocked in the same order in which they were blocked. Although this *alone* does not ensure a deterministic multi-user timing order in which VU statements following a `wait` is executed,[1] the additional proper use of the `wait` arguments *min*, *max*, and *adj* allows control over the order in which multiuser operations occur.

If a shared variable's value is modified (by a VU assignment statement, autoincrement [*sv*++] operation, and so on), any subsequent attempt to modify this value — other than through `wait` — blocks execution until all virtual users already blocked on an event defined by *sv* have had an *opportunity* to unblock. This ensures that events cannot appear and then quickly disappear before a blocked virtual user is unblocked. For example, if two virtual users were blocked waiting for *sv* to equal or exceed *N*, and if another virtual user assigned the value *N* to *sv*, then VU guarantees both users the opportunity to unblock before any other virtual user is allowed to modify *sv*.

1. UNIX or Windows NT determines the order of the scheduling algorithms. For example, if two virtual users are unblocked from a wait in a given order, the user unblocked last may be allowed to execute its next VU statement before the user who unblocked first.

Offering the *opportunity* for all virtual users to unblock does not guarantee that all users actually unblock, because if `wait` had been called with a nonzero value of *`adj`* by one or more of the blocked users, the shared variable value would change during the unblocking script. In the previous example, if the first user to unblock *had* called `wait` with a negative *`adj`* value, the event waited on by the second user would no longer be true after the first user unblocked. With proper choice of *`adj`* values, you can control the order of events.

Example

This example blocks until the value of the shared variable `ev` equals 2, 3, or 4, and returns 1:

```
wait(&ev, 2, 4);
```

This example blocks until the value of the shared variable `ev` equals 0, and before returning the integer value 1, adjusts the value of `ev` to 1 (by adding 1 to its value of 0):

```
wait(&ev, 0, 0, 1);
```

This example blocks until the value of the shared variable `ev` is 1 (returning the integer 1), or until 10 seconds have elapsed (returning the integer 0):

```
wait(&ev, 1, 1, 0, 10000);
```

This example blocks until the value of the shared variable `ev` is 2, 3, 4, or 5, and before returning the integer value 1, assigns the value (2, 3, 4, or 5) to `ret`, and subtracts 10 from `ev`:

```
wait(&ev, 2, 5, -10, -1, &ret);
```

This example allows only one user to access a critical section of code. The `wait` routine blocks until `inuse` equals 0 (the initial value for all shared variables), and upon obtaining access, uses an *`adj`* value of 1 to lock out all other virtual users. Upon completion of the critical section, `inuse` is reset to zero to allow access to other virtual users (who are executing identical code segments). Recall that if virtual users are blocked concurrently, access is granted on a first-come, first-served basis.

```
shared inuse;
wait(&inuse, 0, 0, 1);
/* critical section of code */
inuse = 0;
```

Assume that an application is licensed for five users. This example sets the variable `inuse` so that no more than five people can log on at one time. As a user logs on, the value of `inuse` is decremented:

```
shared inuse;
wait(&inuse, 0, 4, 1);
/* critical section of code */
--inuse
```

Suppose that for stress testing purposes, all virtual users must submit a certain transaction sequence at once. In this example, each virtual user increments `nready` and proceeds when all virtual users are ready (`_nusers` contains the number of virtual users in the emulation session).

```
shared nready;
nready++;
wait(&nready, _nusers, _nusers);
/* Synchronized activity takes place here */
```

This example resynchronizes so that the same condition can be tested repeatedly:

```
shared ready_cnt, control;
for (attempts = 0; attempts < 100; attempts++) {
    ready_cnt++;
    if (_uid == 1) {
        wait(&ready_cnt, _nusers, _nusers, -(_nusers));
        control = 2;
    }
    else
        wait(&control, _uid, _uid, 1);
    /* Synchronized activity takes place here */
}
```

Suppose that all virtual users are required to take turns at executing a certain transaction in round-robin fashion, with no specific execution order. This example successively grants access to the critical section of code to virtual users 1 through *n* in ascending order of user ID (`_uid`). After the last virtual user has taken his turn, he resets `turn` to 0, allowing the next iteration to begin anew with user 1:

```
shared turn;
for (attempts = 1; attempts < 100; attempts++) {
    wait(&turn, _uid-1, _uid-1);
    /* critical section of code */
    if (_uid == _nusers)
        turn = 0;
    else
        turn++;
}
```

In the following example, you need to execute code in a specific order, but it is unrelated to ascending or descending order of user IDs. Ten virtual users are to perform a certain transaction repeatedly in the following arbitrary order: 5, 1, 2, 6, 3, 10, 4, 7, 9, 8. Stated in a different way, user 1 is second, user 2 is third, user 3 is fifth, user 4 is seventh, ... and user 10 is sixth.

The example successively grants access to the critical section of code to virtual users 5, 1, 2, 6, 3, 10, 4, 7, 9, and 8 successively. After the last user (user 8) has taken his turn, he resets `turn` to 0, allowing the next iteration to begin anew with the first virtual user (user 5).

```
shared turn;
int exec_order[10] = {2,3,5,7,1,4,8,10,9,6};
myturn = exec_order[_uid - 1];
lastturn = limitof(exec_order) + 1;

for (attempts = 0; attempts < 100; attempts++) {
    wait(&turn, myturn - 1, myturn - 1);

    /* Critical section of code */
    if (myturn == lastturn)
        turn = 0;
    else
        turn++;
}
```

See Also `sync_point`

while

Flow Control Statement

Description Repeatedly executes a VU statement.

Syntax

```
while (exp1)
            statement1;
```

Syntax Element	Description
exp1	The integer expression to evaluate.
statement1	A VU language statement or, if enclosed in braces, multiple VU language statements.

Comments The execution of the `while` loop occurs in the following steps:

1. *exp1* is evaluated.
2. If the value of *exp1* is not 0, *statement1* is executed. If the value of *exp1* is 0, execution of the `while` loop ends.
3. If the `while` loop execution has not ended, steps 1 and 2 are repeated.

Example In this example, the statements within the `while` loop execute until the `while` condition is false.

```
#include <VU.h>
while ((c = fgetc(infile_des)) != EOF)
if (c >= ‘A’ && c <= ‘Z’)
    fputc(c, outfile_des);
else
   {
    ungetc(c, infile_des);
    break;
}
```

See Also `for` `do-while`

Part I

Appendixes

▸▸▸ A P P E N D I X A

Jolt-Specific VU Functions

This chapter provides a general introduction to the Jolt protocol. It includes the following topics:

- Jolt overview
- PerformanceStudio/Jolt function overview
- PerformanceStudio/Jolt function reference

Jolt Overview

The following sections describe how PerformanceStudio supports the Jolt protocol.

BEA Jolt is a product that extends the BEA TUXEDO middleware framework to provide pure Java-based clients access to TUXEDO application services. This enhanced functionality is provided by a combination of a new set of Jolt classes on the client and some new Jolt system processes on the server.

Jolt clients (pure Java applications or applets) communicate with the Jolt system processes via the Jolt protocol. PerformanceStudio emulates Jolt client activity by reproducing the recorded native Jolt protocol messages originating from the client, effectively becoming a Jolt client from the Jolt server's perspective.

Jolt support is implemented with `sock_send` and `sock_nrecv` emulation commands. Therefore, it uses the same set of VU environment variables, timeouts, and so on, that the socket protocols use. Jolt, in effect, sits on top of socket.

