OpenSTA Documentation

Index of available Guides, Documents and References

Download

Zip Archives

User Guide
>> **HTTP**, **FTP**

Prd. Monitoring
>> **HTTP**, **FTP**

SCL Ref(old)
>> **HTTP**, **FTP**

GS. Guide
>> **HTTP**, **FTP**

A Warning!

Out of date documentation

Most of the documentation listed below is older than that installed with the product in the form of Microsoft HTMLHelp documents. The advice is to first check the Help installed with the product - accessible from the Help menu of the Modeler and the Commander. Use the documents here only if you really need the text in HTML format. The exception to this rule is the SCL Reference which is now more up to date than the product help.

If you are here looking for **PDF documentation** then you will unfortunately be disappointed. The current documentation was provided by **CYRANO SA** specifically licensed so that only they could provide fixed nicely formatted versions. Hardcopy and PDF versions of these manuals were made available, before the dissolution of CYRANO SA, but are no longer current. If you still wish to get copies of these then please ask on the **Users List**, as there are still some floating around.

We're working towards getting both of these issues resolved by completely rewriting the documentation in **DocBook XML**.

User Guide

OpenSTA General UG

This document is entitled *HTTP/S Load User's Guide*. It was named this way because it was intended to document the parts of OpenSTA that are used for producing load tests of HTTP and HTTPS servers. There are chapters on all aspects of OpenSTA relevent to this, from recording your scripts and modeling, then through test runs and finishing with some analysis of the results.

This document, in its most up to date form, is available from the OpenSTA Commander and Modeler Utilities under the *Help>Contents* menu items.

The **Online HTML version of the User Gude** is a little out of date but **available here** anyway.

Production

Prd. Monitoring Guide

This document covers the parts of OpenSTA that allow you to monitor and collect statistics about your whole system performance using NT Performance Monitor and SNMP facilities. These topics are also covered in the UG but from a slightly different perspective.

The **Online HTML version of the Production Monitoring Guide** is **available here**.

SCL Reference

Script Control Language

This document is a reference of the structure, syntax, commands and features of the version of the SCL language produced by HTTP recording and used for HTTP playback within OpenSTA.

The **Online HTML version of the SCL Reference is available here**.

An older **French Translation** of this document was donated. It's only **available here online**.

The GSG
Getting Started Guide

A short tutorial taking the user through the basics of using OpenSTA for load testing a simple dynamic Web application. Unfortunately, not really up to date with the current version of the toolset, screenshots don't match, some instructions are incorrect, add to this that the section on modeling mysteriously ends mid instructions... may still be of some use for beginners though. This guide uses a simple Web application that we wrote specifically for use in this introduction to OpenSTA, you can find details about this on the **Demosite page**.

The **Online HTML version of the Getting Started Guide** is **available here**.

Community FAQ
Frequently Asked Questions

This is a live document of contributed information covering all aspects of OpenSTA's use. The document is stored in a **Wiki** that anyone with a, freely available, **Community Portal** user account can edit. Users are encouraged to contribute questions, answers, hints, tips and comments with the purpose of helping other users and strengthening the OpenSTA community.

The FAQ is only accessible at **http://portal.opensta.org/faq.php**, as it is constantly changing you should bookmark this and check back often.

---

*Proud to be **Open**, prouder to be **Free***

Questions, Comments, Suggestions?

Last Updated: 2005-MAR-28

OpenSTA User Home

For those Researching, Learning and Using OpenSTA

News

2007-OCT-19: OpenSTA 1.4.4 released
2005-JUN-09: BView 1.0.3 released
2005-MAY-12: OpenSTA 1.4.3 released
2005-MAR-28: SCL Reference Rewritten
2004-SEP-07: BView gets an update and new home
2003-DEC-26: OpenSTA.org moves home and gets a refresh
2003-MAY-01: OpenSTA 1.4.2 released

What is OpenSTA?

Open, Systems Testing Architecture

OpenSTA is a distributed software testing architecture designed around CORBA, it was originally developed to be commercial software by **CYRANO**. The current toolset has the capability of performing scripted HTTP and HTTPS heavy load tests with performance measurements from Win32 platforms. However, the architectural design means it could be capable of much more.

Web Load Testing

HTTP Stress & Performance Tests

The applications that make up the current OpenSTA toolset were designed to be used by performance testing consultants or other technically proficient individuals. This means testing is performed using the record and replay metaphor common in most other similar commercially available toolsets. Recordings are made in the tester's own browser producing simple scripts that can be edited and controlled with a special high level scripting language. These scripted sessions can then be played back to simulate many users by a high performance load generation engine. Using this methodology a user can generate realistic heavy loads simulating the activity of hundreds to thousands of virtual users.

Data Collection

Timers, Windows Performance & SNMP Statistics

Results and statistics are collected during test runs by a variety of automatic and user controlled mechanisms. These can include scripted timers, SNMP data, Windows Performance Monitor stats and HTTP results & timings. Much of the data logged can be monitored

live during the test runs; once test runs are complete, logs can be viewed, graphed, filtered and exported for use by more sophisticated report generation software.

Completely Free & Open Source

The OpenSTA toolset is Open Source software licensed under the GNU GPL (General Public License), this means it is free and will always remain free. If you wish to build your own customized version of OpenSTA or take part in its development then the complete toolset source code, buildable in Microsoft Visual Studio 6, and all related information is available from **OpenSTA. SourceForge.net**, the developer home site.

Community Supported

Development Driven by the Users

Much more information can be found out about OpenSTA by checking the **online documentation** or simply **downloading and installing** the toolset. The **FAQ** contains lots of other useful background information and helpful tips, this should be the first place you look if you need help with anything not covered in the documentation. There is no need to stop at reading the FAQ either, it is hosted on the **OpenSTA Community Portal** and, in common with every other resource on this site, it is user editable. This site is a great place for every OpenSTA user to share their experiences with the product and help others learn and use OpenSTA: Remember, the toolset is completely free and any time the developers spend helping users is time they are not enhancing, or fixing problems with, the toolset. By helping other users you are in fact helping OpenSTA and its community become stronger. The premier place for free OpenSTA support and discussions is the **OpenSTA Users Mailing List**, here the developers and many long time users of this toolset give as much help as their freetime will allow.

hosting donated by tcNOW.com

---

*Proud to be **Open**, prouder to be **Free***

Questions, Comments, Suggestions?

Last Updated: 2007-OCT-20

OpenSTA Product Download

Getting and Installing OpenSTA

Latest Stable

Download Installable

1.4.4 Here

Latest Unstable

No current unstable release. Next unstable release 1.5.0

Addons

Extend OpenSTA's functionality with contributed addons

Prerequisites

What You Need First

The OpenSTA Windows installation we are producing uses the *Windows Installer* mechanism that is part of Windows 2000. Installation under NT works using a Microsoft update that will normally be already installed. Instructions given below show how to get the the update if it has not already been installed. The Windows Installer mechanism packages a toolsets binaries and installation instructions in a file with an MSI extension.

To install OpenSTA you must have the following Microsoft Windows configuration:

- **either,** Microsoft Windows NT 4.0 updated with at least service pack 5.
  The OpenSTA installation process requires at least version 1.1 of the Windows Installer for NT, this is not part of the basic installation of NT4, an up to date version may be **downloaded from Microsoft** and installed if required.
  Your installation of NT must also have an up-to-date HTML Help sytem, the update package may be **downloaded from Microsoft**.
- **or,** Microsoft Windows 2000/XP (NT5) or later.

This product also requires a version 2.5 (or later) of Microsoft Data Access Components (MDAC). This may be **downloaded from Microsoft** if you do not have an **up to date version**.

Version Numbers

What They Mean

The release version numbers for OpenSTA will go by the common convention of three integers seperated by dots, of the form ***M.m.P*** - Where:

- ***M*** - is a decimal number representing the major version number of the release.
- ***m*** - is a decimal number representing the minor version number of the release. Odd numbers represent developer/unstable releases.
- ***P*** - is a decimal number representing the patch, or fix, level of the release.

Unstable (or developer) versions are intended for experimental features and have no guarantees of compatibility between patch releases. Defaults may change, features will be added, functionality could be modified.

Stable releases should always be compatible between their patch releases. Only bug fixes will be included in the patch releases of a stable version.

Old Versions
and alternate download

For fast downloads from one of many international mirrors all our distributables are now stored in our **SourceForge Files Area**. If you can't find what you are looking for there, or are having any issues with downloading then **ftp.opensta.org** is fully up to date and contains all the files we have ever made available.

Source Code
To Build Your Own

All information related to building your own copy of OpenSTA, including the source code itself, is referenced from the **OpenSTA developer site hosted by SourceForge**.

---

*Proud to be **Open**, prouder to be **Free***

Questions, Comments, Suggestions?

Last Updated: 2007-Oct-20

OpenSTA User Contacts

Getting in touch with others related to OpenSTA

Bugs

Don't email anyone
Instead **follow these instructions in the FAQ**.

Docs

docs@opensta.org
For suggestions and corrections to the product documentation.

Webmaster

webmaster@opensta.org
Please **only** use regarding spelling and grammer mistakes, bad links, etc. on **this** Web site.

Need Help?

OpenSTA Support

If you are having issues learning or using OpenSTA at any level then the best suggestion we can give you is to thoroughly **read the FAQ** and if you still can't find the answer there then send email to the **Users Mailing List**. Contacting any specific developer or project member directly is generally discouraged, we are all on the mailing lists and contribute as much as we have time to. Get to know us, and let us get to know you on these lists before sending us any personal mail, Thanks.

If you are really desperate for help and are willing to spend some money, then **here's a shortcut to the FAQ entry about purchasing support**.

Commercial

Spending Money

There is nothing for sale here! Although if you are desperate to give money to help out the project, then we are in the process of working out a good way to be able to donate money in a way that can measurably help everyone involved. Please stay tuned to the **Users Mailing List** in the coming months for more information.

There are companies out there who can sell you OpenSTA related products and services, we encourage these companies to **edit these FAQ entries** to help OpenSTA users find their offerings. Please check out **this FAQ item** and if you unsure about spending money on OpenSTA related items with anyone then please ask on the **Users Mailing List** for advice and recommendations.

If you want to sell us something, have a business proposition for us, want to become a partner, etc. etc. Then I can assure you that **whatever it is**, we are probably **not interested**. If you really want to help out then **check out this FAQ entry**.

OpenSTA Contrib Downloads

Contributed Addons, Mods and Plugins for OpenSTA

Also Check

The **OpenSTA Community Portal** maintains an **Addons and Helpers section** as part of its **Links collection**. It's worth checking there for any new modules.

Addons and Helpers

Extending OpenSTA

From time to time people have really great ideas on how to extend OpenSTA to provide extra features or simply make it easier to use. This page is intended to collect together some of the results of those ideas. We encourage this type of effort and some of these items will eventually make it into the core OpenSTA installation. Until then the downloads and links here should help you getting these extensions running. OpenSTA doesn't really have a well documented or stable plugin architecture (yet) so these extensions might not be quite as seemless to install or use as they should be. We're working on changing this to make it easier for plugin writers and end users.

Covansys BView

Browser View for the Modeler

Many people have assumed (wrongly) that when they run their SCL scripts from within the Modeler that the browser style display should update realtime to show what is going on. The browser display within the Modeler currently only shows the results of the HTTP GET's made at recording time. Anoop Joy and his colleagues at **Covansys India** decided to do something about this and came up with BView. Bview is a seperate browser that is started from Modeler and updates as you run your script.

Unfortunately it seems that Anoop & Co. no longer have any time to keep up with BView and the original site set up to distribute it is dead. Fortunately the work was released with source under the GPL. As it is a fairly popular addon we've put it into the SourceForge filestore and are attempting to keep binary distributions available that match with the current version of OpenSTA. Eventually something similar will probably exist in the core distribution.

The source and binary packages for the latest releases can be found here on SourceForge. Installation and build instructions are included in the relative packages. If you have any problems or thoughts on the binary release then please send these to the **OpenSTA Users mailing list** Any discussion regarding building, changing or improving the tool should be directed to the **OpenSTA Developer mailing list** This type of functionality will likely find its way into a future version of OpenSTA so your comments and fixes can help to shape this.

Jerome Delamarche is freelance consultant that maintains the

Trickytools

a number of helper apps

excellent **trickytools.com** Web site with his various software donations to the world. His free **tools for OpenSTA** have their own **download & info page**. At last check, his extremely useful tools included a command line tool to export the binary OpenSTA results files into text versions (opensta2txt), and a tool to help running OpenSTA tests in batch mode (opstabatch). Please check to see if he's produced more though...

Join In

your idea here

The **Links collection** mentioned in the sidebar is user editable, so if you think you have something that should be added here - then first **add your OpenSTA addon there**. Once in a while we'll check those links and update the applicable items here. If you have an idea for your own tool/addon then the depending on your skill level and the intention of the tool there's a couple of places to start: if you think your tool can work without requiring any changes to OpenSTA itself the the **OpenSTA Users Mailing List** would be a good place to start trying to find help and/or testers, if you think you need to change parts of OpenSTA or need intimate details of the data structures used then the **OpenSTA Developers Mailing List** and **OpenSTA Developer Web site** are the best places to start looking.

---

*Proud to be **Open**, prouder to be **Free***

Questions, Comments, Suggestions?

Last Updated: 2004-SEP-07

Contents

HTTP/S Load User's Guide

HTTP/S Performance Testing under OpenSTA
Version 1.3.2

Copyright

This document was published October, 2001.

Manual reference number: OS-HTTP-10-201

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q - R - S - T - U - V - W - X - Y - Z

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Glossary

Collector

An OpenSTA Collector is a set of user-defined queries which determine the performance data that is monitored and recorded from target Hosts when a Test is run. They are used to monitor and collect performance data from the components of Web Application Environments (WAEs) and production systems during Test-runs to help you evaluate their performance.

Collectors are stored in the Repository and are included in Tests by reference, so any changes you make to a Collector will have immediate affect on all the Tests that use them.

The HTTP/S Load release of OpenSTA (Open Source release) supplies the NT Performance Module and the SNMP Module for Collector creation.

NT Performance Collectors are used for collecting performance data from Hosts running Windows NT or Windows 2000.

SNMP Collectors are used for collecting SNMP data from Hosts and other devices running an SNMP agent or proxy SNMP agent.

Collector Pane

The Collector Pane is the workspace used to create and edit Collectors. It is displayed in the Commander Main Window when you open a Collector from the Repository Window.

Commander

OpenSTA Commander is the Graphical User Interface used to develop and run HTTP/S Tests and to display the results of Test-runs for analysis.

Each OpenSTA Module, provides its own Plug-ins and supplies Module-specific

Test Configuration, data collection, Test-run monitoring and Results display facilities. All Plug-in functionality is invoked from Commander.

Cookie

A packet of information sent by a Web server to a Web browser that is returned each time the browser accesses the Web server. Cookies can contain any information the server chooses and are used to maintain state between otherwise stateless HTTP transactions.

Typically cookies are used to store user details and to authenticate or identify a registered user of a Web site without requiring them to sign in again every time they access that Web site.

CORBA

Common Object Request Broker Architecture.

A binary standard, which specifies how the implementation of a particular software module can be located remotely from the routine that is using the module. An Object Management Group specification which provides the standard interface definition between OMG-compliant objects. Object Management Group is a consortium aimed at setting standards in object-oriented programming. An OMG-compliant object is a cross-compatible distributed object standard, a common binary object with methods and data that work using all types of development environments on all types of platforms.

CYRANO

http://cyrano.com/

CYRANO is a public company listed on the EuroNM of the Paris Bourse (Reuters: CYRA.LN, Sicovam 3922). Created in 1989 and publicly trading since 1998, CYRANO is headquartered in Paris, France, with regional headquarters in the UK and USA.

CYRANO is a sponsor and lead developer on the OpenSTA™ project. CYRANO is an end-to-end quality assurance provider to its customers, helping them maximize their IT investments and ensure uninterrupted e-business. CYRANO offers integrated solutions, service and support to companies that want to minimize risk, benchmark Service Level Agreements, and enable Capacity Planning for their IT infrastructures.

Document Object Model or DOM

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents (Web pages). It defines the logical structure of documents and the way a document is accessed and manipulated.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.

For more information:

- What is the Document Object Model?

  www.w3.org/TR/1998/REC-DOM-Level-1-19981001/introduction.html

- The Document Object Model (DOM) Level 1 Specification

  www.w3.org/TR/REC-DOM-Level-1/

Gateway

The OpenSTA Gateway interfaces directly with the Script Modeler Module and enables you to create Scripts. The Gateway functions as a proxy server which intercepts and records the HTTP/S traffic that passes between browser and Web site during a Web session, using SCL scripting language.

Host

An OpenSTA Host is a networked computer or device used to execute a Task Group during a Test-run. Use the Test Pane in Commander to select the Host you want to use a to run Task Group.

Host also refers to a computer or device that houses one or more components of a Web Application Environment under Test, such as a database. Use Collectors to define a Host and the type of performance data you want to monitor and collect during a Test-run

HTML

Hypertext Markup Language. A hypertext document format used on the World-Wide Web. HTML is built on top of SGML. Tags are embedded in the text. A tag consists of a **<**, a case insensitive directive, zero or more parameters and a **>**. Matched pairs of directives, like **<TITLE>** and **</TITLE>** are used to delimit text which is to appear in a special place or style.

.HTP file

See Scripts.

HTTP

HyperText Transfer Protocol. The client-server TCP/IP protocol used on the

World-Wide Web for the exchange of HTML documents. HTTP is the protocol which enables the distribution of information over the Web.

HTTPS

HyperText Transmission Protocol, Secure. A variant of HTTP used by Netscape for handling secure transactions. A unique protocol that is simply SSL underneath HTTP. See SSL.

HTTP/S

Reference to HTTP and HTTPS.

Load Test

Using a Web Application Environment in a way that would be considered operationally normal with a normal to heavy number of concurrent Virtual Users.

Modules

See OpenSTA Modules.

Monitoring Window

The Monitoring Window lists all the display options available during a Test-run or a single stepping session in a directory structure which you can browse through to locate the monitoring option you need. Each Task Group included in the Test is represented by a folder which you can double-click on to open and view the display options contained.

Use the Monitoring Window to select and deselect display options in order to monitor the Task Groups included in your Test and to view additional data categories, including summary data and an error log. The monitoring display options available vary according to the type of Task Groups included in a Test.

The Monitoring Window is located on the right-hand side of the Monitoring Tab by default, but can be moved or closed if required.

Name Server

See OpenSTA Name Server.

O.M.G.

Object Management Group. A consortium aimed at setting standards in object-oriented programming. In 1989, this consortium, which included IBM Corporation, Apple Computer Inc. and Sun Microsystems Inc., mobilized to create a cross-compatible distributed object standard. The goal was a common

binary object with methods and data that work using all types of development environments on all types of platforms. Using a committee of organizations, OMG set out to create the first Common Object Request Broker Architecture (CORBA) standard which appeared in 1991. The latest standard is CORBA 2.2.

Open Source

A method and philosophy for software licensing and distribution designed to encourage use and improvement of software written by volunteers by ensuring that anyone can copy the source code and modify it freely.

The term Open Source, is now more widely used than the earlier term, free software, but has broadly the same meaning: free of distribution restrictions, not necessarily free of charge.

OpenSTA Dataname

An OpenSTA Dataname comprises between 1 and 16 alphanumeric, underscore or hyphen characters. The first character must be alphabetic.

The following are not allowed:

- Two adjacent underscores or hyphens.
- Adjacent hyphen and underscore, and vice versa.
- Spaces.
- Underscores or hyphens at the end of a dataname.

Note: Where possible avoid using hyphens in the names you give to Tests, Scripts and Collectors. The hyphen character functions as an operator in SCL and conflicts can occur during Test-runs.

OpenSTA Modules

OpenSTA is a modular software system that enables users to add additional functionality to the system by installing new OpenSTA Modules. When a new Module is installed existing functionality is enhanced, enabling users to develop their configuration of OpenSTA in line with their performance Testing requirements. Each Module comes complete with its own user interface and run-time components.

OpenSTA Modules are separate installables that bolt on to the core architecture to add specific functionality, including performance monitoring and data collection for all three layers of Web Application Environment activity:

- Low-level - Hardware/Operating System performance data
- Medium-level - Application Performance Data
- High-level - Transaction Performance Data

OpenSTA Name Server

The OpenSTA Name Server allows the interaction of multiple computers across a variety of platforms in order to run Tests. The Name Server's functionality is built on the Object Management Group's CORBA standard.

Performance Test

One or more Tests designed to investigate the efficiency of Web Application Environments (WAE). Used to identify any weaknesses or limitations of target WAEs using a series of stress Tests or load Tests.

Proxy Server

A proxy server acts as a security barrier between your internal network (intranet) and the Internet, keeping unauthorized external users from gaining access to confidential information on your internal network. This is a function that is often combined with a firewall.

A proxy server is used to access Web pages by the other computers. When another computer requests a Web page, it is retrieved by the proxy server and then sent to the requesting computer. The net effect of this action is that the remote computer hosting the Web page never comes into direct contact with anything on your home network, other than the proxy server.

Proxy servers can also make your Internet access work more efficiently. If you access a page on a Web site, it is cached on the proxy server. This means that the next time you go back to that page, it normally does not have to load again from the Web site. Instead it loads instantaneously from the proxy server.

Repository

The OpenSTA Repository is where Scripts, Collectors, Tests and results are stored. The default location is within the OpenSTA program files directory structure. A new Repository is automatically created in this location when you run Commander for the first time.

You can create new Repositories and change the Repository path if required. In Commander click **Tools > Repository Path**.

Manage the Repository using the Repository Window within Commander.

Repository Host

The Host, represented by the name or IP address of the computer, holding the OpenSTA Repository used by the local Host. A Test-run must be started from the Repository Host and the computer must be running the OpenSTA Name Server.

Repository Window

The Repository Window displays the contents of the Repository which stores all the files that define a Test. Use the Repository Window to manage the contents of the Repository by creating, displaying, editing and deleting Collectors, Scripts and Tests.

The Repository Window is located on the left-hand side of the Main Window by default and displays the contents of the Repository in three predefined folders **Collectors**, **Scripts**, and **Tests**. These folders organize the contents of the Repository into a directory structure which you can browse through to locate the files you need.

Double-click on the predefined folders to open them and display the files they contain.

Right-click on the folders to access the function menus which contain options for creating new Collectors, Scripts and Tests.

Results Window

The Results Window lists all the results display options available after a Test-run or a single stepping session is complete. The display options are listed in a directory structure which you can browse through to locate the results option you need. Each Collector-based Task Group included in the Test is represented by a folder which you can double-click on to open and view the display options contained.

Use the Results Window to select and deselect display options in order to view and analyze the results data you need. The results display options available vary according on the type of Task Groups included in a Test.

The Results Window is located on the right-hand side of the Results Tab by default, but can be moved or closed if required.

SCL

See Script Control Language.

SCL Reference Guide

Use the *SCL Reference Guide* for information on the SCL commands used in Script modeling.

Hard copy and soft-copy versions of this guide are available.

You can view or download it from OpenSTA.org.

An on-line version is available in Script Modeler; click **Help > SCL Reference**.

Script

Scripts form the basis of HTTP/S load Tests using OpenSTA. Scripts supply the HTTP/S load element used to simulate load against target Web Application Environments (WAE) during their development.

A Script represents the recorded HTTP/S requests issued by a browser to WAE during a Web session. They are created by passing HTTP/S traffic through a proxy server known as the Gateway, and encoding the recorded data using Script Control Language (SCL). SCL enables you to model the content of Scripts to more accurately generate the Web scenario you need to reproduce during a Test.

Scripts encapsulate the Web activity you need to test and enable you to create the required Test conditions. Use Commander to select Scripts and include them in a Test then run the Test against target WAEs in order to accurately simulate the way real end users work and help evaluate their performance.

Scripts are saved as a .HTP file and stored in the Repository.

Script Control Language

SCL, Script Control Language, is a scripting language created by CYRANO used to write Scripts which define the content of your Tests. Use SCL to model Scripts and develop the Test scenarios you need.

Refer to the *SCL Reference Guide* for more information.

Script Modeler

Script Modeler is an OpenSTA Module used to create and model Scripts produced from Web browser session recordings, which are in turn incorporated into performance Tests by reference.

Script Modeler is launched from Commander when you open a Script from the Repository Window.

Single Stepping

Single stepping is a debugging feature used to study the replay of Script-based Task Groups included in an HTTP/S load Test. Run a single stepping session to follow the execution of the Scripts included in a Task Group to see what actually happens at each function call, or when a process crashes.

SNMP

Simple Network Management Protocol. The Internet standard protocol developed to manage nodes on an IP network. SNMP is not limited to TCP/IP. It can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs, toasters and jukeboxes.

For more information visit the NET_SNMP Web site:

- What is it? (SNMP)

  http://net-snmp.sourceforge.net/

SSL

Secure Sockets Layer. A protocol designed by Netscape Communications Corporation to provide encrypted communications on the Internet. SSL is layered beneath application protocols such as HTTP, SMTP, Telnet, FTP, Gopher, and NNTP and is layered above the connection protocol TCP/IP. It is used by the HTTPS access method.

Stress Test

Using a Web Application Environment in a way that would be considered operationally abnormal. Examples of this could be running a load test with a significantly larger number of Virtual Users than would normally be expected, or running with some infrastructure or systems software facilities restricted.

Task

An OpenSTA Test is comprised of one or more Task Groups which in turn are composed of Tasks. The Scripts and Collectors included in Task Groups are known as Tasks. Script-based Task Groups can contain one or multiple Tasks. Tasks within a Script-based Task Group can be managed by adjusting the Task Settings which control the number of Script iterations and the delay between iterations when a Test is run.

Collector-based Task Groups contain a single Collector Task.

Task Group

An OpenSTA Test is comprised of one or more Task Groups. Task Groups can be of two types, Script-based or Collector-based. Script-based Task Groups represent one or a sequence of HTTP/S Scripts. Collector-based Task Groups represent a single data collection Collector. Task Groups can contain either Scripts, or a Collector, but not both. The Scripts and Collectors included in Task Groups are known as Tasks.

A Test can include as many Task Groups as necessary.

Task Group Definition

An OpenSTA Task Group definition constitutes the Tasks included in the Task Group and the Task Group settings that you apply.

Task Group Settings

Task Group settings include Schedule settings, Host settings, Virtual User

settings and Task settings and are adjusted using the Properties Window of the Test Pane. Use them to control how the Tasks and Task Group that comprise a Test behave when a Test is run.

Schedule settings determine when Task Groups start and stop.

Host settings specify which Host computer is used to run a Task Group.

Virtual User settings control the load generated against target Web Application Environments during specification of the number of Virtual Users running a Task Group. Set Logging levels to determine the amount of performance statistics collected from Virtual Users running the Tasks. You can also select to Generate Timers for each Web page returned during a Test-run.

Task settings control the number of times a Script-based Tasks are run including the delay you want to apply between each iteration of a Script during a Test-run.

Test

An OpenSTA Test is a set of user controlled definitions that specify which Scripts and Collectors are included and the settings that apply when the Test is run. Scripts define the test conditions that will be simulated when the Test is run. Scripts and Collectors are the building blocks of a Test which can be incorporated by reference into many different Tests.

Scripts supply the content of a Test and Collectors control the type of results data that is collected during a Test-run. Test parameters specify the properties that apply when you run the Test, including the number of Virtual Users, the iterations between each Script, the delay between Scripts and which Host computers a Test is run.

Commander provides you with a flexible Test development framework in which you can build Test content and structure by selecting the Scripts and Collectors you need. A Test is represented in table format where each row within it represents the HTTP/S replay and data collection Tasks that will be carried out when the Test is run. Test Tasks are known as Task Groups of which there are two types, either Script-based and Collector-based.

Test Pane

The Test Pane is the workspace used to create and edit Tests, then run a Test and monitor its progress. After a Test-run is complete results can be viewed and compared here. The Test Pane is displayed in the Commander Main Window when you open a Test from the Repository Window.

Test Table

The Test table is a workspace located within the Configuration tab of the Test Pane, used to add the contents and develop the structure of a Test, and to specify the Task Group settings that control how the Test runs.

Use it in combination with the Repository Window to add Scripts and Collectors, which are represented in the Test table as Tasks within Task Groups. A Task occupies one cell within a Task Group which in turn occupies one row in the Test Table.

Most of the cells in a Task Group have functions associated with them which enable you to control the Tasks they contain. Select a Task Group cell in the Test table and use the Properties Window to configure the Task Group settings.

Transaction

A unit of interaction with an RDBMS or similar system.

URI

Uniform Resource Identifier. The generic set of all names and addresses which are short strings which refer to objects (typically on the Internet). The most common kinds of URI are URLs and relative URLs.

URL

Uniform Resource Locator. A standard way of specifying the location of an object, typically a Web page, on the Internet. Other types of object are described below. URLs are the form of address used on the World-Wide Web. They are used in HTML documents to specify the target of a hyperlink which is often another HTML document (possibly stored on another computer).

Variable

Variables allow you to change the fixed values recorded in Scripts. A variable is defined within a Script. Refer to the Modeling Scripts section for more information.

Virtual User

A Virtual User is the simulation of a real life browser user that performs the Web activity you want during a Test-run. The activity of Virtual Users is controlled by recording and modeling the Scripts that represent the Web activity you want to Test. They are generated when a Script-based Task Group is executed during a Test-run and are used to produce the load levels you need against target WAEs.

Web Application Environment, WAE

The applications and/or services that comprise a Web application. This includes database servers, Web servers, load balancers, routers, applications servers, authentication/encryption servers and firewalls.

Web Applications Management, WAM

Consists of the entirety of components needed to manage a Web-based IT environment or application. This includes monitoring, performance testing, results display, results analysis and reporting.

Web Site

Any computer on the Internet running a World-Wide Web server process. A particular Web site is identified by the host name part of a URL or URI. See also Web Application Environment

---

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Appendix:
HTTP Test Executer Initialization File

The Initialization file TestExecuter_web.ini is copied to the OpenSTA Engines directory when OpenSTA is installed. This file contains parameters that can be modified to customize the operation of the HTTP Test Executer.

If the TestExecuter_web.ini file is not found, the HTTP Test Executer uses the default parameter values.

This file has four sections FILES, SOCKET, TEST and THREAD POOL. The parameters that may appear in each section are listed below.

FILES

This section contains parameters related to the HTTP Test Executer Trace file, Trace.txt. This file is located in the OpenSTA Engines directory.

Parameters:

TraceLevel:

Filters what information is output to the trace file. Range: 0-1000.

If this parameter is set to zero, or is not specified, the trace level is set to the value specified in the Trace Settings dialog within Commander. However, if the trace level specified here is higher than that specified in Commander, the higher trace level is used.

This allows the trace level for the HTTP Task Group Executer on each Host to be set independently.

Current supported values:

0 = Errors only (Default value)

10 = Low level tracing

20 = Medium level tracing

50 = Detailed tracing

1000= Full trace (This value can produce a large Trace file)

SOCKET

This section contains parameters related to socket I/O.

Parameters:

MaxSocketDataBuffersCount:

The number of memory buffers reserved to store received data. Each buffer is the size of the operating system's memory page (4Kb on x86). Too high a value for this parameter will cause an unnecessarily large amount of memory to be reserved. This is not necessarily a problem since the memory is not committed until it is actually required. Too low a value will cause a test to fail, because there is an insufficient number of buffers.

Default: 64000.

SocketDataBuffersGrowingCount:

The number of buffers allocated to store received data when more buffers are required. Each buffer is the size of the operating system's memory page (4Kb on x86). The buffers are allocated from the reserved pool, whose size is specified by the MaxSocketDataBuffersCount parameter.

Default: 2000.

MaxSSLConcurrentReq:

The maximum number of SSL buffers that it is estimated will be used at the same time.

This should be set to: No. of Virtual Users * No. of sockets (1 to 4) per Virtual User

Default: 8000.

SSLGrowingBuffCount:

The number of SSL buffers that will be allocated when more buffers are required.

Default: 1000.

TCP_KeepAlives:

Enable or disable TCP Keepalives. If this parameter is set to a value of 1, TCP Keepalives are enabled for all TCP connections established by the HTTP Task Group Executer. This causes the Executer to emit a TCP Keepalive, every second, after a TCP connection has been inactive for a period of time. On Windows 2000, this period is specified by the KeepAliveInterval parameter. On Windows NT, it is fixed at 2 hours. If an error is detected by a TCP Keepalive, an error message is logged to the Audit Log and Error Log and the associated virtual user is aborted.

TCP Keepalives can be used, to prevent virtual users 'hanging' when no response is received for TCP requests issued on their behalf, e.g. because of the failure of a TCP connection. There is a slight performance hit in using this feature, so for greatest efficiency, it should be disabled if it is not required.

If this parameter is not specified, or is set to a value of 0, TCP Keepalives are disabled and virtual users will wait indefinitely for TCP requests to complete.

Default: 0

KeepAliveInterval:

When TCP_KeepAlives is set to a value of 1 and the Executer Host is running Windows 2000, this parameter specifies the time period in milliseconds after which the HTTP Task Group Executer will emit TCP Keepalives for an inactive TCP connection.

Default: 30000

TEST

This section contains Test related parameters.

Parameters:

BrowserParallelism:

Maximum number of requests that the browser normally manages at the same time.

According to RFC 2616 this should be 2 for HTTP 1.1, although in practice it can frequently be as high as 4. The Scripts generated by the Script Modeler, can be used to determine the value for this parameter for your browser(s).

Default: 4.

InitialVirtualUserCount:

The number of Virtual User Control Blocks pre-allocated at the start of a Test. Pre-allocating Control Blocks avoids the overhead of allocating them during the Test. The optimum value for this parameter is the total number of Virtual Users that are to run during a Test. This way no Control Blocks will need to be allocated during the Test-run, and, if at some time during the Test, all Virtual Users are executing simultaneously, all the Control Blocks will be in use.

Default: 1000.

VirtualUserGrowBy:

The number of Virtual User Control Blocks allocated when more Virtual Users are required during a Test-run.

Default: 20.

THREAD POOL

This section contains parameters controlling the behavior of the thread pool.

Parameters:

ThreadPoolConcurrentThreads:

The number of concurrent threads. A value of zero indicates one thread per CPU.

Recommended range: 0 - (4 * number of CPUs).

Default: 0 (1 thread per CPU).

ThreadPoolSize:

The number of threads available in the thread pool. A value of zero creates a thread pool size of 25 * ThreadPoolConcurrentThreads.

Recommended range: 0 - 100.

Default: 0 (25 * ThreadPoolConcurrentThreads).

Setting the MaxSocketDataBuffersCount Parameter

This parameter should ideally be set to the maximum the number of buffers that are required at any one time. This means that no superfluous space is reserved and all reserved space is used.

One way of calculating this value, is to estimate the maximum number of buffers required for a socket on a thread and then to perform the following

calculation:

No. of Sockets per VU * No. of VUs * Max. no. of buffers required per socket.

This allocates enough buffers for each Virtual User to process the largest item concurrently. This may not be realistic, for example, if the largest item is very large compared to others and is not processed very often.

Another way of calculating the value, is to determine a more realistic value for the number of buffers required by an individual user across all sockets and then to perform the following calculation:

No. of VUs * No. of buffers required per VU (across all sockets).

The received data buffer size is equal to the size of the system's memory page (4Kb on x86).

How the above may be used in practice, is probably best illustrated by an example. Consider a very simple HTTP test specifying 10 virtual users, each issuing no more than 2 requests in parallel: a 2Kb HTML page, containing a 23Kb GIF image.

The first formula above would result in a value of 120 for MaxSocketDataBuffersCount, i.e.:

2 * 10 * 6 (No. of Sockets per VU * No. of VUs * Max. no. of buffers required per socket)

Why 6? Because 6 buffers (of 4Kb each) are required to receive 23Kb (the size of the largest item). However, in this example there are only two items to be processed, so if one socket is processing the GIF image (23Kb) then the other socket must be processing the HTML page (2Kb). Therefore, the second formula above would be more appropriate and would result in a value of 70 for MaxSocketDataBuffersCount, i.e.:

10 * 7 (No. of VUs * No. of buffers required per VU (across all sockets)).

Why 7? Because 7 buffers (of 4Kb each) are required to receive 25Kb (23Kb + 2Kb the maximum size of the items to be processed concurrently by a thread).

Although the example is very simple, it does illustrate how the two formulae can be applied in practice.

Below is a sample INI file:

[FILES]

TraceLevel=500

```
[SOCKET]
MaxSocketDataBuffersCount=64000
SocketDataBuffersGrowingCount=2000
MaxSSLConcurrentReq=8000
SSLGrowingBuffCount=1000
[TEST]
BrowserParallelism=4
InitialVirtualUserCount=1000
VirtualUserGrowBy=20
[THREAD POOL]
ThreadPoolSize=0
ThreadPoolConcurrentThreads=0
```

See also:

Test Executers

The OpenSTA Architecture

OpenSTA (Open System Testing Architecture) is a distributed software architecture for developing, executing and analyzing the results of Tests.

A Test may include Scripts or Collectors or both. Scripts define the operations performed by Virtual Users. Collectors define sets of SNMP, NT Performance data or other data to be retrieved during all or part of a Test-run. They can provide useful information about system activity and the Results can be analyzed alongside those from other OpenSTA Modules.

The OpenSTA Architecture provides generic facilities that may be used by other OpenSTA Modules. This chapter describes this architecture and its components.

See also:

OpenSTA Modules

An OpenSTA Test

The Test Manager and Task Group Executers

A Distributed Architecture

The Web Relay Daemon

The OpenSTA Repository

SNMP Collectors

NT Performance Collectors

Architecture Module Installed Files

Script-Based Module Installed Files

OpenSTA Modules

The OpenSTA Architecture Module is the base OpenSTA Module and must be installed before all others. It is installed, as are all Modules, using Microsoft Installer. The graphical Commander utility is used to develop and run Tests. It is also used to display the results of Test-runs. Each Module, provides its own Plug-ins to provide Module-specific Configuration, Test-run Monitoring and Results display facilities. They are invoked by Commander.

See also:

An OpenSTA Test

The OpenSTA Architecture

An OpenSTA Test

A Test is represented in Commander as a table. This table may contain any number of rows, each defining one of two types of Task Group to be executed, a Script-based Task Group and a Collector-based Task Group.

A Script-based Task Group contains one, or a sequence of Tasks, to be performed by one or more Virtual Users. Each Task is represented by a Script, written in the SCL scripting language (Script Control Language) developed by CYRANO, which in HTTP/S Load, represents a recorded Web browser session. When a Test is run, the SCL compiler is invoked to compile these Scripts into object files for execution by Task Group Executers.

Each OpenSTA Module provides its own facilities for creating and maintaining Module-specific Scripts. For example, HTTP/S Load provides the Script Modeler module for producing Scripts from Web browser session recordings. The file extension for SCL Script source files is Module-specific, for HTTP scripts the extension is .HTP. The object file extension is .TOF. A .SCD file is also created by the SCL compiler; this contains a list of script dependencies and is used to

identify items required to compile and run a Script.

A Collector-based Task Group, defines a set of data to be retrieved from one or more Hosts at user-specified intervals during all or part of a Test-run. This data can be viewed alongside other Results to provide comprehensive information about a Test-run. Each such Task Group consists of a single Task, known as a Collector, defining the data to be retrieved. Collectors can be defined for retrieving performance data from Hosts running Windows NT or Windows 2000 and for retrieving SNMP data from Host computers, or other devices, running an SNMP agent or proxy SNMP agent. Collectors may retrieve data for all or part of a Test-run. Each Collector is held as a file. NT Performance Collectors have the extension .NTP. SNMP Collectors have the extension .SMP.

The definition of a Test, as represented in the table displayed by Commander, is stored in a Test Definition file. This is held within the Test's subfolder. For example, if OpenSTA is installed in the default location, the Test Definition file for the Test MYTEST would be:

```
C:\Program Files\OpenSTA\Repository\Tests\MYTEST\MYTEST.tst
```

The Test Definition file is read by the Test Manager and used to initiate and control the execution of a Test.

When a Test is initiated all the Task Groups, identified by rows in a test table, are started in accordance with the start criteria specified in the Test Definition file. Each Task Group may be started when the Test starts, after a fixed time period from the start of the Test or at a specified day and time of day.

See also:

The Test Manager and Task Group Executers

The OpenSTA Architecture

The Test Manager and Task Group Executers

When a Test is executed, a Test Manager process and one or more Task Group Executer processes are created to execute the Test and its constituent Scripts and Collectors.

When a Test is initiated, by clicking the Start Test button  , a single Test Manager process is created on the Repository Host to execute the TestManager. exe image. This reads the Test Definition file and schedules the execution of the Task Groups that make up the Test. The Test Manager creates a new Task Group Executer process for each Collector-based Task Group and a single Task Group Executer process for each host on which an HTTP Task Group is to be executed. The Task Group definition specifies the Host on which its Executer

process is created.

Task Groups containing Scripts are executed by Module-specific Task Group Executers. For example, a Task Group containing Scripts from HTTP/S Load will be run by the HTTP Task Group Executer (TExecuter_htp.exe), which can be configured using the initialization file TestExecuter_web.ini.

For more information, see Appendix: Test Executer Parameter File. One Test Executer process is created for each Task Group to be executed on a Host.

Collector-based Task Groups are also executed by Module-specific Executers. The Host on which the Task Group Executer runs is specified in the Task Group settings. This is not the Host from which data will be retrieved during a Test-run. The target Hosts for data retrieval are specified in the Collector and defined in the queries it contains.

NT Performance Collector Task Groups are executed by the NT Performance Task Group Executer (TExecuter_ntp.exe). These Executers may run for all or part of a Test-run.

SNMP Collector Task Groups are executed by the SNMP Task Group Executer (TExecuter_smp.exe). These Executers may also run for all or part of a Test-run.

Script-based Task Group Executers close down when all Task Group execution on a host is complete. When all Script-based Execution is complete the Collector-based Task Group Executers and the Test Manager close down.

See also:

A Distributed Architecture

The OpenSTA Architecture

A Distributed Architecture

OpenSTA allows the Task Groups that comprise a Test to be executed on remote Hosts. In order to do this, OpenSTA must be installed on each remote Host and the OpenSTA Name Server on each configured to specify the Repository Host for the Test. Each Name Server must then be restarted. The Name Server on the Repository Host must always be started first.

The Test Manager process created for a Test, always runs on the Repository Host, from which the Test must be initiated. Task Group Executer processes are created by the Test Manager on the Hosts on which the Task Groups are to be executed.

OpenSTA's distributed architecture is based on the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group

(OMG), and uses the omniORB Object Request Broker (ORB) and Naming Service from AT&T Laboratories Cambridge.

See also:

The OpenSTA Repository

The OpenSTA Architecture

The Web Relay Daemon

The Web Relay Daemon

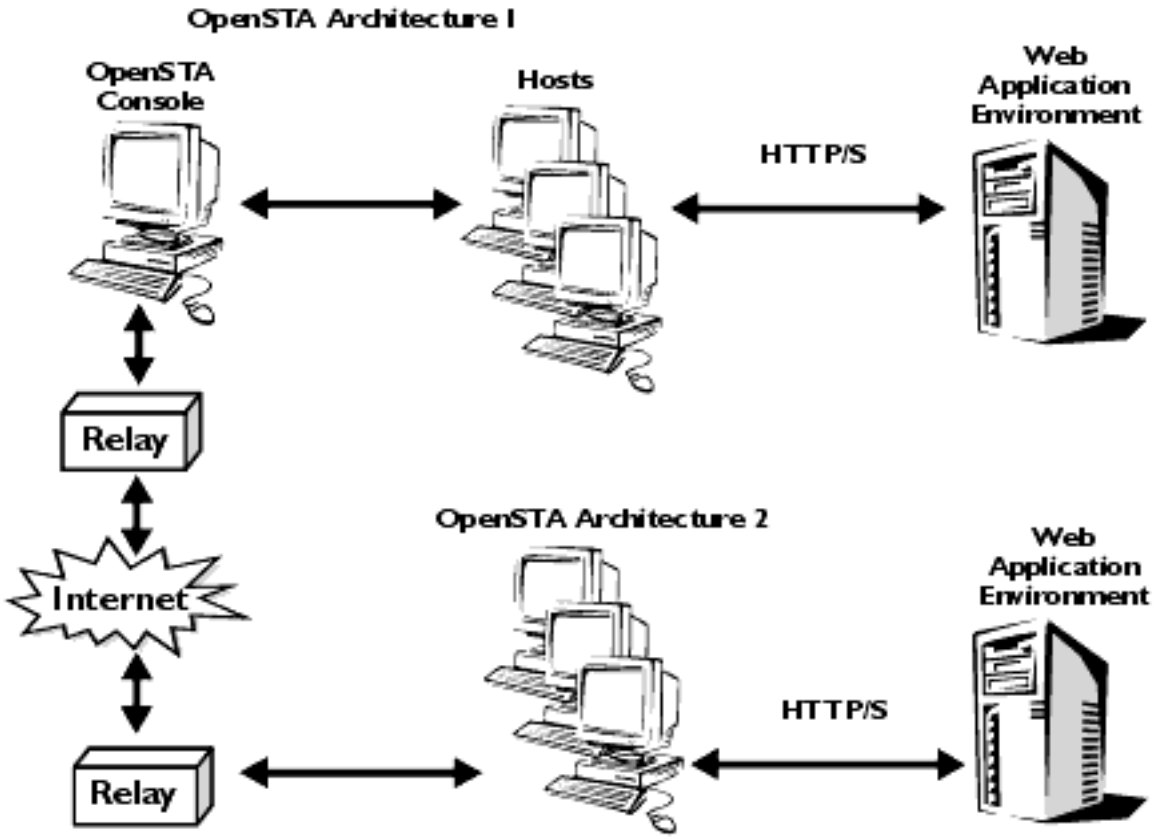
OpenSTA's distributed software architecture enables Test execution on remote Web-based Hosts. This is achieved using a Web Relay Daemon facility which allows the CORBA-based communications within the OpenSTA architecture to be transmitted between machines that are located over the Web.