PerformanceStudio models seven message types within the Jolt protocol:

Jolt Message Type	Usage
Authenticate/Challenge	session management
Authenticate/Ticket	session management
Check Authorization Level	session management
Close Connection	session management
Data Transfer	application service
Establish Connection	session management
Reconnect	session management

The Data Transfer message is the primary means of exchanging application data between the Jolt client and the Jolt server, hence it is called an **application service message**. The other messages, called **session control messages**, establish and maintain Jolt sessions. PerformanceStudio provides emulation functions that let you construct request messages and extract information from response messages of these types.

PerformanceStudio/Jolt Function Overview

PerformanceStudio provides a number of emulation functions that, with the `sock_send` and `sock_recv` emulation commands, can create virtual user scripts that communicate directly with Jolt application services using the native Jolt protocol.

The following sections describe the main classes of Jolt emulation functions.

Request Construction Functions

The request construction function class contains only one function, `jolt_request()`. This function builds a complete Jolt request that can then be sent to a Jolt server via `sock_send`. It requires the assistance of a Message Construction function to supply the body of the request.

Message Construction Functions

Message construction functions build the body of a Jolt request as required by `jolt_request()`. Each Jolt message type has a message construction function. Some of the functions require message parameters, others do not. Message construction functions contain two special subclasses:

- Attribute construction functions, which build attribute lists used by Application Service functions.
- Parameter construction functions, which build parameter lists that may accompany certain attributes.

Response Query Functions

The two primary response query functions are `jolt_response_header()` and `jolt_response_body()`. These functions interface with the `sock_recv` emulation command to retrieve response messages from the Jolt servers. A special subclass of response query functions extracts information from the received Jolt header.

Response Header Query Functions

Response Header Query functions extract specific Jolt message header variables from a Jolt response.

Message Query Functions

These functions, which complement the message construction functions, extract specific information from the body of Jolt responses. The two special subclasses of message query functions are:

- Attribute query functions, which extract specific attributes from a Jolt response.
- Parameter query functions, which extract specific parameters from an attribute.

In addition to the function classes listed above, the Jolt emulation functions are further classified into two functional areas, Jolt Session Control functions and Jolt Application Service functions. In general, for automatically generated virtual user scripts, you should be concerned only with Jolt Application Service functions. Jolt Session Control functions set up the environment in which the Application Service functions operate.

Session Control Functions

PerformanceStudio provides seven categories of session control functions. These establish and maintain working sessions between PerformanceStudio and Jolt Server Handlers (JSHs) during script playback. The following table lists each category and its corresponding VU function prefix:

Category	VU Function Prefix
Authenticate/Challenge	`jolt_challenge`
Check Authorization Level	`jolt_checkauth`
Close Connection	`jolt_close`
Establish Connection	`jolt_estcon`
Reconnect	`jolt_reconnect`
Authenticate/Ticket	`jolt_ticket`
Header Information	`jolt_header`

PerformanceStudio uses a number of session control functions to manage Jolt sessions. However since proper use of these functions is critical to the correct Jolt script playback, do not modify any PerformanceStudio-scripted session control function calls. Improper use of session control functions may result in fatal Jolt server failures.

Application Service Functions

Once a session is established, PerformanceStudio uses application service functions to communicate application data with the Jolt services. There are five categories of Application Service functions:

Category	VU Function Prefix
Data Transfer	`jolt_dataxfer`
Attribute Construction	`jolt_setatt`
Attribute Query	`jolt_getatt`
Parameter Construction	`jolt_setpar`
Parameter Query	`jolt_getpar`

The Data Transfer messages are the primary means of communicating with the Jolt server. A Data Transfer request message encapsulates all of the data that a specific Jolt service requires to execute. Likewise, a Data Transfer response message contains all of the result data that a Jolt service produces. The Data Transfer functions manage both message types.

A Data Transfer message may contain a list of name-value data components called attributes. In general, attributes have predefined meanings and supply information required by the Jolt system. Each attribute has a specific data type and a corresponding value. The attribute construction functions build attribute lists when constructing a request. The attribute query functions locate and extract specific attributes from messages.

One attribute, the data attribute, may also contain a list of name-value data components called parameters. Unlike attributes, parameters are user-defined and encapsulate data required by the Jolt services themselves. Like their attribute equivalents, the Parameter Construction functions build parameter lists for request construction, and the attribute query functions extract specific parameters from messages.

For details about the functions in each Application Service category, see *PerformanceStudio/Jolt Function Reference* on page A-8.

Request Construction

Building a Jolt request involves associating a number of construction functions together to create the correct raw octet sequence of the request message. The octet sequence is then passed to the `sock_send` emulation command, which, in turn, sends it to the Jolt server.

Associating Construction Functions

Construction functions are associated by passing the result of a construction producer function as an input parameter to a construction consumer function. Each construction consumer capable of associating a construction producer has an association parameter of a specific construction type. Only a construction producer function of the same construction type should be associated with a given association parameter construction type. The three construction types are Message, Attribute List, and Parameter List. The construction functions related to each type are described below.

The following table lists the construction consumer functions:

Construction Consumer Function	Association Parameter	Construction Type
`jolt_request()`	message	Message
`jolt_dataxfer()`	attribute_list	Attribute List
`jolt_setatt_data()`	parameter_list	Parameter List

The following table lists the construction producer functions:

Construction Type	Construction Producer Function

Message	`jolt_challenge()`
	`jolt_checkauth()`
	`jolt_close()`
	`jolt_dataxfer()`
	`jolt_estcon()`
	`jolt_reconnect()`
	`jolt_ticket`

Attribute List	See the Attribute List Construction functions.
Parameter List	See the Parameter List Construction functions.

Building Requests

The following steps show how to build a Jolt request:

1. Construct a message by calling one of the message construction functions. Each Jolt message type has its own construction function and may require one or more parameters. If you are constructing a data transfer request you may also need to call and associate the results of one or more attribute or parameter construction functions.

```
string msg;
..msg = jolt_dataxfer(sessionid, JOLT_CALL_RQST, attlst));
/* see 2.3.2.1. example for attlst construction */
```

2. Construct a Jolt request by associating the result of a message construction function with the request construction function `jolt_request()`.

```
string req;
...
req = jolt_request(0, sessionid, handlerid, 1, msg);
```

3. Pass the result of `jolt_request()` to the `sock_send` emulation function.

```
sock_send ["request1"] req;
```

You can combine these steps into one statement as follows:

```
sock_send
    jolt_request(0, sessionid, handlerid, 1,
        jolt_dataxfer(sessionid, JOLT_CALL_RQST,
        jolt_setatt_name("TRANSFER") +
        jolt_setatt_data(
        jolt_setpar_long(1, 309270) +
        jolt_setpar_long(2, 202463) +
        jolt_setpar_double("9500.00"))));
```

Building Attribute Lists and Parameter Lists

Attribute lists and parameter lists are built by combining the results of individual Attribute Construction and Parameter Construction functions with the VU string concatenation operator (+). For example:

```
string attlst;
string parlst;
...
/* create parameter list with two longs and a double */
parlst = jolt_setpar_long(1, 309270) +/* from account */
    jolt_setpar_long(2, 202463) +/* to account */
    jolt_setpar_double("9500.00");/* transfer amount */
/* create attribute list with the NAME and DATA attributes set */
attlst = jolt_setatt_name("TRANSFER") +/* TRANSFER service */
    jolt_setatt_data(parlst);/* parameter list */
```

Note that attributes can be placed within an attribute list in any order.

Likewise, the order of parameters within a list is not significant.

Response Query

Once a Jolt request has been successfully constructed and sent to the Jolt server, receiving and extracting information from the Jolt server response requires the use of the response query functions.

These functions operate in conjunction with the `sock_nrecv` emulation command to access the response data. Receiving the complete Jolt response is a two-stage process. First the Jolt header must be received using a `sock_nrecv`/ `jolt_response_header()` combination statement. For example:

```
sock_nrecv ["rsphdr1"] jolt_response_header();
```

Once this is successfully executed, the contents of the Jolt header may be accessed using the appropriate query functions. The second step is to receive the body of the Jolt response. This is done using a `sock_nrecv/jolt_response_body()` combination statement. For example:

```
sock_nrecv ["rspbod1"] jolt_response_body();
```

Once this is successfully executed, the contents of the response message, including attributes and parameters, may be accessed using the message query functions.

PerformanceStudio/Jolt Function Reference

You should not modify PerformanceStudio-scripted Session Control function calls. Therefore, only the Application Service functions of each function class are described below.

The format is:

< functional area and category (when applicable)>

< VU function prototype>

< function description>

Request Construction Functions

```
string jolt_request (int flags, int sessionid, int handlerid, int msgid,
string message)
```

`jolt_request()` is the top-level Jolt request construction function. The result is an asciified string containing a complete Jolt request that may be passed to the `sock_send` emulation command.

`flag` contains protocol mode information (usually 0).

`sessionid` is the JSH-assigned identifier of the current Jolt session. handlerid is the JSL-assigned handler identifier for the current session.

`msgid` is the incrementing per-session message sequence number of the current request.

`message` is the association parameter for the Message construction.

Message Construction Functions

Application Service (Data Transfer)

```
string jolt_dataxfer (int sessionid, int opcode, string attribute_list)
```

This is the construction function for Data Transfer messages. *sessionid* is the WSH-assigned identifier of the current Jolt session. `opcode` specifies the mode of operation of the current Data Transfer request operation. Valid opcodes are:

Opcode	Description
JOLT_CALL_RQST	TUXEDO `tpcall` primitive
JOLT_DEQUEUE_RQST	TUXEDO `tpdequeue` primitive
JOLT_CONNECT_RQST	TUXEDO `tpconnect` primitive
JOLT_SEND_RQST	TUXEDO `tpsend` primitive
JOLT_RECV_RQST	TUXEDO `tprecv` primitive
JOLT_DISCONNECT_RQST	TUXEDO `tpdiscon` primitive
JOLT_SUBSCRIBE_RQST	TUXEDO `tpsubscribe` primitive
JOLT_UNSUBSCRIBE_RQST	TUXEDO `tpunsubscribe` primitive
JOLT_NOTIFY_RQST	TUXEDO `tpnotify` primitive
JOLT_POST_RQST	TUXEDO `tppost` primitive
JOLT_UNSOL_RQST	n/a
JOLT_CHKUNSOL_RQST	n/a
JOLT_GETCONFIG_RQST	n/a
JOLT_LOGON_RQST	Jolt server logon
JOLT_LOGOFF_RQST	Jolt server logoff
JOLT_GETDEF_RQST	get Jolt Repository service definition
JOLT_GETDEFX_RQST	get Jolt Repository service definition

attribute_list is the association parameter for the Attribute List construction.

Attribute List Construction Functions

These functions construct the attribute list associated with the Data Transfer application service function `jolt_dataxfer()`. There is one construction function per attribute. The results of the functions may be tied together using the VU string concatenation operator (+) to form a complex attribute list.

The naming convention for the functions is `jolt_setatt_`*`attribute-name`*, where *`attribute-name`* is the name of the Jolt attribute constructed. The *`value`* argument, a VU language data type, will be mapped to the appropriate Jolt attribute data representation by the function.

Application Service (Attribute Construction)

string **jolt_setatt_appasswd** (string *value*)

string **jolt_setatt_authlevel** (int *value*)

string **jolt_setatt_clientdata** (int *value*)

string **jolt_setatt_corrid** (string *value*)

string **jolt_setatt_data** (string *parameter_list*)*

string **jolt_setatt_e_errno** (int *value*)

string **jolt_setatt_e_reason** (string *value*)

string **jolt_setatt_errno** (int *value*)

string **jolt_setatt_errorq** (string *value*)

string **jolt_setatt_event** (string *value*)

string **jolt_setatt_filter** (string *value*)

string **jolt_setatt_flags** (int *value*)

string **jolt_setatt_groupnm** (string *value*)

string **jolt_setatt_idle** (int *value*)

string **jolt_setatt_joltvers** (int *value*)

string **jolt_setatt_msgid** (string *value*)

string **jolt_setatt_name** (string *value*)

string **jolt_setatt_netmsgid** (int *value*)

string **jolt_setatt_numevents** (int *value*)

string **jolt_setatt_passwd** (string *value*)

string **jolt_setatt_priority** (int *value*)

string **jolt_setatt_reason** (string *value*)

string **jolt_setatt_replyq** (string *value*)

string **jolt_setatt_repname** (string *value*)

string **jolt_setatt_repnrecs** (int *value*) string **jolt_setatt_reppattern** (string *value*)

string **jolt_setatt_repvalue** (string *value*)

string **jolt_setatt_sid** (int *value*)

string **jolt_setatt_timeout** (int *value*)

string **jolt_setatt_tuxvers** (int *value*)

string **jolt_setatt_type** (int *value*)

string **jolt_setatt_username** (string *value*)

string **jolt_setatt_userrole** (string *value*)

string **jolt_setatt_version** (int *value*)

string **jolt_setatt_xid** (int *value*)

NOTE: The special attribute list construction function `jolt_setatt_data()` accepts a single parameter list construction (see below) in place of a VU scalar value as an argument.

Parameter List Construction Functions

These functions construct the parameter list associated with the Attribute List construction function `jolt_setatt_data()`. There is one construction function per parameter. The results of the functions may be tied together using the VU string concatenation operator (+) to form a complex parameter list.

The naming convention for the functions is `jolt_setpar_`*`parameter-name`*, where *`parameter-name`* is the name of the Jolt parameter constructed. *`fieldid`* is an identifier that uniquely identifies the parameter among other parameters within a list. The *`value`* argument, a VU language data type, will be mapped to the appropriate Jolt parameter data representation by the function. *`asciified-value`* is the asciified form of the parameter value. *`text-value`* is the textual representation of the floating point value (for example, “1.23”).

Application Service (Parameter Construction)

string **jolt_setpar_carray** (int *fieldid*, string *asciified-value*)

string **jolt_setpar_char** (int *fieldid*, int *value*)

string **jolt_setpar_double** (int *fieldid*, string *text-value*)

string **jolt_setpar_float** (int *fieldid*, string *text-value*) string **jolt_setpar_long** (int *fieldid*, int *value*)

string **jolt_setpar_short** (int *fieldid*, int *value*)

string **jolt_setpar_string** (int *fieldid*, string *value*)

Response Query Functions

The Response Query functions extract information from Jolt responses received by the client. All of the query functions, except the Parameter Query group, accept no arguments. They work implicitly with the VU `_response` read-only variable, which is set by the `sock_nrecv` emulation command. Therefore, within a script the Response Query functions must follow the `sock_nrecv` commands on which they operate.

There are two main functions in this class:

int jolt_response_header ()

This function must be passed as an argument to the `sock_nrecv` emulation command to prepare it to receive the header portion of a Jolt response. For example:

```
sock_nrecv ["header_1"] jolt_response_header();
```

This function must always precede its `jolt_response_body()` complement.

int jolt_response_body ()

This function must be passed as an argument to the `sock_nrecv` emulation command to prepare it to receive the body portion of a Jolt response.

```
sock_nrecv ["body_1"] jolt_response_body();
```

This function must always follow its `jolt_response_header()` complement.