The Web Relay Daemon facilitates configuration of the components that comprise the Web Relay environment. These consist of the Web Relay Daemon, a Web server and the OpenSTA architecture. Normal Test control communications use XML over HTTP. OpenSTA Web-based replay allows two modes of file transfer: HTTP or FTP. The system also allows SSL-based data transfer.

Use the Web Relay Daemon to map all the machines that need to connect to one another in an OpenSTA architecture which includes Web-based machines. These facilities offer the full range of OpenSTA communications between single or groups of Web-based machines running OpenSTA.

After configuring the Web Relay Daemon remote Hosts can be selected to run a Task Group as normal. For more information see Select the Host used to Run a Task Group.

Web Relay Daemon Architecture



Note: OpenSTA Console refers to a Host computer that has an installation of OpenSTA. This includes the OpenSTA Architecture and Commander, and may also include the Repository, where all Test related files and results are stored.

See also:

Configuring the Web Relay Daemon

Select the Host Used to Run a Task Group

Test-runs

Configuring the Web Relay Daemon

The configuration of the Web Relay Daemon involves:

- Configuring the Web Server
- Configuring the Relay Map
- Setting the Trace Level

Configuring the Web Server

1. Activate the OpenSTA Web Relay facility:

Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon appears in the task bar.
Note: It is assumed that you already have a Web server installed that supports ISAPI.

2. Right-click on and select **Edit Server Settings** from the pop-up menu to open the Server Settings window.
   Note: If the Web Relay Daemon is inactive the icon is visible.
3. Enter the port number of the local Web server in the **Port** field.
4. Check the **Use SSL** option if SSL security is required.
5. Type the path and root directory of the Web server in the **Root Directory** field.
   A DLL is automatically entered in the **ISAPI Extension** field and a cache file in the **File Cache** field.
6. If you want to use FTP file transfer for data transmission, check the Enable FTP File Transfer option and enter your settings in the complete the optional Local FTP Server Settings fields.
7. Click on **Save** to apply your settings.

See also:

Select the Host Used to Run a Task Group

Test-runs

Configuring the Relay Map

1. Activate the OpenSTA Web Relay facility:
   Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon appears in the task bar.
   Note: It is assumed that you already have a Web server installed that supports ISAPI.
2. Right-click on and select **Edit Relay Map** from the pop-up menu to open the Edit Relay Map Settings window.
   Note: If the Web Relay Daemon is inactive the icon is visible.
3. Click on in the toolbar to add the Relay Map settings of the remote Host you want to connect to.
4. In the Edit Relay Map Settings window enter the name of the remote host in the **Alias** field.
5. In the **IP Address** field, enter the IP address of the remote host.
6. Type the ISAPI extension in the **Extension Path** field.
   Note: This entry is identical to the one in the ISAPI Extension field in the

Web server configuration settings.

7. Enter the port number of the Web server in the **Port** field.
8. In the **User Name** field, enter the user name.
9. Type the password in the **Password** field.
10. Check the **Use SSL** option if SSL security is required.
11. Click **OK** to confirm the Relay Map settings.
    Note: Repeat this process on the remote Host to complete the mapping of the two machines.

See also:

Select the Host Used to Run a Task Group

Test-runs

Setting the Trace Level

1. Activate the OpenSTA Web Relay facility:
   Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon  appears in the task bar.
   Note: It is assumed that you already have a Web server installed that supports ISAPI.
2. Right-click on  and select **Set Trace Level** from the pop-up menu to open the Set Trace Level dialog box.
   Note: If the Web Relay Daemon is inactive the  icon is visible.
3. Click  to the right of the **Trace Level** field and select a trace level setting from the drop down list.
   Tip: The trace level you select effects the amount of information you receive about the Test executer processes if problems are encountered during a Test-run. The default setting is `None'.
4. Click on **OK** to confirm the setting.

See also:

Select the Host Used to Run a Task Group

Test-runs

The OpenSTA Repository

All Test Definition files and the result files produced by Test-runs are stored in a flat-file structure on disk; this serves as the OpenSTA Repository. The default Repository folder, if OpenSTA is installed in the default location, is:

```
C:\Program Files\OpenSTA\Repository
```

An empty OpenSTA Repository is created by Commander when it is invoked, if the local Host is the Repository Host and the Repository does not exist. An empty Repository contains the following files:

`OpenSTA\Repository` - OpenSTA Repository
`OpenSTA\Repository\Captures\` - HTTP Module .ALL files
`OpenSTA\Repository\Data\` - Data files, e.g. for file variables
`OpenSTA\Repository\ObjectCode\` - Script object files
`OpenSTA\Repository\Profiles\` - Collectors
`OpenSTA\Repository\Scripts\` - Script source files
`OpenSTA\Repository\Tests\` - Test files, including test result folders
`OpenSTA\Repository\TraceSettings.txt` - Trace settings
`OpenSTA\Repository\Scripts\Include\` - Script include files
`OpenSTA\Repository\Scripts\Include\global_variables.inc` - SCL global include file
`OpenSTA\Repository\Scripts\Include\response_codes.inc` - SCL HTTP response codes include file

An alternative Repository folder can be specified within Commander using the Repository Path option on the Tools menu.

Below is an example listing of the contents of an OpenSTA Repository with sample Collectors, Scripts and Tests:

```
OpenSTA\Repository\.TMP
OpenSTA\Repository\Captures
OpenSTA\Repository\Data OpenSTA\Repository\ObjectCode
OpenSTA\Repository\Profiles OpenSTA\Repository\Scripts
OpenSTA\Repository\Tests OpenSTA\Repository\TraceSettings.txt
OpenSTA\Repository\Captures\ADDCUST.ALL
OpenSTA\Repository\Captures\CUSTORDER.ALL
OpenSTA\Repository\Captures\LOGIN.ALL
OpenSTA\Repository\ObjectCode\ADDCUST.scd
OpenSTA\Repository\ObjectCode\ADDCUST.tof
OpenSTA\Repository\ObjectCode\CUSTORDER.scd
OpenSTA\Repository\ObjectCode\CUSTORDER.tof
OpenSTA\Repository\ObjectCode\LOGIN.scd
OpenSTA\Repository\ObjectCode\LOGIN.tof
OpenSTA\Repository\Profiles\NTPDATA.NTP
OpenSTA\Repository\Profiles\SNMPDATA.SMP
OpenSTA\Repository\Scripts\ADDCUST.HTP
OpenSTA\Repository\Scripts\CUSTORDER.HTP
OpenSTA\Repository\Scripts\Include
OpenSTA\Repository\Scripts\LOGIN.HTP
```

```
 OpenSTA\Repository\Scripts\Include\global_variables.inc
 OpenSTA\Repository\Scripts\Include\response_codes.inc
 OpenSTA\Repository\Tests\NEWCUST
 OpenSTA\Repository\Tests\NEWORDERS
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001
 OpenSTA\Repository\Tests\NEWCUST\NEWCUST.TST
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\ErrLog.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\ICLog_IPADR.txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\NTPHeader.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\NTPStat.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\SNMPHeader.txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\SNMPStat.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\Summary.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\TestConf.
dat
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\TestLog.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\TestManager_1620.log
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\TEW_IPADR-1676.stat
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\TEW_IPADR-1676.urls
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\TExecuter_ntp_1700.log
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\TExecuter_smp_1908.log
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\Timer.txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\VUsersLog.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001
 OpenSTA\Repository\Tests\NEWORDERS\NEWORDERS.TST
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001\ErrLog.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\ICLog_IPADR.txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\NTPHeader.txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001\NTPStat.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
```

```
14-43-58.001\SNMPHeader.txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\SNMPStat.txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001\Summary.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TestConf.dat
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001\TestLog.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TestManager_1692.log
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TEW_IPADR-1624.stat
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TEW_IPADR-1624.urls
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TExecuter_ntp_1752.log
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\TExecuter_smp_1860.log
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001 14-43-58.001\Timer.
txt
 OpenSTA\Repository\Tests\NEWORDERS\27-06-2001
14-43-58.001\VUsersLog.txt
```

Note: IPADR is the IP address on which the Test Manager or Task Group Executer was executed (with dots replaced by underscores).

When a Test is executed, all Scripts that need to be compiled are compiled into the \Objectcode folder of the Repository. If compilation is successful all the files required to execute the Test, not the Task Groups within it, are copied to the \Engines folder of the Repository Host, e.g.:

```
 C:\Program Files\OpenSTA\Engines
```

When a Test Manager initiates the execution of a Task Group, all the files required by the Task Group Executer are copied by the Executer to the \Engines folder of the Executer Host. These files are temporary and can be deleted when test execution is complete.

See also:

SNMP Collectors

The OpenSTA Architecture

SNMP Collectors

The Simple Network Management Protocol (SNMP) Model and protocol allow state information to be retrieved from nodes in a computer network that are running SNMP agents or are served by proxy agents. A `Network Management Station' sends an SNMP request to the SNMP agent, or proxy agent, which returns the requested data.

The OpenSTA SNMP Module allows SNMP data to be retrieved from Host computers and other devices running SNMP agents using OpenSTA SNMP Collectors. These are created and maintained by the OpenSTA SNMP Configuration Plug-in invoked by Commander. This uses the data in the Management Information Block (MIB) files, supplied with OpenSTA or added by the user, to present the data for selection. A list of the IP addresses of Hosts within a user selected range of IP addresses may also be scanned to identify Hosts running SNMP agents or proxy agents. This list may be used to select an SNMP Collector Host and to view the object data from that Host.

An OpenSTA SNMP Collector defines the data to be retrieved from one or more Hosts. An OpenSTA SNMP Collector is held as a comma-separated data file with the .SMP file extension. SNMP data is retrieved and recorded at user specified intervals throughout all or part of each Test-run by the OpenSTA SNMP Task Group Executer (TExecuter_smp.exe). A different interval may be specified for each SNMP object.

OpenSTA's SNMP Task Group Executer uses Net-SNMP from the University of California at Davis.

SNMP data retrieved by the OpenSTA SNMP Task Group Executer can be monitored as it is retrieved using the SNMP monitoring Plug-in from within Commander. The data is stored in local files, one data file per Executer. These files are closed and copied to the Test-run folder of the OpenSTA Repository on the Repository Host when Test execution is complete, e.g.:

```
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001
14-42-30.001\SNMPHeader.txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\SNMPStat.
txt
```

The SNMPHeader.txt file contains descriptions for the data retrieved and is used in Results display. There is one header file for each data file.

See also:

NT Performance Collectors

The OpenSTA Architecture

NT Performance Collectors

Windows NT and Windows 2000 include a graphical tool, the Performance Monitor, for viewing performance data on these systems. The OpenSTA NT Performance Module allows the performance data displayed by this tool and retrieved by the NT Performance facility to be recorded within OpenSTA.

OpenSTA allows NT Performance data to be retrieved from Hosts running Windows NT or Windows 2000 using OpenSTA NT Performance Collectors, created and maintained by the OpenSTA NT Performance Configuration Plug-in invoked by Commander. An OpenSTA NT Performance Collector defines the data to be retrieved from one or more Hosts. An NT Performance Collector is held as a comma-separated data file with the .NTP file extension.

NT Performance data is retrieved at user specified intervals throughout each Test-run by the OpenSTA NT Performance Task Group Executer (TExecuter_ntp.exe).

OpenSTA's NT Performance Task Group Executer uses the Windows API to retrieve the required data.

Windows Performance data retrieved by the OpenSTA NT Performance Task Group Executer can be monitored as it is retrieved using the NT Performance Plug-in within Commander. The data is stored in local files, one data file per Executer. These files are closed and copied to the Test-run folder of the OpenSTA Repository on the Repository Host when Test execution is complete, e.g.:

```
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\NTPHeader.
txt
 OpenSTA\Repository\Tests\NEWCUST\27-06-2001 14-42-30.001\NTPStat.
txt
```

The NTPHeader.txt file contains descriptions for the data retrieved and is used in Results display. There is one header file for each data file.

See also:

Architecture Module Installed Files

The OpenSTA Architecture

Architecture Module Installed Files

The following files are installed by the OpenSTA Architecture Module:

```
OpenSTA\BaseUI\ - Base User Interface
OpenSTA\Common\ - Common DLL's and Active-X Controls
OpenSTA\Copying - GNU GPL
```

OpenSTA\Engines\ - Test Manager, Task Group Executers and associated files
OpenSTA\Plugins\ - Commander Plug-ins
OpenSTA\README.txt - OpenSTA `readme'
OpenSTA\Server\ - OpenSTA server images and DLL's

OpenSTA\BaseUI\OpenSTACommander.chm - Commander help
OpenSTA\BaseUI\oscommander.exe - Commander
OpenSTA\BaseUI\TestPlugin.exe - Test Configuration Plug-in

OpenSTA\Common\cmax20.dll
OpenSTA\Common\Msvcrt.dll
OpenSTA\Common\omniDynamic303_rt.dll
OpenSTA\Common\omniORB303_rt.dll
OpenSTA\Common\omnithread2_rt.dll
OpenSTA\Common\stlport_vc6.dll
OpenSTA\Common\TEPrfInfo.bat
OpenSTA\Common\TEPrfInfo.dll
OpenSTA\Common\XAuditViewer.ocx
OpenSTA\Common\XChartCtrl.ocx

OpenSTA\Engines\Msglib.dll - Message DLL
OpenSTA\Engines\scl.exe - SCL compiler
OpenSTA\Engines\TestExecuter.odl - Generic Task Group Executer ODL
OpenSTA\Engines\TestInit.exe - Test Initiator utility
OpenSTA\Engines\TestManager.exe - Test Manager

OpenSTA\Plugins\ConfigurationTabDLL.dll
OpenSTA\Plugins\MonitoringTabDLL.dll
OpenSTA\Plugins\ResultsTabDLL.dll

OpenSTA\Server\CyrDmn.exe - OpenSTA Daemon
OpenSTA\Server\CyrVDK002.dll - CORBA Repository services provider
OpenSTA\Server\CyrVDK003.dll - CORBA Naming Service cleaner
OpenSTA\Server\CyrVDK004.dll - CORBA Time service
OpenSTA\Server\CyrVDK010.dll - CORBA Injector Control and Status
OpenSTA\Server\CyrVDK011.dll - CORBA Global Variable Factory
OpenSTA\Server\DaemonCFG.exe - Name Server Configuration utility
OpenSTA\Server\GenericObjects.odl - Generic objects ODL
OpenSTA\Server\Logs\ - omniORB log files
OpenSTA\Server\NSC.log - Naming Service Cleaner log file
OpenSTA\Server\OmniOrb\ - omniORB executable images
OpenSTA\Server\ThreadDefinition.odl - Thread definition ODL

OpenSTA\Server\Logs\Shortcut to README.txt.lnk

OpenSTA\Server\OmniOrb\nameclt.exe - Naming Service Utility
OpenSTA\Server\OmniOrb\omniNames.exe - Naming Service

See also:

Script-Based Module Installed Files

The OpenSTA Architecture

Script-Based Module Installed Files

The following files are installed by the OpenSTA HTTP Module. They are listed as an example of the files installed by a Script-based OpenSTA Module:

`OpenSTA\Common\XHttpStats.ocx` - HTTP Statistics OCX or Active X

`OpenSTA\Engines\TEHttp.odl` - HTTP Task Group Executer ODL
`OpenSTA\Engines\TExecuter_htp.exe` - HTTP Task Group Executer
`OpenSTA\Engines\TestExecuter_web.ini` - HTTP Task Group Executer ini file

`OpenSTA\Engines\Web\` - HTTP Modeler-specific files

`OpenSTA\Engines\Web\Modeller\` - HTTP Modeler-specific files

```
OpenSTA\Engines\Web\Modeller\CaptureBHO.dll
OpenSTA\Engines\Web\Modeller\gateway.exe
OpenSTA\Engines\Web\Modeller\GenericObjects.odl
OpenSTA\Engines\Web\Modeller\gwconscmd.dll
OpenSTA\Engines\Web\Modeller\GWConsole.exe
OpenSTA\Engines\Web\Modeller\gwhttp.dll
OpenSTA\Engines\Web\Modeller\headers.ini
OpenSTA\Engines\Web\Modeller\HttpCaptureCmd.dll
OpenSTA\Engines\Web\Modeller\Recoverer.exe
OpenSTA\Engines\Web\Modeller\SCLReference.chm
OpenSTA\Engines\Web\Modeller\stlport_vc6.dll
OpenSTA\Engines\Web\Modeller\TEHttp.odl
OpenSTA\Engines\Web\Modeller\TEHttpLib.dll
OpenSTA\Engines\Web\Modeller\TModeller_Web.exe
OpenSTA\Engines\Web\Modeller\Tof2Scl.dll
```

`OpenSTA\Plugins\HTTPMonDLL.dll` - HTTP Monitoring Plug-in
`OpenSTA\Plugins\HTTPResultsDLL.dll` - HTTP Results Analysis Plug-in

See also:

SNMP Module Installed Files

The OpenSTA Architecture

SNMP Module Installed Files

The following files are installed by the OpenSTA SNMP Module.

OpenSTA\BaseUI\SNMPPlugin.exe - SNMP Configuration Plug-in

OpenSTA\Common\libsnmp.dll

OpenSTA\Engines\Mibs\ - MIB files
OpenSTA\Engines\TExecuter_smp.exe - SNMP Task Group Executer

OpenSTA\Engines\Mibs\DISMAN-SCRIPT-MIB.txt
OpenSTA\Engines\Mibs\EtherLike-MIB.txt
OpenSTA\Engines\Mibs\Host-RESOURCES-MIB.txt
OpenSTA\Engines\Mibs\Host-RESOURCES-TYPES.txt
OpenSTA\Engines\Mibs\IANAifType-MIB.txt
OpenSTA\Engines\Mibs\IF-MIB.txt
OpenSTA\Engines\Mibs\IP-MIB.txt
OpenSTA\Engines\Mibs\IPV6-ICMP-MIB.txt
OpenSTA\Engines\Mibs\IPV6-MIB.txt
OpenSTA\Engines\Mibs\IPV6-TC.txt
OpenSTA\Engines\Mibs\IPV6-TCP-MIB.txt
OpenSTA\Engines\Mibs\IPV6-UDP-MIB.txt
OpenSTA\Engines\Mibs\Makefile.in
OpenSTA\Engines\Mibs\RFC-1215.txt
OpenSTA\Engines\Mibs\RFC1155-SMI.txt
OpenSTA\Engines\Mibs\RFC1213-MIB.txt
OpenSTA\Engines\Mibs\RMON-MIB.txt
OpenSTA\Engines\Mibs\SNMP-COMMUNITY-MIB.txt
OpenSTA\Engines\Mibs\SNMP-FRAMEWORK-MIB.txt
OpenSTA\Engines\Mibs\SNMP-MPD-MIB.txt
OpenSTA\Engines\Mibs\SNMP-NOTIFICATION-MIB.txt
OpenSTA\Engines\Mibs\SNMP-PROXY-MIB.txt
OpenSTA\Engines\Mibs\SNMP-TARGET-MIB.txt
OpenSTA\Engines\Mibs\SNMP-USER-BASED-SM-MIB.txt
OpenSTA\Engines\Mibs\SNMP-VIEW-BASED-ACM-MIB.txt
OpenSTA\Engines\Mibs\SNMPv2-CONF.txt
OpenSTA\Engines\Mibs\SNMPv2-MIB.txt
OpenSTA\Engines\Mibs\SNMPv2-SMI.txt
OpenSTA\Engines\Mibs\SNMPv2-TC.txt
OpenSTA\Engines\Mibs\SNMPv2-TM.txt
OpenSTA\Engines\Mibs\TCP-MIB.txt
OpenSTA\Engines\Mibs\UCD-DEMO-MIB.inc
OpenSTA\Engines\Mibs\UCD-DEMO-MIB.txt
OpenSTA\Engines\Mibs\UCD-DISKIO-MIB.inc
OpenSTA\Engines\Mibs\UCD-DISKIO-MIB.txt
OpenSTA\Engines\Mibs\UCD-DLMOD-MIB.inc
OpenSTA\Engines\Mibs\UCD-DLMOD-MIB.txt
OpenSTA\Engines\Mibs\UCD-IPFILTER-MIB.inc

```
OpenSTA\Engines\Mibs\UCD-IPFILTER-MIB.txt
OpenSTA\Engines\Mibs\UCD-IPFWACC-MIB.inc
OpenSTA\Engines\Mibs\UCD-IPFWACC-MIB.txt
OpenSTA\Engines\Mibs\UCD-SNMP-MIB-OLD.txt
OpenSTA\Engines\Mibs\UCD-SNMP-MIB.inc
OpenSTA\Engines\Mibs\UCD-SNMP-MIB.txt
OpenSTA\Engines\Mibs\UDP-MIB.txt

OpenSTA\Plugins\SNMPMonDLL.dll - SNMP Monitoring Plug-in
OpenSTA\Plugins\SNMPResultsDLL.dll - SNMP Results Analysis Plug-in
```

See also:

NT Performance Module Installed Files

The OpenSTA Architecture

NT Performance Module Installed Files

The following files are installed by the OpenSTA NT Performance Module.

```
OpenSTA\BaseUI\NTPerfPlugin.exe - NT Performance Configuration Plug-in

OpenSTA\Engines\TExecuter_ntp.exe - NT Performance Task Group Executer

OpenSTA\Plugins\NTPerfMonDLL.dll - NT Performance Monitoring Plug-in
OpenSTA\Plugins\NTPerfResultsDLL.dll - NT Performance Results Analysis Plug-in
```

See also:

Error Reporting and Tracing

The OpenSTA Architecture

Error Reporting and Tracing

OpenSTA creates and maintains a number of Log and Trace files for recording Test-run data. These are described below.

See also:

Results Display

The Audit, Report and History Logs

OpenSTA maintains an Audit Log of its activity and related events for each Test-run. This file contains informational, warning and error messages from the Test Manager, Task Group Executers and, optionally, messages from Scripts written using the SCL LOG command.

All messages in the Audit Log are time-stamped and indicate the name of the Script being processed, the associated User ID and the corresponding script line number, as applicable. All Time-stamps in the Audit Log and elsewhere are based on the time on the Repository Host. This makes it easier to analyze the results of Tests executed on Hosts with different system clock settings or in different time-zones.

The Audit Log can be viewed from the Results tab in Commander. The Audit Log is stored in the Test-run results folder in the OpenSTA Repository. For example, the Audit Log for the Test MYEST initiated on 27-Jun-2001 at 14:27:55 would be held in the following file (if OpenSTA was installed in the default location):

```
 C:\Program Files\OpenSTA\Repository\Tests\MYTEST\27-06-2001
14-27-55.001\TestLog.txt
```

In addition to the Audit Log, OpenSTA may also create two further Test-run logs that may be written to from a Script, a Report Log and a History Log.

The purpose of the Report Log (TestRep.txt) is to record transient information relating to the execution of a Test. Task Group Executers may write messages to this Log, for example to record test-case failures and passes. Messages may also be written to the Log from a Script using the SCL REPORT command. The Report Log can be viewed from the Results tab in Commander.

The purpose of the History Log (TestHis.txt) is to record a history of the executions of a Test. Messages are written to the Log from a Script using the SCL HISTORY command. No OpenSTA process, Test Manager or Task Group Executer, writes messages to this Log. The History Logs for a Test can be viewed from the Results tab in Commander. A separate History Log is maintained for each Test-run. However, all the History Logs for a Test are concatenated to form a single Log when viewed within Commander.

Report and History Logs are stored in the Test-run results folders. Messages within them are time-stamped and indicate the name of the Script being processed, the associated User ID and the script line number, as applicable.

See also:

The Error Log

Test Manager and Task Group Executer Trace Logs

Other Trace Logs

Tracing Script Activity

The Error Log

Within OpenSTA there is an Error Log (ErrLog.txt). This file will contain all significant error messages from the Test Manager, Task Group Executers and OpenSTA Daemon. The Error Log can be viewed from the Monitoring tab in Commander during a Test-run and from the Results tab.

Error Logs are stored in the Test-run results folders. Messages within them are time-stamped.

See also:

The Audit, Report and History Logs

Test Manager and Task Group Executer Trace Logs

Other Trace Logs

Tracing Script Activity

Test Manager and Task Group Executer Trace Logs

For each Test-run, a Trace Log is created for the Test Manager and each Task Group Executer. These Logs contain informational, warning and error messages logged by the Test Manager and Task Group Executers respectively. Error messages will also be written to the Audit Log. They are created in the same folder as the corresponding executable images, i.e. \Engines, and are copied to the Test-run results folder on test completion. The Log file names have the following format:

- TestManager_PID
- TExecuter_htp_IPADR
- TExecuter_EID_PID

Note: IPADR is the IP address of the Host on which the Task Group Executer was executed (with dots replaced by underscores), PID is the Process ID of the Test Manager or Task Group Executer and EID is the Executer Identifier (e.g.

SMP for the SNMP Task Group Executer).

Each message that may be written to the Test Manager and Task Group Executer Trace Logs has an associated level. The level is a number between 1 and 1000 and indicates the importance, or severity, of the message:1 is the most important and 1000 the least important. These levels are used to control the level of messages that are recorded for a Test-run.

The highest level of messages to be recorded for the Test Manager and each class of Task Group Executer during a Test-run, may be specified on the Trace Settings dialog within Commander. This allows each trace level to be set to one of four values:

None (0): Errors only Low (10): Low level tracing Medium (20): Medium level tracing High (50): Detailed tracing

The numbers in parentheses indicate the corresponding trace level numbers.

The trace settings are saved in the file TraceSettings.txt in the Repository folder. When a Test is started, this file is copied to the \Engines folder on each Executer host. The Injector Control object within the OpenSTA Daemon on each of these hosts reads the Trace Settings file and uses it to set the trace level in the command line it creates to initiate each Task Group Executer.

The trace settings apply to all subsequent Test-runs, or until the Trace Settings are modified.

Initially, tracing is switched off, i.e. the trace levels for the Test Manager and all Executers are set to zero, in order to make execution as fast as possible. Tracing is typically enabled during Script, Test and Collector development, in order to help resolve problems when Tests are not running as expected.

See also:

The Audit, Report and History Logs

The Error Log

Other Trace Logs

Tracing Script Activity

Other Trace Logs

In addition to the Logs described above, OpenSTA also maintains the following Logs, which may be of use in diagnosing problems:

- OpenSTA\Engines\ICLog_IPADR_PID.log

  Injector Control object Log, records the activity of the OpenSTA

Daemon's Injector Control object. This is responsible for controlling the execution of Task Groups on a host. This file is reset for each Test-run and is written to the Test-run results folder in the OpenSTA Repository on Test-run completion. IPADR is the IP address of the associated OpenSTA Daemon host (with dots replaced by underscores). PID is the process ID of the OpenSTA Daemon process.

- OpenSTA\Engines\ISLog_IPADR_PID.log

  Injector Status object Log, records data related to the OpenSTA Daemon's Injector Status object. This is responsible for retrieving Task Group status data for the Executers running on a host, e.g. for monitoring Task Group activity during a Test-run. IPADR is the IP address of the associated OpenSTA Daemon host (with dots replaced by underscores). PID is the process ID of the OpenSTA Daemon process. A new log file is created each time the OpenSTA Daemon is started.

- OpenSTA\Server\cyrdmn_PID.log

  OpenSTA Daemon Log, records the activity of the OpenSTA Daemon. PID is the process ID of the OpenSTA Daemon process.

- OpenSTA\Engines\Web\Modeller\gateway.log

  Gateway Log, created during Script capture.

- OpenSTA\Server\NSC.log

  Naming Service cleaner Log.

A higher level of tracing my be set for the OpenSTA Daemon by checking the "Turn on Tracing" check box on the Name Server Configuration utility's "Configuration" dialog.

See also:

The Audit, Report and History Logs

The Error Log

Test Manager and Task Group Executer Trace Logs

Tracing Script Activity

Tracing Script Activity

Script activity may also be traced at run-time using the SCL NOTE command within Scripts. This command allows a message to be associated with a virtual user. The messages associated with a virtual user, if any, may be viewed within Commander, through the Monitoring tab. By including NOTE commands within

Scripts it is possible to trace the flow of execution for virtual users at run-time.

See also:

The Audit, Report and History Logs

The Error Log

Test Manager and Task Group Executer Trace Logs

Other Trace Logs

Starting OpenSTA

When OpenSTA is installed, the Name Server Configuration utility (DaemonCFG.exe) is configured to startup automatically when a user logs in to Windows and to start the Naming Service (omninames.exe), if the local Host is the Repository Host, and the OpenSTA Daemon (cyrdmn.exe) on the local Host.

Before a Test can be executed within OpenSTA, the Naming Service, omninames, must be running on the Repository Host. The Naming Service is used to hold the names and types of OpenSTA CORBA objects. The Naming Service provides the means by which a program can locate the object reference for an object and thereby reference it.

By default, after a user has logged in to a Host on which OpenSTA has been installed, the following images will be running:

- DaemonCFG.exe - Name Server Configuration utility
- omninames.exe - Naming service (if local Host is Repository Host)
- cyrdmn.exe - OpenSTA Daemon

These images will continue to run until, either they are explicitly shutdown by the user, using the OpenSTA Name Server, or they terminate abnormally. If "Automatic Notification" is enabled, the Name Server Configuration utility displays a warning dialog box, if either the Naming Service or OpenSTA Daemon terminates abnormally.

Omninames and the OpenSTA Daemon can be started from the command lines follows:

- omninames - start 1250
- cyrdmn

After the Naming Service and OpenSTA Daemon on the Repository Host have been started, the OpenSTA Daemon on each remote host on which Task Groups are to be executed must be started.

See also:

The Name Server Configuration Utility

The OpenSTA Architecture

The Name Server Configuration Utility

The Name Server Configuration utility (DaemonCFG.exe), accessible from the Windows Programs menu, provides a "Configure..." option. This displays a configuration dialog containing four fields:

- **Repository Host:**

  This identifies the name, or IP address, of the Host holding the OpenSTA Repository to be used by the local Host. Tests must be initiated from the Repository Host and the Naming Service must run on the Repository Host.

- **Repository Path:**

  This is a readonly field identifying the Repository path on the local Host. This is configured from Commander. It is not relevant and not used, unless the local host is the Repository Host.

- **Turn on tracing:**

  If this check-box field is checked, additional tracing data will be logged to the OpenSTA Daemon log file cyrdmn.log.

- **Automatic Notification:**

  If this check-box field is checked, a timer is initiated to `fire' every five seconds. When the timer fires, the system is checked to see that the Naming Service and OpenSTA Daemon process are still running, if either is not, a warning dialog is displayed.

The Name Server Configuration utility also provides the following menu options:

- **Start Name Server:**

  Starts the Naming Service (omninames.exe) and the OpenSTA Daemon (cyrdmn.exe) on the local Host. The Naming Service, will only be started if the Repository Host is configured to be the local Host.

- **Stop Name Server:**

  Stops the OpenSTA Daemon (cyrdmn.exe) on the local Host.

- **Shutdown:**

  Shuts down the Naming Service (omninames.exe), the OpenSTA Daemon (cyrdmn.exe) and the Name Server Configuration utility (DaemonCFG. exe) on the local Host. The Naming Service, will only be shutdown if the Repository Host is configured to be the local Host.

- **Registered Objects:**

  Lists the OpenSTA CORBA objects registered with the Naming Service. The omniORB nameclt utility (held in the Server folder) may also be used to view the list of registered objects, command "nameclt list".

When a Test is executed:

- The Repository is located on the Repository Host in the folder identified by the "Repository path", which can be configured through Commander.
- The Naming Service (omninames.exe) will run on the Repository Host.
- The Test must be initiated from the Repository Host.
- An OpenSTA Daemon (cyrdmn.exe) must be running on each Host on which a Task Group is to be executed and must have been started after the Naming Service on the Repository Host.

See also:

The OpenSTA Daemon

The OpenSTA Architecture

The OpenSTA Daemon

The OpenSTA Daemon process starts a CORBA Factory object and loads a set of `provider' modules, used to provide CORBA `services' to OpenSTA components. These modules are held in the OpenSTA Server folder and have file names of the form "CyrVDKnnn.dll", where "nnn" is a numeric identifier.

Below is a list of the OpenSTA provider modules, together with the names that they register with the Naming Service:

CyrVDK002.dll - Repository interface IPADR_CyrStProvider_001.CyranoProvider IPADR_RegistryProvider_001.CyranoProvider IPADR_CyranoVDK002.CyranoLog

CyrVDK003.dll - Naming Service cleaner

CyrVDK004.dll - Time service IPADR.TimeService

CyrVDK010.dll - Injector Control and Status services IPADR.InjectorControl IPADR.InjectorStatus

CyrVDK011.dll - Global Variable Factory IPADR.CyrVariableFactory

Where IPADR is the Host's IP address (with dots replaced by underscores).

After the OpenSTA Daemons have been started, there will be nine CORBA objects registered with the Naming Service for each host registered with the Repository Host:

IPADR.InjectorStatus // Injector Status

IPADR.InjectorControl // Injector Control

IPADR.TimeService // Time service

IPADR.CyrVariableFactory // Variable factory

IPADR.CyranoFactory // CORBA object factory

IPADR_CyranoDaemon.CyranoLog // Daemon logger

IPADR_CyranoVDK002.CyranoLog // Repository logger

IPADR_CyrStProvider_001.CyranoProvider // Repository provider

IPADR_RegistryProvider_001.CyranoProvider // Registry provider

These objects should always be registered with the Naming Service, they should only disappear if the corresponding OpenSTA Daemon process shuts down or terminates abnormally.

See also:

Command Line Formats

The OpenSTA Architecture

Command Line Formats

There is a command line interface to most OpenSTA executable images. The formats of those that may be useful to OpenSTA users are listed below.

Test Initiator (TestInit.exe)

OpenSTA Daemon (CyrDmn.exe)

Script Compiler (scl.exe)

See also:

OpenSTA Modules

Test Initiator (TestInit.exe)

The Test Initiator utility on the Repository Host may be used to start, stop and kill Tests.

Format:

TestInit -start -T Testname

TestInit -stop

TestInit -kill

Description:

The -start switch initiates execution of a specified test (Testname).

The -stop and -kill switches will close the Test Manager currently registered with the OpenSTA naming service. If there is a fault, and more than one Test Manager is running, TestInit will prompt for the Test Manager to stop. A stop attempts to close down a Test-run gracefully, while a kill terminates the Test Manager and Task Group Executers immediately. OpenSTA will attempt to return Test results to the OpenSTA Repository for both a stop and a kill.

See also:

OpenSTA Daemon (CyrDmn.exe)

Script Compiler (scl.exe)

OpenSTA Daemon (CyrDmn.exe)

Before a Test can be executed, an OpenSTA Daemon (cyrdmn.exe) must be running on each Host on which a Task Group is to be executed and each must have been started after the naming Service on the Repository Host. The OpenSTA Daemon process on a host is normally started automatically when a user logs in to Windows, by execution of the Name Server Configuration utility (DaemonCFG.exe). It may also be started by selecting Start > Programs > OpenSTA > OpenSTA Name Server or by selecting Start Name Server within the Name Server Configuration utility.

Format:

cyrdmn --help Display utility help

--trace n Trace logging level (0 to 1000)

Description:

The --trace switch is not required.

See also:

Test Initiator (TestInit.exe)

Script Compiler (scl.exe)

Script Compiler (scl.exe)

The script compiler (SCL) is used to compile Scripts created using a Module-specific Plug-in. Scripts are held in the Scripts folder and have an application specific extension, e.g. .HTP for HTTP Scripts.

SCL generates object files which are executed by a Task Group Executer. A dependency file is also created for each Script that is successfully compiled, this is used to identify items required to compile and run the Script.

Note that the SCL command line used by Commander includes the -I switch to specify the include directory \Scripts\Include, within the Repository.

SCL has the following command line format:

Format:

scl {option(s)} SCL_file

-h --help

-i --confirm

-I inc_path --include=inc_path

-l [on|off] --list=[on|off]

-o obj_file --object=obj_file

-v --log

-V v1+v2{+...} --variant=v1+v2{+...}

Example:

```
scl -o myscript.tof -I Include -v myscript.htp
```

Compile the Script myscript.htp and output compilation messages to standard output, for a successful compilation these will identify the names of the files created. SCL will look for any include files in the local folder and, if not found there, in the Include subfolder. The following files will be created:

myscript.tof - Object file (for execution)

myscript.scd - Script dependency file

See also:

Test Initiator (TestInit.exe)

OpenSTA Daemon (CyrDmn.exe)

The OpenSTA Architecture

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Results Display

Results Display Overview

HTTP/S Load provides a variety of data collection and display options to assist you in the analysis of Test results. Running a Test and displaying the results enables you to identify whether the Web Application Environments (WAEs) under test are able to meet the processing demands you anticipate will be placed on them. After a Test-run is complete use Commander to control which results are displayed and how they are presented, in order to help you analyze the performance of target WAEs and the network used to run the Test.

Open the Test you want from the Repository Window and click on the  **Results** tab in the Test Pane, then choose the results you want to display using the Results Window. Depending on the category of results you select, data is displayed in graph or table format. You can choose from a wide range of tables and customizable graphs to display your results which can be filtered and exported for further analysis

and print. Use the Results Window to view multiple graphs and tables simultaneously to compare results from different Test-runs.

When a Test is run a wide range of results data is collected automatically. Virtual User response times and resource utilization information is recorded from all Web sites under test, along with performance data from WAE components and the Hosts used to run the Test. Results categories include the Test Configuration option which presents a brief description of the Test and the Task Groups settings that applied during a Test-run. The Test Audit log records significant events that occur during a Test-run and the HTTP Data List records the HTTP/S requests issued, including the response times and codes for every request. The Timer List option records the length of time taken to load each Web page defined in the Scripts referenced by a Test.

Creating and referencing Collectors in a Test helps to improve the quality and extend the range of the results data produced during a Test-run. Collectors give you the ability to target the Host computers and devices used to run a Test and the back-end database components of WAEs under test, with user-defined data collection queries. Use NT Performance and SNMP Collectors to collect data from Host devices within target WAEs or the test network.

The range of results produced during a Test-run can depend on the content of the Scripts that are referenced by a Test. For example Report and History logs are only produced if the Scripts included have been modeled to incorporate the SCL commands used to generate the data content for these logs.

See also:

Results Tab

General Results Display Procedures

Test Audit Log

Test Report Log

Test History Log

HTTP Data List

HTTP Data Graphs

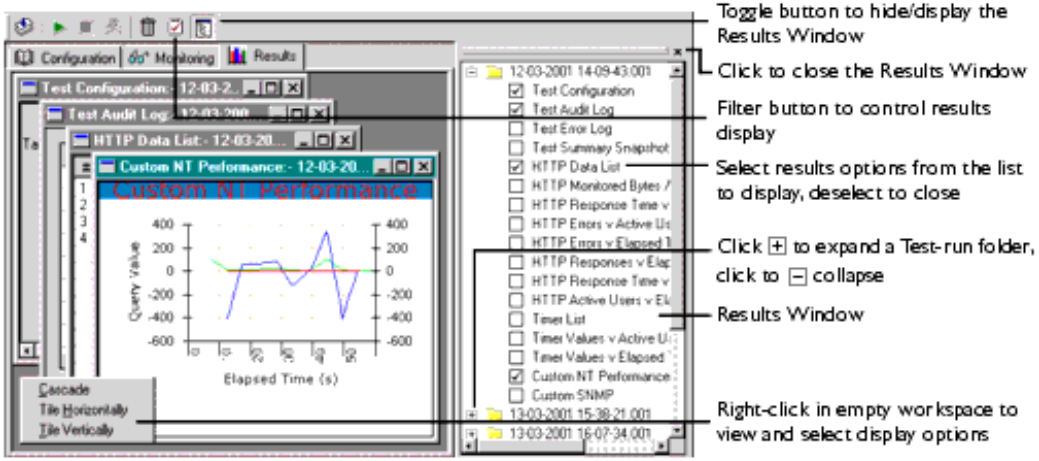
Timer List

SNMP and NT Performance Collector Graphs

Results Tab

Results are stored in the Repository after a Test-run is complete. You can view them by working from the Repository Window to open the Test you want, then click on the  **Results** tab in the Test Pane. Use the Results Window to select the results you want to view in the workspace of the Test Pane. You can reposition the Results Window by floating it over the Main Window to give yourself more room for results display, or close it once you have selected the results options you want to view.

The Results Tab of the Test Pane



Results Tab Display Options

Graphs can be customized to improve the presentation of data by right-clicking within a graph then selecting **Customize**. This function includes options that enable you to modify the graph style from the default line plot to a vertical bar, as well as controlling the color of elements within the graph display.

You can control the information displayed in some graphs and tables by filtering the data they represent. Right-click within a graph or table, then select **Filter** or **Filter URLs**, or click the Filter button  in the toolbar and make your selection. You can also opt to export results data for further analysis and printing. Right-click and select **Export to Excel** or **Export** from the menu.

You can also zoom in on a graph by clicking and dragging over the area of the graph you want to study. Use the **Windows** option to control the presentation of results options in the Test Pane, or right-click within the empty workspace of the Test Pane to access these functions as illustrated in the diagram above.

See also:

The Results Window

Display Test Results

Customize Graph Display

Zoom In and Out of a Graph

Export Test Results

Close Test Results

Delete Test Results

The Results Window

When you click on the **Results** tab, the Results Window opens automatically. Its default location is on the right-hand side of the Test Pane

where it is docked. Use it to select and display results from any of the Test-runs associated with the current Test.

Test-runs are stored in date and time stamped folders which you can double-click on to open, or click ⊞ . When you open a Test-run folder, the available results are listed below. Display the results you want by clicking on the options and ticking the check boxes to the left of the results options. The results you choose are displayed in the Test Pane.

Multiple graphs and tables from different Test-runs associated with the current Test can be displayed concurrently. Use the Results Window to select additional Test-runs and equivalent results options to compare Test results and help evaluate performance.

Results Window Display Options

The Results Window is located on the right-hand side of the Test Pane. It can be closed to increase the workspace area available, or moved to a new position by floating it over the Main Window.

See also:

Hide/Display The Results Window

Move The Results Window

Resize The Results Window

Display Test Results

Results Tab

Hide/Display The Results Window

- Click  , in the double bar at the top of the Results Window to close it.
- Click  in the toolbar to toggle between hiding and displaying the Results Window.

Move The Results Window

1. Click on the double bar at the top of the Results Window.
2. Drag, then drop it in the new position within the Main Window.

   Note: The Results Window docks with the Main Window's borders if it contacts them. Hold down the **Control** key while you reposition the Results Window to avoid this.

Resize The Results Window

1. Move your cursor over part of the window edge.
2. Click and drag, then drop the border in the required position.

General Results Display Procedures

- Display Test Results
- Zoom In and Out of a Graph
- Customize Graph Display
- Export Test Results
- Close Test Results
- Delete Test Results

Display Test Results

1. In the Repository Window, double-click **Tests** to expand the directory structure.
2. Double-click the Test , whose results you want to display.
3. In the Test Pane click the **Results** tab.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.
4. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and view a list of results display options and Task Group results folders.
5. Click ⊞ next to a results option to display your selection in the Test Pane or open a Task Group folder and select from the display options listed.

   A ticked check box to the left of a display option indicates that it is open in the Test Pane.

   Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

   Tip: All available results have display and output options associated with them. These options may include filtering, customizing and exporting. Right-click within a graph or table to display and select from the choices available.

   Use the **Windows** option in the Menu Bar to control the display of graphs and tables. Or, right-click within the empty workspace of the Test Pane to access these functions.

See also:

Customize Graph Display

Zoom In and Out of a Graph

Export Test Results

Customize Graph Display

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.
2. In the Results Window, double-click on a Test-run folder or click ⊞ , to open it and display the available results.

3. Click on a graph display results option to open your selection in the Test Pane.
4. Right-click inside the graph and select **Customize**.
5. Select the Graph Type you want:

- **Line plot**: A single line connecting values.
- **Vertical bars**: A single, solid vertical bar per value.
- **Area under points**: The area beneath the line plot is filled.

Other options control the colors, type face and graph title show or hide.

6. Click **OK** to apply your choices.

Note: The customize settings you select are not saved when you close a graph.

See also:

Display HTTP Data Graphs

Display Custom Collector Graphs

Zoom In and Out of a Graph

1. Open a Test and click the **Results** tab of the Test Pane.

The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and view a list of results display options and Task Group results folders.
3. Click ⊞ next to a graph option to display your selection in the Test Pane.
4. Click and drag over the area of the graph you want to zoom in on and release your mouse button.

The data range you select is magnified to fill the graph window.

5. Double-click anywhere in the graph to zoom out and return to the full graph display.

Export Test Results

1. Open a Test and click the **Results** tab of the Test Pane.

The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and view a list of results display options and Task Group results folders.
3. Click ⊞ next to a results option to display your selection in the Test Pane.
4. Right-click inside the graph or table and select either **Export to Excel** (graphs), or **Export** (tables and lists).

Note: The **Export to Excel** option automatically launches Excel and converts the data into Microsoft Excel Workbook format. Save and edit your results as required.

The **Export** option enables you to export results as a .CSV file. The Test Configuration results option only supports text file format for exporting data.

Close Test Results

- Click , in the Title Bar of a graph or table to close it.
- Use the Results Window to close a graph or table by clicking on the results option and unchecking the check box to the left of the option.

  Note: Click in the toolbar to open or close the Results Window
- Open a different Test or a Collector from the Repository Window.

  Note: You can move between the display tabs within the currently selected Test without affecting the display options you have chosen in the Results tab.

Delete Test Results

1. Open a Test and click the **Results** tab of the Test Pane.
2. Click , in the toolbar.
3. In the Delete Test-runs dialog box, select the Test-runs you want to delete.

   Note: Test-runs are labelled with a date and time stamp to help you identify them.
4. Click **Delete** to remove the results from the Repository.

Test Configuration

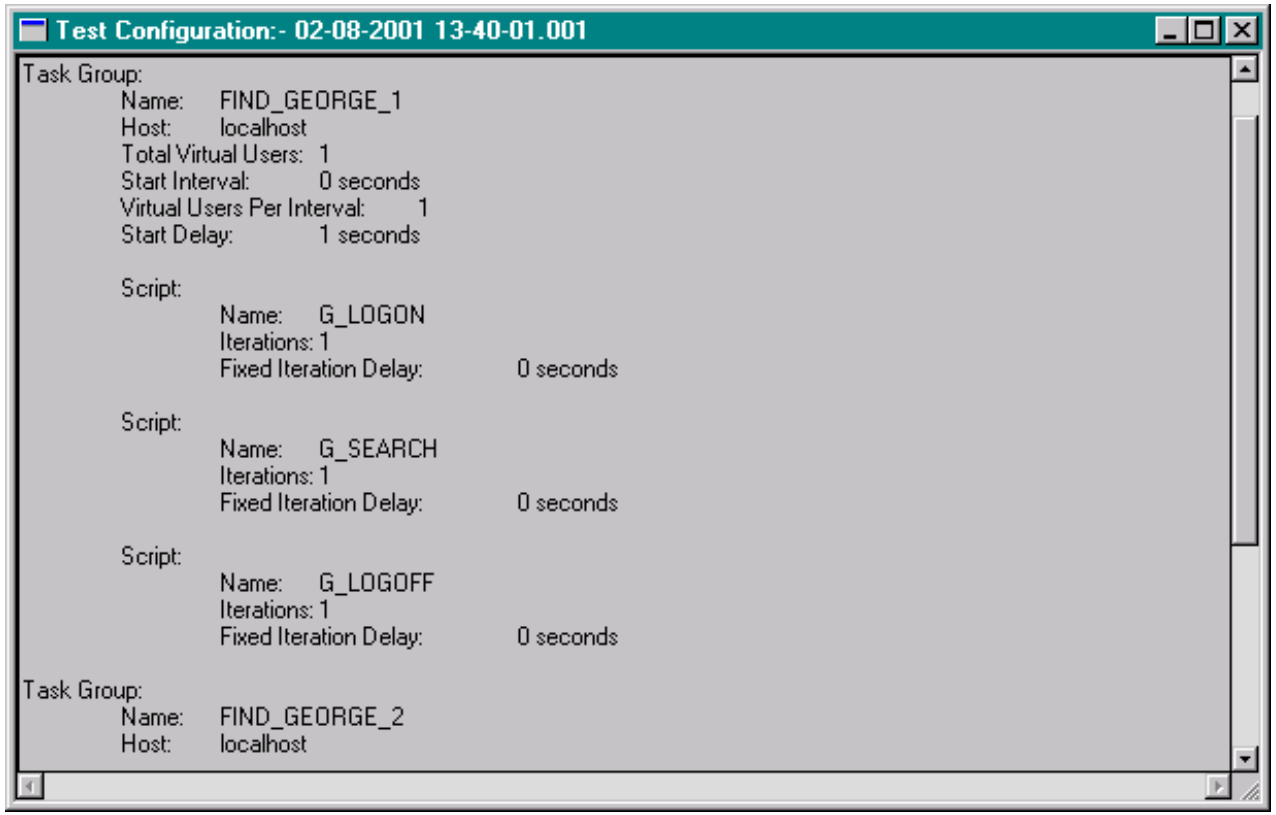
The Test Configuration display option consists of a summary of data collected during a Test-run. It provides data relating to the Task Groups, Scripts, Hosts and Virtual Users that comprised the Test-run.

See also:

Display Test Configuration

Display Test Configuration

1. Open a Test and click the **Results** tab in the Test Pane.
2. In the Results Window, double-click on a Test-run folder or click ⊞ , to open it and display the available results.
3. Click the **Test Configuration** results option in the list.
   Test configuration information is displayed in the Results tab in the following format:



Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

See also:

Test Configuration

Test Audit Log

The Test Audit log contains a list of significant events that have occurred during a Test-run. These include the times and details of Test initiation and completion, errors that may have occurred and Virtual User details.

Additional Audit log entries may be written to the log if the Scripts included in the Test have been modeled to incorporate the appropriate SCL code. Use the **LOG** SCL command in a Script, to generate the data content for the Test Audit log. For more information on SCL refer to the *SCL Reference Guide*; an on-line copy is available within the Script Modeler, Help menu.

See also:

Display Test Audit Log Data

Error Reporting and Tracing

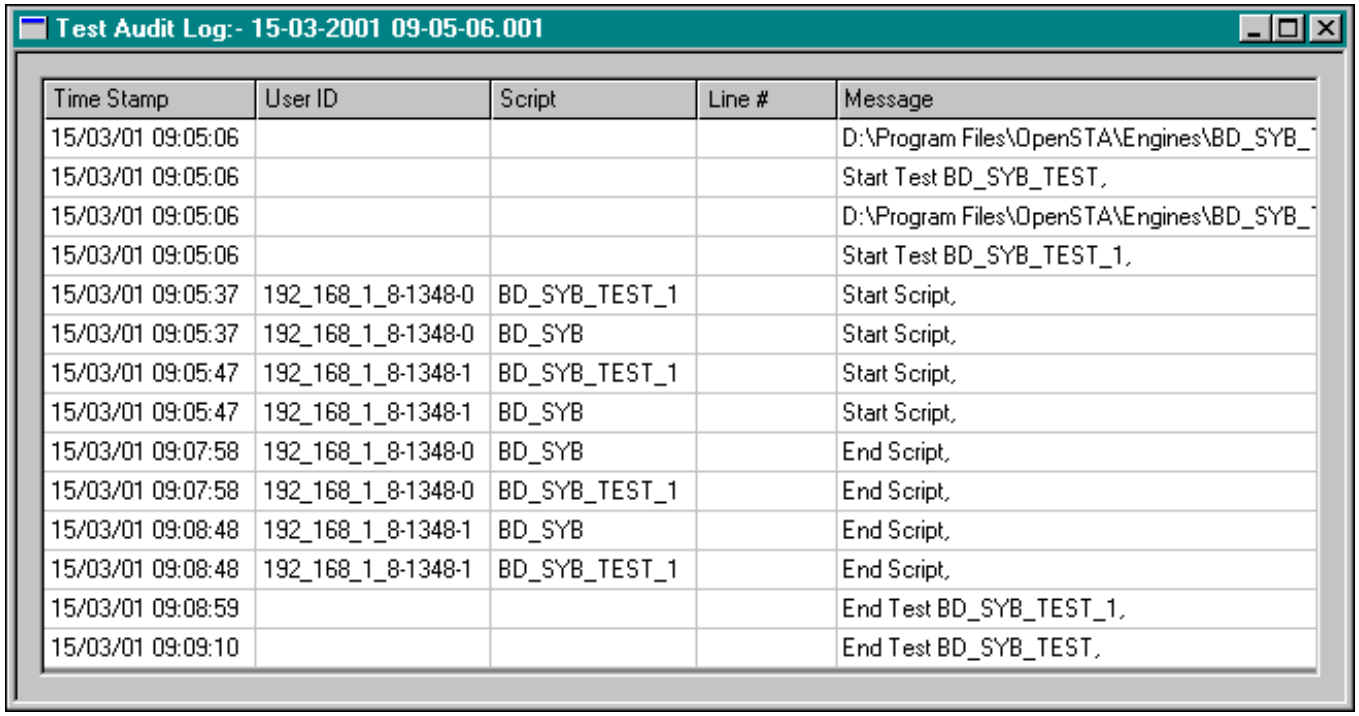
Display Test Audit Log Data

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **Test Audit Log** results option in the list.

   Audit information is displayed in the Results tab in table format:



Test Audit Log:- 15-03-2001 09-05-06.001

Time Stamp	User ID	Script	Line #	Message
15/03/01 09:05:06				D:\Program Files\OpenSTA\Engines\BD_SYB_
15/03/01 09:05:06				Start Test BD_SYB_TEST,
15/03/01 09:05:06				D:\Program Files\OpenSTA\Engines\BD_SYB_
15/03/01 09:05:06				Start Test BD_SYB_TEST_1,
15/03/01 09:05:37	192_168_1_8-1348-0	BD_SYB_TEST_1		Start Script,
15/03/01 09:05:37	192_168_1_8-1348-0	BD_SYB		Start Script,
15/03/01 09:05:47	192_168_1_8-1348-1	BD_SYB_TEST_1		Start Script,
15/03/01 09:05:47	192_168_1_8-1348-1	BD_SYB		Start Script,
15/03/01 09:07:58	192_168_1_8-1348-0	BD_SYB		End Script,
15/03/01 09:07:58	192_168_1_8-1348-0	BD_SYB_TEST_1		End Script,
15/03/01 09:08:48	192_168_1_8-1348-1	BD_SYB		End Script,
15/03/01 09:08:48	192_168_1_8-1348-1	BD_SYB_TEST_1		End Script,
15/03/01 09:08:59				End Test BD_SYB_TEST_1,
15/03/01 09:09:10				End Test BD_SYB_TEST,

Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

Tip: You can export the data displayed in the Test Audit Log by right-clicking within the table and selecting **Export**. The data is exported in CSV format.

See also:

Test Audit Log

Error Reporting and Tracing

Test Report Log

The Test Report log is a sequential text file that is used to record information about a single Test-run. Usually, a single record is written to the Report log whenever a Test case passes or fails.

Additional Report log entries may be written to the log if the Scripts included in the Test have been modeled to incorporate the appropriate SCL code. Use the **REPORT** SCL command in a Script, to generate the data content for the Test Report log. For more information on SCL refer to the *SCL Reference Guide*; an on-line copy is available within the Script Modeler, Help menu.

See also:

Display Test Report Log Data

Error Reporting and Tracing

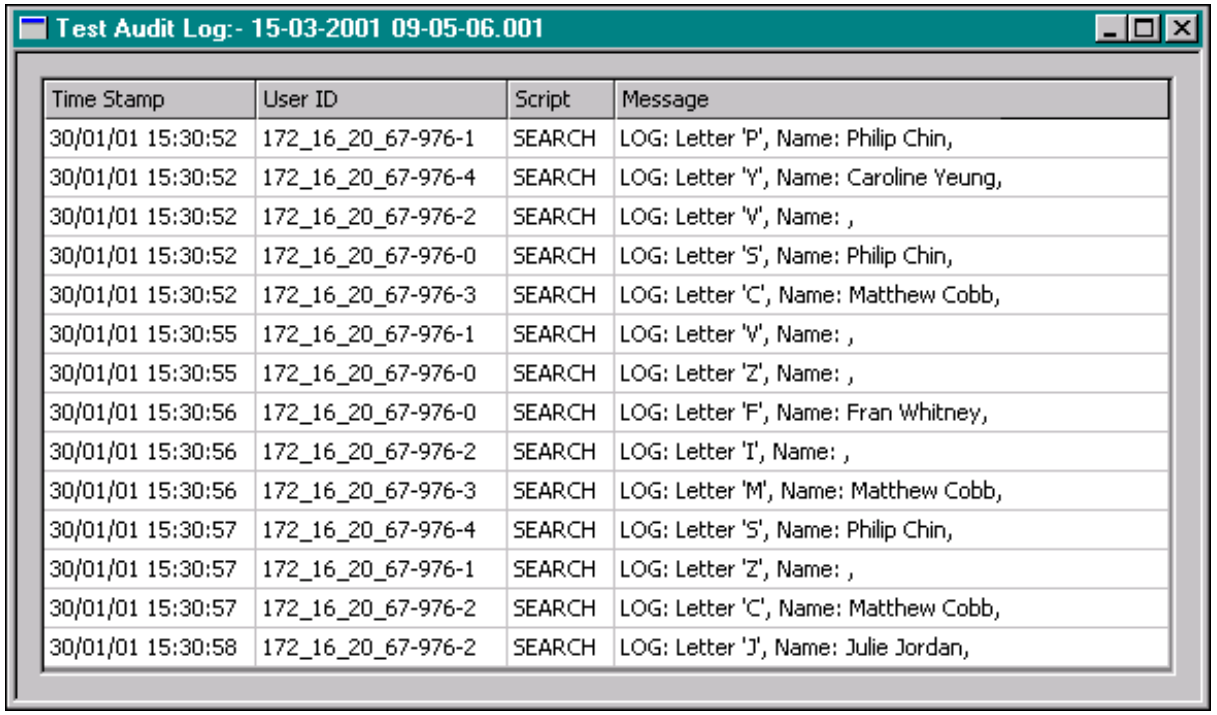
Display Test Report Log Data

1. Open a Test and click the  **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **Test Report Log** results option in the list.

   Report information is displayed in the Results tab in table format:



Time Stamp	User ID	Script	Message
30/01/01 15:30:52	172_16_20_67-976-1	SEARCH	LOG: Letter 'P', Name: Philip Chin,
30/01/01 15:30:52	172_16_20_67-976-4	SEARCH	LOG: Letter 'Y', Name: Caroline Yeung,
30/01/01 15:30:52	172_16_20_67-976-2	SEARCH	LOG: Letter 'V', Name: ,
30/01/01 15:30:52	172_16_20_67-976-0	SEARCH	LOG: Letter 'S', Name: Philip Chin,
30/01/01 15:30:52	172_16_20_67-976-3	SEARCH	LOG: Letter 'C', Name: Matthew Cobb,
30/01/01 15:30:55	172_16_20_67-976-1	SEARCH	LOG: Letter 'V', Name: ,
30/01/01 15:30:55	172_16_20_67-976-0	SEARCH	LOG: Letter 'Z', Name: ,
30/01/01 15:30:56	172_16_20_67-976-0	SEARCH	LOG: Letter 'F', Name: Fran Whitney,
30/01/01 15:30:56	172_16_20_67-976-2	SEARCH	LOG: Letter 'I', Name: ,
30/01/01 15:30:56	172_16_20_67-976-3	SEARCH	LOG: Letter 'M', Name: Matthew Cobb,
30/01/01 15:30:57	172_16_20_67-976-4	SEARCH	LOG: Letter 'S', Name: Philip Chin,
30/01/01 15:30:57	172_16_20_67-976-1	SEARCH	LOG: Letter 'Z', Name: ,
30/01/01 15:30:57	172_16_20_67-976-2	SEARCH	LOG: Letter 'C', Name: Matthew Cobb,
30/01/01 15:30:58	172_16_20_67-976-2	SEARCH	LOG: Letter 'J', Name: Julie Jordan,

Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

Tip: You can export the data displayed in the Test Report Log by right-clicking within the table and selecting **Export**. The data is exported in CSV format.

See also:

Test Report Log

Error Reporting and Tracing

Test History Log

The Test History log is a sequential text file that is used to maintain a chronological history of each occasion on which the Test was run, together with the results of that Test. Usually, a single record is written to the History log when the Test-run is complete.

In addition, further Test History log entries may be written to the log if the Scripts included in the Test have been modeled to incorporate the appropriate SCL code. Use the **HISTORY** SCL command in a Script, to generate the data content for the Test History log. For more information on SCL refer to the *SCL Reference Guide*; an on-line copy is available within the Script Modeler, Help menu.

See also:

Display Test History Log Data

Error Reporting and Tracing

Display Test History Log Data

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **Test History Log** results option in the list.

   History information is displayed in the Results tab in table format.

   Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

   Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

   Tip: You can export the data displayed in the Test History Log by right-clicking within the table and selecting **Export**. The data is exported in CSV format.

See also:

Test History Log

Error Reporting and Tracing

Test Error Log

The Test Error Log records all significant error messages from the Test Manager, Task Group Executers and OpenSTA Daemon.

Data included in the log are: Time Stamp, Test Name, Location and Message.

See also:

Display Test History Log Data

Error Reporting and Tracing

Display the Test Error Log

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **Test Error Log** display option in the list to open it in the Test Pane.

Test Error Log data is displayed in table format.

Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

Note: Click  , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

Tip: You can export the data displayed in the Test Error Log by right-clicking within the table and selecting **Export**. The data is exported in CSV format.

See also:

Test History Log

Error Reporting and Tracing

Test Summary Snapshots

The Test Summary Snapshots option displays a variety of Test summary data captured during a Test-run. Snapshots of Test activity are recorded at defined intervals and summarized in table format. You can set this interval in seconds using the Task Monitor Interval button. The test statistics provided relate mainly to Task and HTTP request behavior. They are particularly useful in determining the number of HTTP requests issued, request duration and the time elapsed between request issue and results receipt during Tests-runs.

See also:

Display Test Summary Snapshots

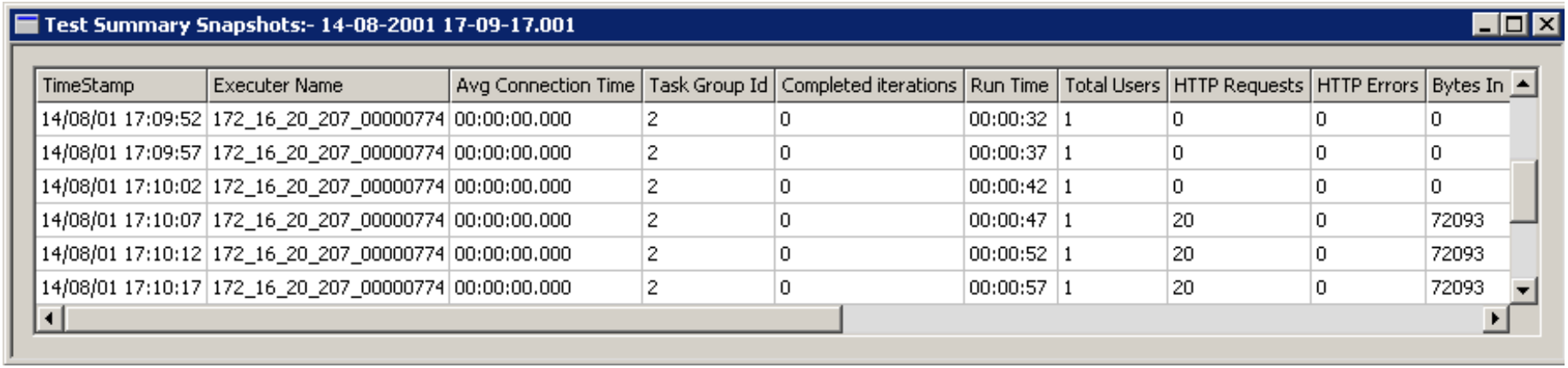
Task Monitoring Interval

Display Test Summary Snapshots

1. Open a Test and click the  **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **Test Summary Snapshots** display option in the list to open it in the Test Pane.
   Test Summary Snapshots data is displayed in table format:



Test Summary Snapshots:- 14-08-2001 17-09-17.001

TimeStamp	Executer Name	Avg Connection Time	Task Group Id	Completed iterations	Run Time	Total Users	HTTP Requests	HTTP Errors	Bytes In
14/08/01 17:09:52	172_16_20_207_00000774	00:00:00.000	2	0	00:00:32	1	0	0	0
14/08/01 17:09:57	172_16_20_207_00000774	00:00:00.000	2	0	00:00:37	1	0	0	0
14/08/01 17:10:02	172_16_20_207_00000774	00:00:00.000	2	0	00:00:42	1	0	0	0
14/08/01 17:10:07	172_16_20_207_00000774	00:00:00.000	2	0	00:00:47	1	20	0	72093
14/08/01 17:10:12	172_16_20_207_00000774	00:00:00.000	2	0	00:00:52	1	20	0	72093
14/08/01 17:10:17	172_16_20_207_00000774	00:00:00.000	2	0	00:00:57	1	20	0	72093

Test Summary Snapshots data categories are:

- **TimeStamp:** Gives the time of the Task execution.
- **Executer Name:** Provides the IP address of the machine on which the test executes.
- **Avg Connection Time:** Shows the average length of time for a TCP connection.
- **Task Group ID:** Shows the ID corresponding to the Task Group.
- **Completed Iterations:** Shows the number of times a task has been executed.
- **Run Time:** Indicates the total execution time of the Task.
- **Total Users:** Gives the total number of users.
- **HTTP Requests:** Shows the total number of HTTP requests within the Task.

  HTTP Errors: Indicates the number of 4XX and 5XX error codes returned from the Web browser after the HTTP request has been sent. These error codes adhere to the World Wide Web Consortium (W3C) standards. For more information visit: **http://w3.org/Protocols/HTTP/HTRESP**.

- **Bytes In:** Gives the number of bytes received for the HTTP request results.
- **Bytes Out:** Shows the number of bytes sent for the HTTP request.
- **Min Request Latency:** Indicates the minimum length of time elapsed in milliseconds between sending an HTTP request and receiving the results.
- **Max Request Latency:** Shows the maximum length of time elapsed in milliseconds between sending an HTTP request and receiving the results.
- **Average Request Latency:** Gives the average length of time elapsed in milliseconds between sending an HTTP request and receiving the results.
- **Task 1(VUs):** Shows the number of virtual users for a Task.
- **Task 1(Iterations):** Gives the number of iterations for a Task.
- **Task 1(Period):** Shows the duration of a Task.
  Note: If your Task Group consists of multiple Tasks, extra columns corresponding to the respective Task numbers are included in the Test Summary Snapshots table.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

Tip: You can export the data displayed in the Test Summary Snapshots table by right-clicking within the table and selecting **Export**. The data is exported in CSV format.

See also:

Test Summary Snapshots

HTTP Data List

The HTTP Data List stores details of the HTTP requests issued by the Scripts included in a Test when it is run. This data includes the response times and codes for all the HTTP requests issued. The amount of HTTP data recorded depends on the Logging level specified for a Script-based Task Group when you created the Test and defined the Virtual User settings to be applied. The Logging level setting controls the number of Virtual Users that statistics are gathered for and can be edited from the Configuration tab of the Test Pane.

The data is presented in a table and can be sorted by clicking on the column headings to reverse the display order of the data entries. These results can also be filtered by right-clicking inside the table and selecting the **Filter** option. Use the **Export** right-click menu option to export data in .CSV text file format which allows them to be imported into other data analysis and report generating tools.

See also:

Display the HTTP Data List

Filter HTTP Data List

Virtual User Settings

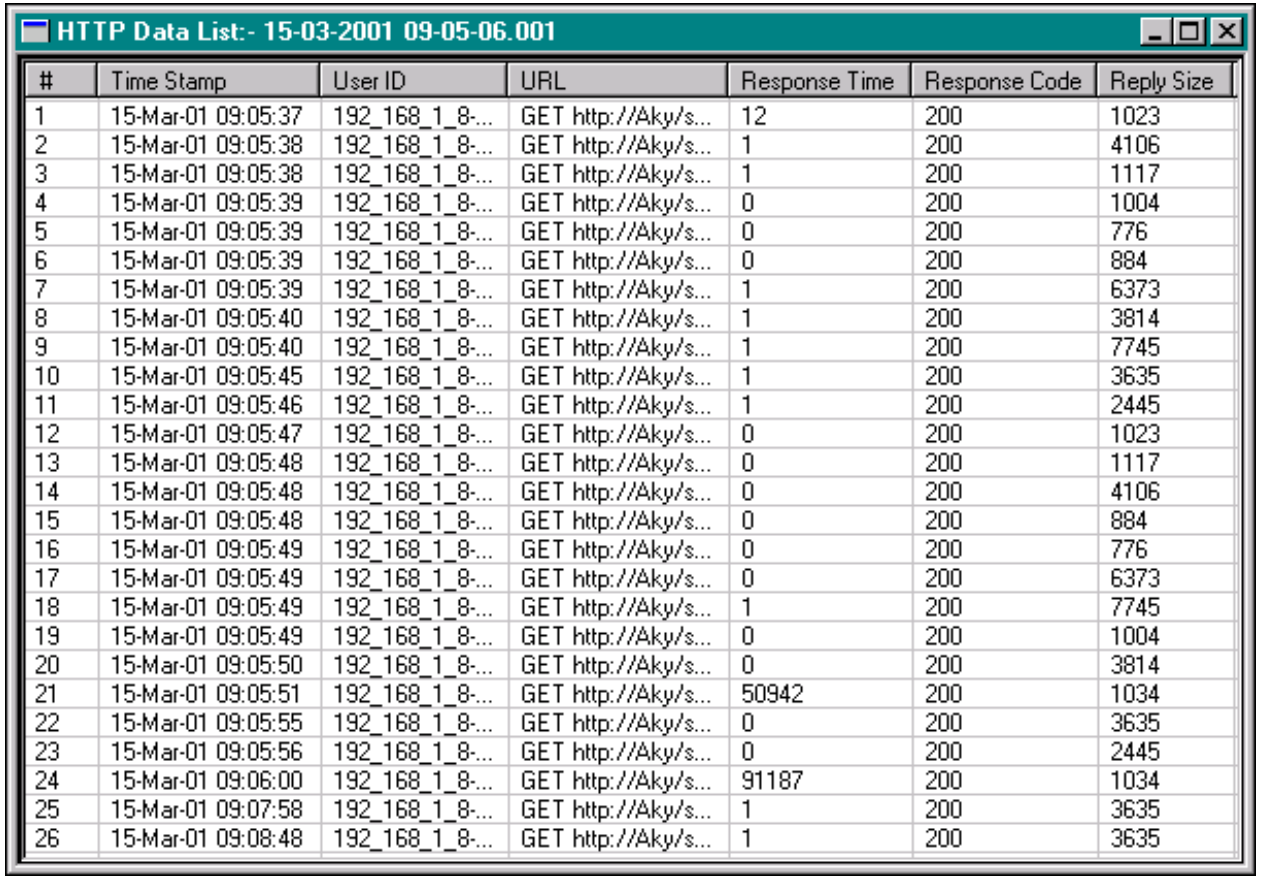
Display the HTTP Data List

1. Open a Test and click the  **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.
3. Click the **HTTP Data List** display option in the list to open it in the Test Pane.

   HTTP Data List information is displayed in table format:



HTTP Data List:- 15-03-2001 09-05-06.001

#	Time Stamp	User ID	URL	Response Time	Response Code	Reply Size
1	15-Mar-01 09:05:37	192_168_1_8-...	GET http://Aky/s...	12	200	1023
2	15-Mar-01 09:05:38	192_168_1_8-...	GET http://Aky/s...	1	200	4106
3	15-Mar-01 09:05:38	192_168_1_8-...	GET http://Aky/s...	1	200	1117
4	15-Mar-01 09:05:39	192_168_1_8-...	GET http://Aky/s...	0	200	1004
5	15-Mar-01 09:05:39	192_168_1_8-...	GET http://Aky/s...	0	200	776
6	15-Mar-01 09:05:39	192_168_1_8-...	GET http://Aky/s...	0	200	884
7	15-Mar-01 09:05:39	192_168_1_8-...	GET http://Aky/s...	1	200	6373
8	15-Mar-01 09:05:40	192_168_1_8-...	GET http://Aky/s...	1	200	3814
9	15-Mar-01 09:05:40	192_168_1_8-...	GET http://Aky/s...	1	200	7745
10	15-Mar-01 09:05:45	192_168_1_8-...	GET http://Aky/s...	1	200	3635
11	15-Mar-01 09:05:46	192_168_1_8-...	GET http://Aky/s...	1	200	2445
12	15-Mar-01 09:05:47	192_168_1_8-...	GET http://Aky/s...	0	200	1023
13	15-Mar-01 09:05:48	192_168_1_8-...	GET http://Aky/s...	0	200	1117
14	15-Mar-01 09:05:48	192_168_1_8-...	GET http://Aky/s...	0	200	4106
15	15-Mar-01 09:05:48	192_168_1_8-...	GET http://Aky/s...	0	200	884
16	15-Mar-01 09:05:49	192_168_1_8-...	GET http://Aky/s...	0	200	776
17	15-Mar-01 09:05:49	192_168_1_8-...	GET http://Aky/s...	0	200	6373
18	15-Mar-01 09:05:49	192_168_1_8-...	GET http://Aky/s...	1	200	7745
19	15-Mar-01 09:05:49	192_168_1_8-...	GET http://Aky/s...	0	200	1004
20	15-Mar-01 09:05:50	192_168_1_8-...	GET http://Aky/s...	0	200	3814
21	15-Mar-01 09:05:51	192_168_1_8-...	GET http://Aky/s...	50942	200	1034
22	15-Mar-01 09:05:55	192_168_1_8-...	GET http://Aky/s...	0	200	3635
23	15-Mar-01 09:05:56	192_168_1_8-...	GET http://Aky/s...	0	200	2445
24	15-Mar-01 09:06:00	192_168_1_8-...	GET http://Aky/s...	91187	200	1034
25	15-Mar-01 09:07:58	192_168_1_8-...	GET http://Aky/s...	1	200	3635
26	15-Mar-01 09:08:48	192_168_1_8-...	GET http://Aky/s...	1	200	3635

Tip: Right-click within the table and use the menu options to **Filter** and **Export** the data.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

See also:

Filter HTTP Data List

Export Test Results

HTTP Data List

Filter HTTP Data List

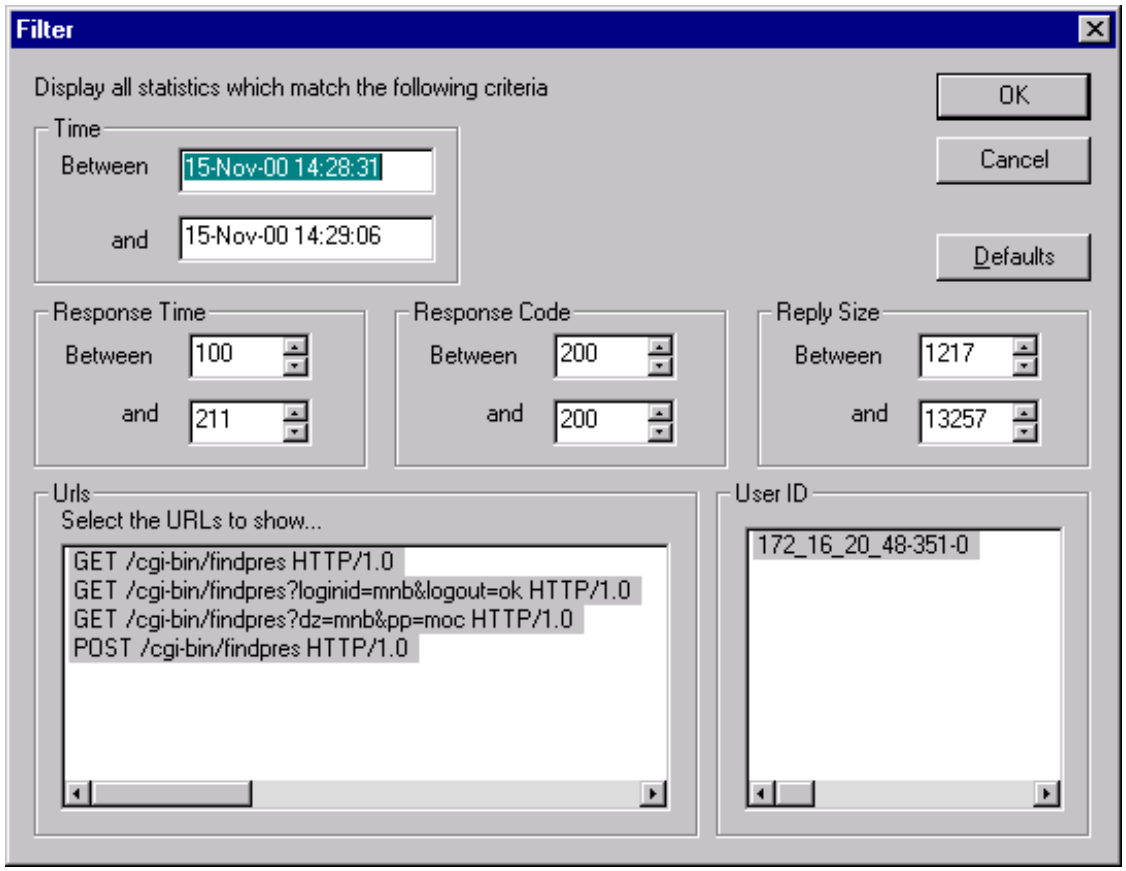
1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.

3. Click on a graph display, results option to open your selection in the Test Pane.
4. Click  , in the toolbar or right-click inside the graph and select **Filter**.

   The Filter dialog box offers a variety of selection criteria as illustrated:



5. The filter criteria available correspond to the column categories in the HTTP Data List table. Select your settings from the filter options:

- **Time**: Date and time HTTP GETs and POSTs were issued.
- **Response Time**: Web site response time to GETs in milliseconds.
- **Response Code**: Code issued by Web site in response to GETs.
- **Reply Size**: Size of data response to GETs issued by Web site in bytes.
- **URLs**: Filter by URL.
- **User ID**: Filter by Virtual User(s) identity (IP address).

6. Select the filter options you want then click **OK** to apply them.

   Note: Click the **Defaults** button to apply the original filter settings, which reflect the full range of data measurements of the HTTP

Data List listed.

Note: The filter settings you apply are not saved when you close the table.

See also:

Display the HTTP Data List

HTTP Data List

HTTP Data Graphs

The volume of HTTP data recorded is controlled by the Logging level you set for a Task Group's Virtual Users. The Logging level determines the number of Virtual Users that data is collected for and controls the quality of the data displayed in the graphs. The HTTP data collected relates only to responses to HTTP requests issued as part of Test.

The HTTP data collected during a Test-run can be displayed in a number of different graphs where you can scrutinize your Test results. There are seven graphs in total which you can display using the Results Window.

Right-click within a graph and select to **Customize**, **Export to Excel Filter URLs**.

See also:

Display HTTP Data Graphs

Filter URLs in HTTP Data Graphs

Customize Graph Display

HTTP Response Time (Average per Second) v Number of Responses Graph

HTTP Errors v Active Users Graph

HTTP Errors v Elapsed Time Graph

HTTP Responses v Elapsed Time Graph

HTTP Response Time v Elapsed Time Graph

HTTP Active Users v Elapsed Time Graph

Virtual User Settings

Display HTTP Data Graphs

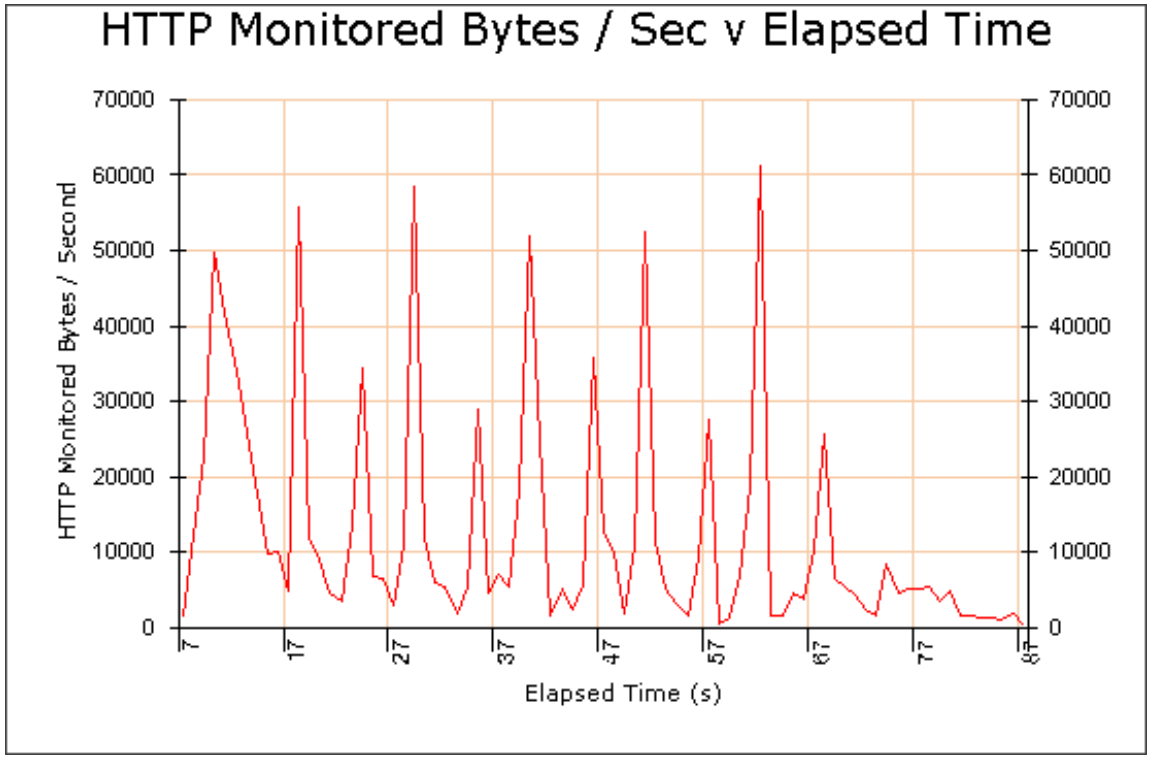
1. Open a Test and click the  **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, click ⊞ next to a Test-run folder or double-click on it to open the folder and display the available results.

3. Click on an HTTP data list option such as **HTTP Monitored Bytes / Second v Elapsed Time** to open your selection in the Test Pane.

   This graph shows the total number of bytes per second returned during the Test-run.



   Note: Graphs are displayed in the default line plot style.

   Tip: Right-click within the graph and use the menu options to **Customize**, **Filter URLs** and **Export to Excel**.

   Note: Click  , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

See also:

Filter HTTP Data List

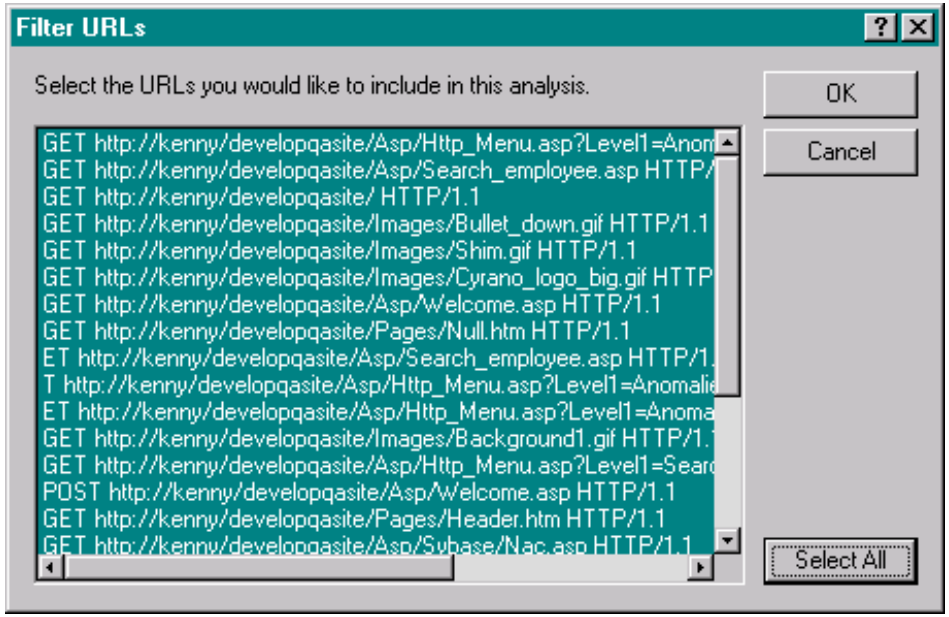
Customize Graph Display

HTTP Data Graphs

Filter URLs in HTTP Data Graphs

1. Open a Test and display an HTTP data graph in the Test Pane.
2. Click  , in the toolbar or right-click inside a graph then select **Filter URLs**.

Use the Filter URLs dialog box to select the URLs you want to display. Click **Select All** to display all the URLs.



3. In the Filter URLs dialog box select the URLs you want to view.
4. Click **OK** to display the selected URLs.

   Note: The filter settings you apply are not saved when you close the table.