Message Query Functions

These functions extract specific field values from the message body portion of the Jolt responses. The naming convention used for these functions is `jolt_`*`message-name_field-name`*, where *`message-name`* is the name of the message to be examined and *`field-name`* is the name of the field to be extracted.

Application Service (Data Transfer)

string jolt_dataxfer_attribute_list ()

Response Attribute Query Functions

These functions extract specific attribute values from Jolt Data Transfer response messages. The actual attribute value is mapped to an appropriate VU language data type as necessary. The naming convention for these functions is `jolt_getatt_`*`attribute-name`*, where *`attribute-name`* is the name of the attribute to extract.

Application Service (Attribute Query)

`string` **`jolt_getatt_appasswd`** `()`

`int` **`jolt_getatt_authlevel`** `()`

`int` **`jolt_getatt_clientdata`** `()`

`string` **`jolt_getatt_corrid`** `()`

`string` **`jolt_getatt_data`** `()`

`int` **`jolt_getatt_e_errno`** `()`

`string` **`jolt_getatt_e_reason`** `()`

`int` **`jolt_getatt_errno`** `()`

`string` **`jolt_getatt_errorq`** `()`

`string` **`jolt_getatt_event`** `()`

`string` **`jolt_getatt_filter`** `()`

`int` **`jolt_getatt_flags`** `()`

`string` **`jolt_getatt_groupnm`** `()`

`int` **`jolt_getatt_idle`** `()`

`int` **`jolt_getatt_joltvers`** `()`

`string` **`jolt_getatt_msgid`** `()`

`string` **`jolt_getatt_name`** `()`

`int` **`jolt_getatt_netmsgid`** `()`

`int` **`jolt_getatt_numevents`** `()`

`string` **`jolt_getatt_passwd`** `()`

`int` **`jolt_getatt_priority`** `()`

`string` **`jolt_getatt_reason`** `()`

`string` **`jolt_getatt_replyq`** `()`

string **jolt_getatt_repname** ()

int **jolt_getatt_repnrecs** ()

string **jolt_getatt_reppattern** ()

string **jolt_getatt_repvalue** ()

int **jolt_getatt_sid** ()

int **jolt_getatt_timeout** ()

int **jolt_getatt_tuxvers** ()

int **jolt_getatt_type** ()

string **jolt_getatt_username** ()

string **jolt_getatt_userrole** ()

int **jolt_getatt_version** ()

int **jolt_getatt_xid** ()

Response Parameter Query Functions

These functions extract specific parameter values from Jolt Data Transfer response messages. The actual parameter value will be mapped to an appropriate VU language data type as necessary. The naming convention for these functions is `jolt_getpar_`*`parameter-name`*, where *`parameter-name`* is the name of the parameter to extract. *`fieldid`* is the application-assigned identifier used to distinguish a particular parameter from a list of parameters.

Application Service (Parameter Query)

string **jolt_getpar_carray** (int *fieldid*)

int **jolt_getpar_char** (int *fieldid*)

string **jolt_getpar_double** (int *fieldid*)

string **jolt_getpar_float** (int *fieldid*)

int **jolt_getpar_long** (int *fieldid*)

int **jolt_getpar_short** (int *fieldid*)

string **jolt_getpar_string** (int *fieldid*)

▸▸▸ APPENDIX B

SAP-Specific VU Functions

If you have purchased a license to play back SAP protocol, and you record a session that accesses a SAP R/3 server, the script that you generate will contain VU functions that emulate SAP clients. This appendix lists the functions that the VU script can contain. The functions begin with the prefix `VuERP`.

This appendix divides SAP-specific VU functions into the following categories:

- Event Manipulation and Communication
- Event Structure Access
- Utilities

Because the VU functions serve as wrappers to the SAP GUILIB API, you need to be familiar with the GUILIB API. For information on the GUILIB API, consult your SAP documentation.

GUILIB uses the term **event** to mean a data representation of a particular SAP screen. The event data structure contains a complete description and instructions necessary for rendering the SAP screen. Therefore, in this appendix, the terms *event* and *screen* are synonymous.

The functions, properties, and fields defined in the GUILIB documentation are shown in ***bold italics***.

For information on testing SAP applications, see the following on-line manuals on the Documentation CD:

- *Rational LoadTest Try it! for Virtual User Testing of SAP Applications*
- *Rational Robot Try it! for GUI Testing of SAP Applications*

Event Manipulation and Communication

Each function in this section is invoked via the VU `emulate`() command. Therefore, all environment variables that affect the `emulate`() command also affect the execution of the functions in this section. Those functions with `Set` in their name set properties in the event or screen; those functions with `Send` in their name send the screen, or event, information to the SAP R/3 server.

Functions

`func` **`VuErpSetHeight`**`(`*`Height`*`) int` *`Height`*`; {}`

Sets the ***`screen.dimrow`*** field of the event. If *`Height`* is greater than 255, it is set to 255. If the event is a ***`modal screen`*** 0, the function returns 0. Otherwise it returns 1. A return of 0 indicates a failure since modal events/screens are not resizable.

`func` **`VuErpSetWidth`**`(`*`Width`*`) int` *`Width`*`; {}`

Sets the ***`screen.dimcol`*** field of the event. If *`Width`* is greater than 255, it is set to 255. If the event is a ***`modal screen`*** 0, the function returns 0. Otherwise it returns 1.

`func` **`VuErpSetHScroll`**`(`*`Pos`*`) int` *`Pos`*`; {}`

Sets the *`Pos`* field of the event and marks the event type with ***`MES_HSCROLL mask`***. This function always returns 1.

`func` **`VuErpSetVScroll`**`(`*`Pos`*`) int` *`Pos`*`; {}`

Sets the *`Pos`* field of the event and marks the event type with ***`MES_VSCROLL mask`***. This function always returns 1.

`func` **`VuErpSetCurPosByIndex`**`(`*`Index`*`) int` *`Index`*`; {}`

A wrapper for ***`ItEv_SetCurPosByCtrl`***(). Returns 0 if `ItEv_SetCurPosByCtrl` fails and 1 otherwise.

`func` **`VuErpSetCheck`**`(`*`Index`*`,ck) int long, ck; {}`

A wrapper for ***`ItEv_SetCheck`***(). Returns 0 if `ItEv_SetCheck` fails and 1 otherwise.

`func` **`VuErpSetMenuId`**`(`*`id`*`) int` *`id`*`; {}`

A wrapper for ***`ItEv_SetMenuID`***(). Returns 0 if `ItEv_SetMenuID` fails and 1 otherwise.

func **VuErpSetOkCode**(*okCode*) string *okCode*; {}

A wrapper for ***ItEv_SetOKCode***(). Returns 0 if `ItEv_SetOKCode` fails and 1 otherwise.

func **VuErpSetPfKey**(*KeyCode*) int *KeyCode*; {}

A wrapper for ***ItEv_SetPFKey***(). Returns 0 if `ItEv_SetPFKey` fails and 1 otherwise.

func **VuErpSetValue**(*Index*, *value*) int *Index*; string *value*; {}
A wrapper for ***ItEv_SetValue***(). Returns 0 if `ItEv_SetValue` fails and 1 otherwise.

func **VuErpSetValueDecrypt**(*Index*, *value*) int *Index*; string *value*; {}

A wrapper for ***ItEv_SetValue***() that decrypts the encrypted value. Returns 0 if `ItEv_SetValue` fails and 1 otherwise. By default, the user name and password are encrypted in a capture script and are decrypted with the `VuErpSetValueDecrypt`() function before being passed to `ItEv_SetValue`().