See also:

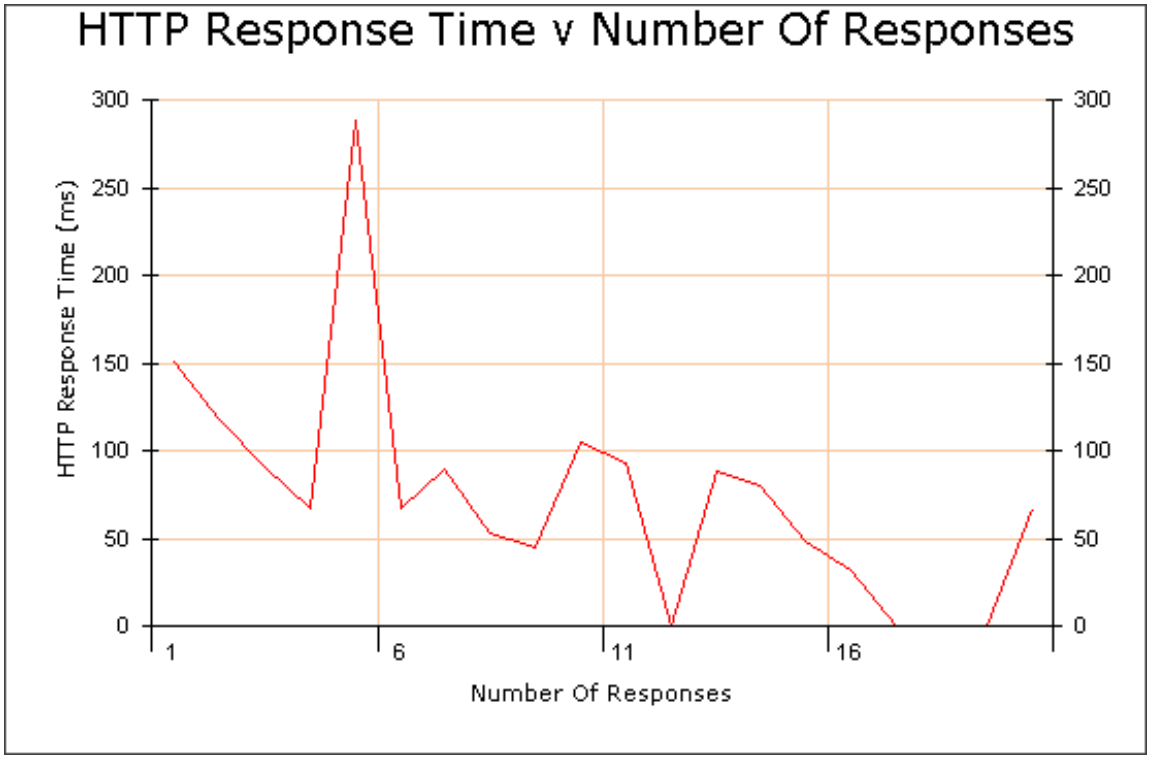
Display HTTP Data Graphs

Customize Graph Display

HTTP Data Graphs

HTTP Response Time (Average per Second) v Number of Responses Graph

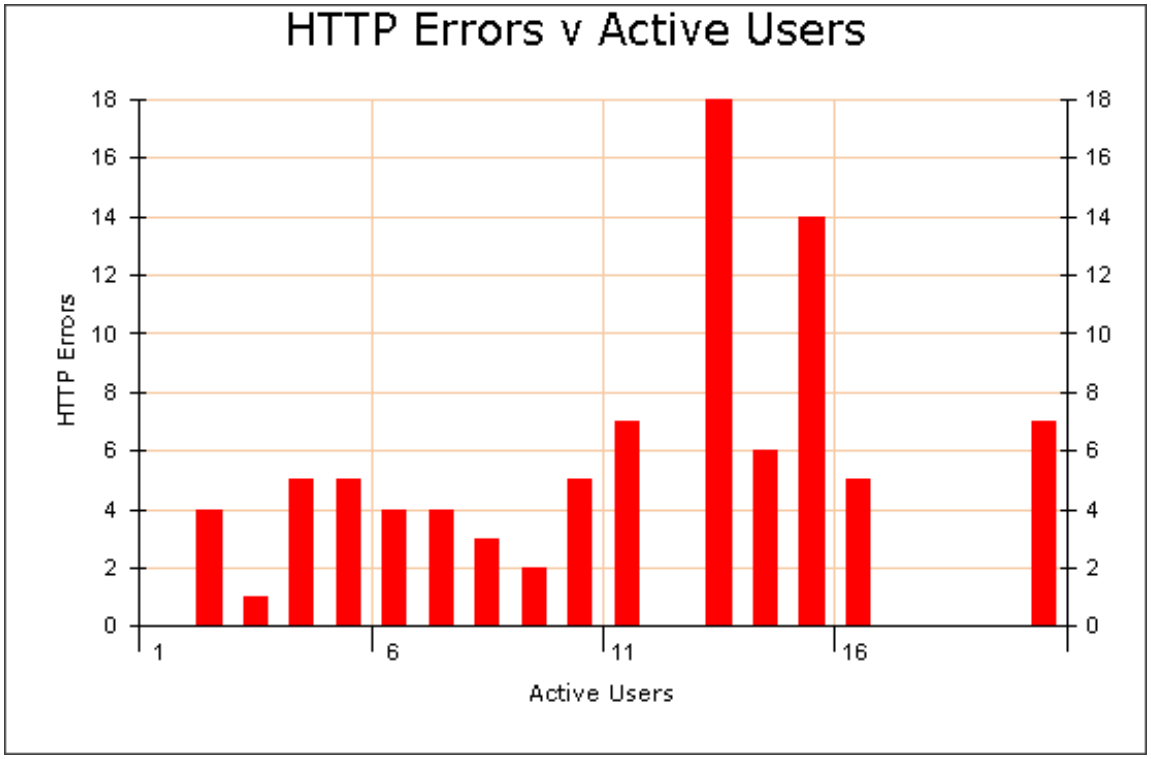
This graph displays the average response time for requests grouped by the number of requests per second during a Test-run.



Tip: Right-click within the graph and use the menu options to **Customize**, **Filter URLs** and **Export to Excel**.

HTTP Errors v Active Users Graph

This graph is used to display the effect on performance measured by the number of HTTP server errors returned as the number of active Virtual Users varies during a Test-run.

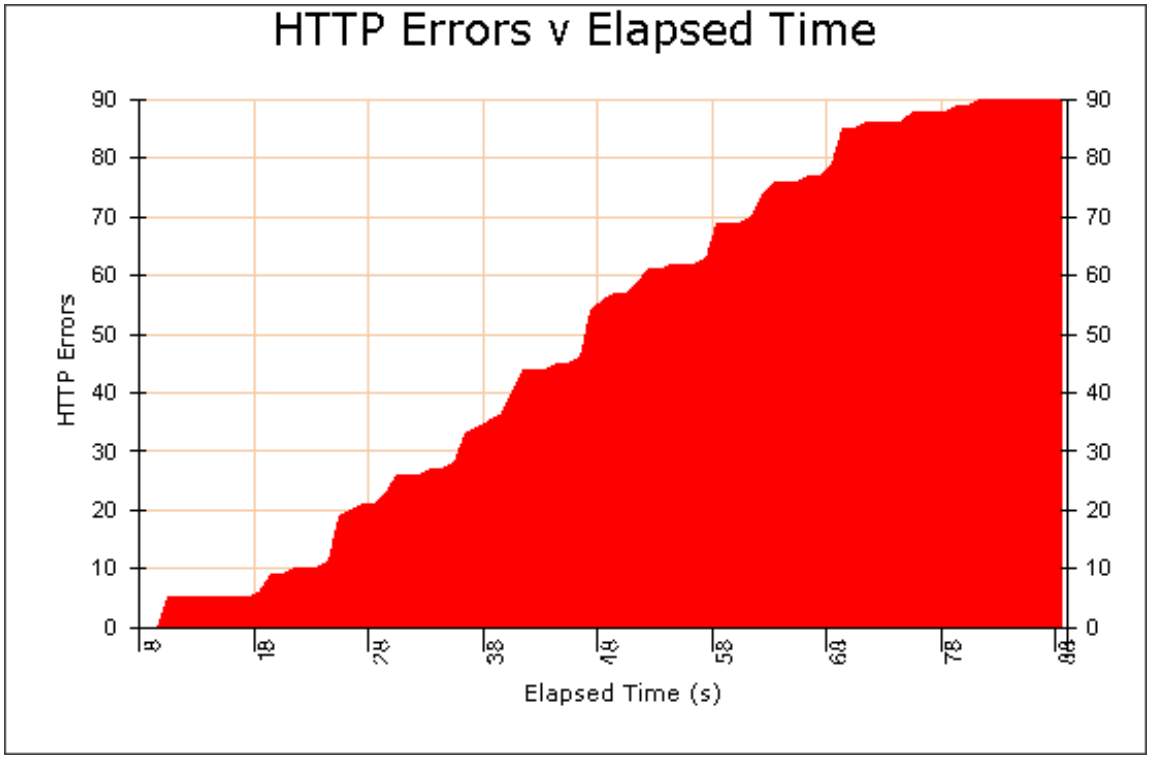


Note: This graph has been customized to display data points as vertical bars. Right-click within a graph and select **Customize**, then select Graph Type, **Vertical bars**.

Make use of the **Filter URLs** and **Export to Excel** options associated with this graph by right-clicking within it.

HTTP Errors v Elapsed Time Graph

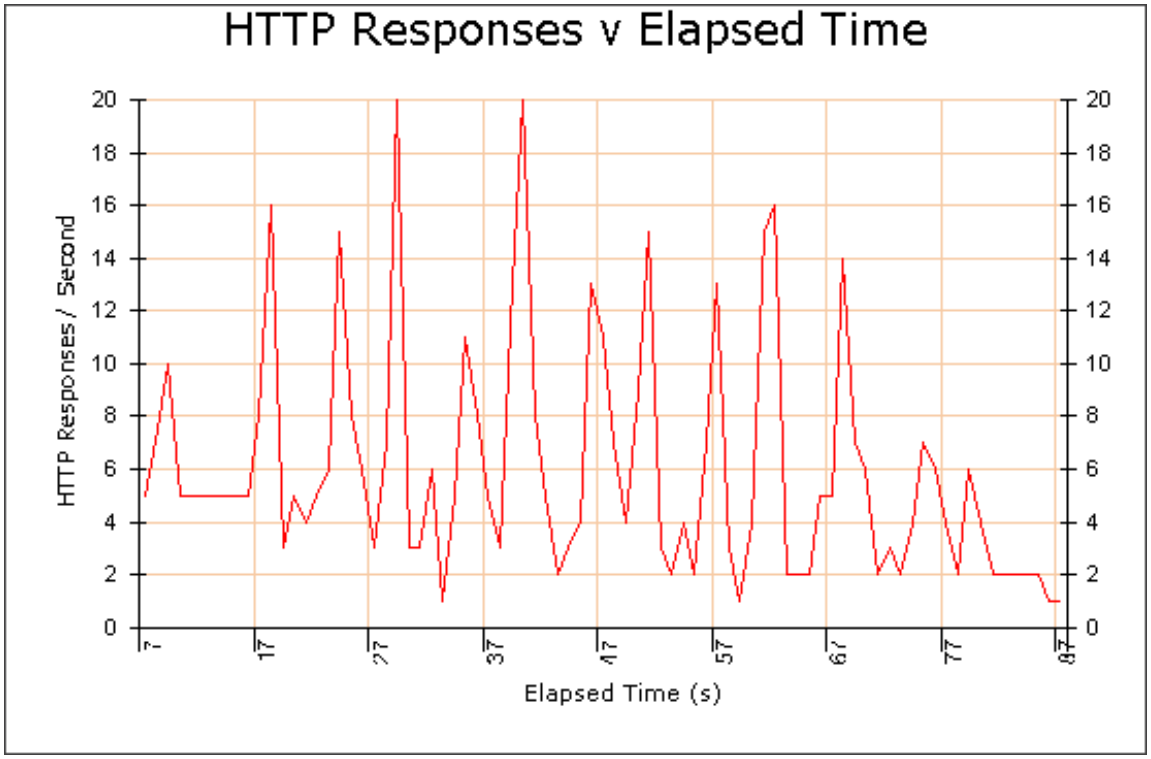
This graph displays a cumulative count of the number of HTTP server errors returned during the Test-run.



Note: This graph has been customized to display the area under the data points as a solid. Right-click within a graph and select **Customize > Area under points** from the menu to change the appearance of your graphs.

HTTP Responses v Elapsed Time Graph

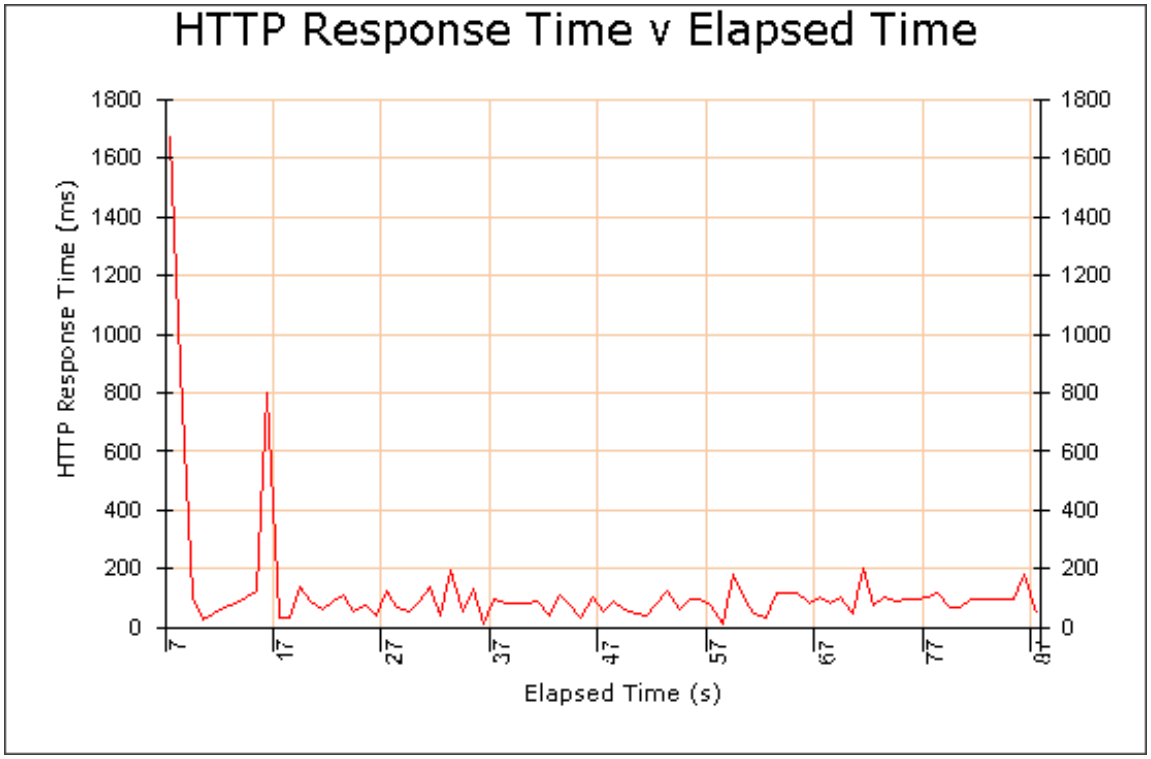
This graph displays the total number of HTTP responses per second during the Test-run.



Right-click within a graph and select to **Customize** or **Export to Excel**.

HTTP Response Time v Elapsed Time Graph

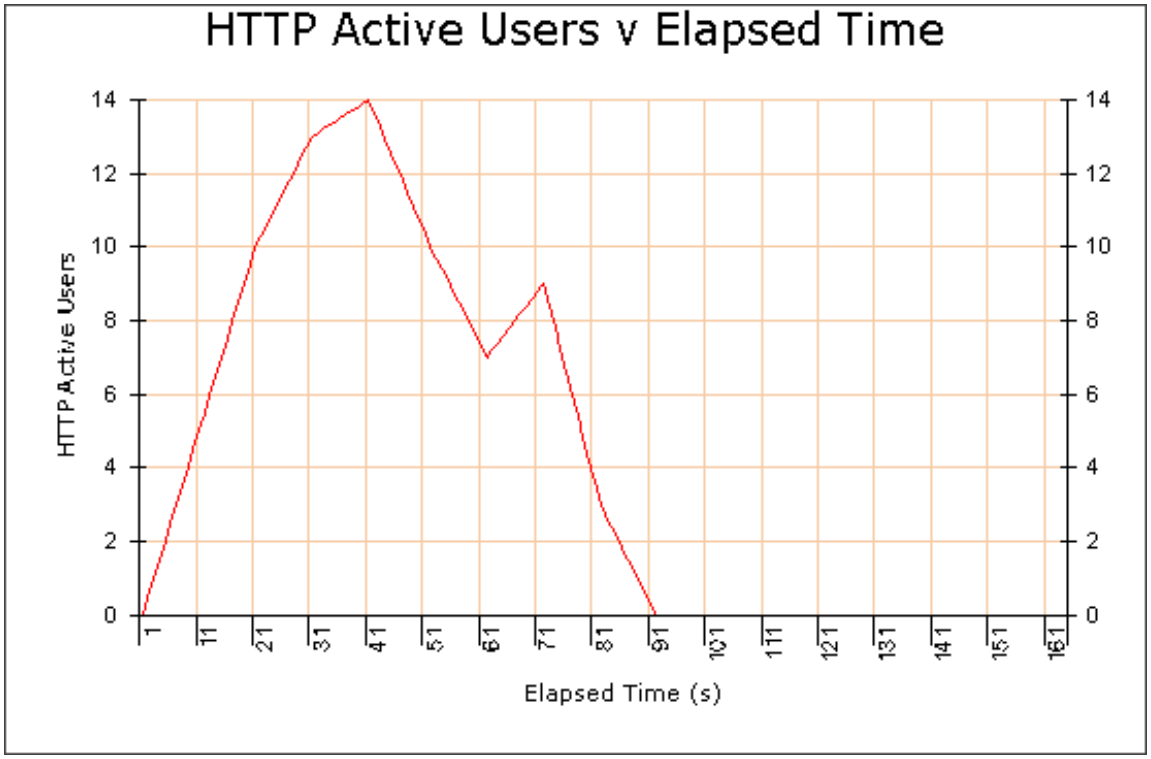
This graph displays the average response time per second of all the requests issued during the Test-run.



Use the right-click menu options to **Customize**, **Export to Excel Filter URLs**.

HTTP Active Users v Elapsed Time Graph

This graph displays the total number of active Virtual Users sampled at fixed intervals during a Test-run.



Right-click within the graph and use the menu options to **Customize** or **Export to Excel**.

Single Step Results

During Test development it is important to check that a Test runs correctly. You can run a single stepping session to help verify a Test by monitoring Task Group replay to check that the WAE responses are appropriate. Then use the Single Step Results option to analyze the results data obtained. The data includes the HTTP requests issued to a target WAE and the HTTP returned in response during a single stepping session.

Single stepping a Test is a useful method to help you verify that a Test with a modular structure runs as you expect. A modular Test incorporates two or more Scripts in one Task Group to simulate a continuous Web browser session when the Test is run and requires some modeling of the Scripts included. After single stepping the Task Group that contains the Script sequence, open up the Single Step Results option and double-click on an HTTP request to display the request details.

View the details of the HTTP request in response to which the first cookie was issued during a Test-run. In the Response Header section of the Request Details window look for the **Set-Cookie** entry and make a note of the cookie ID including its name and value. Then view first request included in the next Script in the sequence and look in the Request section of the Request Details window for the **Cookie** entry. The cookie ID recorded here should be the same as the first cookie value issued at the end of the previous Script. Ensure that the value of the last cookie issued in each Script is handed onto the next Script in the sequence, for all the Scripts in the Task Group.

See also:

Display Single Step Results

Single Stepping

Timer List

Developing a Modular Test Structure

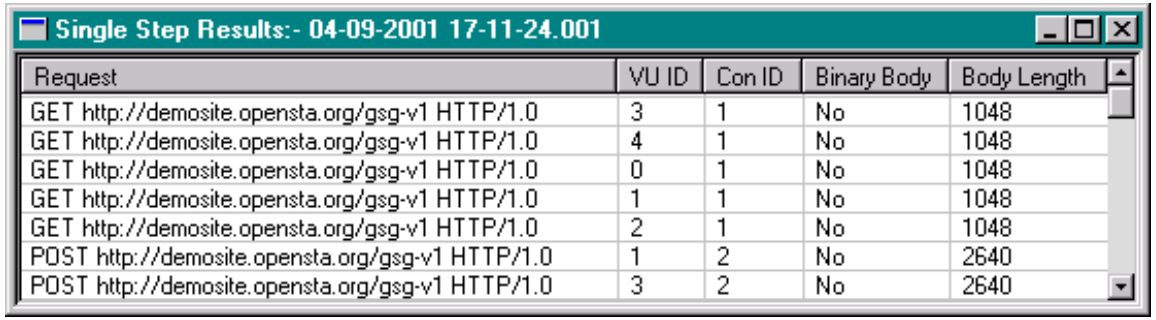
Display Single Step Results

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, double-click on a single stepping Test-run folder or click ⊞, to open it and display the available results.
3. Click the **Single Step Results** display option to open your selection in the Test Pane.

   Single step results are displayed in table format:

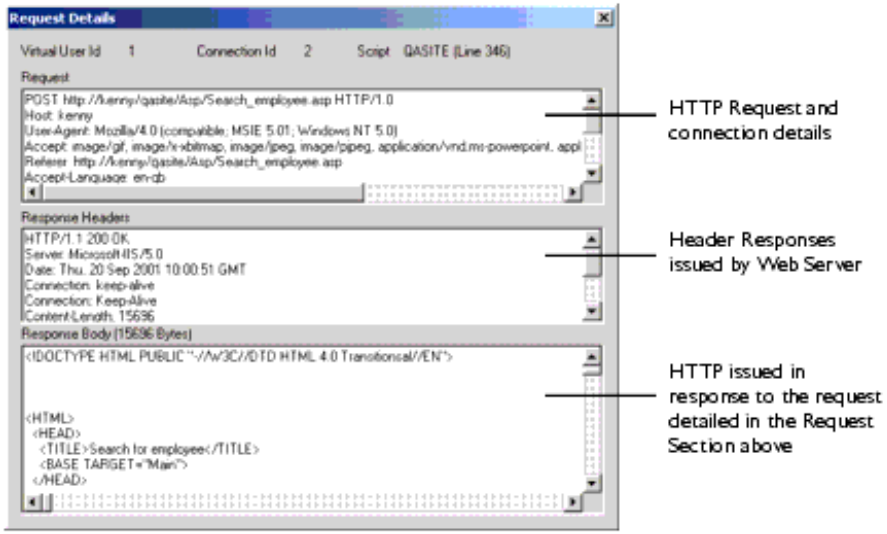


Request	VU ID	Con ID	Binary Body	Body Length
GET http://demosite.opensta.org/gsg-v1 HTTP/1.0	3	1	No	1048
GET http://demosite.opensta.org/gsg-v1 HTTP/1.0	4	1	No	1048
GET http://demosite.opensta.org/gsg-v1 HTTP/1.0	0	1	No	1048
GET http://demosite.opensta.org/gsg-v1 HTTP/1.0	1	1	No	1048
GET http://demosite.opensta.org/gsg-v1 HTTP/1.0	2	1	No	1048
POST http://demosite.opensta.org/gsg-v1 HTTP/1.0	1	2	No	2640
POST http://demosite.opensta.org/gsg-v1 HTTP/1.0	3	2	No	2640

   Single Step Results data categories are:

- **Request:** Displays the HTTP request details.
- **VU ID:** Gives the ID of the Virtual User associated with the HTTP request.
- **Con ID:** Shows the Connection ID corresponding to the number of connections to the Web Server.
- **Binary Body:** Indicates whether the file loaded in response to the HTTP request is binary or non-binary.
- **Body Length:** Gives the size in bytes of the file loaded in response to the HTTP request.

4. Double-click on a request to display more details about your selection.



The HTTP data in the Response Body section is the same data displayed in the HTTP section when you replay a Task Group during a single stepping session.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

See also:

Single Stepping

Timer List

The Timer List file gives details of the Timers recorded during a Test-run. Timer results data records the time taken to load each Web page specified by a Script for every Virtual User running the Script during a Test-run. The level of Timer information recorded is controlled by adjusting the Virtual User settings in the Test's Script-based Task Groups. Open the Test with the Configuration tab of the Test Pane displayed, then click on a **VUs** table cell in a Task Group and check the activate the Generate Timers for each page option in the Properties Window. The Logging level you select here controls the volume of HTTP data and the number of timers recorded.

The information collected is presented in a table and can be sorted by clicking on the column headings to reverse the display order of the data entries.

Timer List can be exported to a .CSV text file which allows results to be imported into many other data analysis and report generating tools.

See also:

Display the Timer List

Timer Values v Active Users Graph

Timer Values v Elapsed Time Graph

Virtual User Settings

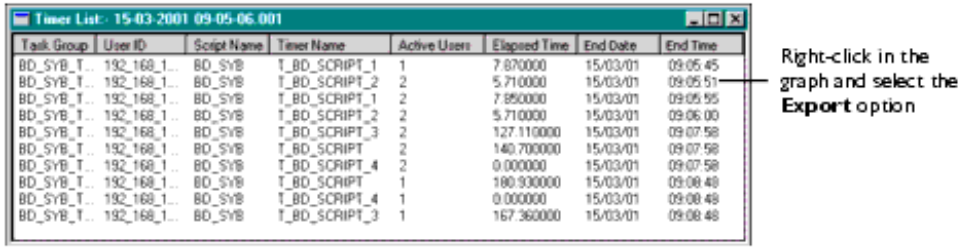
Display the Timer List

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, double-click on a Test-run folder or click ⊞ , to open it and display the available results.
3. Click the **Timer List** display option to open your selection in the Test Pane.

   Timer List information is displayed in table format:



   Note: Right-click within the graph and select **Export** to save the data to a .CSV text file, which allows results to be imported into other data analysis and report generating tools.

   Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

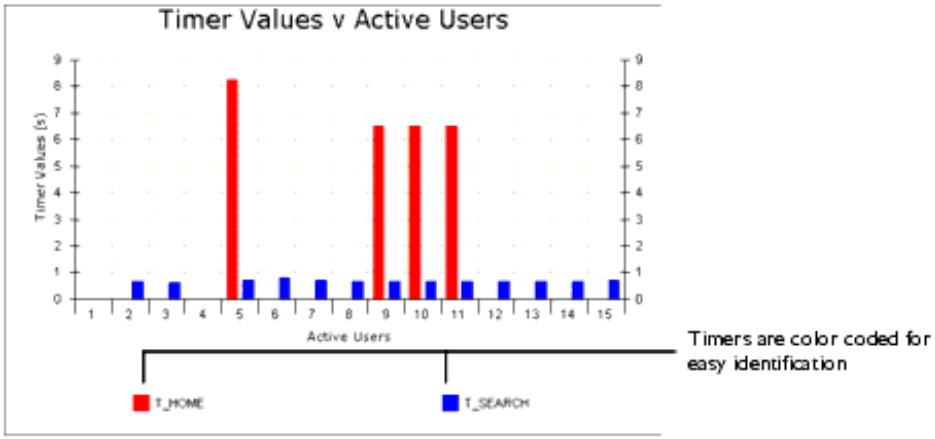
   Note: Click ☒ , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

   Tip: To improve the display of your results use the Customize, option to display your data in vertical bars style. If your timer names and color coding key is not displayed, you can maximize the display area by double-clicking in the title bar of the graph.

Timer Values v Active Users Graph

This graph is used to display the effect on performance as measured by timers, as the number of Virtual Users varies.

You can control the information displayed by filtering the timers. The Select Timers to display dialog box appears when you choose this option from the Results Window. Use it to select the timers you want to view, then click **OK** to proceed.

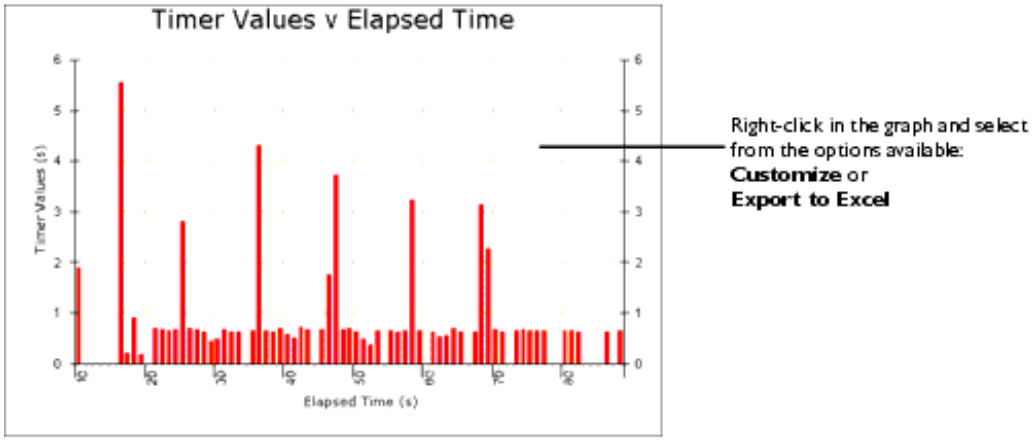


Right-click within a graph and select to **Customize**, **Export to Excel Filter URLs**.

Timer Values v Elapsed Time Graph

This graph is used to display the average timer values per second.

You can control the information displayed by filtering the timers. The Select Timers to display dialog box appears when you choose this option from the Results Window. Use it to select the timers you want to view, then click **OK** to proceed.



Right-click within a graph and select to **Customize**, **Export to Excel Filter URLs**.

SNMP and NT Performance Collector Graphs

The data collection queries defined in a Collector generate results data that can be displayed in custom graphs. A maximum of two custom graphs are produced per Test-run. All NT Performance Collector data is displayed in the Custom NT Performance graph. All SNMP Collector data is displayed in the Custom SNMP graph.

If your Test includes more than one NT Performance or SNMP Collector, the appropriate custom results graph combines the data collection

queries from all Collectors of the same type and displays them in one graph which you can then filter to display the data you require.

Use the **Filter** option to select and display specific data collection queries defined in the Collectors. The unique names you assigned to each query are displayed below the graph in a color coded key. The IP address of the Host used to run the Collector Task during a Test-run is automatically listed alongside the query name.

Right-click within a graph and select to **Customize**, **Export to Excel Filter**.

See Also:

Display Custom Collector Graphs

Filter Custom Collector Graphs

Custom SNMP Graph

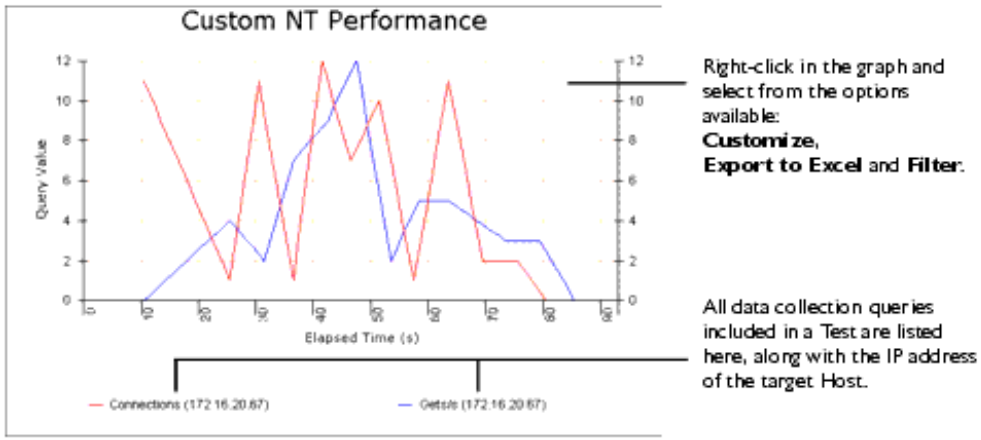
Display Custom Collector Graphs

1. Open a Test and click the **Results** tab in the Test Pane.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

2. In the Results Window, double-click on a Test-run folder or click ⊞ , to open it and display the available results.
3. Click the **Custom NT** or **Custom SNMP** from the list results option to open your selection in the Test Pane.

   The Custom NT Performance Graph is displayed below:



Note: Graphs are displayed in the default line plot style. Right-click within a graph and select **Customize** from the menu to change their appearance.

Tip: Right-click within the graph and use the menu options to **Customize**, **Export to Excel** and **Filter** the data.

Tip: Display multiple graphs and tables concurrently to compare results using the Results Window.

Note: Click ☒ , in the Title Bar of a graph or table to close it or deselect the display option in the Results Window.

See Also:

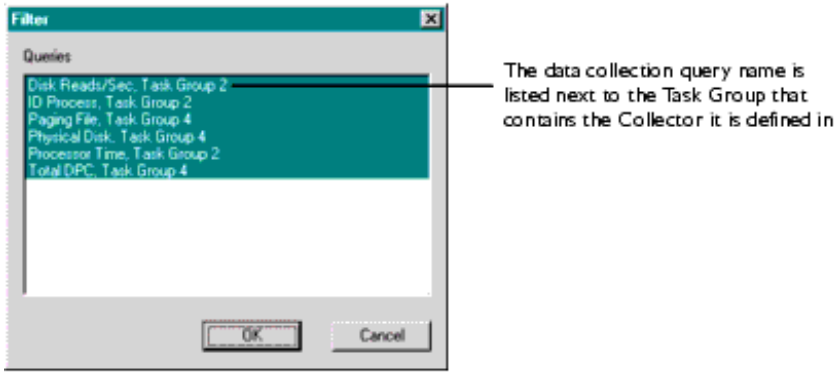
Filter Custom Collector Graphs

Custom SNMP Graph

Filter Custom Collector Graphs

1. Open a Test and display a Custom Collector graph in the Test Pane.
2. Click  , in the toolbar or right-click inside a custom graph then select **Filter**.

   Use the Filter dialog box to select the data collection queries you want to display.



   Note: If you have more than one Collector of the same type referenced in a Test, all the results collected are merged and displayed in one custom graph.

   The Filter dialog box displays the data collection queries alongside the Task Group name indicating which Collector a data collection query belongs to.

3. In the Filter dialog box select the data collection queries you want to view.
4. Click **OK** to display the selected queries.

   Note: The filter settings you apply are not saved when you close a graph.

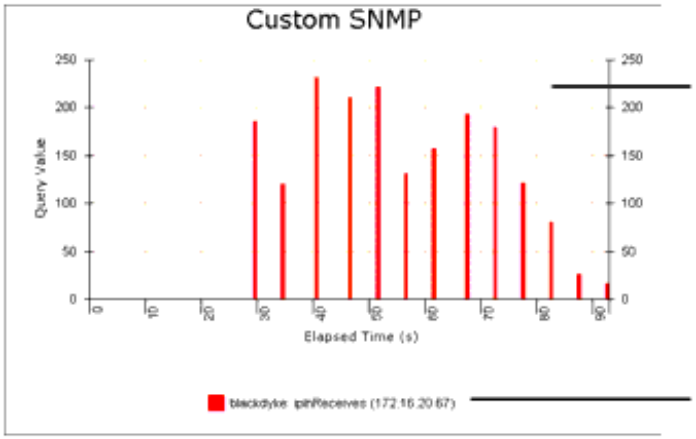
See Also:

Display Custom Collector Graphs

Custom SNMP Graph

Custom SNMP Graph

The Custom SNMP graph displays results returned by all the SNMP Collectors executed during a Test-run. You can filter the data collection queries displayed to control the amount of data displayed.



The data collection queries as defined in the Collectors referenced by a Test are color coded for easy identification. Each query displays the IP address of the Host targeted during a Test- run.

There is a right-click menu associated with the custom graph. Use the **Customize** option to change the appearance of the graph. Other options include the **Export to Excel** option which enables you to convert data for further analysis and output, and the **Filter** option which is used to display specific data collection queries.

See Also:

Display Custom Collector Graphs

Filter Custom Collector Graphs

Single Stepping

Single Stepping HTTP/S Load Tests

Make use of the single stepping functionality provided during Test development to check your HTTP/S load Tests and to help resolve errors that may occur during a Test-run.

When you run a Script-based Task Group within a single stepping session HTTP is returned from target WAEs in response to the Scripts that are executed during replay. You can single step through the browser requests contained in Scripts and monitor HTTP responses to check that the Task Group is behaving as required.

The main difference between single stepping a Script-based Task Group and a normal Test-run, is that replay can be paused by inserting breakpoints against the HTTP requests included in Scripts. When a breakpoint is reached Task Group replay is halted allowing you to study the HTTP returned from the target WAE for all HTTP requests issued before the breakpoint. Breakpoints can be inserted before and during a Test-run, or you can single step through the Test-run using the Single Step button  on the toolbar.

The number of Virtual Users running a Task Group can be set within the session, either to the number previously configured, or to one, for the duration of the replay. Cutting down the number of Virtual Users reduces the amount of HTTP data returned, which is useful if you are running a large volume load Test that involves hundreds or thousands of Virtual Users.

You can also select the Script items you want to view during replay using the toolbar display buttons to help make monitoring easier. Comments, Transaction Timers and Call Scripts need to be manually inserted by modeling Scripts before running a single stepping session. Comments are useful to help explain the HTTP requests and other Script content during replay. Transaction Timers are used to measure the duration of user-defined HTTP transactions when a Test is run. The Call Script command enables you to execute a Script that is not included in the Task Group. For more information on preparing Scripts for a single stepping session see General Modeling Procedures.

When you replay a Task Group during a single stepping session it runs in the same way as a normal Test-run. It is executed according to existing Task Group settings and any changes you may have made to the Virtual User settings from within the single stepping session.

Results collected during a single stepping session are unrealistic in comparison to data from a a normal Test-run. In this mode the continuous replay characteristic of a regular Test-run, is disrupted by breakpoints and some Task Group settings are overridden. Single Stepping results can assist you in Test development but are unreliable for WAE performance analysis.

See also:

Single Stepping Procedure

Add a Comment to a Script for Display During a Single Stepping Session

Add a Transaction Timer to a Script

Modify Wait Command Values in a Script

Call a Script

Single Step Results

Single Stepping Procedure

Begin a single stepping session by opening a Test, right-clicking on a Script-based Task Group in the Test table, then selecting **Single Step Task Group** from the menu.

The next step is to configure the session so that the Task Group runs as required. Run and monitor the replay then make use of the results generated.

Configuration

Use the Monitoring tab to setup your Scripts before running the Task Group. Configuration includes selecting the Script items to display, inserting

breakpoints, controlling the HTTP that is displayed during replay and selecting the number of Virtual Users to run the Task Group.

The Script Item list in the Monitoring tab displays a summary of a Script's content. HTTP requests are always displayed, whereas Secondary Gets, Timers, Waits, Comments, Transaction Timers and Call Scripts can be shown or hidden. Use the Monitoring tab toolbar buttons to show and hide the Script items.

You can double-click on a Script item to switch to the Script tab and display the SCL commands associated with your selection, or right-click and select **Go to Script**. The Script tab displays the full content of a Script in a read-only view. If you need to model a Script open it up from the Repository Window and use Script Modeler and make your changes.

Insert the breakpoints you need by right-clicking on an HTTP request, then selecting **Insert/Remove Breakpoint**. Breakpoints can also be added during replay.

You can control the HTTP responses that are returned for all the requests issued during Task Group replay using the HTTP check boxes. Make sure that the check box next to a request is checked before you run the Task Group in order to monitor and record the HTTP returned.

The amount of HTTP data recorded is also affected by the number of Virtual Users running the Task Group. You can select to apply the number of Virtual Users the Test was originally configured to include or you can run a single Virtual User.

Running

After configuring your Task Group it is ready to run. When you click the Run button  in the Monitoring tab toolbar the Task Group is replayed according to the Timer values contained in the Scripts. Replay is halted when a breakpoint is reached allowing you to view the WAE HTTP responses, which are displayed in the HTTP section at the bottom of the Monitoring tab.

If you are using the single step method there is no need to add breakpoints in the Script Item list before running a Task Group. Simply click on the Monitoring tab, then click  to run the Task Group. After an HTTP request is issued and the HTTP response is complete, replay is automatically paused. Move through the Task Group from one HTTP request to the next by clicking  until replay is complete. You can click  at any stage to revert to continuous replay and use the Break button  in the toolbar to insert breakpoints and pause the run.

Some Task Group settings are overridden during replay. A Task Group is run only once during a single stepping session so if you have configured a Task Group iteration setting using the Schedule settings option, these parameters

are not applied. Script iteration settings configured using the Task settings option are applied during replay.

Also the Virtual User settings relating to Batch Start Options are overridden during replay. When breakpoints are reached the continuous replay of the Task Group is halted. This means that if you have configured a staggered start for the Virtual Users in order to ramp up the load generated, after reaching a breakpoint all Virtual Users are restarted simultaneously.

Monitoring

During a single stepping session Task Group replay can be monitored from the List tab which displays a list of Script items, or the Script tab located within the main Monitoring tab display which displays the SCL commands that correspond to the Script Items. As replay proceeds HTTP requests are consecutively highlighted in yellow to indicate the progress of the run. The corresponding HTTP is displayed in the HTTP section below the Script item list.

You can choose to monitor a specific Virtual User from these tabs, or you can display an approximation of the activity of all Virtual Users, depending whether you selected to run a single Virtual User or all configured Virtual Users within the session. Use the Virtual User and Script selection boxes in the Monitoring tab to pick a Virtual User or Script to monitor during replay.

The Users tab appears to the right of the Script tab when you replay a Task Group. It lists all the Virtual Users involved in the replay and supplies data for each one. The HTTP request, Script name and Script line number that each Virtual User is currently running is displayed. Double-click on a Virtual User to monitor your selection in the List tab or Script tab.

Task Group replay can also be monitored in the same way as a normal Test-run using the Monitoring Window to select and display the options you need. Default categories of performance data are recorded as usual, along with results data using the Collectors that you may have included in the Test.

Task Group replay must be stopped before you can end a single stepping session. Stop Task Group replay from the Monitoring tab by clicking  in toolbar, then click  to end the single stepping session.

Results

The WAE responses which are displayed in the HTTP section at the bottom of the Monitoring tab during replay are recorded for later display and analysis.

After replay is complete, results can be accessed from the Results Window. Open up a single stepping Test-run folder  and select the **Single Step Results** option.

See also:

The Single Stepping Test Pane

Single Stepping a Script-based Task Group

The Single Stepping Test Pane

Use the Single Stepping Test Pane to configure and run a Script-based Task Group. Apply the Task Group settings you require to control how it behaves during replay.

Run and monitor the Task Group then display your results for analysis.

The Single Stepping Test Pane is displayed in the Main Window when you begin a single stepping session. It has two sections represented by the following tabs:

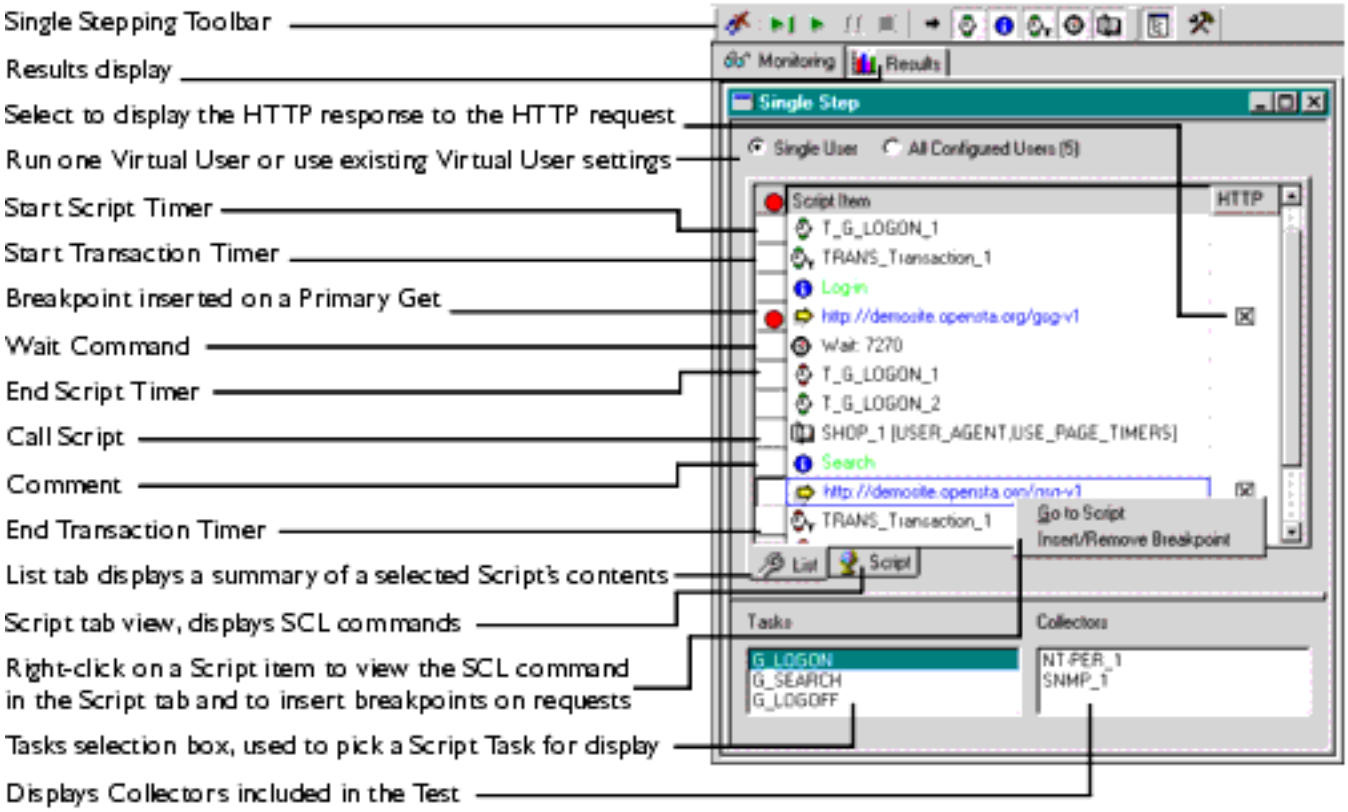
-  **Monitoring**: This is the default view when you begin a single stepping session and is the workspace used to configure a Task Group and monitor replay. Use this tab to select and configure the Scripts included in the Task Group by controlling the Script items that are displayed and inserting the breakpoints you need. Select the number of Virtual Users to run the Task Group and pick the requests whose HTTP responses you want to monitor and record.

  Run and monitor Task Group replay from this tab. For more information, see Running Tests.

-  **Results**: Use this tab to view the results collected during Task group replay in graph and table format. Use the Results Window to select and view the display options available. For more information, see Results Display.

Single Stepping Test Pane Features

The Monitoring tab view of the Single Stepping Test Pane is displayed below:



See also:

Single Stepping a Script-based Task Group

Single Stepping a Script-based Task Group

Note: Before beginning a single stepping session you should compile the Test by clicking  , in the toolbar, to check that the Scripts it contains are valid.

1. Open a Test with the  **Monitoring** tab of the Single Stepping Test Pane displayed.
2. Right-click on a Script-based Task Group and select **Single Step Task Group** from the menu.

   The first Script Task in a sequence is displayed by default in the Monitoring tab.
3. The Script or sequence of Scripts included in the Task Group are listed in the Tasks selection box at the bottom of the window.

   Click on a Script Task to display it in the workspace above.
4. The top section of the window displays the List tab view by default which includes a list of Script items included in the selected Script.

   Use the toolbar to select the Script items you want to display. Primary HTTP requests are always displayed. Choose from:

Secondary URLs **Timers**
Comments **Transaction Timers**
Waits **Call Scripts**

Note: The display options you select apply to all the Scripts in the Task Group.

Tip: Use the Script tab to view the SCL commands that constitute the Script.
You can double-click on any Script item to display the SCL commands associated with your selection in the Script tab.

5. Insert a breakpoint on an HTTP request by right-clicking on the request then selecting **Insert/Remove Breakpoint**.

   Breakpoints can be inserted on Primary HTTP requests and Secondary Gets. They are indicated by to the left of the HTTP request.

   Note: Breakpoints inserted using this method are saved after you end the single stepping session. Breakpoints inserted using the Break button are temporary and not saved.

6. Use the HTTP check boxes to the right of an HTTP request to control whether HTTP responses are displayed when the Task Group is run. By default the check boxes are checked and HTTP is returned for all requests.

   Click on a check box to check or uncheck it.

   Tip: You can quickly configure you HTTP data display option for all HTTP requests by clicking on the column title **HTTP** to select the entire HTTP column, then uncheck or check a single HTTP check box to uncheck or check all boxes.

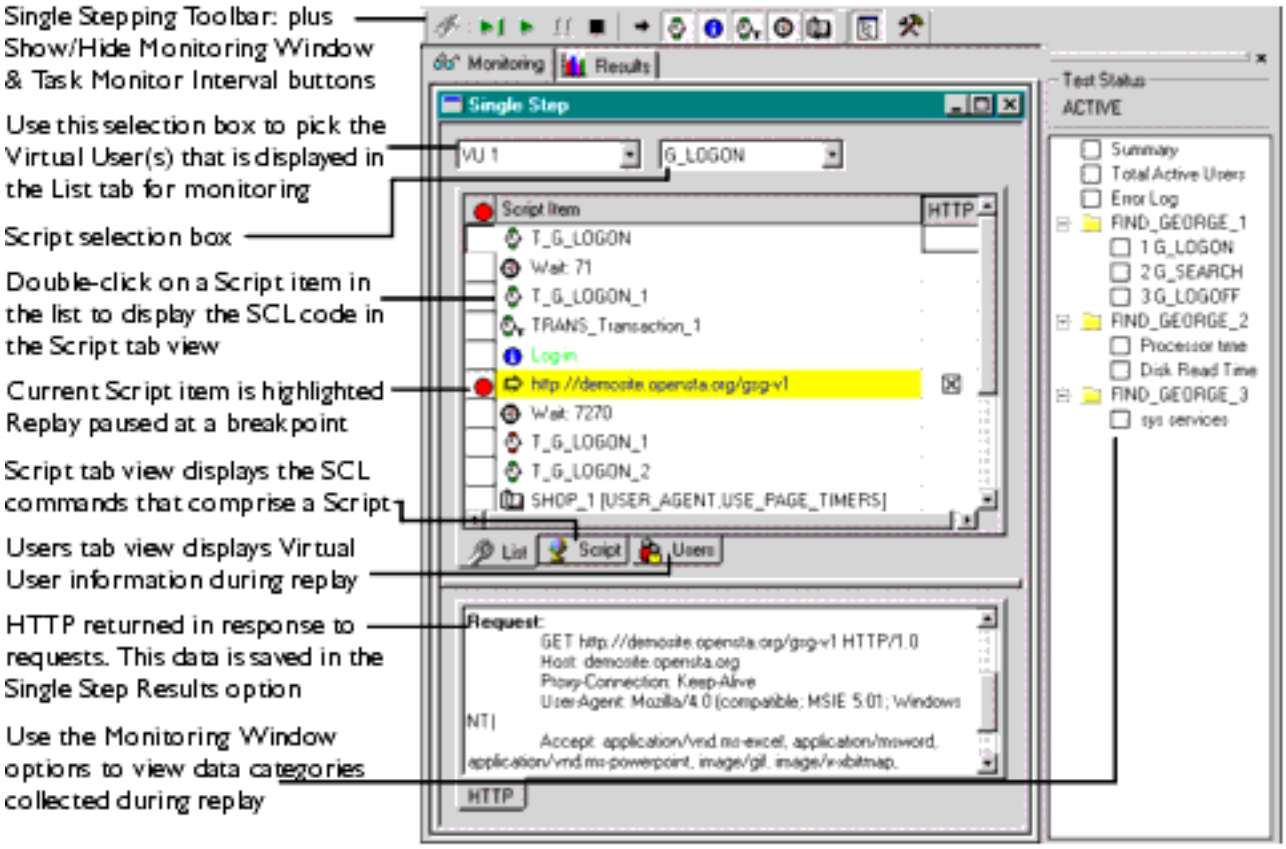
7. Run the Task Group from the Monitoring tab by clicking in the toolbar to replay up to the first breakpoint.

   Or click in the toolbar to replay the Task Group one HTTP request at a time.
   Replay is automatically halted after the response is complete. Keep clicking to single step through the Task Group.

   Tip: Use the break button , to pause the replay.

   You can monitor the replay of the Task Group from the List tab, Script

tab or Users tab in the Scripts Item list.



Note: While the replay is paused you can reconfigure your Task Group replay options from the Monitoring tab if required. You can insert new breakpoints, edit Task Group settings control the HTTP returned and change the Script items you display.

8. Click the Run button  or the Step button  to restart the Task Group replay.

9. Click  to stop the Task Group replay.

10. End a single stepping session from the Monitoring tab by clicking  in the toolbar.

    On completion of the Test-run click the  **Results** tab and use the Results Window to access the **Single Step Results** option.
    The Test-run folders that store single stepping session results are identified  , to distinguish them from normal test-run folders  .

See also:

Add a Comment to a Script for Display During a Single Stepping Session

Add a Transaction Timer to a Script

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Running Tests

Test-runs

Running a Test enables you to simulate real end user Web activity and to collect performance data from the components of the system under test. Use the results you produce during a Test-run to help evaluate the performance of target Web Application Environments (WAEs). You can also use HTTP/S Load to create and run Collector-only Tests to monitor WAEs in a production scenario. The ability to run load Tests and production monitoring Test means that you can directly compare the performance of your target system within these two environments.

HTTP/S Load's distributed software testing architecture enables you to run the Task Groups that comprise a Test on remote Hosts during a Test-run. Distributing Task Groups across a network enables you to run Tests that generate realistic heavy loads simulating the activity of many Virtual Users.

HTTP/S Load provides a variety of data collection and monitoring functions. When a Test is run a wide range of results data is collected automatically. This information includes Virtual User response times and resource utilization from all WAEs under test. You can also create and reference Collectors in your Tests to enhance the Test-run monitoring and data collection options available.

Create and add Collectors to your Tests to enhance the results data produced during a Test-run. Use SNMP and NT Performance Collectors to monitor, graph and record performance data from Host computers and other devices that form part of the system under Test, as well as the Test network.

You can monitor the progress of a Test-run and all of the Task Groups it contains from the Monitoring tab view of the Test Pane. Select NT Performance and SNMP Collector Task Groups to track the data collection queries they define. Select a Script-based Task Group to track the Scripts and the Virtual Users that are currently running.

Running a Test is a straightforward procedure, because the Task Group settings of the Collectors and Scripts you include in the Test have already been specified during Test creation.

Open the Test you want to run and click the Start Test button  , in the toolbar.

At the end of the Test-run all results are stored in the Repository in date and time stamped folders. Display the data collected to help analyze the performance of the target system from the Test Pane of Commander.

See also:

Dynamic Tests

Distributed Tests

The Web Relay Daemon

Test-run Procedure

Monitoring a Test-run

Trace Settings

Single Stepping HTTP/S Load Tests

Dynamic Tests

In HTTP/S Load Tests are dynamic, which means that the Test contents and settings can be modified while it is running, giving you control over a Test-run and the results that are generated.

New Task Groups can be added and the contents and settings of the existing Task Groups that comprise a Test can be individually edited by temporarily stopping the Task Group, making the changes required then restarting them. These facilities give you control over the load generated and enable you to modify the type of performance data you monitor and record without stopping the Test-run.

Note: It is not possible to remove a Task Group from a Test during a Test-run.

While a Test is running you can:

- Add a new Task Group.

  Scripts and Collectors can be added to a Test and the Task Groups that contain them started.
- View the settings and status of Task Groups using the Properties Window and the Status column of the Configuration tab.
- Modify Task Group settings when the selected Task Group has stopped.

  These settings are:

  Schedule settings

  Host settings

  Virtual User settings

  Task settings

- Stop/Start a Task Group.

  Task Groups can be stopped and started during a Test-run using the **Stop** and **Start** buttons in the new Control column of the Configuration tab. The **Stop** button is displayed if the Task Group is Active and a **Start** button is displayed if the Test is running and the Task Group is stopped, otherwise no button is displayed.

See also:

Distributed Tests

Test-run Procedure

Run a Test

Results Display

Distributed Tests

HTTP/S Load supplies a distributed software testing architecture based on CORBA which enables you to utilize remote Host computers to run the Task Groups that comprise a Test. A Task Group can be run by a Task Group Executer process on a remote Host or the Repository Host during a Test-run.

Define the Host you want to run a Task Group when you add a Script or Collector to a Test. Open the Configuration tab of the Test Pane, then click on the Host column table cell in the selected Task Group and using the Properties Window to select a Host, for more information, see Select the Host Used to Run a Task Group.

OpenSTA Name Server

Before you can start a distributed Test the Hosts you have chosen to run the Task Groups must have the OpenSTA Name Server installed, running and correctly configured. Use the Name Server Configuration utility to configure the OpenSTA Name Server settings on all the Hosts running Windows in your Test network.

Before starting a Test-run, make sure that the OpenSTA Name Server is running on the Repository Host and that the Repository Host setting points to itself. You can configure this by right-clicking  , in the Task Bar and selecting the **Configure** option. Then specify the Repository Host setting by typing **localhost**, the computer name or the IP address of the Repository Host in the Repository Host text box. You will need to restart the OpenSTA Name Server to implement the configuration changes you make.

Then configure the remote Hosts you are using to run your Task Groups. The Repository Host setting must point to the Repository Host, which is the machine from where the Test will be run.

When a Host is running Commander, the OpenSTA Name Server and the Name Server Configuration utility should be running by default, because they are setup to launch automatically when you launch Windows. When they are both running, they are represented in the Task bar by the Name Server Configuration utility icon,  . If no icon appears, you need to launch the OpenSTA Name Server and configure it before running a Test.

See also:

Launch the OpenSTA Name Server and the Name Server Configuration Utility

Change the Repository Host Setting of the OpenSTA Name Server

Test-run Procedure

Launch the OpenSTA Name Server and the Name Server Configuration Utility

- Click **Start > Programs > OpenSTA > OpenSTA Name Server**.

  Or,

1. Click **Start > Run**.
2. Enter the application path and program file:

   \Program Files\OpenSTA\Server\DaemonCFG.exe

   or click **Browse**, then locate and double-click the program file.
3. Click **OK**.

See also:

Change the Repository Host Setting of the OpenSTA Name Server

Change the Repository Host Setting of the OpenSTA Name Server

1. In the Task Bar, right-click  .
2. Select **Configure** from the menu, to display the OpenSTA Name Server dialog box.
3. Remote Hosts must point to the Repository Host:
   In the Repository Host text box enter the computer name or IP address of the Repository Host which will be used to run the Test.

   The Repository Host must point to itself:
   In the Repository Host text box type **localhost**, or enter the computer name or IP address of the Repository Host.
4. Click **OK**.

   Note: You will need to restart the OpenSTA Name Server to implement the configuration changes you make.

   Always start the OpenSTA Name Server on the Repository Host before configuring and restarting the OpenSTA Name Server on remote Hosts. If the OpenSTA Name Server shuts down on the Repository Host, you must restart the OpenSTA Name Server on all remote Hosts.

See also:

Launch the OpenSTA Name Server and the Name Server Configuration Utility

Start the OpenSTA Name Server

Start the OpenSTA Name Server

When the OpenSTA Name Server has stopped, the Name Server Configuration utility icon appears in the Task Bar with a small crossed red circle over it,  .

1. In the Task Bar, right-click  .
2. Select **Start Name Server** from the menu.

   The Name Server Configuration utility icon appears  in the Task Bar.

See also:

Change the Repository Host Setting of the OpenSTA Name Server

Stop the OpenSTA Name Server

When the OpenSTA Name Server is running, the Name Server Configuration utility icon appears in the Task Bar with a small ticked green circle over it,  .

1. In the Task Bar, right-click  .
2. Select **Stop Name Server** from the menu.
3. Click **Yes** in the dialog box that appears to confirm your choice.
4. Click **OK** to close the confirmation dialog box.

   The Name Server Configuration utility icon appears  in the Task Bar.

See also:

Start the OpenSTA Name Server

Shutdown the OpenSTA Name Server

1. In the Task Bar, right-click  or  .
2. Select **Start Name Server** from the menu.
3. Click **Yes** in the dialog box that appears to confirm your choice.

See also:

Launch the OpenSTA Name Server and the Name Server Configuration Utility

Test-run Procedure

If you have not already compiled your Test, clicking the Start Test button  in the toolbar will do so. The compiled Test is then distributed to the Host computers you have chosen to run the Test on and executed according to the Task Group settings specified.

A Test is run from one or more Host computers across a network. Target WAEs are injected with HTTP/S and load is generated against them. Collectors included in a Test are also launched and begin collecting the performance data as instructed from the devices you have targeted. You can monitor the data collection recorded by Collectors during a Test-run and the activity of the Scripts and Virtual Users. WAE responses are recorded as performance data which can be displayed after the Test-run is complete.

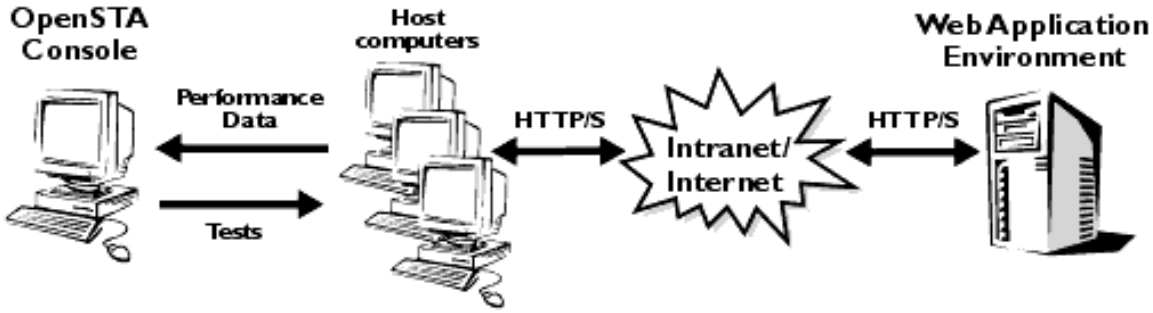
HTTP/S Load Test

During an HTTP Test, target WAEs are injected with HTTP/S and load is generated against them

using. The load you generate is only limited by the availability of system resources.

Use Commander to monitor the activity of the Scripts and Virtual Users included in your Test then display the HTTP data on completion.

The Test-run process within a Web Application Environment is illustrated below:



Note: OpenSTA Console refers to a computer which has an installation of OpenSTA. This includes the OpenSTA Architecture, Commander and the Repository.

Tests can only be run one at a time in order to achieve consistent and reliable results that can be reproduced. The Host computers used to run a Test should be dedicated to this purpose.

You can terminate a Test-run at any stage using the Stop Test button , or the Kill Test-tun button in the toolbar. Use to save HTTP results up to the end of a Test-run. Use to stop a Test-run if you are not interested in results data generated up to the point where the Test was terminated. Minimal results are saved using this option.

When you run a Test, a Test-run folder with the date and time stamp is automatically created in the Repository. Display the results collected during a Test-run by clicking on the Results tab of the Test Pane then using the Results Window to open up Test-run folders and selecting results options for display.

See also:

Run a Test

Monitoring a Test-run

Results Display

Run a Test

1. In the Repository Window, double-click **Tests** to open the folder and display the Tests contained.
2. Double-click the Test you want to run, to open it in the Test Pane.
3. Check the Test contains the Scripts and Collectors you want and that the Task Group settings are correct, then click in the toolbar to run the Test, or click **Test > Execute Test**.

   Note: When you run a Test the Scripts it contains are automatically compiled. If there are errors during compilation the Compile Errors dialog box appears displaying the location and name of the Script that failed and details of the SCL error.
   You can export these error messages to a text editor for easier viewing by right-clicking within the Details section of the Compile Errors dialog box, then selecting, copying and

pasting the text. You can also make use of the single stepping functionality available to help identify errors that may occur during a Test-run.

After your Test has been compiled successfully the Starting Test dialog box appears, which displays a brief status report on the Test-run.

When the Test is running the entry in the Test Status box at the top of the Monitoring Window reads **ACTIVE**.

Tip: Click on the **Monitoring** tab within the Test Pane during a Test-run and select the Task Groups required, to monitor the performance of target WAEs and Hosts in graph and table format.

On completion of the Test-run, click the **Results** tab within the Test Pane, to display the results generated.

See also:

Monitoring a Test-run

Single Stepping HTTP/S Load Tests

Monitoring a Test-run

During a Test-run all Task Groups, the Tasks they contain and summary information can be monitored using Commander, from the Monitoring tab of the Test Pane. Open the Test that is currently running from the Repository Window, then click the **Monitoring** tab in the Test Pane. Use the Monitoring Window, which is displayed by default on the right-hand side of the Main Window, to pick the options you want to display in the workspace of the Test Pane.

The display options listed in the Monitoring Window include all the Task Groups defined in the Test, plus a Summary **display option** which you can select to display an overview of Test-run activity that includes the Task Group name, type and the length of time it has been running. The display options available depend on the type of Test you are running. If your Test includes Scripts the Total Active Users graph is populated with HTTP-related data. The type of Collectors you include in a Test and the data collection queries they define also affects the display options available. Adding Collectors to a Test enhances your monitoring options by enabling you to select and monitor the data collection queries they define. Open up a Task Group folder that contains the Collector you want to monitor, then select the queries you want to display using the Monitoring Window.

When you run a Test the following display options are available for monitoring:

- **Summary:** Provides a summary of Test-run and Task Group activity including Task Group name, status and duration, HTTP data, Virtual User and Task-related details.
- **Total Active Users:** Displays a graph indicating the number of active Virtual Users.
- **Error Log:** Enables you to monitor errors as they occur giving details of the Time, Test Name, Location and Message for each error.
- Collector-based Task Groups: Performance Anomalies and data collection queries.
- Script-based Task Groups: Task details and the number of Virtual Users running.

After you have selected your monitoring display options, you can hide the Monitoring Window to increase the workspace area available for displaying your data.

Click , in the toolbar to hide and display the Monitoring Window.

Use the Task Monitor Interval function to control the frequency at which performance data is collected and returned to Commander for display. Data collection takes up processing resources and can affect the performance of the Test network so it is best to set the Task Monitor Interval to a high value. This function relates to HTTP data and does not affect the data collection interval or polling time you set in Collectors.

If you encounter errors during a Test-run make use of the single stepping functionality provided to check your Tests and to help resolve them. You can monitor the replay of Script-based Task Groups included in a Test and check the HTTP data returned. For more information see Single Stepping HTTP/S Load Tests.

See also:

Monitor Scripts and Virtual Users

Monitor NT Performance and SNMP Collectors

Results Display

Trace Settings

Select a Test to Monitor

Use this function to open up the Test that is running by switching from whatever function you are currently performing in HTTP/S Load to the Configuration tab of the Test Pane.

1. Click **Tools > Monitor**.
2. In the Select Test To Monitor dialog box, click the Test name displayed in the selection box.
3. Click the **Monitor** button.

   This procedure opens the Test displaying the Configuration tab of the Test Pane.

   The Status cell of a Task Group displays.

Set the Task Monitoring Interval

Use this function to control the frequency at which HTTP, Error Log and Summary data is collected and returned to Commander for monitoring.

1. Open the Test that is running and click the  **Monitoring** tab of the Test Pane.
2. Click the Task Monitor Interval button  , in the toolbar.
3. Enter a value in seconds in the text box.
4. Click **OK** to apply your setting to control the data refresh rate.

Monitor a Summary of Test-run Activity

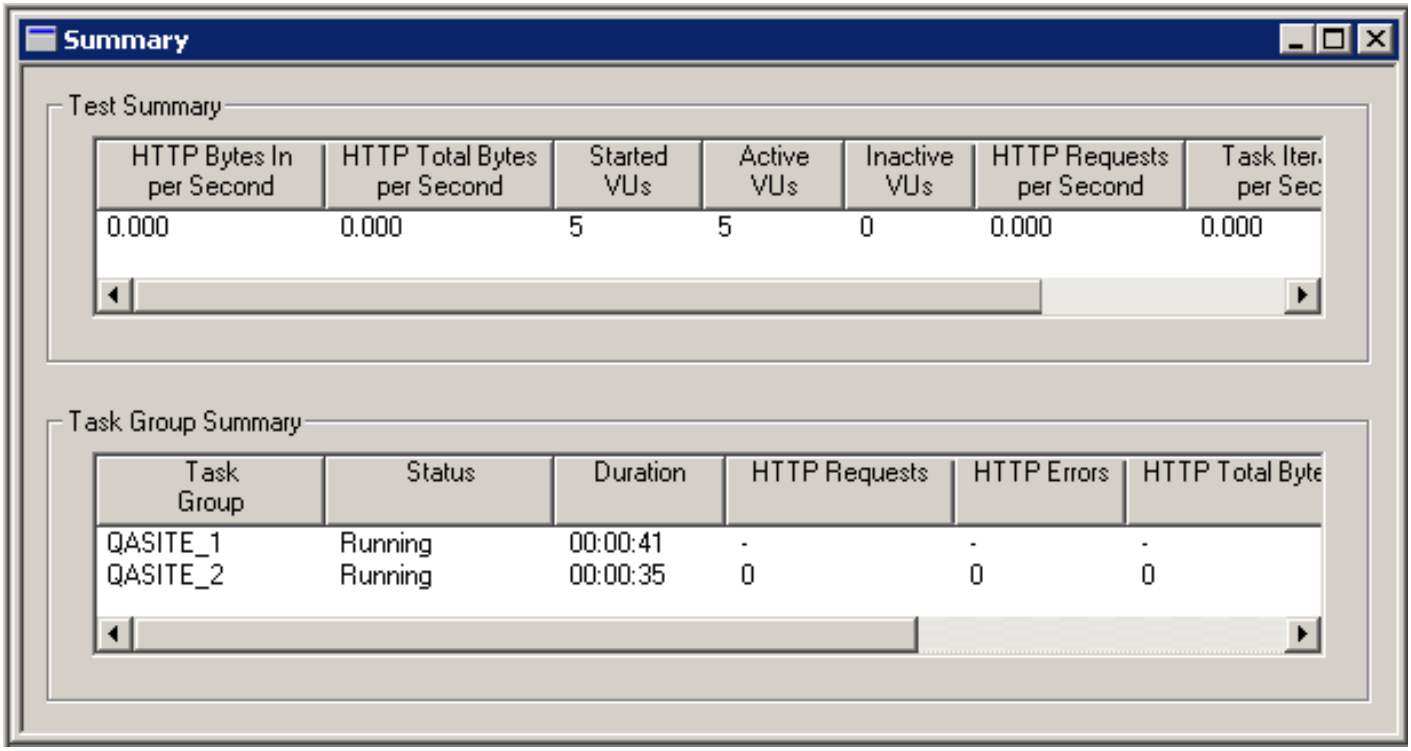
1. Open the Test that is running and click the  **Monitoring** tab of the Test Pane.

   Make sure that the entry in the Test Status box at the top of the Monitoring Window reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click the **Summary** option.

   A summary of Test-run data is displayed in the Monitoring tab. The Summary Window is

divided into 2 panes, the Test Summary pane and the Task Group Summary pane:



The Test Summary pane displays test-specific statistics regarding overall test activity. Data categories provide a range of statistics relating to virtual users, HTTP requests and errors, and connection times.

The HTTP Bytes In per Second and HTTP Bytes Out per Second columns give details of the number of bytes issued and received for HTTP requests per second. The HTTP Total Bytes per Second column measures the total number of bytes for HTTP issue and receipt per second.

The HTTP Errors per Second column indicates the number of 4XX and 5XX error codes returned from the Web browser after the HTTP request has been sent per second. These error codes adhere to the World Wide Web Consortium (W3C) standards. For more information visit: **http://w3.org/Protocols/HTTP/HTRESP**.

Other data categories include:
Minimum Request Latency, Maximum Request Latency and Average Request Latency. These categories give details of the minimum, maximum and average time elapsed between sending an HTTP request and receiving the results.

Tip: Right-click in the Test Summary pane of the Summary Window to control the column display by selecting or deselecting the column names from the pop-up menu.

The Task Group Summary pane presents Task-specific statistics relating to individual Task activity. There is a row entry for each Task Group, facilitating Task monitoring.

Data categories include those described in detail above. Also available are Task Group name, status, duration and Task period.

The following data categories included are also measured per second:

HTTP Requests, HTTP Errors, HTTP Bytes In, HTTP Bytes Out, HTTP Total Bytes.

Tip: Right-click in the Task Group Summary pane of the Summary Window to control the column display by selecting or deselecting the column names from the pop-up menu.

Note: Click , in the Title Bar of a graph or table to close it or deselect the display

option in the Monitoring Window.

Monitor Scripts and Virtual Users

1. Open the Test that is running and click the  **Monitoring** tab of the Test Pane.

   Make sure that the entry in the Test Status box at the top of the Monitoring Window reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click ⊞ , next to a Script-based Task Group folder to open it. The Script-based Task Group folder lists the Script Tasks it contains.
3. Select a Script from the Monitoring Window to track Virtual User activity.

   Data for all the Virtual Users running the selected Script-Task are displayed in the Test Pane. The data categories are Virtual User ID, Duration, Current Script-Task iteration and Note Text connected with each Virtual User. Note text is included for the last NOTE command executed by a Virtual User.

   Note: When a Test-run is complete, the entry in the Test Status box at the top of the Monitoring Window reads **INACTIVE** and the display options in the Monitoring Window are cleared.

See also:

Set the Task Monitoring Interval

Monitor NT Performance and SNMP Collectors

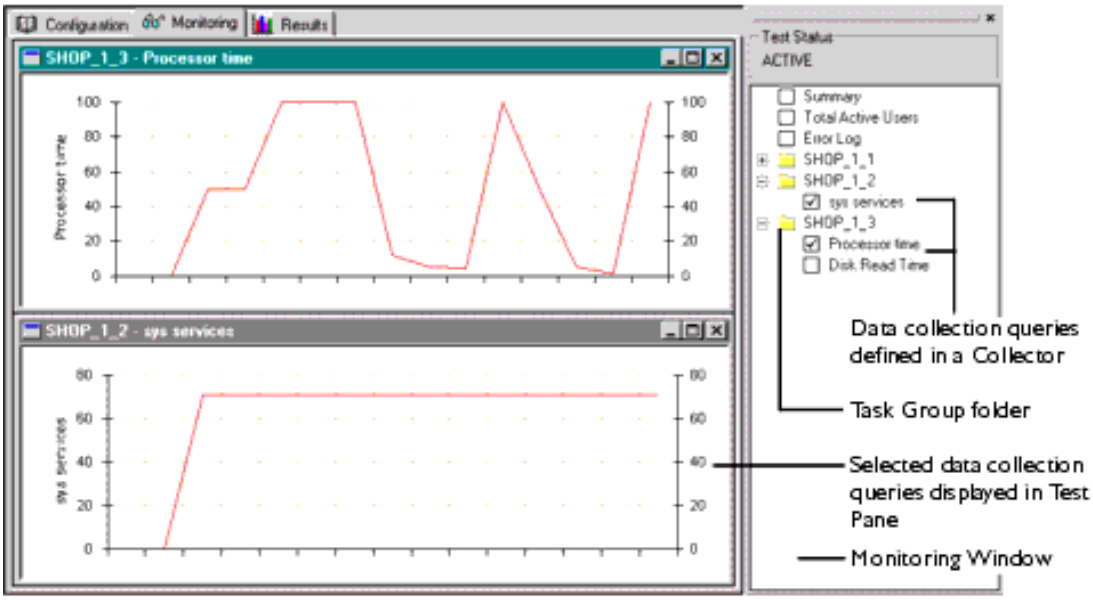
1. Open the Test that is running and click the  **Monitoring** tab of the Test Pane.

   Make sure that the entry in the Test Status box at the top of the Monitoring Window reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click ⊞ , to open a Task Group folder that contains an NT Performance or an SNMP Collector.

   The data collection queries defined in the selected folder are listed below. They represent the display options available.

3. Select one or more of the data collection queries you want to monitor from the Monitoring Window.



Note: When a Test-run is complete, the entry in the Test Status box at the top of the Monitoring Window reads **INACTIVE** and the display options in the Monitoring Window are cleared.

Stop/Start a Task Group

1. Open the Test that is running with the **Configuration** tab of the Test Pane displayed.

   Tip: Click **Tools > Monitor**, to display the Test that is currently running.

   Select the Test you want from the list in the Select Test to Monitor dialog box, then click the **Monitor** button. This procedure opens the Test in the Test Pane.

2. In the Status column of the Test table, click the **Stop** button that appears when a Task Group is running to terminate the selected Task Group.

   After a Task Group has been stopped the Stop button toggles to a Start button. Click **Start** to restart a selected Task Group.

Terminate a Test-run

Note: The Test you want to terminate must be open in the Commander Main Window.

1. Open the Test that is running in the Test Pane.

   Tip: Click **Tools > Monitor**, to display a list of the Tests that are currently running.

2. Click the Stop Test button , in the toolbar to stop the Test, or select **Test > Stop Test**.

   Note: Use , if you want to save results up to the point where the Test-run is stopped.

   Or,

   Click the Kill Test button , in the toolbar to stop the Test.

   Note: Use to stop a Test-run and save minimal HTTP results. Other results categories

such as the Audit log, are generated for the duration of the Test-run.

See also:

Monitoring a Test-run

Trace Settings

The Trace Settings function is used to record the activity of the Test Executer processes that execute the Test and its constituent Script and Collector Task Groups when a Test is run. If you encounter problems during a Test-run, use the Trace settings option to supply you with information on the Test Executer processes and help identify the problem.

The level of data logged using this option can be increased from None, which is the default setting to High, until information relating to the problem is recorded. The amount of Tracing you select will have an affect on the performance of your system so it is best to increase tracing levels gradually over a series of Test-runs until you are able to identify the cause of a problem using the Trace files that are generated.

If you have additional OpenSTA Modules installed extra process entries are added to the Target Name entries column in the Trace settings dialog box, which is used to set the Trace level you want.

After a Test-run is complete the Trace files generated for the Test Executers you have selected are copied to the Repository where they can be found in the Test-run folders.

See also:

Specify Trace Settings

Specify Trace Settings

1. Click **Tools > Trace Settings**.
2. In the Trace Settings dialog click three times on a table cell to the right of the Target Name whose Trace setting you want to edit.
3. Select a trace level from the list, which can be either **None**, **Low**, **Medium** or **High**.
4. Click **OK** to apply your settings.

Creating and Editing Tests

Test Development

After planning your Test, use Commander to coordinate the Test development process.

The contents and structure of your Test will depend on the type of Test you are conducting, the nature of the system you are targeting and the aims of your performance Test. In its simplest form a Test can consist of just one Task Group running a single Collector or a Script. However, to produce a fully automated performance Test that accurately simulates the test scenario you want, as well as producing the results data required, it is usually necessary to develop a more detailed Test structure.

Script and Collector Tasks are contained by Task Groups in a Test. Task Groups enable you to control when Tasks run and how they operate during a Test-run. A Test can include one or more Collector-based Task Groups, one or more Script-based Task Groups or a combination of both, depending on whether you are developing an HTTP/S load Test or a production monitoring Test. Add Scripts to generate the HTTP/S load levels required against target systems during a Test-run. Add Collectors to monitor and record performance data from Hosts.

The Task Groups that comprise a Test can be enabled or disabled, before or during a Test-run. Disabling the Script-based Task Groups means that no load is generated when the Test is run. This gives you the ability to use the same Test within both load Test and production monitoring scenarios and enables you to directly compare the performance of a target system within these two environments. After you have added the Scripts and Collectors you need and applied the Task Group settings required, the Test is ready to run.

Tests can be run using networked computers on remote Hosts to execute the Task Groups that comprise a Test. Distributing Task Groups across a network enables you to utilize the processing resources of multiple networked computers. It is then possible to run HTTP/S load Tests that generate realistic heavy loads simulating the activity of many users. In order to do this, HTTP/S Load must be installed on each Host and the OpenSTA Name Server must be running and configured to specify the Repository Host for the Test. For more information on configuring the OpenSTA Name Server to run a distributed Test, see Distributed Tests.

During a Test-run you can monitor Task Group replay from within the Test Pane. Then display the

results collected after a Test-run is complete to assist you in analyzing and improving the performance of target systems.

See also:

Test Creation

Distributed Tests

The Web Relay Daemon

Running Tests

Test Creation

After you have created the Scripts and Collectors you need, you are ready to create a new Test. Use the right-click menu function associated with the **Tests** folder in the Repository Window, or select **File > New Test > Tests** in the Menu Bar. Give your Test a name, then double-click the new Test , in the Repository Window to open it.

In Commander an open Test is represented as a table which is displayed in the Test Pane. This is the workspace where you can develop the contents of a Test by adding the Scripts and Collectors you need from the Repository. Select them individually working from the Repository Window. Drag and drop them into the Test Pane in the required order.

When you add a Script or Collector to a new row in the Test table, a new Task and Task Group are automatically created. Select a Task Group table cell and apply the settings you require using the Properties Window located at the bottom of the Test Pane to control when a Task Group starts, the Host used to run the Task Group and the load generated by Script-based Task Groups when a Test is run. Click , in the toolbar to hide and display the Properties Window.

There is only a single instance of the Scripts and Collectors you create. They are included in Tests by reference which means they can be included in many different Tests or in the same Test as separate Tasks. Deleting a Test has no affect on the Scripts and Collectors it contains, and similarly, removing Tasks from a Test does not delete them from the Repository. Any changes you make to Scripts and Collectors are immediately reflected in all the Tests that use them. The Scripts and Collectors that you incorporate in a Test can be removed by overwriting them with new selections or by deleting them from the Test Pane.

Make use of the Duplicate a Task Group function to duplicate a Task Group then edit the Task Group Definition to speed up the Test development process. You can duplicate or delete a Task Group by right-clicking on a Task Group and selecting the required popup menu option.

Tests are saved as .TST files and are stored in the Repository. A Test name must be defined according to the rules for OpenSTA Datanames, with the exception that the name can be up to 60 characters long.

As part of the Test creation process you can make use of the single stepping functionality provided to check that your HTTP/S load Tests run as required during replay. For more information see Single Stepping HTTP/S Load Tests.

If you are developing a Test which includes Scripts that run in sequence within the same Task Group you need to model the Scripts for the Task Group to replay correctly when the Test is run. If your WAE uses cookies or issues session identities, then each Script you create will contain a unique identity that has no connection to the other Scripts included in the Task Group. You need to establish a connection between the Scripts by modeling them.

See also:

The Test Pane

Tasks and Task Groups

The Test Development Process

Developing a Modular Test Structure

The Test Pane

Use the Test Pane to create and edit a Test, then apply the Task Group settings you require to control how they behave during a Test-run. Run and monitor the Test-run then display your results for analysis.

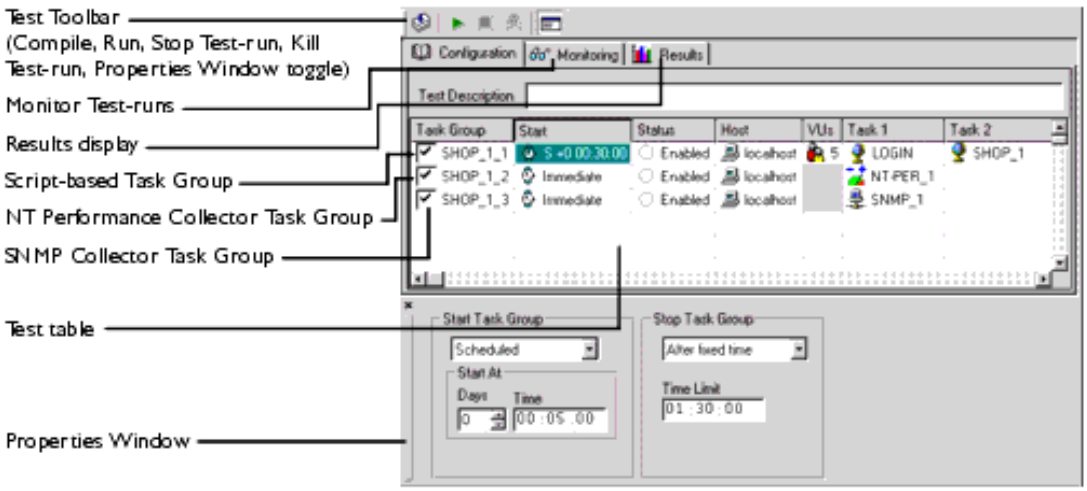
The Test Pane is displayed in the Main Window when you open a Test by double-clicking a new Test , or an existing Test , in the Repository Window.

The Test Pane comprises three sections represented by the following tabs:

- **Configuration**: This is the default view when you open a Test and the workspace used to develop a Test. Use it in combination with the Repository Window to select and add Scripts and Collectors. It displays the Test table which has categorized column headings that indicate where Script and Collector Tasks can be placed and the Task Group settings that apply to the contents of the Test.
  Select a Task Group cell to view and edit the associated settings using the Properties Window displayed below the Test table. For more information, see Tasks and Task Groups.
- **Monitoring**: Use this tab to monitor the progress of a Test-run. Select the display options you want from the Monitoring Window, including a Summary and data for individual Task Groups. For more information, see Running Tests.
- **Results**: Use this tab to view the results collected during Test-runs in graph and table format. Use the Results Window to select the display options available which are dependent on the type of Test you are running. For more information, see Results Display.

Test Pane Features

The Configuration tab view of the Test Pane is displayed below:



See also:

Tasks and Task Groups

The Test Development Process

Tasks and Task Groups

Work from the Repository Window to create new Tests and to open existing ones. Its default location is on the left-hand side of the Commander Main Window,

The Repository Window displays the contents of the Repository and functions as a picking list from where you can select the Scripts and Collectors you want to include in a Test. Use it in conjunction with the Configuration tab of the Test Pane to develop the contents of a Test. Select a Script or Collector from the Repository Window then drag and drop it on to a Task column in the Test table.

The Scripts and Collectors you add to a Test are known as Tasks which are contained within Task Groups. When you add a Script or Collector to a new row in the Test table a new Task and Task Group are automatically created. Each Task Group occupies a single row within the Test table and can be one of two types, either Script-based or Collector-based. A Script-based Task Group can incorporate one Script or a sequence of Script Tasks. Collector-based Task Groups contain a single Collector Task.

Task Groups enable you to control the behavior of the Scripts and Collectors they contain during a Test-run. Select a Task Group cell and apply the settings you require using the Properties Window of the Configuration tab, located at the bottom of the Test Pane. Task Group settings control when a Task Group starts, the Host used to run the Task Group and the load generated by Script-based Task Groups when a Test is run.

When you add a Script or Collector to a Test, you can apply the Task Group settings you require or you can accept the default settings and return later to edit your settings.

The Task Group cells in the Test table are dynamically linked to the Properties Window below, select them one at a time to display and edit the associated Task Group settings in the Properties Window.

Select the **Start** or **Host** cells in a Task Group row to control the Schedule and Host settings. Script-based Task Groups and the Script Tasks they contain have additional settings associated with them. Select the **VUs** and **Task** cells to control the load levels generated when a Test is run.

Use the Disable/Enable a Task Group option to control which Task Groups are executed when a Test is run. This is a useful feature if you want to disable Script-based Task Groups to turn off the HTTP/S load element. The Test can then be used to monitor a target system within a production scenario.

Task Group Settings include:

- Schedule Settings: apply to Script-based and Collector-based Task Groups
- Host Settings: apply to Script-based and Collector-based Task Groups
- Virtual User Settings: apply to Script-based Task Groups only
- Task Settings: apply to Script Tasks only

See also:

Task Group Settings

The Test Development Process

Task Group Settings

Schedule Settings: apply to Script-based and Collector-based Task Groups

Schedule settings enable data to be collected, or an HTTP/S load to be applied, over specific periods by controlling when Task Groups start and stop during a Test run. Click on the **Start** cell in a Task Group and use the Properties Window to specify your Schedule settings. Once a Test is running, Schedule

settings cannot be edited, but they can be overridden manually using the **Start** and **Stop** buttons that appear in the Status column of a Task Group.

The default setting for a Task Group to start is when the Test is run. The Scheduled option starts a Task Group after the number of days and at the time you set. The Delayed option starts a Task Group after the period of time you set, relative to when the Test was started.

There are three options for stopping Task Groups. Manually, means that the Task Group will run continuously until you intervene to end it using the **Stop** button in the Status column of the Task Group that becomes active during a Test run. You can also schedule a Task Group to stop after a fixed period of time and for Script-based Task Groups only you can instruct the Task Group to stop after completing a number of iterations.

For more information, see Edit the Task Group Schedule Settings.

Host Settings: apply to Script-based and Collector-based Task Groups

Specify the Host computer you want to use to run a Task Group during a Test-run. Click on the **Host** cell in a Task Group and use the Properties Window to select a Host.

For more information, see Select the Host Used to Run a Task Group.

Virtual User Settings: apply to Script-based Task Groups only

The load generated against target Web Application Environments (WAEs) during a Test-run is controlled by adjusting the Virtual User settings. Click on the **VUs** cell in a Script-based Task Group and use the Properties Window to specify your Virtual User settings.

Specify the number of Virtual Users you want to run the Task Group to control the HTTP/S load generated when the Task Group is run. Logging levels can be set here to specify the amount of HTTP/S performance statistics gathered from the Virtual Users running the Task Group. Select the Generate Timers option to record the time taken to load each Web page specified in a Script by each Virtual User running the Script.

Use the Batch Start Option to ramp up the load you generate by controlling when the Virtual Users you have assigned to a Task Group run. This is achieved by starting batches of Virtual Users at intervals over a period of time, with a delay between the start of each batch period.

For more information, see Specify the Virtual Users Settings for a Script-based Task Group.

Task Settings: apply to Script Tasks only

Edit the Task settings to control how many times a Script is run. Click on a **Task** cell in a Script-based Task Group and use the Properties Window to specify your Task settings.

You can schedule a Script Task to stop after a fixed period of time or after completing a number of iterations. You can also specify a Fixed or Variable delay to be applied between each Script iteration completed by a Virtual User during a load Test.

For more information, see Edit the Number of Script Iterations and the Delay Between Iterations.