Users wishing to datapool unencrypted user names and passwords should replace the `VuErpSetValueDecrypt`() calls with `VuErpSetValue`(), i.e.:

Line from captured script (that uses a datapool with encrypted password):

```
emulate ["RatlErp_sun_exception_on001"]
VuErpSetValueDecrypt(5,

datapool_value(VuErp_DP, "RSYST_BCODE")),
VuErp_log_message;
```

Line from modified script (uses a datapool with unencrypted password):

```
emulate ["RatlErp_sun_exception_on001"]
datapool_value(VuErp_DP, "RSYST_BCODE"),
VuErp_log_message;
```

func **VuErpFreeConnection**() {}

A wrapper for ***It_FreeConnection***(). Returns 0 if `It_FreeConnection` fails and 1 otherwise.

func **VuErpFreeEvent**() {}

A wrapper for ***It_FreeEvent***(). Returns 0 if `It_FreeEvent` fails and 1 otherwise.

```
func VuErpGetEventEx(long flags) {}
```

A wrapper for ***It_GetEventEx***(). Returns 0 if `It_GetEventEx` fails and 1 otherwise.

```
func VuErpLogoff() {}
```

A wrapper for ***It_Logoff***(). Returns 0 if `It_Logoff` fails and 1 otherwise.

```
func VuErpNewConnection(Host,SystemNo,flags)
string Host, SystemNo; int flags; {}
```

A wrapper for ***It_NewConnection***(). Returns 0 if `It_NewConnection` fails and 1 otherwise.

```
func VuErpSendEvent() {}
```

A wrapper for ***It_SendEvent***(). Returns 0 if `It_SendEvent` fails and 1 otherwise.

```
func VuErpSendReturn() {}
```

A wrapper for ***It_SendReturn***(). Returns 0 if `It_SendReturn` fails and 1 otherwise.

```
func VuErpSetCtlVScroll(Index, pos) int Index, pos; {}
```

Set ***TabVerScrollbarStartRow*** field of the ***IT_TABLEINFO*** structure for the control indexed by *Index*. Returns 1 if successful and 0 otherwise.

Event Structure Access

Each function in this section is invoked via the VU Language `emulate`() command. Therefore, all environment variables that affect the `emulate`() command also affect the execution of the functions in this section. Each function attempts to get the value of an event or screen returned from the server. If the value is not assigned, each function continues to check the value until the value is assigned or `Timeout_val` is reached. (This is true for any function called by `emulate`()).

Functions

func **VuErpGetEventPtr**() {}

Returns a pointer to the current event structure. Returns a NULL if there is no valid event at the time of the call.

func **VuErpGetCtrlCnt**() {}

Returns ***screen.iCtrlCnt*** field of the event structure that indicates the number of controls present in the current event.

string func **VuErpGetCtrlName**(*Index*) int *Index*; {}

Returns the name of the control indexed by *Index*. If *Index* is invalid, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

string func **VuErpGetCtrlValue**(*Index*) int *Index*; {}
Returns a value of the control indexed by *Index*. If *Index* is invalid, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

string func **VuErpGetCtrlFieldName**(*Index*) int *Index*; {}

Returns a field name of the control — a ***szFieldName*** field of the IT_CTRL structure indexed by *Index*. If the field name is not available or *Index* is invalid, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

string func **VuErpGetScrnName**() {}

Returns a screen name of the event — a ***screen.szScreenName*** field of the event structure. If the screen name is not available, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

string func **VuErpGetProgName**() {}

Returns a program name of the event — a ***screen.szProgramName*** field of the event structure. If the program name is not available, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

```
string func VuErpGetEventMsg() {}
```

Returns a status message of the event — a ***szMessage*** field of the event structure. If the status message is not available, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

```
string func VuErpGetTitle() {}
```

Returns a title of the event — a ***szNormTitle*** field of the event structure. If the title is not available, an empty string is returned. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

Utilities

Each function in this section, except for `VuErp_VerifyEvent`(), is invoked via the VU `emulate`() command. Therefore,, all VU environment variables that affect the `emulate`() command also affect the execution of the functions in this section. Each function, except the last two functions (`VuErpDecrypt` and `VuErpEncrypt`), verifies that the value of a property of an event screen is the expected value. The last two functions either encrypt or decrypt a text string.

Functions

```
int func
VuErp_VerifyEvent(scrn,prog,title,msg,ctrlCnt,verifyScrn,
verifyMsg,verifyCnt)()string scrn,prog,title,msg;
int ctrlCnt,verifyScrn,verifyMsg,verifyCnt;
```

This function verifies that the screen (event) returned from the SAP server is the expected screen.

The verification is done by comparing the following five parameters of the `VuErp_VerifyEvent` function call with the corresponding event properties actually returned by the server:

`scrn`: Internal screen name as defined in Advanced Business Application Programming (ABAP).

`prog`: Internal program name as defined in ABAP

`title`: Screen title (caption)

`msg`: Message appearing in the status bar of the screen

`ctrlCnt`: Number of controls on the screen

Comparison of attributes can be turned off with the last three parameters of `VuErp_VerifyEvent`, as follows:

`verifyScrn`: If, and only if, the value of *`verifyScrn`* is 0, then *`scrn`*, *`prog`*, and *`title`* are not compared with the actual values returned by the server.

`verifyMsg`: If, and only if, the value of *`verifyMsg`* is 0, then *`msg`* are not compared with the actual value returned by the server.

`verifyCnt`: If, and only if, the value of *`verifyCnt`* is 0, then `verifyCnt` are not compared with the actual value returned by the server.

The default values for *`verifyScrn`*, *`verifyMsg`*, and *`verifyCnt`* (the variables, `VuErp_VerifyScreenInfo`, `VuErp_VerifyMessageLine`, and `VuErp_VerifyCtrlCount)` are defined as 1 by default. You can change the values of these variables or substitute another integer for the parameters *`verifyScrn`*, *`verifyMsg`*, and *`verifyCnt`*.

`VuErp_VerifyEvent` returns 1 if all compared parameters of the event returned from the server match all compared parameters of the expected event. If one or more compared parameters do not match, this function returns 0.

This function is added at capture time by the exception handler or by the user during script editing.

`VuErp_VerifyEvent`() is written in the VU Language and is contained in the file ~ Program Files\Rational\Rational Test 7\include\vuerp1.h.

`func` **`VuErpCompareScreenName`**`(`*`in`*`) string` *`in`*`; {}`

Compares the *`in`* string against the screen name of the event. The function returns 1 if strings are equal and 0 otherwise. If in is NULL, the function always returns 1.

`func` **`VuErpCompareProgramName`**`(`*`in`*`) string` *`in`*`; {}`

Compares the *`in`* string against the program name of the event. The function returns 1 if strings are equal and 0 otherwise. If *`in`* is NULL, the function always returns 1.

`func` **`VuErpCompareTitle`**`(`*`in`*`) string` *`in`*`; {}`

Compares the *`in`* string against the title of the event. The function returns 1 if strings are equal and 0 otherwise. If *`in`* is NULL, the function always returns 1.