See also:

The Test Development Process

Edit the Task Group Schedule Settings

Select the Host Used to Run a Task Group

Specify the Virtual Users Settings for a Script-based Task Group

Edit the Number of Script Iterations and the Delay Between Iterations

Disable/Enable a Task Group

Tasks and Task Groups

The Test Development Process

The HTTP/S load Test development process typically includes the following procedures:

- Create a Test
- Open a Test
- Add Scripts to a Test
- Add Collectors to a Test
- Define Task Group settings, these include:
- Edit the Task Group Schedule Settings
- Select the Host Used to Run a Task Group
- Specify the Virtual Users Settings for a Script-based Task Group
- Edit the Number of Script Iterations and the Delay Between Iterations
- Delete a Script or Collector from a Test
- Duplicate a Task Group
- Disable/Enable a Task Group
- Delete a Task Group
- Replace a Script or Collector in a Test
- Compile a Test
- Save and Close a Test
- Rename a Test
- Delete a Test

See also:

Running Tests

Single Stepping HTTP/S Load Tests

Create a Test

1. In Commander select **File > New Test > Tests**.

   Or: In the Repository Window, right-click **Tests**, and select **New Test > Tests**.

   The Test appears in the Repository Window with a small crossed red circle over the Test icon , indicating that the file has no content. As soon as you open the Test and add a Script or a Collector, the icon changes to reflect this and appears .
2. In the Repository Window give the Test a name, which must be an OpenSTA Dataname, with the exception that the name can be up to 60 characters long.
3. Press **Return**.

   Note: The new Test is saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Open a Test

The Test Development Process

Open a Test

1. In the Repository Window double-click **Tests** to open the folder and display the Tests contained.
2. Double-click a new Test , or an existing Test , to open it in the Configuration tab of the Test Pane, within the Commander Main Window. The Configuration tab displays the Test table where you can add Test Tasks and apply Task Group settings.

   Note: You do not have to close an open Test or Collector before opening another Test.

   Only a single Test or Collector can be open at one time. When you open a Test, the Test or Collector that was already open is closed and any changes you made are automatically saved.

   When a Test is open in the Test Pane, the Test icon in the Repository Window appears with a small, yellow lock icon overlaid, . An open Test cannot be renamed or deleted.

   The name of the open Test is displayed in the Commander Title bar.

See also:

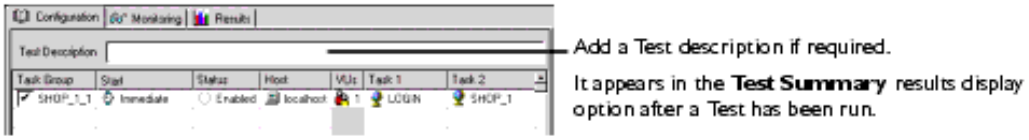
Add Scripts to a Test

Running Tests

The Test Development Process

Add Scripts to a Test

1. Open a Test with the **Configuration** tab of the Test Pane displayed.
2. Double-click **Scripts**, in the Repository Window to open the folder.
3. Click on a Script then drag it across to the Test Pane and drop it in a row under a **Task** column as shown below:



   The selected Script, appears in the first empty row under the first Task column in a new Task Group. Additional Scripts can be added in sequence within the same row.

- The Task Group name is taken from the Test name and includes a number suffix which is automatically incremented for each new Task Group added to the Test.
  Use the Task Group cell to disable and enable a Task Group.

  Note: Uniquely naming Task Groups enables you to select and monitor them during a Test-run from the Monitoring tab.
- The Start column indicates the Task Group Schedule settings. For more information on Task Group scheduling, see Edit the Task Group Schedule Settings.
- The Status column displays Task Group activity and status information.
- The Host column defaults to **localhost**, which refers to the computer you are currently working on. The Host you select here determines which computer or device will run the Task Group during a Test-run. For more information on selecting a Host, see Select the Host Used to

Run a Task Group.

- The VUs column displays the number of Virtual Users assigned to run a Task Group. The default is a single Virtual User **1**.
  The number of Virtual Users running the Task Group can be changed by selecting the VUs cell and using the Properties Window to enter a new value. For more information, see Specify the Virtual Users Settings for a Script-based Task Group.
- With the Script Task you have just added selected, use the Properties Window at the bottom of the Configuration tab to specify the Task settings. For more information, see Edit the Number of Script Iterations and the Delay Between Iterations.

4. If your Task Group incorporates more than one Script, select the next Script from the Repository Window, then drag and drop it into the same Task Group row under the next empty **Task** column cell. Repeat this process until your Script sequence is complete.
5. Note: If the Scripts included in a Task Group target a WAE that uses cookies or issues session identities, you need to model them before the Task Group will replay correctly when the Test is run. Add additional Scripts to a Test in a new Task Group by dragging and dropping them into the next empty row.
   Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Developing a Modular Test Structure

Edit the Task Group Schedule Settings

Select the Host Used to Run a Task Group

Specify the Virtual Users Settings for a Script-based Task Group

Edit the Number of Script Iterations and the Delay Between Iterations

Disable/Enable a Task Group

Add Collectors to a Test

Distributed Tests

Running Tests

The Test Development Process

Add Collectors to a Test

Note: You can add and run new Task Groups during a Test-run.

1. Open a Test with the **Configuration** tab of the Test Pane displayed.
2. Double-click **Collectors**, in the Repository Window to open the folder and display the contents.
3. Click on a Collector then drag it across to the Test Pane and drop it in an empty row under the **Task 1** column, as shown below:

The Collector you add appears in the first empty row under the first Task column, in a separate

Task Group. Collector-based Task Groups can only contain a single Task.

- The Task Group name is taken from the Test name and includes a number suffix which is automatically incremented for each new Task Group added to the Test.

  Use the Task Group cell to disable and enable a Task Group.

  Note: Uniquely naming Task Groups enables you to select and monitor them during a Test-run from the Monitoring tab.

- The Start column indicates the Task Group Schedule settings. For more information on Task Group scheduling, see Edit the Task Group Schedule Settings.
- The Status column displays Task Group activity and status information.
- The Host column defaults to **localhost**, which refers to the computer you are currently working on.

  The Host you select here determines which computer or device will run the Task Group during a Test-run. For more information on selecting a Host, see Select the Host Used to Run a Task Group.

  Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Edit the Task Group Schedule Settings

Select the Host Used to Run a Task Group

Disable/Enable a Task Group

Distributed Tests

Running Tests

The Test Development Process

Edit the Task Group Schedule Settings

1. Open a Test with the **Configuration** tab of the Test Pane displayed.
2. Click on the **Start** cell in a Task Group.

   The current Schedule settings are displayed in the Properties Window at the bottom of the Configuration tab. The default setting is for an **Immediate** start when the Test is run.

3. In the Start Task Group section of the Properties Window, click to the right of the selection box and choose a start option:

- **Scheduled**: The Task Group starts after the number of days and at the time you set.
  Enter a time period using the Days and Time text boxes.
- **Immediately**: The Task Group starts when the Test is started.
- **Delayed**: The Task Group starts after the time period you set, (days: hours: minutes: seconds), relative to when the Test was started.
  Enter a time period using the Days and Time text boxes.

  Note: Your settings are displayed in the Test table.

4. In the Stop Task Group section of the Properties Window, click to the right of the selection box and choose a stop option:

- **Manually**: The Task Group will run continuously until you click the **Stop** button in the Status column of the Task Group that activates during a Test run.
- **After fixed time**: The Task Group is stopped after a fixed period of time.
  Enter a time period using the Time Limit text box.
- **On Completion**: The Script-based Task Group is stopped after completing a number of iterations.
  Enter the number of Task Group iterations in the Iterations text box.

  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

  Note: During a Test-run Schedule settings cannot be edited, but they can be overridden manually using the **Start** and **Stop** buttons in the Status column of each Task Group.

See also:

Select the Host Used to Run a Task Group

Running Tests

The Test Development Process

Select the Host Used to Run a Task Group

Note: Collector-based Task Groups include a Collector which defines a set of data to be recorded from one or more target Hosts during a Test-run. The Host you select in the Test table determines which computer or device will run the Task Group during a Test-run, not the Host from which data is collected.

1. Open a Test with the **Configuration** tab of the Test Pane displayed.
2. Click on the **Host** cell, , in a Task Group.

   The current Host settings are displayed in the Properties Window at the bottom of the Configuration tab. The default setting is localhost, which refers to the computer you are currently using.
3. In the Host Name text box of the Properties Window, enter the name of the Host to run the Task Group. Your settings are then displayed in the Test table.

   Note: The Host you select must have the OpenSTA Name Server installed and running with the Repository Host setting pointing to the local Host. For more information on configuring the Hosts used to run a Test, see Distributed Tests.

   Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

See also:

Specify the Virtual Users Settings for a Script-based Task Group

Running Tests

Distributed Tests

The Test Development Process

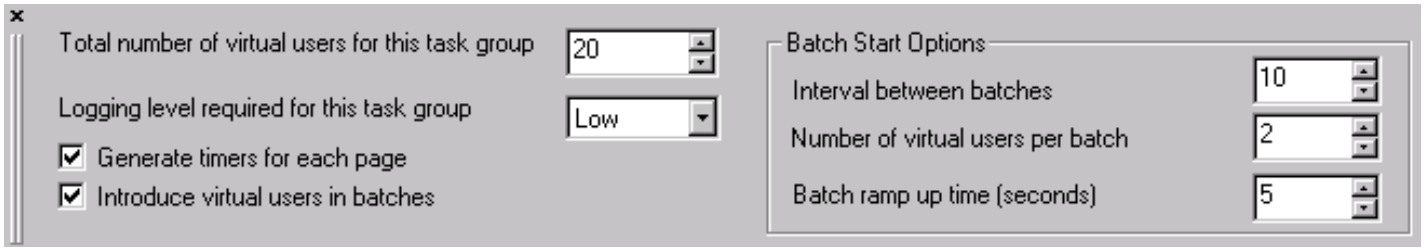
Specify the Virtual Users Settings for a Script-based Task Group

1. Open a Test with the **Configuration** tab of the Test Pane displayed.

2. Click on the VUs cell of the Task Group whose Virtual User settings you want to edit. The current Virtual User settings are displayed in the Properties Window at the bottom of the Configuration tab. Use it to help control the load generated during a Test-run by specifying the number of Virtual Users and when they start.
3. In the Properties Window enter a value in the first text box to specify the total number of Virtual Users for the Task Group, or use to set a value.
4. Select the Logging level required for the Task Group to control the level of performance statistics and Timers gathered from Virtual Users. Click , and select either:
   Low: Information collected from the first 10 Virtual Users in the Task Group.
   High: Information collected from all the Virtual Users in the Task Group.
   None: No performance statistics or Timers are gathered.
5. Click the **Generate Timers For Each Page** check box, to record results data for the time taken to load each Web page specified in the Scripts, for every Virtual User running the Scripts. Timer information is recorded for the duration of the complete Script if the box is checked or unchecked.
6. Click on the **Introduce Virtual Users in batches** check box if you want to ramp up the load you generate by controlling when the Virtual Users you have assigned run. This is achieved by starting groups of Virtual Users in user defined batches.
7. Use the Batch Start Options section to control your Virtual user batch settings.

- **Interval between batches**, specifies the period of time in seconds between each ramp up period. No new Virtual Users start during this time.
- **Number of Virtual Users per batch**, specifies how many Virtual Users start during the batch ramp up time.
- **Batch ramp up time (seconds)**, specifies the period during which the Virtual Users you have assigned to a batch start the Task Group. The start point for each Virtual User is evenly staggered across this period.

  The example below depicts the Properties Window, where 20 Virtual Users are assigned to a Script-based Task Group. When the Task Group is run 2 Virtual Users (the number of Virtual Users per batch) will start over a period of 5 seconds (batch ramp up time) with a 10 second delay between each batch running.



  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

See also:

Select the Host Used to Run a Task Group

Running Tests

The Test Development Process

Edit the Number of Script Iterations and the Delay Between Iterations

1. Open a Test with the **Configuration** tab of the Test Pane displayed.
2. Click on a Script Task , whose Task settings you want to edit, to display the current Task

settings in the Properties Window at the bottom of the Configuration tab.

3. With a Script Task selected, use the Properties Window to specify how long the Task runs. Click on the Task Termination box and select an option, either:

- **On Completion**: set a value to control the number of times (iterations) a Virtual User will run the Script during a Test-run.
- **After Fixed Time**, specify a time period to control when the task completes.

  Enter a value in the text box below or use  .

4. You can specify a Fixed or Variable delay between each iteration of a Script Task.
   In the Properties Window, click on the Delay Between Each Iteration box and select an option, either:

- **Fixed Delay**: set a time value in seconds using the Delay text box.

  Or, you can choose to introduce a variable delay between Scripts:

- **Variable Delay**: set a value range in seconds using the Minimum and Maximum text boxes to control the upper and lower limits of variable iteration delay.

  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

See also:

Add Scripts to a Test

Running Tests

The Test Development Process

Delete a Script or Collector from a Test

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. Select an individual Script or Collector Task from within a Task Group.
3. Press **Delete**, or select **Test > Delete Selection**.

   Note: Deleting a Collector Task removes the Task Group from the Test table, since a Collector-based Task Group can only include a single Collector.

   Note: The Script or Collector you remove is not deleted from the Repository since all Scripts and Collectors are included in Tests by reference.

   Tip: To remove a Script-based Task Group, which contains a sequence of Scripts, right-click on the Task Group and select **Delete Task Group**.

See also:

Running Tests

The Test Development Process

Duplicate a Task Group

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. Right-click on the Task Group you want to copy.
3. Select **Duplicate Task Group** from the popup menu.

A copy of the Task Group is created at the bottom of the Task Group table.
Edit the Tasks included and the settings of the new Task Group as required.

See also:

Running Tests

The Test Development Process

Disable/Enable a Task Group

Note: When you add a Script or Collector to a Test the default setting is Enabled.

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. In the Task Group column click on the check box in the cell of the Task Group that you want to disable.

   When the check box is unchecked, **Disabled** appears in the Status column cell of the Task Group, indicating that the Task Group will not execute when the Test is run.
3. Enable a disabled Task Group by clicking on the unchecked check box.

   When the check box is checked **Enabled** appears in the Status column cell of the Task Group.

   Note: The **Start** and **Stop** buttons that appear in the Status cell of a Task Group during a Test-run are used to control when a Task Group starts, not the Enable/Disable function.

See also:

Running Tests

The Test Development Process

Delete a Task Group

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. Right-click on the Task Group you want to delete, or hold down the **Shift** key to select more than one Task Group for deletion.
3. Select **Delete Task Group** from the popup menu.

   Note: Deleting a Task Group does not remove the Scripts or Collectors it contains from the Repository because they are included in a Task Group by reference.

   Tip: Delete a Script or Collector from a Test to remove the Task Group from the Test table, since a Collector-based Task Group can only include a single Collector.

See also:

Running Tests

The Test Development Process

Replace a Script or Collector in a Test

Note: Only Scripts can overwrite other Scripts and only Collectors of the same type can be overwritten using the method below.

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. Select the Script or Collector you want to add from the Repository Window and drag and drop it

on to the Task cell in the Test Pane containing the Script or Collector that you want to replace.

3. Click **Yes** in the confirmation dialog box to overwrite the Task.

   Note: The Script or Collector you overwrite is not deleted from the Repository since all Tasks are included in Tests by reference.

   Note: Existing Task Group settings are unaffected when you replace a Task.

   Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

See also:

Running Tests

The Test Development Process

Compile a Test

Before you run a Test you need to compile to check its validity and ensure it will run.

If you have not compiled a Test before you run it, clicking the Test-run button  , automatically compiles the Test.

1. Open a Test.
2. Click  in the toolbar to compile or select **Test > Compile Test**.
3. When your Test compiles successfully you are notified with an on-screen message. Click **OK** in the dialog.

   A Compile Errors dialog appears if compilation is unsuccessful, detailing the nature of the problem.

See also:

Running Tests

The Test Development Process

Save and Close a Test

- The Test-related work you perform is automatically saved in the Repository and the Test is closed when you switch to a different function or exit Commander.
- Click  , in the Menu Bar.

  Note: You do not have to close an open Test before opening another Test or Collector.

See also:

Running Tests

The Test Development Process

Rename a Test

Note: An open Test cannot be renamed.

1. In the Repository Window double-click  **Tests**, to open the folder and display the Tests contained.
2. Right-click on a Test and select **Rename** from the menu.

3. Enter a new name and press **Return**.

See also:

Running Tests

The Test Development Process

Delete a Test

Note: An open Test cannot be deleted.

1. In the Repository Window double-click **Tests**, to open the folder and display the Tests contained.
2. Right-click on a Test and select **Delete**, or select a Test and press **Delete**.
3. Click **Yes** to confirm the deletion of the Test from the Repository.

   Note: Collectors are incorporated into a Test by reference, so deleting a Test does not affect the Tasks they contain.

See also:

The Test Development Process

Creating and Editing Collectors

Collectors Overview

Collectors are used to monitor and collect performance data from target components of production systems and Web Application Environments (WAEs) during Test-runs, to help you evaluate their performance.

A Collector is a set of user-defined data collection queries which determine the type of performance data recording carried out from one or more Host computers or devices during a Test-run. Include them in your Tests to target specific components of the WAEs under test and the Hosts used to run a Test, with precise data collection queries to collect the performance data you need. Create Collectors and incorporate them into your Tests, then run the Test to generate the results data required.

Collectors give you the flexibility to collect a wide range of performance data at user defined intervals during a Test-run. A Collector can contain a single data collection query and be used to target a single Host. Or alternatively, they can contain multiple queries and target multiple Hosts. NT Performance Collectors are used for collecting performance data from Hosts running Windows NT or Windows 2000. SNMP Collectors are used for collecting SNMP data from Hosts and other devices running an SNMP agent or proxy SNMP agent.

Collectors are stored in the Repository and are included in Tests by reference. This means that any changes you make to a Collector will have immediate affect on all the Tests that use them.

Collector-based Task Groups can be monitored during a Test-run. The specific data collection queries defined within a Collector can be selected and monitored from the Monitoring tab view of the Test Pane.

After a Test-run is complete, results are stored in the Repository from where they are available for immediate display and analysis. The data collected can be displayed alongside results from previous Test-runs associated with the Test, to provide comparative information about target system performance.

Results collected by all the SNMP Collectors included in a Test are saved in the Custom SNMP file. Results collected by all the NT Performance Collectors you include are saved in the Custom NT Performance file. Results are displayed by opening a Test, then using the Results Window displayed in the Results tab of the Test Pane to open the display options listed. Results data can be can be exported to spreadsheet and database programs for further analysis, or printed directly from Commander.

See also:

Creating Collectors

Creating Collectors

Creating Collectors involves deciding which Host computers or other devices to collect performance data from and the type of data to collect during a Test-run. Create SNMP Collectors to target any Hosts capable of running an SNMP agent or proxy SNMP agent which can include computers and other devices. Or create NT Performance Collectors to collect performance data from Hosts running Windows NT or Windows 2000.

Use the right-click menu function associated with the  **Collectors** folder in the Repository Window to create a new Collector, or select **File > New Collector > SNMP** or **File > New Collector > NT Performance** in the Menu Bar. Give your new Collector a name, then double-click the new Collector icon  (SNMP), or  (NT Performance) in the Repository Window to open it in the Collector Pane. This is the workspace where you can develop a Collector by defining your data collection queries.

In Commander an open Collector is represented as a table in the Collector Pane. This is the workspace where you can develop a Collector. Each data collection query you define occupies a row within the table. When you first open a new Collector there are no rows or data collection queries defined and the Edit Query dialog box appears automatically. Use this dialog box to setup a new query.

Work through the configuration settings to add a unique query name, choose the Host from which performance data will be collected, select the query type and to specify the frequency for data to be collected. You can also select to record the raw value of the data, or the Delta Value which records the difference between the data collected at each interval.

In existing Collectors that already have one or more data collection queries defined, double-click a row in the table to open Edit Query dialog box and make any changes you need. Use  , in the toolbar to add new rows and define additional data collection queries. Select a row and click  , to delete a query. The Collector settings are automatically saved in the Repository when you switch to a different function or exit from Commander.

There is only a single instance of the Collectors you create. They are included in Tests by reference which means that they can be used in many different Tests. The data collection and monitoring settings you define in a Collector apply to all the Tests that use it. Similarly, any changes you make are immediately reflected in all the Tests that reference it. The Collectors you incorporate into Tests can be removed by overwriting them with new selections or deleting them from a Test, but this does not delete them from the Repository.

NT Performance Collectors are saved as .NTP files, SNMP Collectors are saved as .SMP files. A Collector name must be defined according to the rules for OpenSTA Datanames, with the exception that the name can be up to 60 characters long.

See also:

The Collector Pane

Creating and Editing Tests

The Collector Pane

The Collector Pane is displayed in the Commander Main Window, when you open a Collector from the Repository Window. The options, the display and the creation process are similar for both SNMP and NT Performance Collectors.

Double-clicking on a new Collector  (SNMP), or  (NT Performance) opens the Collector Pane with a blank display and the Edit Query dialog box appears automatically. Work through the settings presented here to define a data collection query.

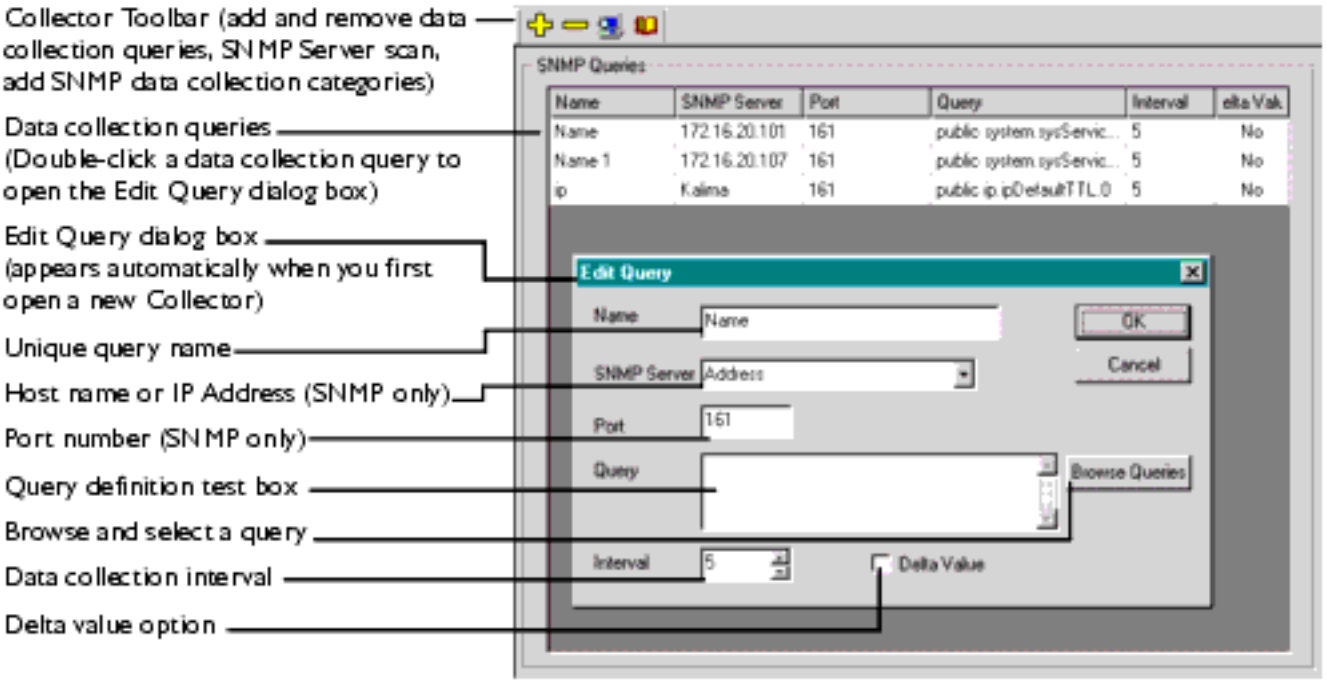
SNMP Collectors supply two additional toolbar options, the SNMP Server Scan  , and Edit SNMP Categories  .

The SNMP Servers Scan identifies all networked SNMP Servers currently running an SNMP agent within a defined IP address range. The SNMP Server text box in the Edit Query dialog box is populated with the returned IP addresses from where you can select a target device.

Use Edit SNMP Categories option to create new SNMP data collection categories which you can use to define a query in the Edit Query dialog box.

Collector Pane Features

The Collector Pane with the Edit Query dialog of an SNMP Collector is displayed below:



See also:

SNMP Collectors

NT Performance Collectors

SNMP Collectors

SNMP Collectors (Simple Network Management Protocol) are used to monitor and collect SNMP data from Host computers or other devices running an SNMP agent or proxy SNMP agent during a Test-run. Creating then running SNMP Collectors as part of a Test enables you to collect results data to help you assess the performance of WAEs under test.

SNMP is the Internet standard protocol developed to manage nodes on an IP network, but SNMP is not limited to TCP/IP. It can be used to manage and

monitor all sorts of equipment including computers, routers, wiring hubs and printers. That is, any device capable of running an SNMP management process, known as an SNMP agent. All computers and many peripheral devices meet this requirement, which means you can create and include SNMP Collectors in a Test to collect data from most components used in target WAEs.

SNMP data collection queries defined in a Collector can be displayed graphically during a Test-run to monitor the performance of the target Host. Select a Task Group that contains an SNMP Collector from the Monitoring Window in the Monitoring tab of the Test Pane then choose the performance counters you want to display.

The results collected using a Collector can be viewed after the Test-run is complete. Select a Test and open up the Custom NT Performance graph from the Results tab of the Test Pane to display your results.

Note: The SNMP Module used to create SNMP Collectors is a component of HTTP/S Load.

See also:

SNMP Collector Development Process

NT Performance Collectors

Creating and Editing Tests

Monitoring a Test-run

Results Display

SNMP Collector Development Process

- Create an SNMP Collector
- Open an SNMP Collector
- Add SNMP Data Collection Queries
- Run the SNMP Server Scan
- Create New SNMP Data Collection Categories
- Edit Collector Settings
- Save and Close a Collector
- Rename a Collector
- Delete a Collector

Create an SNMP Collector

1. In Commander, select **File > New Collector > SNMP**.

   Or: In the Repository Window, right-click **Collectors**, and select **New Collector > SNMP**.

   The Collector appears in the Repository Window with a small crossed red circle over the icon , indicating that the Collector has no content.

   Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon changes to reflect this .

2. Give the new Collector a name within the Repository Window, which must be an OpenSTA Dataname, with the exception that the name can be up to 60 characters long, then press **Return**.

   Note: The new Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Open an SNMP Collector

Open an SNMP Collector

1. In the Repository Window double-click **Collectors**, to open the folder and display the Collectors contained.
2. Double-click a new Collector , or an existing Collector , to open the Collector Pane in the Commander Main Window, where you can setup your data collection queries.

   The Edit Query dialog box opens automatically when you open a new Collector , or double-click on a row of an open Collector. Use this dialog box to Add SNMP Data Collection Queries.

   Note: You do not have to close an open Collector or Test before opening another Collector.
   Only a single Collector or Test can be open at one time. When you open a Collector the Collector or Test that was open is closed and any changes you made are automatically saved.
   When a Collector is open in the Collector Pane, the Collector icon in the Repository Window appears with a small, yellow lock icon overlaid, .
   An open Collector cannot be renamed or deleted.
   The name of the open Collector is displayed in the Commander Title bar.

See also:

Add SNMP Data Collection Queries

Add SNMP Data Collection Queries

1. Open an SNMP Collector.
2. If the Edit Query dialog box does not appear automatically, click  , in the toolbar.
3. In the Name text box enter a unique title for the data collection query.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring tab of the Test Pane. You can select query names to monitor the progress of the Test-run.

   Query names also appear in the Custom SNMP graph with the associated results data. Use the Results Window in the Results tab of the Test Pane to display them.
4. In the SNMP Server text box enter the Host name or the IP address you want to collect data from.

   Tip: You can Run the SNMP Server Scan by clicking  in the toolbar, to identify all networked SNMP Servers currently running an SNMP agent, then click  , to the right of the SNMP Server text box to display the list and select an SNMP Server.
5. In the Port text box enter the port number used by the target SNMP Server.

   Note: Port 161 is the default port number that an SNMP agent runs from.
6. Click the **Browse Queries** button to open the Select Query dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit Query dialog box.
7. In the Select Query dialog box, click  to the right of the Category selection box and choose a category from the drop down list.
8. In the Query selection box below, choose a query associated with the selected category.

   Note: The Current Value of the query must contain a numeric counter in order to generate data to populate the results graphs.
9. Click **Select** to confirm your choices and return to the Edit Query dialog box.
10. In the Edit Query dialog box use the Interval text box to enter a time

period (in seconds) to control the frequency of data collection, or use  , to set a value.

11. Leave the Delta Value column check box unchecked to record the raw data value, or check the box to record the Delta value.

    Note: Delta value records the difference between the data collected at each interval.

12. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

13. Use  , in the toolbar to add additional queries. Double-click on a query to edit it.

    Tip: Select a query then click  , in the toolbar to delete it.

    Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

Run the SNMP Server Scan

Use the SNMP Server Scan option to identify all networked SNMP Servers currently running an SNMP agent within a defined IP address range. Run the scan before defining a query.

1. Open an SNMP Collector.
2. Click  , in the toolbar.
3. In the SNMP Server Scan dialog box, edit the IP address range entered in the Start IP and End IP text boxes to control the range of the scan if required.
4. In the Port text box enter the correct SNMP port number.

   Note: Port 161 is the default port number that an SNMP agent runs from. If you have assigned a different port number you must enter it here for the scan to be successful.

5. Click **Scan** to initiate the scan.

   The scan identifies SNMP Servers and populates the SNMP Server text box in the Edit Query dialog box with the returned IP addresses. Click  , to the right of the SNMP Server text box to display the list and select an SNMP Server.

   Note: If you switch to a different function, such as opening a Test or Collector, the scan results are not saved.

Create New SNMP Data Collection Categories

Use this option to create new SNMP data collection categories which you can select when you define a new query in the Select Query dialog box.

1. Open an SNMP Collector.
2. Click  , in the toolbar.
3. In the Category Definition dialog box, click in the Name text box and enter the title of the new data collection category.

   Note: The new category can be chosen from the Category text box of the Select Query dialog box when you are defining a query.
4. In the Walk Point text box enter the query definition.

   Note: The Walk Point you define can be selected in the Query text box of the Edit Query dialog box and the Category text box of the Select Query dialog box when you are choosing a query.
5. Click **Apply** to make the new category available for selection. Click **Close** to cancel.

   Note: Edit the Walk Point of a category by clicking  , to the right of the Name text box to display and select a category, then enter the new query definition.

NT Performance Collectors

NT Performance Collectors are used to monitor and collect performance data from your computer or other networked Hosts running Windows NT or Windows 2000 during a Test-run. Creating and running NT Performance Collectors as part of a Test enables you to collect comprehensive data to help you assess the performance of WAEs under test.

Use NT Performance Collectors to collect performance data during a Test-run from performance objects such as Processor, Memory, Cache, Thread and Process on the Hosts you specify in the data collection queries. Each performance object has an associated set of performance counters that provide information about device usage, queue lengths, delays, and information used to measure throughput and internal congestion.

NT Performance Collectors can be used to monitor Host performance according to the data collection queries defined in the Collector during a Test-run. Performance counters can be displayed graphically by selecting the Task Group that contains the Collector from the Monitoring Window in the Monitoring tab of the Test Pane.

The results recorded using a Collector can be monitored then viewed after the Test-run is complete. Select a Test and open up the Custom NT Performance graph from the Results tab of the Test Pane to display your results.

If the Web server under test is running Microsoft IIS (Internet Information Server), then you can monitor it by selecting the Web Service performance object from the Performance Object selection box when you add a new data collection query.

Note: The NT Performance Module used to create NT Performance Collectors is a component of HTTP/S Load.

See also:

NT Performance Collector Development Process

Creating and Editing Tests

Monitoring a Test-run

Results Display

NT Performance Collector Development Process

- Create an NT Performance Collector
- Open an NT Performance Collector
- Add NT Performance Data Collection Queries
- Edit Collector Settings
- Save and Close a Collector
- Rename a Collector
- Delete a Collector

Create an NT Performance Collector

1. In Commander, select **File > New Collector > NT Performance**.

   Or: In the Repository Window, right-click **Collectors**, and select **New Collector > NT Performance**.

   The Collector appears in the Repository Window with a small crossed red circle over the Collector icon , indicating that the Collector has no content.

   Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon

changes to reflect this  .

2. Give the new Collector a name within the Repository Window, which must be an OpenSTA Dataname, with the exception that the name can be up to 60 characters long, then press **Return**.

   Note: The new Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Open an NT Performance Collector

Open an NT Performance Collector

1. In the Repository Window double-click  **Collectors** to open the folder and display the Collectors contained.
2. Double-click a new Collector  , or an existing Collector  , to open the Collector Pane in the Commander Main Window, where you can setup your data collection queries.

   The Edit Query dialog box opens automatically when you open a new Collector  , or double-click on a row of an open Collector. Use this dialog box to Add NT Performance Data Collection Queries.

   Note: You do not have to close an open Collector or Test before opening another Collector.
   Only a single Collector or Test can be open at one time. When you open a Collector the Collector or Test that was open is closed and any changes you made are automatically saved.

   When a Collector is open in the Collector Pane, the Collector icon in the Repository Window appears with a small, yellow lock icon overlaid,  . An open Collector cannot be renamed or deleted.

   The name of the open Collector is displayed in the Commander Title bar.

See also:

Add NT Performance Data Collection Queries

Add NT Performance Data Collection Queries

1. Open an NT Performance Collector.
2. If the Edit Query dialog box does not appear automatically double-click on a query.

3. In the Name text box enter a unique title for the data collection query.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring tab of the Test Pane. You can select query names to monitor the progress of the Test-run.
   Query names also appear in the Custom NT Performance graph with the associated results data. Click the Results tab in the Test Pane and display them.

4. Click the **Browse Queries** button to open the Browse Performance Counters dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit Query dialog box.

5. In the Browse Performance Counters dialog box, select the Host you want to collect data from. You can select to either:

   - **Use local computer counters**: Collects data from the computer you are currently using.
   - Or, **Select counters from computer**: Enables you to specify a networked computer. Type **** then the name of the computer, or click  and select a computer from the list.

6. In the Performance object selection box select a performance object. Click  , to the right of the selection box and choose an entry from the drop down list.
7. In the Performance counters selection box choose a performance counter.

   Note: Click **Explain** to open a dialog box which gives a description of the currently selected Performance counter.

8. In the Instances selection box pick an instance of the selected performance counter.
9. Click **OK** to confirm your choices and return to the Edit Query dialog box.
10. In the Interval text box enter a time period (in seconds) to control the frequency of data collection, or use  , to set a value.
11. Leave the Delta Value column check box unchecked to record the raw data value, or check the box to record the Delta value.
    Delta value records the difference between the data collected at each interval.
12. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

13. Use  , in the toolbar to add additional queries.

    Tip: Double-click on a query to edit it. Select a query then click  , in the toolbar to delete it.
    Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

General Collector Procedures

- Edit Collector Settings
- Save and Close a Collector
- Rename a Collector
- Delete a Collector

See also:

Creating and Editing Tests

Edit Collector Settings

1. In the Repository Window double-click  **Collectors** to open the folder and display the Collectors contained.
2. Double-click a Collector to open the Collector Pane in the Commander Main Window, where you can make your edits.
3. If the Edit Query dialog box does not open automatically, double-click on a data collection query to open it.
4. Make your edits to the query using the Edit Query dialog box.
5. Click **OK** to apply your changes.
6. Click  , in the toolbar to add a new Custom SQL request.
7. Select a SQL request then click  , in the toolbar to delete it.

   Note: The changes you make to a Collector have immediate affect on all the Tests that reference it.

   Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

See also:

Creating and Editing Tests

Save and Close a Collector

- The Collector-related work you perform is automatically saved in the Repository and the Collector is closed when you switch to a different function or exit from Commander.
- Click  , in the Menu Bar.

  Note: You do not have to close an open Collector before opening another Collector or Test.

Rename a Collector

Note: An open Collector cannot be renamed.

1. In Commander, double-click  **Collectors**, in the Repository Window to expand the directory.
2. Right-click on a Collector and select **Rename**, or double-click slowly on a Collector.
3. Enter the new name and press **Return**, to save your changes in the Repository.

   Note: When you rename a Collector the Tests that reference it notify you that it is missing by highlighting the Task table cell it occupied in Red. The Test cannot run with a missing Task. Rename or recreate a Collector to match the name of the missing Collector to resolve this problem or delete the Task from the Test.

Delete a Collector

Note: An open Collector cannot be deleted.

1. In Commander, double-click **Collectors**  , in the Repository Window to expand the directory.
2. Right-click on a Collector and select **Delete** from the menu.

   Or, click on the Collector you want to remove and press **Delete**.
3. Click **Yes** to confirm the deletion.

   Note: When you delete a Collector the Tests that reference it notify you that it is missing by highlighting the Task table cell it occupied in Red. The Test cannot run with a missing Task. Recreate a Collector to match the name of the missing Collector to resolve this problem or delete the Task from the Test.

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Modeling Scripts

Modeling Overview

Modeling Scripts enables you to develop realistic Tests and improve the quality of the Test results produced.

There are extensive modeling options available within Script Modeler that can help you to develop realistic performance Tests. When you are familiar with the structure of Scripts and in particular the SCL code they are written in, you will be well equipped to model them. SCL is a simple scripting language that gives you control over the Scripts you create. It enables you to model Scripts to accurately simulate the Web activity and to generate the load levels you need against target WAEs when a Test is run.

How you model the Scripts you record or whether you choose to do so at all, depends on the functionality of the WAE you are testing and the type of Web activity you want to Test.

A key modeling technique involves the addition of variables to a Script which enable you to change the fixed values they record. For example, if a Script records login

details which identify the user who conducted the original browser session, you can replace this information with a variable that changes the user login details each time the Script is replayed during a Test-run.

Variables can be incorporated into Scripts to control a variety of elements including user selections. For example, a Script may record items purchased by the user which you need to vary in order to make your Test more realistic. Introducing a variable to replace the recorded selections enables you to do this.

Using variables to change the activity and the identity of the Virtual Users enables you to use a single modeled Script to simulate multiple unique browser users when the Test is run.

Script Modeling is enhanced beyond the representation of HTTP requests with SCL and the addition of variables to a Script, by providing the capability to include objects from a Web page in a Script. HTTP/S Load provides the capability to use DOM objects from the Web pages that are saved at the same time a Script is recorded, to model the corresponding Script. This modeling technique is known as DOM Addressing and can be used to verify the results of a Test by checking that the WAE responses returned during a Test-run are correct.

If you are developing a Test which includes Scripts that run in sequence within the same Task Group you need to model the Scripts for the Task Group to replay correctly when the Test is run. If your WAE uses cookies or issues session identities, then each Script you create will contain a unique identity that has no connection to the other Scripts included in the Task Group. You need to establish a connection between the Scripts by modeling them. For more information Developing a Modular Test Structure.

Whether you are developing a modular Test structure or you are using Task Groups that reference a single Script, it is important to check that the Test is running correctly. Make use of the DOM Addressing function to help verify a Test-run. You can also run a single stepping session to check that the WAE responses are appropriate.

In addition to the SCL code, a knowledge of HTTP commands is useful in reading and modeling Scripts. Make use of the *SCL Reference Guide* to assist with your modeling tasks. In Script Modeler, click **Help > SCL Reference** to view an on-line copy of the guide.

See also:

SCL Representation of Scripts

SCL Representation of Scripts

SCL, Script Control Language, is a scripting language created by CYRANO. Within Script Modeler, it is used to write the Scripts which define the content of your Tests. Make use of SCL commands to model Scripts and develop the Test scenarios you need. Refer to the *SCL Reference Guide* for more information.

When a Script is recorded through the Gateway, the raw HTTP/S traffic is represented using SCL code. Scripts are written using SCL code which enables you to model them. This gives you control of the content of the Tests you create and enables you to simulate the Test scenarios required no matter how complex. Model a Virtual User by using the menu options available or by keying in the SCL commands you need. Scripts function as interactive text files, which you can edit and manipulate using methods you will be familiar with if you have used any type of text editor. You can enter text, cut and paste, search and replace text elements and variables, scroll up and down through the file and bookmark text lines.

When you open a Script you will notice that the data it contains is represented using syntax coloring to help identify the different elements. For example, SCL keywords and commands are represented in blue. A Script is divided into three sections represented by the following SCL keywords; **Environment**, **Definitions** and **Code**.

See also:

The Environment Section

The Definitions Section

The Code Section

Automated Script Formatting Features

The Environment Section

The Environment section is always the first part of a Script. It is introduced by the mandatory **Environment** keyword. It is preceded by comments written by the Gateway which note the browser used and the creation date.

This section is used to define the global attributes of the Script including a **Description**, if you choose to add one, the **Mode** and **Wait** units, for example:

```
!Browser:IE5
!Date : 11-Dec-00
Environment
        Description ""
        Mode        HTTP
        Wait        UNIT MILLISECONDS
```

See also:

The Definitions Section

The Code Section

The Definitions Section

The Definitions section follows the Environment section and is introduced by the mandatory **Definitions** keyword. It contains all the definitions used in the Script, including definitions of variables and constants, as well as declarations of Timers and file definitions.

```
Definitions
        ! Standard Defines
        Include "RESPONSE_CODES.INC"
        Include "GLOBAL_VARIABLES.INC"
```

The **RESPONSE_CODES.INC** is an include file which contains the definitions of constants which correspond to HTTP/S response codes.

The **GLOBAL_VARIABLES.INC** file is used to hold variable definitions of global and Script scope which are shared by Virtual Users during a Test-run.

See also:

The Environment Section

The Code Section

The Code Section

The Code section follows the Definitions section and is introduced by the mandatory **Code** keyword. It contains commands that represent the Web-activity you have recorded and define the Script's behavior. The Code section is composed of SCL commands that control the behavior of the Script.

See also:

The Environment Section

The Definitions Section

Automated Script Formatting Features

During the recording of a Script the HTTP/S traffic is written in SCL code to produce a Script. A Script combines a variety of automatically encoded features which are incorporated during the creation process. Representing the Web activity recorded during a Web session in SCL code enables Scripts to be modeled more easily and to be replayed as part of a Test without the requirement to model.

Some of the automated formatting features are listed below:

Syntax Coloring

Script data is represented using syntax coloring to help identify the different elements. SCL keywords, commands and clauses are represented in blue, comments are green, strings are magenta, operators are red, numbers and text are black.

Generation of Timers

Generation and insertion of code to time the period that elapses between the issuing of an HTTP/S request and the loading of Web pages and the duration of Script replay.

Generation of Waits

Generation and insertion of code that suspends Script execution for a finite period. A Wait command represents a pause in browser activity.

Creation and Modification of Variables for Cookies

Automatic generation and insertion of variables into the Definitions section of a Script to replace any cookies issued by a WAE with a new variable definition and record them in the Script. Script Modeler automatically substitutes the unique session identity defined by a cookie which enables you to replay a Script as part of a Test to function as one or multiple Virtual Users.

HTTP Commands

The HTTP commands provide facilities for issuing HTTP requests for resources, examining and interrogating the response messages and synchronizing requests. The HTTP commands are as follows:

GET command: Issues an HTTP GET request to retrieve a URI.

POST command: Issues an HTTP POST request for the WAE to accept some data from the client.

HEAD command: Issues an HTTP HEAD request to retrieve a URI, but the WAE does not return the associated Web page or object.

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

LOAD RESPONSE_INFO HEADER command: Loads a character variable with all or one of the HTTP response message header fields for a specified TCP connection.

SYNCHRONIZE REQUESTS command: Causes the thread currently executing to be suspended immediately, until responses have been received for all the requests that have been issued by the thread.

CONNECT command: Used to establish a TCP connection to a nominated Host computer.

DISCONNECT command: Used to close one or all of the TCP connections established using the CONNECT command.

Text Layout and Formatting

Left aligned text including tabs and spaces, are features of a Script which are incorporated to format SCL commands and other content. This formatting aids the legibility of Scripts and has no other effect on compilation.

See also:

Modeling a Script

Modeling a Script

The Tests you develop and the Scripts they contain are largely dependent on the structure and function of the WAE(s) you are testing and the results you want to achieve. Planning and developing the appropriate Scripts is crucial in the development of a successful performance Test.

To produce the Tests you need it may be necessary to model the Scripts they contain. The following example demonstrates some common modeling procedures using Script Modeler, which you can apply in the Test development process.

The example includes the following procedures:

- Open a Script from Commander or Open a Script from Script Modeler
- Create a Variable
- Apply MUTEX Locking
- Locate Login Details and Insert Variables
- DOM Addressing

The example documents the procedures involved in modeling a user name and password to enable the simulation of multiple Virtual Users with unique identities when a Test is run. It uses a single Script captured over an Internet connection that records launching, logging on, conducting a search and then logging off from the OpenSTA demo WAE, *Which US President?*. The search conducted cross-references `Democrat' with `Baptist', and returns the names of three former presidents who fit into this category.

The demosite can be downloaded or launched directly, from **http://opensta.org/**.

For more information on SCL commands, refer to the *SCL Reference Guide*, which is available within Script Modeler's on-line help system.

Note: See Developing a Modular Test Structure, for an overview of modeling Scripts to create a modular Test incorporating a sequence of two or more Scripts.

See also:

Variables

Developing a Modular Test Structure

Open a Script from Commander

1. In the Repository Window within Commander, double-click **Scripts**, to expand the directory structure.
2. Double-click on the Script (or Include file) you want to open represented by or .

   Note: indicates a new Script before HTTP/S traffic has been recorded.

See also:

Open a Script from Script Modeler

Open a Script from Script Modeler

1. Click , or select **File > Open**.
2. In the Open Capture dialog box, double-click the Script you want to open or select a Script and click **OK**.

See also:

Open a Script from Commander

Variables

When you create a new variable you need to choose the attributes it embodies in order to perform the functions you want when a Test is run. You can assign the properties you require from within the Variable Creation dialog box. To setup new variables select **Variable > Create**, or to edit existing variables, select **Variable > Modify**. Use these dialogs to control the function of your variables by selecting the settings you need.

The settings you apply are represented within the Script by SCL code, or option clauses, which are depicted in blue, bold text within the Script Pane. Once you have created a variable it is represented as a text string within the Definitions section of the Script. The example below displays a variable named USER, which contains one SCL keyword and two SCL option clauses:

```
CHARACTER*512 USER ( "user1", "user2", "user3" &
, "user4", "user5" ), LOCAL, RANDOM
```

The **CHARACTER** keyword defines the type of variable, **LOCAL** identifies its scope and **RANDOM** indicates the method this variable uses to select values. The entry ***512**, indicates the maximum character length the variable can contain. The variable values are listed after the variable name. For more information on variables refer to the *SCL Reference Guide*.

See also:

Variable Options

Variable Scope Options

Variable Value Source

Variable Order

Variable Type

Create a Variable

MUTEX Locking

Variable Options

All the options relating to developing new variables can be accessed from within the Variable Creation dialog box, click **Variable > Create**. However, you can define the prefix you use for new variables before you create them.

See also:

Specify The Prefix Name for Your Variables

Variable Scope Options

Specify The Prefix Name for Your Variables

1. Open a Script, then select **Options > Variables**.
2. Enter the prefix text in the Variable Options dialog box.

   Note: The name you give to a variable must conform to the rules of OpenSTA Datanames and cannot be longer than 16 characters.
3. Click **OK**.

   The prefix appears in the Name text box of the Variable Creation dialog box when you next create a variable.

Variable Scope Options

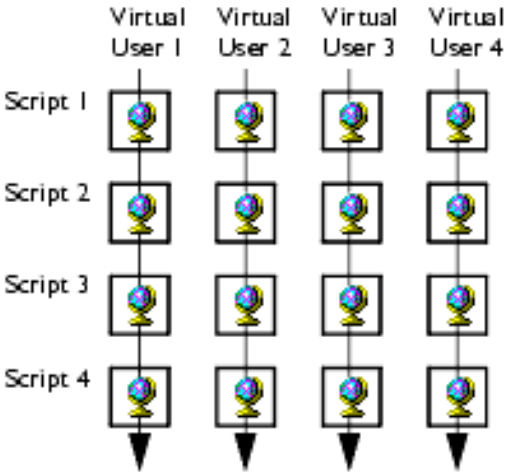
The Scope of a variable determines which Virtual Users and Scripts can make use them during a Test-run. The default variable scope is Local.

There are four variable scope settings available:

Local Scope Variables

Local scope variables are only accessible to the Virtual User running the Script in which they are defined. They cannot be accessed by any other Virtual Users or Scripts. Similarly, a Script cannot access any of the local variables defined within any of the Scripts it calls.

A Local scope variable can only be used by Virtual User 1 in Script 1, Virtual User 1 in Script 2, Virtual User 2 in Script 1, etc. Each Virtual Users copy of the variable can only be referenced and used by them.



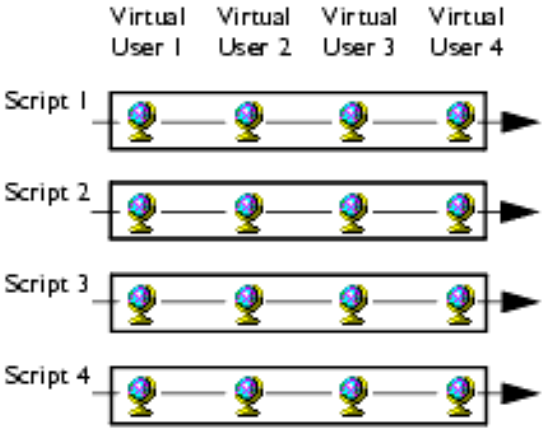
This matrix represents four Virtual Users running the same modular Test comprised of 4 Scripts.

The black squares represent the scope of a Local variable during a Test-run.

Script Scope Variables

Script scope variables can be accessed by any Virtual User running the Script in which they are defined.

A Script scope variable can be used by Virtual Users 1 to 4 in Script 1 or by Virtual Users 1 to 4 in Script 2 etc. There is only one copy of the variable which can be shared by any user. It can only be referenced and used within the Script that it is defined.



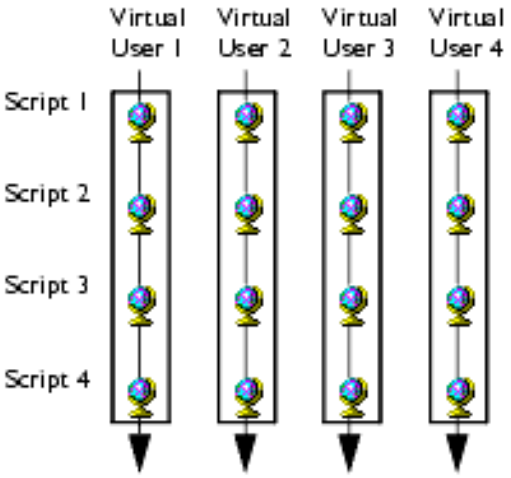
This matrix represents four Virtual Users running the same modular Test comprised of 4 Scripts.

The black rectangles represent the scope of a Script variable during a Test-run.

Thread Scope Variables

Thread scope variables are accessible from any Script run by the Virtual User, or thread, that defines them.

A Thread scope variable can be used by Virtual User 1 in Scripts 1-4, Virtual User 2 in Scripts 1-4, etc. Each Virtual User has their own copy of the variable which cannot be referenced or used by another Virtual User. A Thread scope variable must be defined in every Script in the sequence that uses it. Alternatively, define a thread scope variable in the Global_Variables.INC file to include it in every Script.



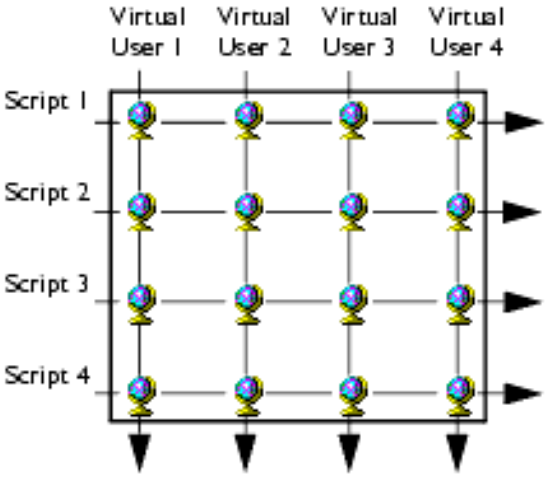
This matrix represents four Virtual Users running the same modular Test comprised of 4 Scripts.

The black rectangles represent the scope of a Thread variable during a Test-run.

Global Scope Variables

Global scope variables are accessible to any thread running any Script under the same Test Manager, that is, any Script in a particular Test-run.

A Global scope variable can be used by any Virtual User in any Script. However, it must be defined in each Script it is to be used in or included in the Global_Variables. INC file.



This matrix represents four Virtual Users running the same modular Test comprised of 4 Scripts.

The black square represent the scope of a Global variable during a Test-run.

See also:

Variable Value Source

Variable Value Source

The Value Source option enables you to create new values for your variables or to use an existing source which can be externally accessed, from a file or a database. If you want to utilize an external value source select either **File** or **Database** from the Value Source drop down list and locate your value data.

The default option is the Value List.

See also:

Variable Order

Variable Order

The Order option controls how the variable values are selected during a Test-run. The choice for the selection of variable values is **Sequential** or **Random**. Choosing a sequential order for the selection of variable values results in values being used consecutively from your value list.

The Random order option dictates that a variable value is selected randomly from a list or range, when the variable is used in conjunction with the GENERATE command (refer to the *SCL References Guide* for more information). The values are selected in a different order each time they are generated. This is achieved by generating a different seed value for the variable each time the variable is initialized. Local scope variables are initialized when a Test-run begins. Script scope variables are initialized by the first thread to run the Script.

See also:

Variable Type

Variable Type

The variable Type option enables you to define whether a variable is numeric or text type. Typically the **Character** option is used for modeling user names, passwords, and other text based elements. The **Integer** type variable facilitates the introduction of equations into the variable for more advanced modeling tasks.

See also:

Create a Variable

Create a Variable

Note: For more information on creating variables see Variables.

1. Open a Script, then select **Variable > Create**.

   Shortcut: Click  in the Variable Toolbar.

2. In the Variable Creation dialog box, enter a name for your new variable. In this example the name is **USERNAME**.

   Note: The name you give must be a OpenSTA Datanames.

3. Select the Scope of your variable. In this example the selection is Script. Click  and choose from:

   Local: Only accessible to the Virtual User running the current Script.

   Script: Accessible to any Virtual User running the current Script.

   Thread: Accessible to any Script run by a specific Virtual User.

   Global: Accessible to any Script and any Virtual User.

   Note: The scope of a variable relates to which Virtual Users and Scripts can make use of the variables you create.

4. Select the Value Source of your variable. In this example the selection is Value List. Click  and choose from:

   Value list: Enter your own variable values.

   File: Use existing variable values from file.

   Database: Use existing variable values stored on a database.

5. Select the order in which the variable values are selected when a Test is run. In this example the selection is Sequential. Choose from:

   Sequential: Assigns variable values will be consecutively from your value list.

Random: Assigns variable values randomly from your value list.

6. Select the data types of the variable. In this example the selection is Character. Choose from:

   Character: Text variable.

   Integer: Numeric variable.

7. Click **Next** when you have made your selections.
8. In the Value List dialog box you need to enter the variable values, or names that will represent the Virtual Users you need when the Test is run. In this example there are five values or user names entered manually within the Value List dialog box, as described below.

- You can enter the variable values you need freehand, by double-clicking inside the Value text box or click the  , and enter your variable value.
- Or, click **Generate Values** to automate the process. In the generation Parameters dialog box, give your values a prefix, then specify the number of Virtual Users you want by entering a number range in the **From** and **To** text boxes. The **Step** function controls

  Note: If you select File or Database as your Value Source, clicking **Generate Values** takes you to different dialog boxes from where you can locate your value sources.

9. After you have created your variable values, click **OK** to return to the Value List dialog box and use the Value List toolbar buttons to manipulate your entries.

   Click on a value in the list, click  to delete it, click  to move the item up the list and click  to move the item down.

10. Click **Finish** when the setup process is complete.
11. Repeat this process to create the **PASSWORD** variable, which your five Virtual Users will need in order to access the *Which US President?* WAE. Note: This WAE requires a password to be the reverse spelling of the login name.

    The variables you have created are represented as text strings within the Definitions section of the Script, as illustrated below:

```
CHARACTER*512 USERNAME ( "phillip", "allan", "david" &
, "robert", "donna" ), SCRIPT
CHARACTER*512    PASSWORD ( "pillihp", "nalla", "divad" &
, "trebor", "annod" ), SCRIPT
```

The variables are now ready to be substituted for the original login identity recorded

in the Script. But before you can do so you must apply MUTEX locking SCL code to ensure there are no sharing violations between Virtual Users during a Test-run.

12. Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar, to compile the Script.

    Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

13. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.

    Select **Capture > Replay** or click  , in the Capture/Replay Toolbar. The replay activity is displayed in the Output Pane.

14. Click  , to save the Script, or click **File > Save**.

Edit a Variable

1. Open a Script, then working in the Script Pane, find the variable you want to edit in the Definitions section.
2. Click an insertion point within the variable string, click **Variable > Modify**, and make your changes in the Variable dialog box.

   Shortcut: Click  in the Variable Toolbar.

3. Click  , to save the Script, or click **File > Save**.

MUTEX Locking

MUTEX locking is a straightforward procedure that you can perform in order to ensure a Test-run which simulates unique, multiple Virtual Users is successful.

During a Test-run that simulates Virtual Users with unique identities by using a Script scope value list, it is possible for them to acquire the same variable values. When a Test is run the Virtual Users specified by the Task Group settings, take their identities sequentially from the list defined in the Script's USERNAME variable, in this example. This list of unique identities defined by a variable's list is shared by the Virtual Users running a Test, which means that when several Virtual Users access the list in rapid succession user identities can be duplicated in the milliseconds between one Virtual User claiming and setting their identity, by the next Virtual User accessing the list. MUTEX locking is used as a fail safe command that prevents this from happening. A variable supporting unique identities for Virtual Users during a Test-run needs to be shared so all Virtual Users can access the variable. Therefore, the scope of the variable must be Script or global scope.

In the current example, the user name and password details of Virtual Users must remain unique for the Test to produce useful results. Inserting a MUTEX lock prevents other Virtual Users from acquiring the MUTEX and therefore ensures the

unique identities of each Virtual User when a Test is run.

AQUIRE MUTEX is the SCL command which gives a Virtual User exclusive access to the MUTEX you define, which in this case is a user name and password value contained in the USERNAME and PASSWORD variables. The SCL code assigns the first user name value from the specified variable and makes a local copy of this value, which prevents any other Virtual User from using it, until the MUTEX is released with **RELEASE MUTEX** command. For more information about MUTEX and the additional options available, refer to the *SCL Reference Guide*.

See also:

Apply MUTEX Locking

DOM Addressing

Apply MUTEX Locking

1. Open a Script, then working in the Script Pane, find the login details in the Code section.
2. You must insert the MUTEX command before the login details referenced in the Script, which in this example is before the **PRIMARY POST URI** entry.
3. The name you give your MUTEX element is up to you, for this example LOGIN is used.

```
ACQUIRE MUTEX "LOGIN"
NEXT USERNAME
NEXT PASSWORD
SET MY_USERNAME = USERNAME
SET MY_PASSWORD = PASSWORD
RELEASE MUTEX "LOGIN"
```

4. The **NEXT** command loads a variable with the next sequential value from the USERNAME and PASSWORD variables. When the **NEXT** command is first executed, it will retrieve the first value. The variable value set is treated as cyclic, so when the last value has been retrieved, the next value retrieved is the first in the set.
5. The **SET** commands make local copies of the variable value loaded using the **NEXT** command.
6. You need to declare the MUTEX elements you have created in the Definitions section. In the current example they should appear as below:

```
CHARACTER*512 MY_USERNAME, LOCAL
CHARACTER*512 MY_PASSWORD, LOCAL
```

7. Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar, to compile the Script.

   Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

8. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.

   Select **Capture > Replay** or click  , in the Capture/Replay Toolbar. The replay activity is displayed in the Output Pane.

9. Click  , to save the Script, or click **File > Save**.

See also:

Locate Login Details and Apply USERNAME and PASSWORD Variables

Locate Login Details and Apply USERNAME and PASSWORD Variables

1. Open a Script, then working in the Script Pane, find the login details in the Code section so you can edit the code and apply your variables.

   Use the scroll bars in the Script Pane, or the text search facility to locate the section of the Script which records the login data.

   Tip: Right-click inside the Script Pane and select **Find**, or click  , then in the Find dialog box type in **loginid** and click **OK**. Press **F3** to find the next instance.

   In the OpenSTA demosite the login details you are looking for are located below the first Primary Post in the Code section of the Script. The relevant section of code is illustrated below:

```
        PRIMARY POST URI"http://demosite.opensta.org/gsg-v1
HTTP/1.0"ON 2 &
        HEADER DEFAULT_HEADERS &
        ,WITH {"Accept: application/vnd.ms-excel, application/msword,
                application/vnd.ms-powerp" & "oint, image/gif, image/
x-xbitmap,
                image/jpeg, image/pjpeg, */*", &
                "Referer: http://demosite.opensta.org/gsg-v1", &
                "Accept-Language: en-us", &
                "Content-Type: application/x-www-form-urlencoded", &
                "Content-Length: 22", &
                "Pragma: no-cache"} &
        ,BODY "loginid=mike&passwd=ekim"
```

```
Login details are located after the SCL clause BODY.
```

2. Replace the original session details as illustrated below:

```
,BODY "loginid="+MY_USERNAME+"&passwd="+MY_PASSWORD
```

3. Click  , to save the Script.

   Your Script is now adequately modeled to simulate the Virtual Users required. However, it useful to incorporate Document Object Model, DOM, references to help you verify the results you get when a Test is run. This modeling method is known as DOM Addressing.

4. Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar, to compile the Script.

   Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

5. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.

   Select **Capture > Replay** or click  , in the Capture/Replay Toolbar. The replay activity is displayed in the Output Pane.

6. Click  , to save the Script, or click **File > Save**.

See also:

Addressing a DOM Element

DOM Addressing

DOM Addressing is a modeling technique that enables you to effectively validate a Test by checking that the HTML responses are correct.

The term DOM Addressing describes the ability to access specific and unique elements within the DOM (Document Object Model) or Web page and use it to model the corresponding Script.

When a Script is recorded, the Web pages returned from the WAE in response to

browser requests issued are saved in a .ALL file, at the same time as the Gateway produces the .HTP or Script file. These HTML responses include DOM information plus other categories of data that are directly related to, and dynamically linked with the Script.

During a Test-run the contents of a particular DOM signature or DOM object address, can be recovered dynamically. This enables modeling of the dynamic nature of some Web pages by:

- Providing information in the Script that can be used to verify that the application is working. For example, by enabling a text-matching Test to be performed.
- Enabling the adaptive loading of objects (probably via secondary GETs) that are part of the dynamic content of the current page, but were not present in the recorded page.
- Emulating the clicking of different links within a Web page that may be unique for individual users.

After you have modeled and run the Test open the Test Report Log from the Results Window to display the responses generated.

Producing a meaningful performance Test requires an thorough knowledge of the WAE functionality you are testing. In the current example, a Virtual User launches the *Which US President?* WAE and conducts a search, cross referencing `Democrat' with `Baptist', which returns three former presidents who fit in to this category. Knowing how the WAE should behave in these circumstances enables you to model the Script to report the results returned by the WAE to each Virtual User involved in the Test-run.

See also:

Addressing a DOM Element

Developing a Modular Test Structure

Addressing a DOM Element

In this example the DOM Addressing procedure is used to verify the Test results by checking that the Web pages returned from the WAE during Test-run and search results they contain are correct.

1. Open a Script, then working in the Script Pane, search for a **`PRIMARY POST URI`** or a **`PRIMARY GET URI`** in the Code section.
   Shortcut: Click in the Address Bar and select the URL you want to view from the list to move to this entry in the Script.
2. Click the URL Details button  , in the Standard Toolbar, or click **View > URL Details**, to open the .ALL file. The **`GET`** or **`POST`** you selected is highlighted by the appearance of  , on the left of the selected URL text

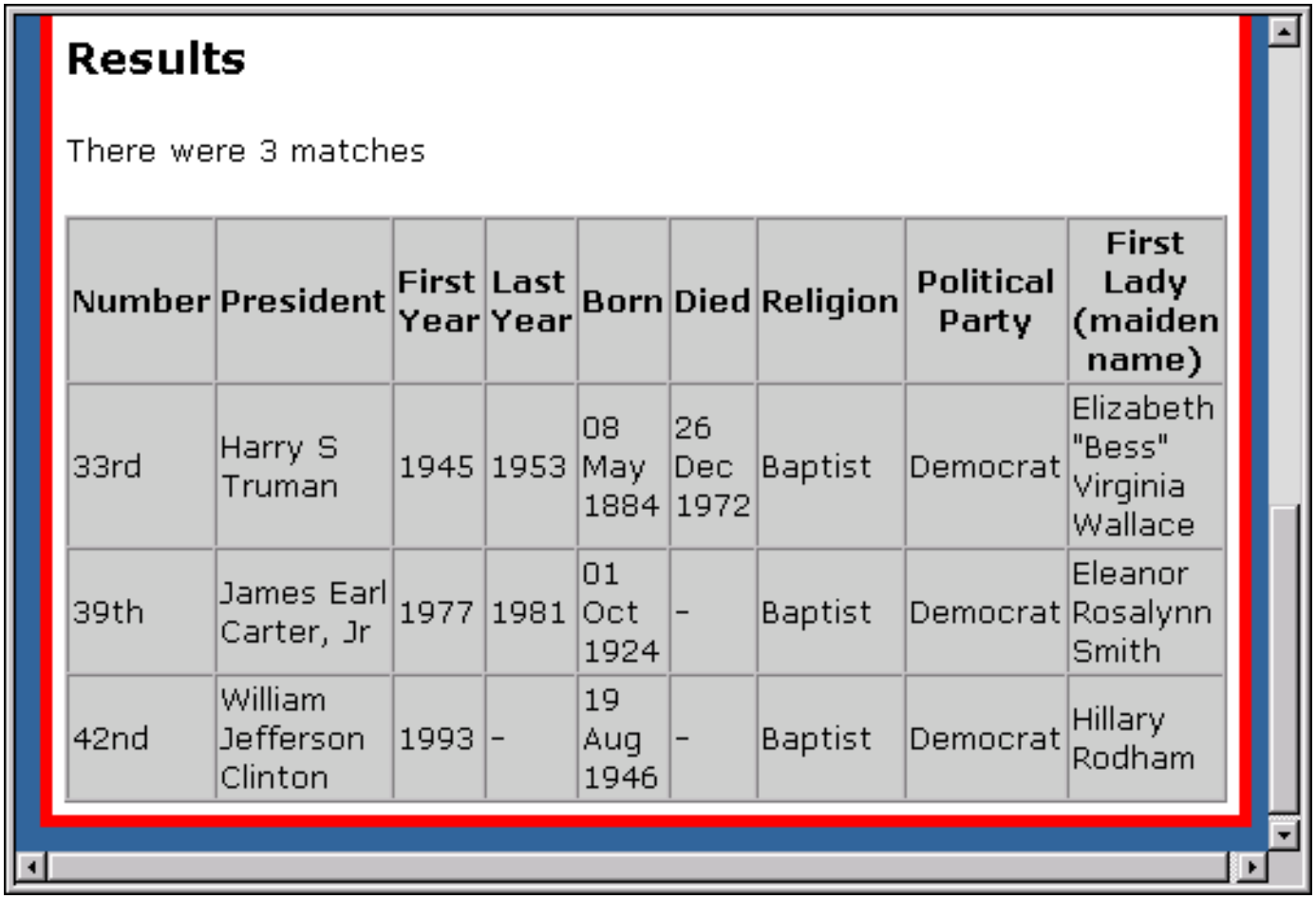
string.

This displays HTML information in the Query Results Pane relevant to the URL you have selected in the Script Pane.

Note: The .ALL file is simultaneously recorded with the .HTP file, or Script, during the original HTTP/S capture process.

3. In the Query Results Pane, click on the **HTML** tab to display the Web page and confirm it contains the results you want to model.

   Click and drag the Query Results Pane borders to expand the work space if necessary.



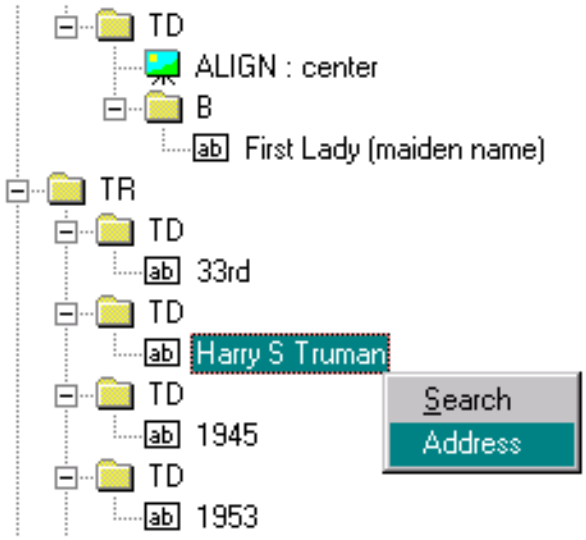
Results

There were 3 matches

Number	President	First Year	Last Year	Born	Died	Religion	Political Party	First Lady (maiden name)
33rd	Harry S Truman	1945	1953	08 May 1884	26 Dec 1972	Baptist	Democrat	Elizabeth "Bess" Virginia Wallace
39th	James Earl Carter, Jr	1977	1981	01 Oct 1924	-	Baptist	Democrat	Eleanor Rosalynn Smith
42nd	William Jefferson Clinton	1993	-	19 Aug 1946	-	Baptist	Democrat	Hillary Rodham

4. After you have located the returned Web page that contains the search results, click the **DOM** tab to display the Document Object Model structure of the Web page.
5. The first returned president is Harry S Truman. Scroll down through the DOM tree structure to locate the DOM object that contains this reference, or use the Search function. Right-click in the DOM view of the Query Results Pane and click **Search**, then enter the text you want to search for in the Find dialog box.

   Right-click on the DOM object **Harry S Truman** and click **Address**. As illustrated in the partial view of the DOM structure sampled from the Query Results Pane below:



6. In the Addressing dialog box, give your variable a name, in this case **PRESIDENT**, and click **OK**.

   The following text string is added in the appropriate place within the current Script displayed in the Script Pane:

```
LOAD RESPONSE_INFO BODY ON 3 &
INTO PRESIDENT &
,WITH  "HTML(0)/BODY(1)/TABLE(2)/TBODY(0)/TR(0)/TD(0)/
        TABLE(4)/TBODY(0)/TR(1)/TD(1):TEXT:(0)"
```

The text string that appears after the **,WITH** command indicates the path or location, of the DOM object within the DOM structure, also known as the DOM address.

7. When you add a DOM address variable the details are stored automatically in the Definitions sections of the Script. Scroll up to check that the DOM variable has been registered. In this example it appears as illustrated below:

```
CHARACTER*512 PRESIDENT ,LOCAL
```

Note: A Script will not compile unless all variables correctly are recorded in the Definitions section of the file.

8. The DOM variable produces no output after Test-run without additional modeling, this involves entering an SCL command.

   Below the **,WITH** clause and the DOM address, in the Code section of the Script, click an insertion point and press **Return**.

   Then type **REPORT** or **LOG**, along with any comments you want included in your results.

The **REPORT** command writes a message to the Report log, and the **LOG** command writes a message to the Audit log, which can be viewed after the Test-run is complete from within Commander.

In the current example, the SCL command and comments are structured as illustrated below:

```
LOAD RESPONSE_INFO BODY ON 3 &
INTO PRESIDENT &
,WITH  "HTML(0)/BODY(1)/TABLE(2)/TBODY(0)/TR(0)/TD(0)/
        TABLE(4)/TBODY(0)/TR(1)/TD(1):TEXT:(0)"

REPORT  "USER ", MY_USERNAME, " SEARCHED FOR DEMOCRAT-BAPTIST
AND
        GOT ", PRESIDENT
```

The **MY_USERNAME** entry ensures that the results generated are written to file for each Virtual User. The comment text also appears to give context to the results and **PRESIDENT** refers to the DOM variable which calls the WAEs responses for each Virtual User during a Test-run.

9. Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar, to compile the Script.

   Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

10. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.

    Select **Capture > Replay** or click  , in the Capture/Replay Toolbar. The replay activity is displayed in the Output Pane.

11. When you are happy with your Script, click  , in the Standard Toolbar to save the Script, or click **File > Save**.
12. Click **File > Close** to close the current Script, or click **File > Exit** to shutdown Script Modeler.

    Note: If you have unsaved Scripts open in Script Modeler, you are automatically prompted to save them before the program closes. Closing down Script Modeler also closes the browser which restores your original browser settings.

Developing a Modular Test Structure

Developing a modular Test structure involves creating two or more Scripts then combining them sequentially in a Test to represent a continuous Web browser session when the Test is run. For example, this may include a log on Script,

followed by one or more Scripts that record the Web services that you want to test, then a log off Script.

A modular Test can take longer to develop than a Test that references a single Script, but once the modular structure is in place maintaining it is easier. It is then possible to re-record and replace individual Scripts which have been affected by functional changes to a WAE, rather than re-recording all the Web activity which would be necessary if your Test used a single Script.

When Web services are modified and Tests need updating you will only need to recreate the affected Scripts within a modular Test structure. This means that you do not waste time re-recording and re-modeling every Script in the sequence, which is work you would have to do if you use a single Script to record a browser session.

Creating Scripts that encapsulate specific Web activity also enables them to be used in different Tests. This can help reduce the amount of time you would otherwise need to spend on recording and modeling Scripts.

With a modular Test structure in place you can configure the Script Tasks in a sequence to repeat using the Task settings option, enabling you to simulate multiple searches by a single Virtual User. For example if one of the Scripts in the Task Group sequence records a search, you can apply a Task iteration setting to this Script Task during Test development in order to repeat the search as often as required during a Test-run. To further enhance the realism of the browser activity simulated during a Test-run you could model the search Script by replace the item searched for with a variable to simulate multiple unique searches using the same Task Group

Recording Scripts and then building a modular Test is no problem if the WAE you are targeting does not generate cookies or URL session identities. But if the WAE under Test uses cookies, you need to be able to manipulate the session identities they record to enable the Scripts to run in sequence. The session identity needs to be shared between Scripts by creating a new variable which unifies the cookies recorded in different Scripts. Minor edits to the Scripts you want to include in your Test are required, as well as the use of the Global_Variables.INC to record the new session identity variable. The Global_Variables.INC is a resource file that can be used to make variable values available across all Scripts in the Repository.

Open the Global_Variables.INC and define your variable, then reference the variable name at the end of each Script in the sequence. At the beginning of each subsequent Script assign the global cookie value to the Script so that they all share the same identity and function correctly during a Test-run.

See also:

Model Scripts to Run in Sequence During a Test-run

Creating Scripts

General Modeling Procedures

The Test Development Process

Model Scripts to Run in Sequence During a Test-run

This following procedure documents the modeling of Scripts for incorporation into a Task Group in sequence, in order to simulate a continuous Web browser session when replayed during a Test-run.

Note: The procedure references Scripts which were captured over an Internet connection from the OpenSTA demo WAE, *Which US President?*. Aspects of this procedure will vary when applied to Scripts recorded from different WAEs depending on their functionality and in particular how they handle cookies.

1. Ensure that Script Modeler is set to automatically model cookies before you record Scripts. For more information see Select Automatic Cookie Modeling. Open the Global_Variables.INC file from Script Modeler by selecting **Tools > Edit** and double-clicking **GLOBAL_VARIABLES.inc** in the Edit File dialog box.

   Or, from Commander, double-click **Scripts**, in the Repository Window, double-click **Include**, then double-click the **GLOBAL_VARIABLES** file.

2. Enter the variable type, name and scope, in this example:

   ```
   CHARACTER*1024 SESSIONID, THREAD
   ```

3. Click , to save the file or click **File > Save**.

   Now you need to replace the hard coded session ID values of the cookies recorded in your Scripts with the variable you have created in the Global_Variables.INC.

4. Open the Script that you want to run first in the Task Group sequence.

   The Script you are modeling may contain several cookies each with different functions. You need to identify which one contains session ID information.

   To do this search for the Load Response_Info Header command in the Script to identify the WAE issued cookie that contains session ID information. For example:

   ```
   Load Response_Info Header on 1          &
           Into cookie_1_0         &
           ,WITH "Set-Cookie,findpresid"
   ```

In this example **findpresid** is the cookie issued by the WAE that contains session ID information and has been assigned to the modeled cookie, **cookie_1_0**.

5. Scroll down to the end of the Script and click an insertion point between the **Endif** and **Exit** commands and type the following:

```
Set SESSIONID = cookie_1_0
```

6. Click  , to save the Script, or click **File > Save**.
7. Then compile the Script by clicking  in the Capture/Replay Toolbar, or selecting **Capture > Syntax Check**. Compilation results are reported in the Output Pane.
8. Open the Script that you want to run next in the Task Group sequence.
9. Before the first HTTP request find the first automatically modeled cookie. Click in the Address Bar and select the first HTTP request in the list to move to this entry in the Script. In this example the command appears:

```
Set S_cookie_2_0 = "findpresid=Dave,996069766"
```

Replace the cookie value with the variable you have created, **SESSIONID**, for example:

```
Set S_cookie_2_0 = SESSIONID
```

The cookie now has the same session ID value as the cookie at the end of the first Script.

10. Scroll down to the end of the Script insert the **SESSIONID** variable between the **Endif** and **Exit** SCL commands. Click an insertion point between these commands and enter the variable name. For example:

```
Set SESSIONID = cookie_2_0
```

Note: In this example **cookie_2_0** contains the session ID information

11. Save and compile the Script.

12. Repeat steps 9-12 in the other Scripts that you want to include in the modular Test structure.

    Note: The last Script in the sequence does not need to include the **Set** command details at the end of the Script.

    After you have modeled the Scripts you are ready to develop the Test by adding them to a Task Group in the planned order.

    Tip: Try single stepping the Task Group to check that it behaves as you expect, in particular to make sure that the modular Task Group replays as a continuous Web browser session. Use the Single Step Results option to view the HTTP responses recorded during Task Group replay to check your modeling has been successful and that the Test is valid.

    You can also access the Web server log to check that the Scripts are running in sequence.

See also:

Developing a Modular Test Structure

Add Scripts to a Test

Run a Test

The Test Development Process

SIngle Stepping

General Modeling Procedures

Comprehensive information on SCL commands and their syntax is documented in the *SCL Reference Guide*, which is available within Script Modeler's on-line help system.

Single Stepping, Comments

Transaction Timers

Wait Commands

Call Scripts

Syntax Check

Find and Replace Variables in Strings

Single Stepping, Comments

Comments can be added to Scripts in order to give some explanation of their

content when monitoring Task Group replay during a single stepping session.

The function of HTTP requests included in a Script are not always obvious, particularly if the WAE you are testing issues the same or similar HTTP requests, but the functionality of each Web page returned is different. You can add Comments before URLs for example, to indicate which commands are about to run.

During a single stepping session Comments can be displayed when a Task Group is replayed. As replay proceeds the information you have added about HTTP requests or other Script items is displayed, which can help to make monitoring Task Group replay easier.

Comments can be inserted while recording a Script or added afterwards. During a recording session, click the Add Comment button  in the Capture/Replay Toolbar, or select **Capture > Insert Comments**. The time taken to add Comments is not included in Scripts.

Comments may be incorporated into Scripts either on lines by themselves or embedded in statements and commands. In both cases, they are identified by the SCL command **!USC:**.

See also:

Add a Single Stepping Comment to a Script

Single Stepping

Create a New Script

SCL Reference Guide

Add a Single Stepping Comment to a Script

Note: Comments can be added during Script recording using the Add Comment button  .

1. Open a Script, then working in the Script Pane, move to the position where you want to add a Comment.

   Tip: Click in the Address Bar and select the URL you want to view from the list to move to this command in the Script.

2. Click an insertion point in the Script where you want to add a Comment and type: **!USC:**

   Note: A Comment can be incorporated into a Script either on a line by itself or embedded in a statement or command.

3. Enter the Comment text after **!USC:**

4. Compile the Script to check that it will replay correctly by clicking  , in the Capture/Replay Toolbar.

Note: Compilation results are reported in the Output Pane.

5. Click  to save your changes.

   Note: Comments can be displayed in the Script Item list during a single stepping session. Click  in the single stepping toolbar to show and hide them.

See also:

Single Stepping

Create a New Script

SCL Reference Guide

Transaction Timers

Scripts can be modeled to include Transaction Timers that are used to measure the duration of user-defined HTTP transactions within a Script-based Task Group when a Test is run.

During Script creation Timers are automatically generated to measure the period of time that elapses between an HTTP request being issued and the corresponding Web page being loaded, and also the duration of Script replay. Transaction Timers enable you to measure a series of HTTP requests that represent a complete Web browser transaction.

A Transaction Timer defines a sequence of SCL code within a Script in order to measure the duration of the HTTP transaction delimited. An HTTP transaction is a user-defined sequence of SCL code within a Script contained by the **Start Timer TRANS_** and the **End Timer TRANS_** SCL commands in the Code section of a Script.

During a single stepping session you can display Transaction Timers when a Task Group is replayed. As the replay proceeds the Start and End Transaction Timer commands are listed along with the HTTP requests that comprise the HTTP transaction they are measuring.

Transaction Timer results are available in the Timer List, the Timer Values v Active Users graph and the Timer Values v Elapsed Time graph.

See also:

Add a Transaction Timer to a Script

Timer List

Single Stepping

SCL Reference Guide

Add a Transaction Timer to a Script

1. Open a Script, then working in the Script Pane, move to the Definitions section of the Script to define your Transaction Timer name using the Timer statement.
2. Click an insertion point in a new line and type **Timer TRANS_** and the Transaction Timer name, which must be an OpenSTA Dataname, for example: **Timer TRANS_First_Purchase**

   After using the Timer statement to declare the HTTP transaction, use the Start Timer TRANS_ and End Timer TRANS_ commands to delimit the code you want to measure.
3. Click an insertion point in the Code section of the Script where you want to start your Transaction Timer and type **Start Timer TRANS_** and the Timer name, for example: **Start Timer TRANS_First_Purchase**
4. Click an insertion point in the Code section of the Script where you want to end your Transaction Timer and type **End Timer TRANS_** and the Timer name, for example: **End Timer TRANS_First_Purchase**
5. Click  in the Capture/Replay Toolbar to compile the Script and check it replays correctly. Compilation results are reported in the Output Pane.
6. Click  to save your changes.

   Note: Transaction Timers can be displayed in the Script Item list during a single stepping session. Click  in the single stepping toolbar to show and hide them.

   Note: Results generated using Transaction Timers are displayed in the Timer List, the Timer Values v Active Users graph and the Timer Values v Elapsed Time graph.

See also:

Timer List

Single Stepping

SCL Reference Guide

Wait Commands

Wait SCL commands are generated and inserted automatically during Script creation and represent a pause in Web browser activity. When a Script is replayed during a Test-run, Script execution is suspended for a finite period according to the Wait command values included in the Script.

As response times improve during WAE development, the Wait command values recorded no longer reflect the speed of the Web application. In these circumstances you can either re-create the Script or edit existing Wait command values to improve the accuracy of the Script when it is replayed during a Test-run.

During a single stepping session you can display Wait commands to help make monitoring Task Group replay more transparent. As the replay proceeds the Wait commands are displayed including the time value of each Wait. Wait command time periods are either recorded in seconds or milliseconds depending which unit is defined by the Wait UNIT statement in the Environment section of the Script.

See also:

Edit Wait Values in a Script

Single Stepping

SCL Reference Guide

Edit Wait Values in a Script

1. Open a Script, then locate a Wait command you want to edit.

   Tip: Right-click inside the Script Pane and select **Find**, or click  , then in the Find dialog box type in **Wait** and click **OK**. Press **F3** to find the next instance.

2. Click an insertion point within a Wait command.
3. Delete the existing value and enter the new Wait period in its place.

   Note: Find the Wait UNIT statement in the Environment section of the Script to check whether Wait command values are recorded in seconds or milliseconds.

4. Compile the Script to check that it will replay correctly by clicking  , in the Capture/Replay Toolbar.

   Note: Compilation results are reported in the Output Pane.

5. Click  to save your changes.

   Note: Wait commands can be displayed in the Script Item list during a single stepping session. Click  in the single stepping toolbar to show and hide them.

See also:

Single Stepping

SCL Reference Guide

Call Scripts

The Call Script SCL command enables you to execute a Script that is not included in a Task Group when a Test is run. A Script modeled to include this command can call a named Script, or a variable can be introduced to call a Script at random. When this command is executed control is transferred to the called Script. Control is returned to the calling Script when the called Script exits. There is no limit on the number of Scripts that may be called by a Script.

During a single stepping session you can display the Call Script commands included in a Script which can help improve your monitoring options during Task Group replay. As replay proceeds Call Script commands are executed and the called Script's name is displayed in the Script selection box in the Single Stepping Test Pane. The Script items it contains such as HTTP requests, are displayed in the Monitoring tab.

Scripts that are called by another Script cannot be configured before running a single stepping session because they are not part of the Task Group being tested. But when Task Group replay is paused at a breakpoint, or if you are using the Single Step button to run a Task Group, it is possible to insert breakpoints and to select the requests whose HTTP responses you want to capture.

See also:

Call a Script

Single Stepping

SCL Reference Guide

Call a Script

1. Open a Script, then working in the Script Pane, move to the position where you want to call a Script from. This must be in the Code section of the Script.
2. Type **Call Script " "** inserting the Script name between the quotes, for example:

   Call Script "SHOP_1"

   Note: Refer to the *SCL Reference Guide*, available within Script Modeler's on-line help system for more information on the Call Script command.

   The Call Script command must also define which variables are to receive values passed as parameters from a calling Script.

3. Search for the Entry command within the Script and copy the information it includes.
   The Entry command is the first item in the Code section of the Script.

Tip: Right-click inside the Script Pane and select **Find**, or click  , then in the Find dialog box type in **Entry** and click **OK**.

4. Add this information to your Call Script command, for example:

   Call Script "SHOP_1" [USER_AGENT,USE_PAGE_TIMERS]

5. Compile the Script to check that it will replay correctly by clicking  , in the toolbar.

   Note: Compilation results are reported in the Output Pane.

6. Click  to save your changes.

   Note: Call Script commands are displayed in the Script Item list during a single stepping session.

See also:

Single Stepping

SCL Reference Guide

Syntax Check

When you perform any of these modeling procedures, it is important to verify the syntax by compiling the Scripts you record and model, to ensure the validity of the code and contents. If compilation fails an error message(s) appears in the Output Pane, in which case you may need to repeat the modeling procedure to resolve the problem. After compilation it is a good idea to replay the Script to check the activity you have recorded. When you are happy with your modeled Script make sure you save your work.

See also:

Syntax Check a Script

Syntax Check a Script

1. Open a Script, then select **Capture > Syntax Check** to compile the Script.

   Shortcut: Click  , in the Capture/Replay Toolbar.

   Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

2. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.
   Select **Capture > Replay**

Shortcut: Click  , in the Capture/Replay Toolbar.

The replay activity is displayed in the Output Pane.

3. Click  , to save the Script, or click **File > Save**.

Find and Replace Variables in Strings

If a variable string within a Script is excessively long it is truncated. The variable string is wrapped around on to the next line in the Script and the sections of the string are joined by an ampersand at the end of each line. If you need to replace a truncated variable you cannot enter the variable name to locate it using the Find and Replace function because the breaks in the text string will not always occur in the same place. The Search and Replace in Strings function is an intelligent search facility that can help you locate and substitute truncated variables.

See also:

Search and Replace a Variable in Strings

Search and Replace a Variable in Strings

1. Open a Script, then select **Variable > Replace In Strings**.

   Shortcut: Click  in the Variable Toolbar.

2. Enter the name of the variable in the Find text box and the new name in the Replace text box.
3. Click **Replace All** to substitute the new variable name.
4. Click  , to save the Script, or click **File > Save**.

Find Script Text

Use the Find function to locate any of the elements contained within the current Script.

1. Open a Script, select click **Edit > Find**

   Shortcut: Click inside the Script Pane and click  , or press **Ctrl + F**. Right-click within the Script Pane and click **Find**.