`func` **`VuErpCompareMessage`**`(`*`in`*`) string` *`in`*`; {}`

Compares the in string against the status message of the event. The function returns 1 if strings are equal and 0 otherwise. If *`in`* is NULL, the function always returns 1.

```
func VuErpCompareEvent(title,scrn,prog,msg,ctrlCnt)
string title,scrn,prog,msg; long ctrlCnt; {}
```

This function combines the functionality of the previous four and also compares the number of controls. Just as for the previous functions, passing NULL for any string parameter causes the comparison of that parameter to always succeed. If *ctrlCnt* is -1, the controls count comparison always succeeds.

```
string func VuErpCrypt(char *str)
```

Returns an encrypted version of *str*. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

```
string func VuErpDecrypt(char *str)
```

Returns a decrypted version of *str*. The space allocated for the string is reused on each successive call. To preserve the return value, assign it to another VU string variable before calling this function again.

Glossary

action object – In TestFactory, an object in the application map that represents an action to which a control in the application responds. Typical actions are mouse left-click, mouse right-click, and mouse left-double-click; the corresponding action objects in the application map are LeftClick, RightClick, and LeftDoubleClick.

ActiveX control – A reusable software control that takes advantage of Object Linking and Embedding (OLE) and Component Object Modeling (COM) technologies. Developers can use ActiveX controls to add specialized functions to applications, software development tools, and Web pages. Robot can test ActiveX controls in applications.

actual results – In a functional test, the outcome of testing an object through a verification point in a GUI script. Actual results that vary from the recorded baseline results are defects or intentional changes in the application. See also *baseline results*.

Administrator – See *Rational Administrator*.

Agent computer – In LoadTest, a computer that has the Rational Agent software installed and that plays back a virtual user or GUI script. In a LoadTest schedule, you can identify the Agent computer on which to run a script. See also *Rational Agent*.

API recording – In Robot, a virtual user recording method that captures API calls between a specific client application and a server. These calls are captured on the client computer.

application map – In TestFactory, a hierarchical list of controls and actions in the application-under-test, as well as the states of the application-under-test and the transitions between those states. An application map can include UI objects and action objects, as well as TestFactory objects such as Pilots, Test Suites, and scripts.

application-under-test – The software being tested. See also *system-under-test*.

Asset Browser – A window that displays testing resources such as builds, queries, scripts, schedules, reports, report output, and logs. The Asset Browser is available in TestManager and LoadTest.

AUT – See *application-under-test*.

automated testing – A testing technique in which you use software tools to replace repetitive and error-prone manual work. Automated testing saves time and enables a reliable, predictable, and accurate testing process.

AutoPilot – In TestFactory, a tool for running scripts, Test Suites, and Pilots. The scripts and Test Suites can run on your local computer or on computers in the Test Lab. The Pilots run on your local computer, and the scripts they generate can run on your local computer or on computers in the Test Lab.

base state – In TestFactory, the known, stable state in which you expect the application-under-test to be at the start of each script segment. See also *script segment*.

baseline results – In a functional test, the outcome of testing an object through a verification point in a GUI script. The baseline results become the expected state of the object during playback of the script. Actual test results that vary from the baseline results are defects or intentional changes in the application. See also *actual results*.

best script – In TestFactory, an optimized script generated by a Pilot. A best script contains the fewest number of script segments that provide the most coverage of the source code or user interface in the application-under-test.

breakpoint – A feature of the Robot debugger. When you assign a breakpoint to a line of code, and then run the script in the debugger environment, the script stops executing at that line of code. Control returns to you, and the breakpoint line is displayed. From here you can view variables, perform other debugging activities, and continue executing the script.

build – A version of the application-under-test. Typically, developers add new features or enhancements to each incremental build. As team members test a build, they enter defects against those features that do not behave as expected. You use TestManager to define and manage builds.

built-in data test – A data test that comes with Robot and is used with the Object Data verification point. A data test uses a specific property of the object, in conjunction with other parameters, to determine the data to capture. Although built-in data tests cannot be edited, renamed, or deleted, they can be copied and then edited, and they can be viewed. See also *custom data test*.

ClearQuest – See *Rational ClearQuest*.

client/server – An architecture for cooperative processing in which the software tasks are split between server tasks and client tasks. The client computer sends requests to the server, and the server responds.

code coverage – In TestFactory, the percentage of code that is tested by a script. This percentage is based on the portion of the code that a script touches, relative to all code in the application-under-test. A Pilot can use code coverage to determine the best script for a run. See also *UI coverage*.

command ID – In LoadTest's VU language, an identifier for a command. Robot automatically assigns a unique command ID, composed of an alphanumeric prefix and a three-digit number, to each emulation command. Because command IDs appear in both the virtual user script and the LoadTest report output, they enable you to determine the relationship between an emulation command and its response times.

command ID prefix – In LoadTest, a prefix for a unique emulation command ID. The prefix defaults to the script name (up to the first seven characters). However, you can define the prefix in the Generator tab of the Virtual User Record Options dialog box.

custom data test – A customer-defined data test used with the Object Data verification point. A data test uses a specific property of the object, in conjunction with other parameters, to determine the data to capture. Custom data tests are created within your organization and are stored in the repositories that were active when they were created. They can be edited, renamed, and deleted. See also *built-in data test*.

data test – A test that captures the data of an object with the Object Data verification point. See also *built-in data test* and *custom data test*.

datapool – A source of test data that GUI scripts and virtual user scripts can draw from during playback. You can automatically generate datapools using TestManager, or you can import datapool data from other sources such as your database.

dependency – In LoadTest, a method of coordinating an object in a schedule with an event. For example, if the script Query is dependent upon the script Connect, then Connect must finish executing before Query can begin executing. See also *event*.

distributed architecture – Architecture in which computer systems work together and communicate with each other across LAN, WAN, or other types of networks. A client/server system is an example of distributed architecture.

distributed functional test – In LoadTest, a test that uses multiple Agent computers to execute multiple GUI scripts written in the SQABasic language.

dynamic load balancing selector – A type of selector in a LoadTest schedule. Items in the selector, such as scripts, are executed according to a weight that you set.

emulation commands – VU language statements or commands that emulate client activity, evaluate the server's responses, and perform communication and timing operations. LoadTest stores the results of emulation commands in a log file, which you can view from the LogViewer.

emulation functions – VU language functions that emulate client activity and evaluate the server's responses. Unlike emulation commands, emulation functions do not perform communication and timing operations, and they are not logged.

environment control commands – VU language commands that let you control a virtual user's environment by changing the VU environment variables. For example, you can set the level of detail that is logged or the number of times that virtual users attempt to connect to a server.

event – An item in a LoadTest schedule upon which another item is dependent. For example, if the script Connect sets an event and the script Query depends on this event, Connect must finish executing before Query can begin executing. See also *dependency*.

external script – A script that runs a program created with any tool. You plan and run external scripts in TestManager.

fixed user group – In LoadTest, a group that contains a scalable number of users. When you create a fixed user group, you indicate the maximum number of users that you will run in the group. Typically, you use fixed user groups in functional tests, which do not add a workload to the system.

flow control statements – In the VU and SQABasic languages, statements that let you add conditional execution structures and looping structures to a script.

functional test – A test to determine whether a system functions as intended. Functional tests are performed on GUI objects and objects such as hidden DataWindows and Visual Basic hidden controls.

Grid Comparator – The Robot component for reviewing, analyzing, and editing data files for text and numeric verification points in grid formats. The Grid Comparator displays the differences between the recorded baseline data and the actual data captured during playback.

GUI script – A type of script written in the SQABasic language. It contains GUI actions such as keystrokes and mouse clicks. Typically, a GUI script also contains verification points for testing objects over successive builds of the application-under-test.

GUI user – The type of user that is emulated when a GUI script is executed. Only one GUI user at a time can run on a computer.

hidden object – An object that is not visible through the user interface. Hidden objects include objects with a visible property of False and objects with no GUI component.

IDE – Integrated Development Environment. This environment consists of a set of integrated tools that are used to develop a software application. Examples of IDEs supported by Robot include Oracle Forms, PowerBuilder, Visual Basic, and Java.

Image Comparator – The Robot component for reviewing and analyzing bitmap image files for Region Image and Window Image verification points. The Image Comparator displays differences between the recorded baseline image and the actual image captured during playback. The Image Comparator also displays unexpected active windows that appear during playback.

instrumentation – In TestFactory, the process of inserting code coverage counters into the application-under-test. These counters record how much code is executed during a script run. See also *object code instrumentation* and *source code instrumentation*.

load – See *workload*.

load balancing – See *workload balancing*.

LoadTest – See *Rational LoadTest*.

log – A repository object that contains the record of events that occur while playing back a script or running a schedule. A log includes the results of all verification points executed as well as performance data that can be used to analyze the system's performance.

LogViewer – See *Rational LogViewer*.

low-level recording – A recording mode that uses detailed mouse movements and keyboard actions to track screen coordinates and exact timing. During playback, all actions occur in real time, exactly as recorded.

manual script – A set of testing instructions to be run by a human tester. The script can consist of steps and verification points. You create manual scripts in TestManager.

Master computer – A computer that executes LoadTest. From this computer, you create, run, and monitor schedules. When the run is finished, you use it to analyze test results.

mix-ins – See *Pilot mix-ins*.

network recording – In Robot, a virtual user recording method that records packet-level traffic. This traffic is captured on the wire.

next available selector – In LoadTest schedules, a selector that distributes each item such as a script, delay, or other selector to an available computer or virtual user. This type of selector is used in a GUI schedule. The next available selector parcels out the items sequentially, based on which computers or virtual users are available.

object – An item on a screen, such as a window, dialog box, check box, label, or command button. An object has information (properties) associated with it and actions that can be performed on it. For example, information associated with the window object includes its type and size, and actions include clicking and scrolling. In some development environments, a term other than *object* is used. For example, the Java environment uses *component*, and the HTML environment uses *element*.

object code instrumentation – In TestFactory, the process of inserting code coverage counters into the executable file of the application-under-test. These counters record how much of the program a script tests. See also *instrumentation* and *source code instrumentation*.

Object-Oriented Recording® – A script recording mode that examines objects in the application-under-test at the Windows layer. Robot uses internal object names to identify objects, instead of using mouse movements or absolute screen coordinates.

Object Properties Comparator – The Robot component that you use to review, analyze, and edit the properties of objects captured by an Object Properties verification point. The Object Properties Comparator displays differences between recorded baseline data and the actual data captured during playback.

Object Scripting commands – A set of SQABasic commands for accessing an application's objects and object properties. You add Object Scripting commands manually when editing a script.

Object Testing® – A technology used by Robot to test any object in the application-under-test, including the object's properties and data. Object Testing lets you test standard Windows objects and IDE-specific objects, whether they are visible in the interface or hidden.

OCI – Object Code Insertion. The Rational technology used in TestFactory to instrument object code and measure how much of the application-under-test a script tests. See also *code coverage* and *object code instrumentation*.

performance test – A test that determines whether a multi-client system performs within user-defined standards under varying loads. Performance tests are always run from a schedule in LoadTest.

Pilot – In TestFactory, a tool for generating scripts automatically.

Pilot mix-ins – In TestFactory, a list of Pilots that are executed on a random basis during the run of a lead Pilot. Mix-ins are useful for randomly testing multiple areas of the application-under-test. To make tests more realistic, you can combine mix-ins and scenarios.

Pilot scenario – An ordered list of Pilots that are executed during the run of a Pilot. A Pilot scenario is useful for testing UI objects that need to be exercised in a specific order. To make tests more realistic, you can combine scenarios and mix-ins.

project – A collection of data, including test assets, defects, requirements, and models, that can facilitate the development and testing of one or more software components.

proxy recording – In Robot, a virtual user recording method that captures the client/server conversation on the network wire rather than on the client computer. Proxy recording allows Robot to capture network packets that are not visible to it during network recording — for example, if the client and server are in different network segments.

query – A request for information stored in the repository. A query consists of a filter and several visible attributes — the columns of data to display, the width of the column, and the sort order.

random selector – A type of selector in a LoadTest schedule. Items in the selector, such as scripts, are randomly executed. Random selectors can be with replacement, where the odds are the same, or without replacement, where the odds change with each iteration.

Rational Administrator – The component for creating and maintaining repositories, projects, users, groups, computers, and SQL Anywhere servers.

Rational Agent – The LoadTest software that resides on a shared network drive and runs on each computer where testing occurs. The entries specified in a schedule play back on the Agent computer, which reports on their progress and status as they run. See also *Agent computer*.

Rational ClearQuest – The Rational product for tracking and managing defects and change requests throughout the development process. With ClearQuest, you can manage every type of change activity associated with software development, including enhancement requests, defect reports, and documentation modifications.

Rational LoadTest – The Rational Test component for running performance, stress, scalability, multi-user, and distributed functional tests on multiple Agents connected by a network. With LoadTest, you can initiate test runs and monitor tests from a master computer that manages the test process. LoadTest is available only in Rational Suite PerformanceStudio.

Rational LogViewer – The Robot component for displaying logs, which contain the record of events that occur while playing back a script or running a schedule. Also, the component from which you start the four Comparators.

Rational PerformanceArchitect – The Rational component that lets you test the performance of COM/DCOM applications. With Rational PerformanceArchitect, you can create a Rose sequence or collaboration diagram, convert it to a virtual user script, and then use Rational Suite PerformanceStudio to edit the script and run the performance tests.

Rational repository – A database that stores application testing information, such as test requirements, scripts, and logs. All Rational Suite TestStudio and Rational Suite PerformanceStudio products and components on your computer update and retrieve data from the same connected repository. A repository can contain either a Microsoft Access or a Sybase SQL Anywhere database.

Rational RequisitePro – The Rational product for organizing, managing, and tracking the changing requirements of your system.

Rational Robot – The Rational product for recording, playing back, debugging, and editing scripts.

Rational SiteCheck – The Robot component for managing your intranet or World Wide Web site. You can use SiteCheck to visualize the structure of your Web site, and you can use it with Robot to automate Web site testing.

Rational Synchronizer – The Rational tool that ensures the consistency of data across several Rational products.

Rational TestAccelerator – An agent application that executes scripts. TestFactory uses computers running TestAccelerator as remote machines on which to run automated distributed tests.

Rational TestFactory – The Rational Test component for mapping an application-under-test and generating scripts automatically. TestFactory is available in Rational Suite TestStudio and Rational Suite PerformanceStudio.

Rational TestManager – The Robot component for managing the overall testing effort. You use it to define and store information about test documents, requirements, scripts, schedules, and sessions.

Report Layout Editor – The TestManager component for customizing the layout of reports.

repository – See *Rational repository*.

RequisitePro – See *Rational RequisitePro*.

Robot – See *Rational Robot*.

scalable user group – In LoadTest, a group that contains a varying number of users. When you create a scalable user group, you assign it a percentage of the total workload. Assume you have a scalable user group that is 50 percent of the workload. If you run a test with 10 users, the group will contain 5 users. If you run a test with 100 users, the group will contain 50 users.

scenario – In LoadTest, a modular group of scripts and other items in a schedule that is used by more than one user group. A scenario can contain scripts, delays, and synchronization points.

scenario – See *Pilot scenario*.

schedule – In LoadTest, structure that you create to specify how scripts should be played back. A schedule can contain GUI scripts and virtual user scripts, and can indicate the number of times to repeat a script and the computer on which the script will run. In performance testing, a schedule is used to create a workload. In distributed functional testing, a schedule is used to distribute scripts among various computers.

script – A set of instructions used to navigate through and test an application. You can generate scripts in a variety of ways. You can use Robot to record scripts used in functional testing and performance testing. You can also use TestManager to create and manage manual scripts, and to manage external scripts created with a third-party testing tool. A script can have properties associated with it, such as the purpose of the script and requirements for the script. See also *external script*, *GUI script*, *manual script*, and *virtual user script*.

script outline – In TestFactory, the readable version of a script. A script outline contains a description of the actions that Robot performs while running the script.

script segment – In TestFactory, a section of a script that tests a particular element of product functionality. A Pilot generates a script segment by starting the application-under-test in a base state, navigating through the part of the product that you are testing, and returning the application-under-test to the base state. See also *base state*.

seed – An initial number fed to a random number generator. Using the same seed produces the same series of random numbers. In LoadTest, you use seeds to generate think times.

selector – An item that you insert in a LoadTest schedule to indicate how often and in what order to run scripts.

sequential selector – In a LoadTest schedule, a type of selector that executes each script, delay, or other item in the same order in which it appears in the schedule.

session – In virtual user recording, one or more scripts that you record from the time you begin recording until the time you stop recording. Typically, the scripts in a session represent a logical flow of tasks for a particular user, with each script representing one task. For example, a session could be made up of three scripts: *login*, *testing*, and *logout*. In TestFactory, a session is the period of time that the TestFactory application or a window is open.

shared variable – An integer variable that multiple scripts and multiple virtual users can read and write to. You can see the value of a shared variable while monitoring a LoadTest schedule. For example, you can set a shared variable as a flag to end a playback session. Each script can check the flag to see if the session should end. When that flag is set, exit tasks can be performed.

shell script – A script that calls or groups several other GUI scripts and plays them back in sequence. Shell scripts provide the ability to create comprehensive tests and then store the results in a single log.

SiteCheck – See *Rational SiteCheck*.

source code instrumentation – In TestFactory, the process of inserting code into the source code of the application-under-test. This code measures how much of the source code a script tests. See also *instrumentation* and *object code instrumentation*.

SQABasic – The Robot scripting language for recording GUI actions and verifying GUI objects. SQABasic contains most of the syntax rules and core commands that are contained in the Microsoft Basic language. In addition, SQABasic has commands that are specifically designed for automated testing. See also *VU*.

stable load – In LoadTest, a condition that occurs when a specified number of virtual users have logged on to the system-under-test and are active. When the stable load criterion is met, LoadTest begins measuring the load.

streak – When running a virtual user schedule in LoadTest, a series of successes or failures for emulation commands. You can see a streak while monitoring a schedule.

structural test – A test to determine whether the structure of a Web site is consistent and complete. A structural test ensures that an application's interdependent objects are properly linked together. You perform a structural test using SiteCheck.

synchronization point – In LoadTest, a place where emulated virtual users stop and wait until all other synchronized users reach that point. When all users reach the synchronization point, they are released and continue executing.

Synchronizer – See *Rational Synchronizer*.

system tuning – In LoadTest, the process of optimizing a system's performance by changing hardware resources and software configuration parameters while using a constant workload.

system-under-test – The system being tested. This includes the computers and any software that can generate a load on the system, networks, user interfaces, CPUs, and memory. See also *application-under-test*.

test assets – The resources that facilitate the planning or development phases of the testing effort. Examples of test assets include scripts, schedules, sessions, test documents, and test requirements.

test development – The process of developing tests to verify the operation of a software application. This includes creating scripts that verify that the application-under-test functions properly. Test development lets you establish the baseline of expected behavior for the application-under-test.

test documents – Test plans, project schedules, resource requirements, and any other documents that are important to your project. You develop your test documents using your own word processing or scheduling program; you then reference the name and location of the document in TestManager. This lets members of the test and development team locate documents quickly.

Test Lab – A collection of computers on which TestAccelerator is running. In TestFactory, you can distribute the scripts associated with a Pilot, a Test Suite, or the AutoPilot to run on computers in the Test Lab. See also *Rational TestAccelerator*.

Test Suite – In TestFactory, a tool for running a collection of scripts as a group.

TestAccelerator – See *Rational TestAccelerator*.

TestFactory – See *Rational TestFactory*.

TestManager – See *Rational TestManager*.

Text Comparator – The Robot component for reviewing, analyzing, and editing data files for text and numeric verification points in any format except grids. The Text Comparator displays the differences between the recorded baseline results and the actual results.

think time – In virtual user and GUI scripts, think times are delays that simulate a user's pauses to type or think while using an application. With virtual user scripts, LoadTest calculates the think time at runtime, based on think time VU environment variables that are set in the script. You can set a maximum think time in Robot. With GUI scripts, Robot uses the actual delays captured between keystrokes, menu choices, and other actions.

transaction – In LoadTest, a logical unit of work performed against a server. For example, submitting a search query or submitting a completed form to a Web server are both transactions.

transaction rate – In LoadTest, the playback speed calculated as a function of number of transactions per unit of time. For example, if a script contains one transaction, and each script is started at half-second intervals, your transaction rate would be 2 per second.

transactor – In LoadTest, an item that you insert in a LoadTest schedule to indicate the number of user-defined transactions that a virtual user performs in a given time period.

UI coverage – In TestFactory, the percentage of objects in the application map that are tested by a Pilot-generated script. This percentage is the proportion of UI objects that the script touches, relative to all UI objects available to the Pilot. A Pilot can use UI coverage to determine the best script for a run. See also *code coverage*.

UI object properties – Attributes of object classes and UI objects that TestFactory uses to map applications and generate scripts.

unexpected active window – A window that appears during script playback that interrupts the script playback process and prevents the expected window from being active. For example, an error message generated by the application-under-test is an unexpected active window. You can view unexpected active windows in the Image Comparator.

user group – In LoadTest, a collection of users that execute similar tasks and generate the same basic workload. Accountants and data entry operators are examples of user groups.

verification – The process of comparing the test results from the current build of the software to its baseline results.

verification point – A point in an SQABasic script that confirms the state of one or more objects. During recording, a verification point captures object information from the application-under-test and stores it as the baseline. During playback, a verification point recaptures the object information and compares it to the baseline. In a manual script, a verification point is a question about the state of the application-under-test.

virtual user – In LoadTest, a type of user that is emulated when a virtual user script is executed. A computer can run multiple virtual users simultaneously.

virtual user script – A type of script written in the VU language. Virtual user scripts contain client/server requests and responses as well as user think times.

VU – The Robot scripting language for recording a client's requests to a server. VU provides most of the syntax rules and core commands available in the C programming language. In addition, VU has emulation commands and functions that are specifically designed for automated performance testing. See also *SQABasic*.

wait state – A delay or timing condition that handles time-dependent activities.

workload – In LoadTest, the set of all activities that users perform in an actual production setting of the system-under-test. You can use LoadTest to emulate a workload.

workload balancing – In LoadTest, the act of distributing activities so no one system or device becomes a bottleneck.

workload model – In LoadTest, the workload model is represented as a schedule. You can play back this schedule and analyze the response times.

▸▸▸ Index

D

E

M

N

O

P

Q

R

U

V

W