2. Enter the text to search for in the What text box, or click  and choose from a list of previous search items.

   Note: Click on a word or integer in the Script that you want to find before you begin the search process and the selected item appears in the What text box automatically.

3. Click **Find** to run the search.
4. Press **F3** to locate the next instance of the search item.

Find and Replace Script Text

Use the Find and Replace function to locate and substitute any of the elements contained within the current Script.

1. Open a Script, then select **Edit > Replace**.

   Shortcut: Click inside the Script Pane and click  , or press **Ctrl + H**. Right-click within the Script Pane and click **Replace**.

2. Enter the text to search for in the Find text box and enter the replacement text in the Replace with text box. Click  and choose from a list of previous search and replace items.

   Note: Click on a word or integer in the Script that you want to find and replace before you begin the search process and the selected item appears in the Find text box automatically.

3. Click **Find Next** to run the search, click **Replace** to substitute an instance, click **Replace All** for a global substitution.

Find in SCL Files

Use the Find In SCL Files function to locate and substitute any of the elements contained within the current Script.

1. Open a Script, then click  .
2. Enter the text you want to locate in the current and other Scripts.
3. Click **Find** to run the search, click **Cancel**.

Creating Scripts

Script Development

After you have planned your performance Test structure, you need to develop the Test contents by recording the Scripts that will be included in your Tests. Launch Script Modeler from Commander to record and model Scripts, then incorporate them into your Tests using Commander.

Scripts are HTTP/S recordings of the Web sessions you conduct using Script Modeler which represent the HTTP/S traffic they record as SCL code. The browser requests recorded are automatically encoded using SCL during the capture process. This gives the HTTP/S data an intelligible structure and makes it possible to model the Script. Script Modeler's editing capabilities enable you to record and edit Scripts to simulate the behavior of thousands of virtual users when a Test is run.

Scripts form the content of your Tests and enable you to generate the Web activity you want during a Test-run. They are stored in the Repository, which is displayed in the Commander Main Window. From here they can be selected for inclusion by reference in multiple Tests.

When you run a Test, the Scripts that it incorporates are run according to the

Task Group settings you have specified in Commander. These settings determine the load directed against the target WAE when a Test is run. Scripts encapsulate the Web activity you need to reproduce and the Task Group settings control the way the Scripts are run. Together, these elements control the type of the test environment that is simulated when a Test is run.

Before you begin to record your Scripts you should check your recording configuration. There are several options to choose from depending on the software setup you have on your computer and the network configuration you are working within. You can select the type of browser you want to use to record your Scripts, as well as the method of connecting to the target WAE.

See also:

The Script Development Process

The Script Development Process

The Script development process typically includes the following procedures:

Configuring Script Modeler for Script Recording

- Check Your LAN Proxy Server Settings
- Set Your Proxy Server Settings for a Dial Up Connection
- Select Browser Type for Script Recording
- Select the Gateway's Local Recording Mode
- Select the Gateway's Remote Recording Mode
- Select Automatic Cookie Modeling
- View Gateway HTTP/S Traffic During Script Recording

Script Recording

- Create a New Script
- Create Additional Scripts
- Save a Script
- Close a Script
- Rename a Script
- Delete a Script

See also:

The Gateway and Script Creation

Script Modeler enables you to create Scripts by rerouting the HTTP/S traffic that passes between a browser and WAEs, through the Gateway and recording the data it intercepts. The Gateway achieves this by temporarily overriding the Local Area Network (LAN), settings specified in your browser when you begin recording a Script. The browser's proxy server settings are temporarily modified in order to force it to use a proxy server when the browser is launched. This override directs the browser's internet connection through the Gateway which functions as the specified proxy server.

The Gateway listens for browser requests then forwards them to the target WAE, or to another proxy server if your browser normally has one defined, and then on to the WAE. When you start recording a Script, your Web browser is launched and the Gateway records the HTTP/S requests along with the WAEs responses to your browser.

The home page, address settings of your browser are also temporarily altered by the Gateway during Script recording. This removes all reference to the home page URL from the Script, since it is unlikely to form a part of the Script you create. Enter a URL in the Address text box to launch the WAE of your choice. When you have completed your recordings the original browser settings are restored.

See also:

Local Area Network Settings

Local Area Network Settings

The Local Area Network (LAN) settings for your browser are temporarily modified by Script Modeler, when you begin recording a Script and for the duration of the recording session. Script Modeler forces your browser to use a proxy sever in order to re-route HTTP/S traffic through the Gateway where it is recorded. The configuration of any existing proxy server settings you may have, are unaffected by this temporary alteration of your browser settings. So browser requests are correctly forwarded from the Gateway to the WAE. Your browser's original proxy settings are restored after the capture process is complete.

The Gateway has two recording modes, local and remote. In local mode the

internet connection settings are overridden as described above. Whereas in remote recording mode you need to modify the proxy settings of the remote browser manually in order to use the proxy Gateway.

See also:

Check Your LAN Proxy Server Settings

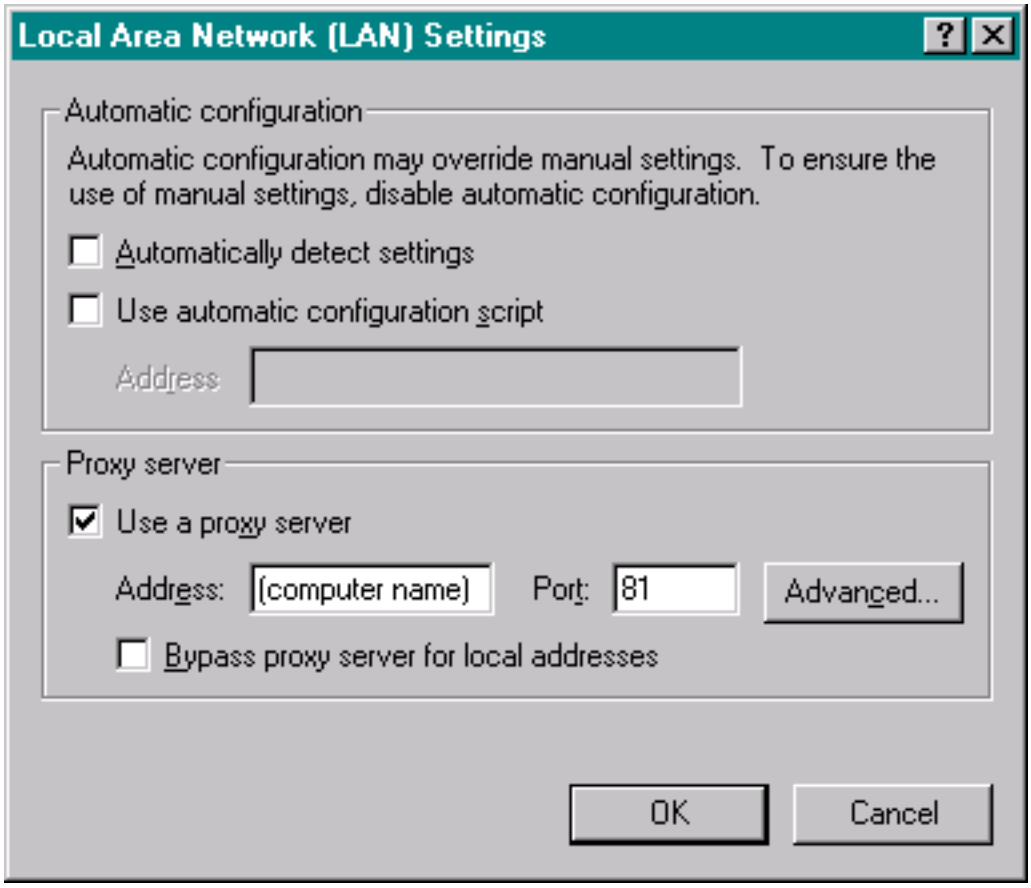
Using a Dial Up Connection

The Script Recording Process

Check Your LAN Proxy Server Settings

- Select **Tools > Internet Options > Connections > LAN Settings**.

  The screen shot below shows the proxy server settings for a Local Area Network connection in Internet Explorer 5:



  Note: Script Modeler cannot establish an internet connection if the Proxy server settings include **Automatically detect settings** or **Use automatic configuration script**.

  To overcome this difficulty you can either disable the use of all proxy severs or manually enter the name and port of the proxy sever. You may need to see your system administrator for this information.

Note: The Address and Port details will vary according to your network configuration and the Internet Service Provider you use.

Using a Dial Up Connection

A dial up internet connection setup involves establishing a connection within Script Modeler when you begin an HTTP/S recording and the browser is launched. As well as entering your dial up settings, including user name and password, you must manually change the proxy server settings. Script Modeler only automates the configuration of the proxy server settings for a LAN connection. You need to select to use a proxy server to direct your internet connection through the Gateway.

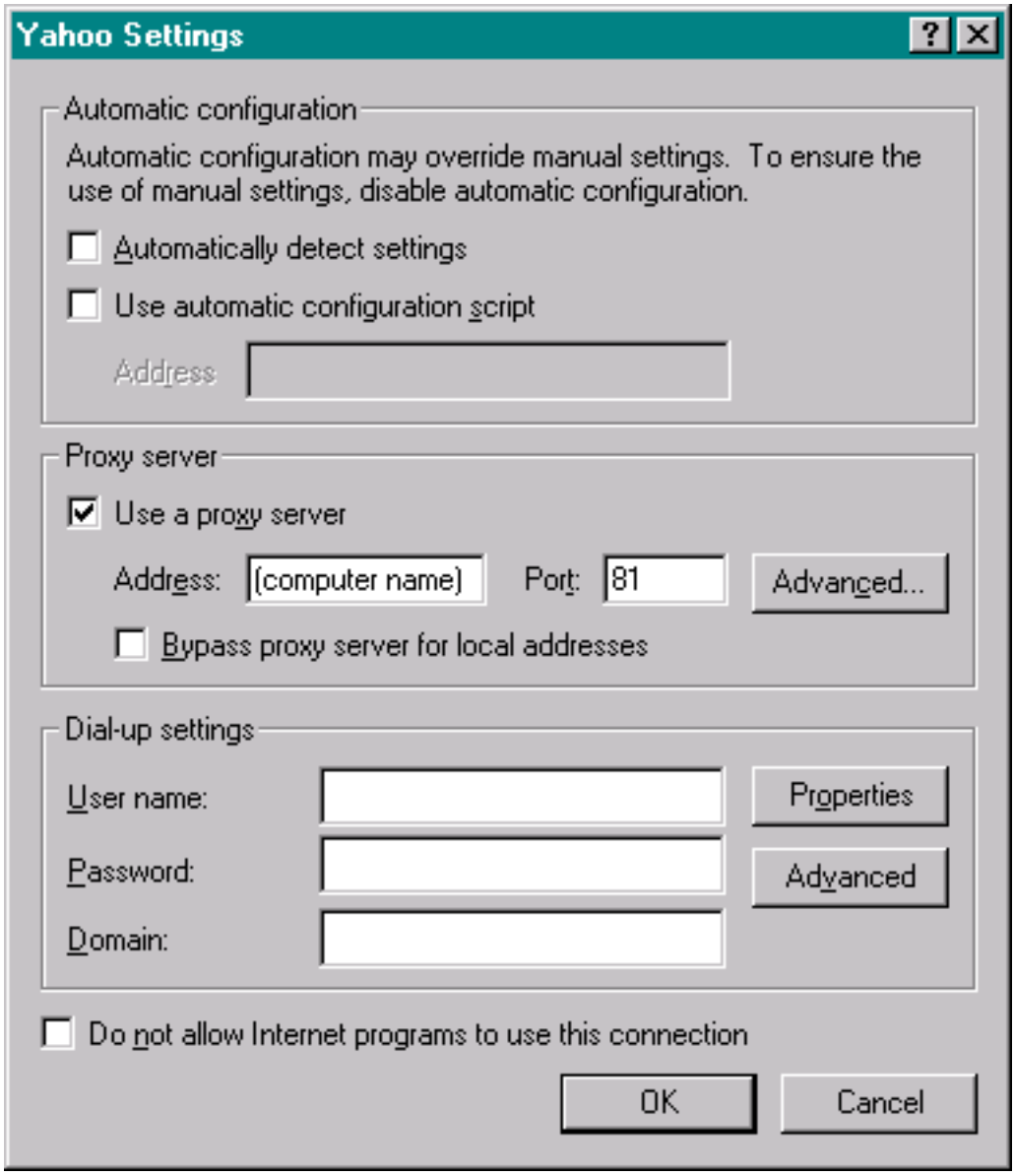
See also:

Set Your Proxy Server Settings for a Dial Up Connection

Set Your Proxy Server Settings for a Dial Up Connection

1. Select **Tools > Internet Options > Connections > Settings**.

   The screen shot below shows the proxy settings for a dial up connection:



2. In the Settings dialog box, click the **Use a proxy server** check box, enter the **Address** (your computer name) and **Port** details (port no. 81).
3. Enter your details as usual, then click **OK** to connect.

The Script Recording Process

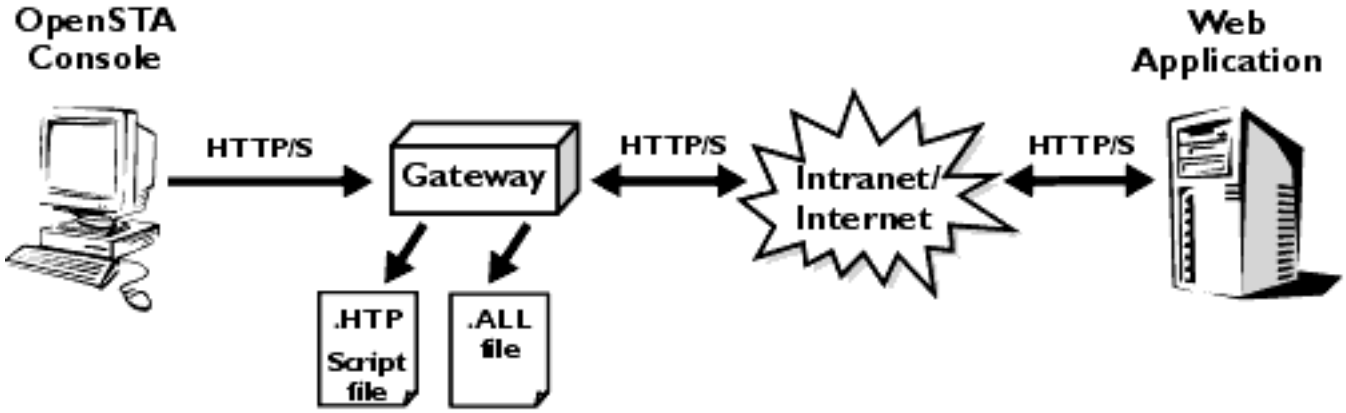
Script Modeler creates a Script exactly as the browser requested the Web pages and their contents. They are created by the Gateway and consist of SCL code, including GET, POST and HEAD commands, which represent corresponding HTTP/S instructions.

Scripts represent HTTP/S browser requests in SCL code and are saved in a .HTP file. During the same recording session the corresponding WAE responses are recorded by the Gateway in a .ALL file. This includes DOM, HTML and Web page structure data. The full detail of a Web session is stored in these two files.

After you have clicked the Record button  in the Capture/Replay Toolbar and entered the first URL in your browser's Address text box, the WAE responds by sending the HTTP/S data that forms the content of the Web page displayed by your browser.

Loading a Web page involves parsing or compiling the Web page structure from the raw HTTP/S data returned by the WAE in response to the URL or PRIMARY GET. The content is then rendered on screen by the browser whilst concurrently making additional, asynchronous requests on other TCP connections via secondary GETs for the remaining contents of the Web page. The browser continues to issue requests and render any remaining content until the Web page is fully loaded. The Gateway records and formats this information.

The Script Modeler HTTP/S capture process is illustrated below:



Browser requests hit the target WAE via the Gateway, across the Internet or other network. Browser requests are recorded by the Gateway as a Script (.HTP file). WAE responses are recorded by the Gateway in a .ALL file.

Note: OpenSTA Console refers to a computer which has an installation of OpenSTA. This includes the OpenSTA Architecture and Commander and it may also include the Repository, where all Test related files and results are stored. The PC will also have a Web browser installed and is typically the home for the Gateway. In this diagram the Gateway is shown as separated from the OpenSTA Console to clarify the Script recording process.

See also:

Script Modeler Configuration Options

Script Modeler Configuration Options

Before you capture a new Script there are some configuration options available within Script Modeler that you may want to check or change. Some of these options relate to the type of Web browser(s) you have installed on your computer, the settings you have specified for them and the Gateway recording

mode you want to use, either Local or Remote.

Other configuration options available relate to Gateway settings that can assist you during the recording process and afterwards if you need to model your Script.

You can opt to display a command line console which displays the activity of the Gateway during recordings and to trace this activity.

Another Gateway setting allows the automatic generation of variables to replace any cookies received during a Web session. Automating the substitution of a variable for the unique identity of the original cookie transmitted helps to simplify modeling and speed up the development of your Tests.

Use the **Options** menu choices to configure your settings.

See also:

Browser Settings

Configuring The Gateway: Local and Remote Recording

Gateway Settings

Creating New Scripts

Browser Settings

If you have more than one type of browser installed on your computer, you can choose which one you want to launch in order to record your Scripts.

OpenSTA currently supports Internet Explorer 4 and 5, and Netscape Navigator 4.7, for use in HTTP/S captures. If you have several browsers installed on your computer you may want to specify which browser is launched when you begin recording a Script. If you have more than one of the browser types mentioned installed and you do not select one, Script Modeler defaults to the latest version of Internet Explorer installed on your computer.

See also:

Select Browser Type for Script Recording

Select Browser Type for Script Recording

1. In Script Modeler, select **Options > Browser**.
2. In the Select Browser dialog box, click  , and select the browser you want from the list, either **Internet Explorer 4**, **Internet Explorer 5** or **Netscape**.

Note: The **Netscape** option refers to Netscape Navigator version 4.7

3. Click **OK** to save your settings.

Configuring The Gateway: Local and Remote Recording

You can choose to record Scripts on your own computer or through another remote computer by selecting the appropriate Gateway recording mode, either Local or Remote.

In local recording mode the browser you have selected to use is launched when you begin the HTTP/S capture process you are recording from. The default mode for capturing HTTP/S traffic through the Gateway is Local. In this mode the Gateway listens on port 81 for HTTP/S traffic by default.

If there is no browser installed, or if you want to use another computer to record your Scripts, you can choose the Remote recording mode. You can use any networked computer with a browser installed to conduct remote Script recording. If you use Remote recording mode you must manually modify the proxy server settings of the browser on the remote computer. Make a note of these settings because you need to copy them exactly into the Gateway settings on the computer from where you are recording.

See also:

Select the Gateway's Local Recording Mode

Select the Gateway's Remote Recording Mode

Select the Gateway's Local Recording Mode

Note: Local Recording Mode is the default setting.

1. In Script Modeler, select **Options > Gateway**.

   Note: If you selected to use Netscape Navigator 4.7 to conduct your recordings, then a browser Information dialog appears.

   Click  and locate the Netscape preferences file **prefs.js**. After you have selected the file, click **OK** to move on to the Gateway dialog box.

2. In the Gateway dialog box Capture section, click **Local**.
3. Type in an **Administration Port** and **Port** number if the defaults displayed here are in use. Otherwise accept the defaults displayed.

   Note: The Administration Port is used for internal communication between Script Modeler and the Gateway.

4. Click **OK** to save your settings.

Note: These settings apply to all subsequent Script recordings until you change them.

Select the Gateway's Remote Recording Mode

1. In Script Modeler, select **Options > Gateway**.

   Note: If you selected to use Netscape Navigator 4.7 to conduct your recordings, then a browser Information dialog appears.

   Click  and locate the Netscape preferences file **prefs.js**. After you have selected the file, click **OK** to move on to the Gateway dialog box.

2. In the Gateway dialog box Capture section, click **Remote**.
3. Type in an **Administration Port** and **Port** number if the defaults displayed here are in use. Otherwise accept the defaults displayed.

   Note: The Administration Port is used for internal communication between Script Modeler and the Gateway.

4. In the Proxy section of the Gateway dialog box, the selections you make must reflect the settings specified in the Proxy settings dialog of the remote computer you are using, these settings include:

   - **Proxy Address** and **Port**: Enter the address and port number of the proxy server you want to use to connect to the Internet.
   - **Secure** and **Port**: Enter the address and port number of the secure proxy server you want to use to connect to the Internet.
   - **Bypass proxy server for local addresses**: Check the box if you do not want to use the proxy server for all local addresses, including intranet addresses. Note: You might be able to gain access to local addresses easier and faster if you do not use the proxy server.
   - **Do not use proxy server addresses beginning with**: Enter the Web addresses that do not need to be, or should not be, accessed through the Proxy server.

5. Click **OK** to save your settings.

   Note: These settings apply to all subsequent recordings until you change them.

Gateway Settings

The Settings section of the Gateway dialog box offers you options which you can select to enhance the quality of the Scripts you record and improve the visibility of the recording process.

Select the Console option to display a command line window during the recording process. The console displays Gateway activity during the Web session.

Select the Automatic Cookie Generation option to automate the processing of any cookies you record in your Scripts. If the WAE you are testing generates cookies, then enabling this function can help to speed up the Test creation process. This option automatically substitutes the unique identities of any cookies received from a WAE with a variable. The automatic modeling of cookies is an optional feature but active by default.

Automatic Cookie Generation

Any cookies issued by a WAE under test are recorded in the Scripts you create. If cookies include unique session identity information such as a time stamp, the Scripts that contain them will be rejected by the target WAE when replayed during a Test-run unless they are modeled.

Modeling cookies is an essential procedure that enables Scripts to be used in Tests. It involves replacing the unique session identity encapsulated by a cookie from the original Web session and it is achieved by assigning the fixed values recorded to a variable. Script Modeler gives you the option to automate this task or to manually model Scripts depending on your requirements. The default setting in Script Modeler is for automatic cookie generation.

Selecting to automatically model cookies can help improve the efficiency of the Test development process. It also enables you to develop a modular Test structure by incorporating a sequence of Scripts in a Task Group to function as one browser session during a Test-run, without having to model the cookie information. For more information on developing a Task Group that combines multiple Scripts, see Developing a Modular Test Structure.

See also:

Select Automatic Cookie Modeling

View Gateway HTTP/S Traffic During Script Recording

Developing a Modular Test Structure

Select Automatic Cookie Modeling

1. In Script Modeler, select **Options > Gateway**.
2. Click the **Automatic Cookie Generation** check box.

View Gateway HTTP/S Traffic During Script Recording

1. In Script Modeler, select **Options > Gateway**.

2. Click the **Console** check box to view a Command Line window during the Script capture process.

   This console displays the Gateway initiating HTTP/S connections and receiving WAE responses, concurrent to the actions you perform using the browser.

Creating New Scripts

After you have configured the Gateway and chosen the Web browser you want to use for Script recording you are ready to begin the HTTP/S capture process.

Recording Scripts is straightforward.

In the Commander Menu Bar select **File > New Script > HTTP**, or right-click **Scripts**  in the Repository Window and select **New Script** from the menu. Give the Script a name, press return and double-click the new Script icon  in the Repository Window to launch Script Modeler.

Use the Capture/Replay Toolbar or the **Capture** menu option in Script Modeler to control the recording process.

After you have recorded a Script you can either save it or begin the next recording by selecting **File > New**. You can have several Scripts open at the same time if required so you can either save your Scripts as you record them or before you exit. Press **Ctrl > Tab** to switch between the Scripts you have open.

Two files are generated during a recording session and stored in the Repository after you save, a .HTP file, or Script, which records the Web browser requests, and a .ALL file which records the WAE responses and contains HTML data.

Script names must be defined according to the rules for OpenSTA Datanames, with the exception that the name can be up to 60 characters long.

After you have saved the Script it appears in Repository Window  . The small crossed red circle is removed indicating that the Script is ready for use. Drag and drop the Scripts you need from the Repository Window into your Tests.

Note: If you want to create a Test that simulates a first time user and you have previously accessed the target WAE, make sure you clear the browser's memory cache before recording your Scripts. In Internet Explorer 5, temporary Internet files are removed by selecting **Tools > Internet Options** then clicking the **Delete Files** button.

See also:

Capture/Replay Toolbar

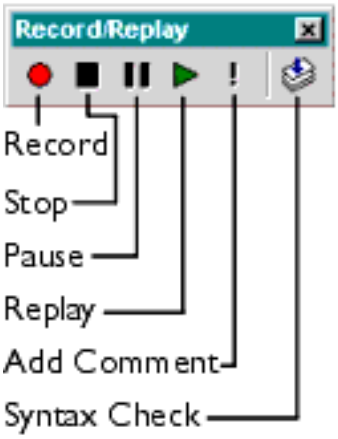
Create a New Script

Modeling Scripts

Developing a Modular Test Structure

Capture/Replay Toolbar

The Capture/Replay Toolbar is used to control the Script recording process. It is located below the Menu Bar in the Script Modeler window. You can use it to record multiple Scripts during the same Web browser session



.

Click the Record button , in the Capture/Replay Toolbar to begin the HTTP/S capture process. This action launches your Web browser which in turn activates the Gateway, or Proxy Server. The HTTP/S traffic generated during the recording session is intercepted by the Gateway and the HTTP/S requests are encoded to produce a Script written in SCL. This data is displayed in the Script Pane after the recording process is complete. The HTML pages accessed during the Web session are recorded in a separate file and can be displayed in the Query Results Pane.

Use the Add Comment button , to add comments during a recording session. They are used to assist you when monitoring a single stepping session.

Use the Pause button to suspend Script recording. Click to resume recording.

When you have recorded everything you need, end your recording session by clicking the Stop button , or close down the browser.

Use the Syntax Check button to compile the current Script. The SCL compiler is launched and generates an object file called a .TOF file. This is the file that is executed by a Task Group Executer when a Test is run.

Use the Replay button  to compile the current Script and replay it within Script Modeler to ensure that it is valid. Replay activity can be monitored using the Output Pane.

See also:

Create a New Script

Single Stepping Comments

Create a New Script

1. In Commander select **File > New Script > HTTP**.

   Or: In the Repository Window, right-click  **Scripts**, and select **New Script > HTTP**.

   The Script appears in the Repository Window with a small crossed red circle over the Script icon  , indicating that the file has no content. As soon as you open the Script and record a Web session, the icon changes to reflect this and appears  .

2. Give the new Script a name within the Repository Window, which must be an OpenSTA Dataname, with the exception that the name can be up to 60 characters long, then press **Return**.

3. Double-click the new Script icon  , to launch Script Modeler.

4. Click the Record button  , in the Capture/Replay Toolbar, or select **Capture > Record**, to begin the HTTP/S capture process.

   This action launches the Gateway and the Web browser you have selected.

   Your browser's Home page Internet option is overridden by Script Modeler when you start recording. The setting is replaced with **about: blank**, which specifies that your home page will be a blank HTML page. This ensures that your normal Home page is not launched and recorded in the Script.

   Note: The Gateway is launched in Local recording mode by default unless you have chosen Remote recording mode.

5. Type in a URL and hit **Return** or select a URL from the browser's URL Address bar. Then use the browser as normal to perform the actions you want to record in your Script.

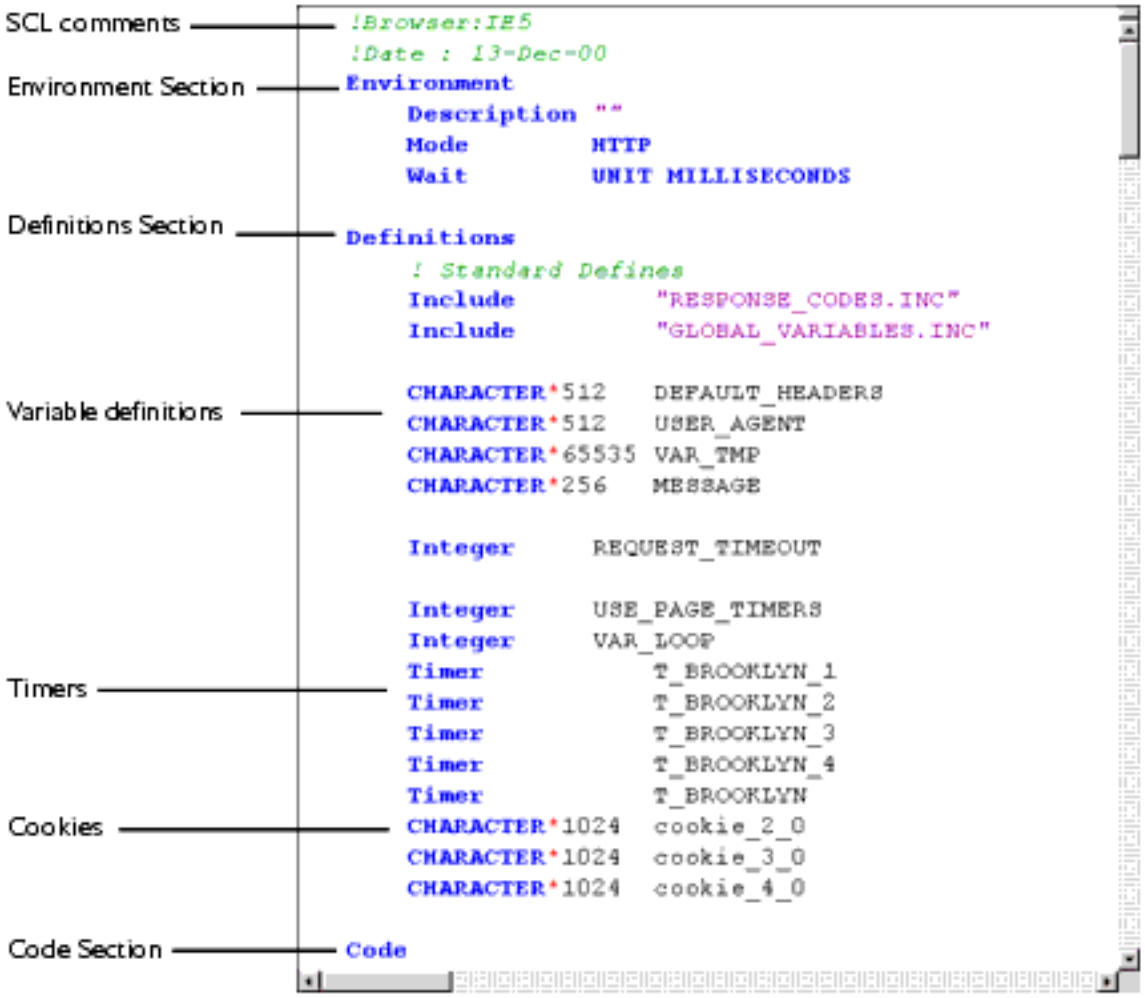
   Tip: Use the Add Comment button  in the Capture/Replay Toolbar to add comments while you are recording a Web session, or select **Capture**

> Insert Comments. They are used to assist you when monitoring a single stepping session. The time taken to add a comment is not recorded in the Script.

6. After you have completed the browser actions you need, switch back to Script Modeler from the browser and click the Stop button , in the Capture/Replay Toolbar to end the recording. Or, close the browser.

   Tip: If you have more than one Script to record use to end a recording, to save repeatedly closing and opening the browser.

   When you have finished recording the Script the SCL formatted data is displayed in the Script Pane as illustrated below:



7. Before you save your new Script you need to compile it using the Syntax Check option to ensure the validity of the recording.

   Select **Capture > Syntax Check** or click , in the Capture/Replay Toolbar. Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-record the Script or model the contents to resolve the problem.

Note: You can record over the top of an existing Script until you achieve the content you need.

8. After compilation replay the Script to check the activity you have recorded.

   Select **Capture > Replay** or click  , in the Capture/Replay Toolbar

9. When you have finished recording, click  , in the Standard Toolbar to save your Script in the Repository, or click **File > Save**.
10. Select **File > Close** to close the current Script or **File > Exit** to exit Script Modeler.

    Note: If you have unsaved Scripts open in Script Modeler, you are automatically prompted to save them before the program closes. Closing down Script Modeler also closes the browser which restores your original browser settings.

See also:

Create Additional Scripts

Modeling Scripts

Developing a Modular Test Structure

Single Stepping Comments

Create Additional Scripts

Once you have launched Script Modeler you can record additional Scripts.

1. In Script Modeler, select **File > New**.

   Shortcut: Click  , in the Standard Toolbar, or press **Ctrl + N**.

2. Click the Record button  , in the Capture/Replay Toolbar, or select **Capture > Record**, to begin the HTTP/S capture process.
3. In the Script Name dialog box, give the new Script a name, which must be an OpenSTA Dataname, with the exception that the name can be up to 60 characters long.

   Click **OK** to launch the Gateway and the Web browser you have selected and begin recording. Create your Script as usual, see Create a New Script for details.

Save a Script

- Click  , to save your changes.

  Note: The Script is saved in the Repository.

Close a Script

- Select **File > Close**.

  Note: If you make any changes to a Script and attempt to close the file before saving, a dialog box appears prompting you to do so. Click **Yes** to save the changes.

Rename a Script

1. In Commander, double-click  **Scripts**, in the Repository Window to expand the directory.
2. Right-click the Script and select **Rename**, or double-click slowly on the Script.
3. Enter the new name and press **Return**, to save your changes in the Repository.

   Note: When you rename a Script the Tests that reference it notify you that it is missing by highlighting the Task table cell it occupied in red when the Test is opened. The Test cannot run with a missing Task. Rename or recreate a Script to match the name of the missing Script to resolve this problem or delete the Task from the Test.

Delete a Script

1. In Commander, double-click  **Scripts**, in the Repository Window to expand the directory.
2. Right-click on the Script and select **Delete** from the menu.

   Or click on the Script you want to remove and press **Delete**.
3. Click **Yes** to confirm the deletion.

   Note: When you delete a Script, the Tests that reference it notify you that it is missing by highlighting the Task table cell it occupied in red when the Test is opened. The Test cannot run with a missing Task. Recreate a Script with the same name as the missing Script to resolve this problem or delete the Task from the Test.

   Note: Your changes are automatically saved in the Repository.

HTTP/S Scripts

What are Scripts?

Scripts form the content of an HTTP/S performance Test using HTTP/S Load.

After you have planned a Test the next step is to develop its content by creating the Scripts you need. Launch Script Modeler from Commander to create and model Scripts, then incorporate them into your performance Tests.

Script Modeler records HTTP/S traffic during a Web session and creates a Script exactly as the browser issued its requests. When a Test that includes the Script is run, the Script is replayed exactly as the browser made the original requests. This means that the target Web Application Environment (WAE) receives the same number of concurrent, asynchronous connections and requests from each simulated browser user, or Virtual User, just as it would receive from real end users.

Scripts represent the recorded HTTP/S requests issued by a browser to a target

WAE during a Web session. They are created by passing HTTP/S traffic through a proxy server, the Gateway, and replacing the original HTTP/S commands with SCL commands. The Gateway interfaces directly with the Script Modeler and uses Script Control Language (SCL), to represent HTTP/S recordings.

Using SCL to write Scripts gives you control over their content. It enables you to model them by introducing variables to modify the fixed values they record. In turn, this gives you control of the performance Tests that incorporate them and enables you to effectively test the Web activity you want with the load levels you require.

When you record a Web session a .HTP file and a .ALL file are produced. The .HTP file contains the HTTP/S browser requests issued during the Web session written in SCL. This file is the Script which is designed to be modeled and replayed as part of a Web performance Test. The .ALL file is directly related to the .HTP file and stores the WAE responses in several categories, including the DOM, which can be utilized to model the accompanying Script.

All Scripts are stored in the Repository from where they can be included by reference into multiple Tests.

See also:

Understanding Scripts

Creating Scripts

Modeling Scripts

Understanding Scripts

Successfully recording, modeling and incorporating Scripts into Tests requires an understanding of the components and concepts which are related to them in HTTP/S Load.

See also:

Tests

The Gateway

Scripts and SCL

HTTP/S Scripts and Test-runs

Virtual Users

DOM Addressing

Cookies and Automatic Cookie Modeling

The Repository

Planning Your Scripts

Tests

A Test is a set of user controlled definitions that specify which Scripts and Collectors are included and the parameters that apply when you run the Test. They also include the results that are recorded which can be monitored while they are being generated during a Test-run and displayed in graph and table format after the Test-run is complete.

Scripts and Collectors are the building blocks of a Test which can be incorporated by reference into many different Tests. Scripts determine the contents of a Test and Collectors define the data collection to be carried out during a Test-run. The Scripts and Collectors you add to a Test are organized in Task Groups. Select the Settings you want to apply to each Test Task Group to control how the Test is run and the level of load that is generated against the target WAE. Task Group settings include the number of Virtual Users, the Host computers used and the number of times a Script is replayed during a Test-run.

Develop your performance Tests by planning their structure and content, then create the Scripts and Collectors you need in order to simulate the type of activity you want to test. Scripts record HTTP/S activity during a Web session and Collectors control the type of data that is collected when a Test is run. Use Commander to manage and run your Tests along with the Scripts and Collectors they contain.

See also:

Understanding Scripts

Creating and Editing Tests

The Gateway

The Gateway is a component of OpenSTA which interfaces with Script Modeler to enable you to record HTTP/S traffic and create Scripts. It functions as a proxy server, residing between the client browser and the remote Web server hosting the WAE under test. When you begin a recording using Script Modeler the Gateway overrides some of your browser's internet connection settings, forcing the use of a proxy server, in this case the Gateway. The Gateway can then record the Web activity between browser and WAE and produce a Script, which is represented using SCL scripting language. The Script can then be modeled using the Script Modeler.

The Gateway records browser requests in a .HTP file, or Script, and WAE responses are stored in a .ALL file. The .ALL file contains HTML data which is directly related to the .HTP file content. It can be used to model the Script by manipulating information it contains including the DOM, the Document Object Model.

See also:

Understanding Scripts

The Gateway and Script Creation

Scripts and SCL

HTTP/S Load uses a scripting language called Script Control Language (SCL), developed by CYRANO, to represent Scripts. SCL is used to control and represent the HTTP/S traffic they record.

Using SCL to write Scripts gives you the modeling capabilities you need to develop realistic performance Tests. You can model a Script or a sequence of Scripts, to simulate thousands of Virtual Users in order to generate the load you require against one or more target WAEs when you run a Test.

See also:

Understanding Scripts

The Gateway and Script Creation

SCL Representation of Scripts

HTTP/S Scripts and Test-runs

Using HTTP/S Scripts to develop performance Tests has several advantages over client level or browser based replay techniques. HTTP/S traffic is the key information that is generated during a Web session. Capturing Web activity at this level enables you to record the activity of a variety of browser types across various platforms. Scripts can be modeled then referenced in Tests that simulate realistic conditions when they are run. After capturing and modeling Scripts you can replay them as part of a Test to exactly replicate the original browser commands. The HTTP/S requests are concurrently and asynchronously run on as many TCP connections as the original Web session, multiplied by the number of Virtual Users you select to run the Test.

Developing and executing HTTP/S based Tests is much less resource intensive than other techniques involving full browser emulation, which enables you to develop Tests that run a larger number of Virtual Users.

See also:

Understanding Scripts

Creating Scripts

Creating and Editing Tests

Running Tests

Virtual Users

Virtual Users are a key feature in OpenSTA.

A Virtual User is the simulation of a real life browser user that performs the Web activity you want during a Test-run. The activity of Virtual Users is controlled by recording and modeling the Scripts that represent the Web activity you want to Test. They are generated when a Script-based Task Group is executed during a Test-run and are used to produce the load levels you need against target WAEs.

The identity and activity of Virtual Users running a Test can be modified by modeling the Scripts they run to include variables that replace fixed values in the Scripts. For example, if a Script includes logon details they can be modeled to replace the original browser user's identity with a variable. You then have the ability to replay the Script as part of a Test which includes multiple Virtual Users each with their own unique identity.

Open a Test and select a Script-based Task Group to configure your Task Group, which includes your Virtual User settings. Specify the number of Virtual Users you need in order to generate the level of load required against the target WAEs when the Test is run.

See also:

Understanding Scripts

Virtual User Settings

DOM Addressing

The Document Object Model (DOM), is an application programming interface for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. When a Script is being recorded the Web pages returned are saved in a .ALL file. During the replay of a Script as part of a Test the contents of a particular DOM signature or DOM object address, can be recovered dynamically. This enables modeling of the dynamic

nature of some Web pages.

The term DOM addressing describes the ability to access and model specific and unique elements within the DOM or Web page, using Script Modeler. Address DOM elements to modify Scripts to improve the representation of the unique behavior of users during Test-run, such as the selection of different links within a Web page.

See also:

Understanding Scripts

DOM Addressing

Cookies and Automatic Cookie Modeling

A cookie is a packet of information sent by a Web server to your browser when you connect to it, that is saved on your computer. Typically they contain user identification details such as name, password and other preferences. The next time you connect to the same WAE, the cookie is automatically retrieved from your computer. This allows the WAE to recall information you have already given, freeing you from the task of re-entering it. The WAE can then process the cookie information and customize the Web page it sends back to you.

Cookies form part of the HTTP/S traffic recorded in a Script if the WAE you are targeting issues them. When you use Script Modeler to record a Script, cookies are automatically modeled, which means that the cookie identity is copied into a variable that replaces its unique identity with a new variable definition. This is an essential requirement for the replay of a Script as part of a Test. A Script containing unmodeled cookies records a unique session identity that would be rejected by the WAE if it was replayed during a Test-run to represent one or more Virtual Users.

See also:

Understanding Scripts

Automated Script Formatting Features

The Repository

The Repository is the storage area on your hard drive or networked computer. All the files that define a Test, including Scripts and Collectors, and the result files produced during Test-runs are stored here.

The contents and structure of the Repository are immediately visible through the Repository Window in Commander. It is located on the left-hand side of the Main Window and displays all the Scripts, Collectors and Tests that are stored

there. You can work from the Repository Window to initiate the creation of new Scripts and to open existing Scripts.

See also:

Understanding Scripts

Planning Your Scripts

Make sure you plan your Scripts before you begin recording them. Planning the creation of Scripts involves identifying which WAE functions you want to test, deciding who your clients are and how you expect them to use the Web services, and establishing the type of Test structure you want to develop.

Recording a sequence of Web activity in a Script is a straightforward process once you have planned the functionality you want to test. Making the behavior of the Virtual Users you generate during a Test-run more realistic can be achieved by modeling the Scripts.

The type of users you want to represent may be a consideration during the planning stage, depending on the nature of the WAE functions you are testing. Representing first time users or repeat users during a Test-run should influence the steps you take before recording the Scripts. If you want to simulate first time users it is important your Scripts are accurate, so make sure that you have cleared the browser memory cache before recording. This ensures that the WAE is stressed with a realistic load during the Test-run. Alternatively, if you are simulating repeat users, ensure the Script recording conditions support this requirement. Visit the WAE and conduct the activity you want to test before you record the Script so that the Web pages and their content are held in the memory cache and the Scripts reflect these circumstances.

Commander supports a versatile Test structure. Tests can include a single Script but a modular Test structure can be more efficient. If you create smaller Scripts that record specific client browser actions, such as logging on and entering a password, it is possible to reuse them in other Tests that involve accessing the same WAE. You can incorporate the same Scripts in any Test stored in the same Repository.

Also, it is easier to maintain a Test that represents browser activity with a structure that incorporates several smaller Scripts in a modular Test structure. A Test incorporating a single or a few large Scripts, takes much longer to update when there are changes to the WAEs functionality, than a modular Test structure comprising several smaller Scripts

So capturing the right activity in your Scripts is important in the development of a successful performance Test.

See also:

The Core Functions of Script Modeler

The Core Functions of Script Modeler

1. Recording Scripts.
2. Modeling Scripts.

See also:

Launch Script Modeler

Script Modeler Interface

Launch Script Modeler

You can move directly into Script Modeler from Commander.

1. In the Repository Window within Commander, double-click **Scripts**, to expand the directory structure.
2. Double-click on a Script icon or , within the tree structure.

See also:

Creating Scripts

Modeling Scripts

Script Modeler Interface

Script Modeler supplies versatile Script creation and modeling functionality. Use the menu bar and right-click menu options to create and model Scripts.

After you create a Script or when you open one, the .HTP file is displayed in the Script Pane on the left-hand side of the main window. It is represented using SCL code which enables you to model it using the menu options or directly by keying in the SCL commands you need.

The Query Results Pane is used to display WAE responses. HTML information is recorded during the same Web session as the corresponding Script and is directly related to it, which enables additional modeling options.

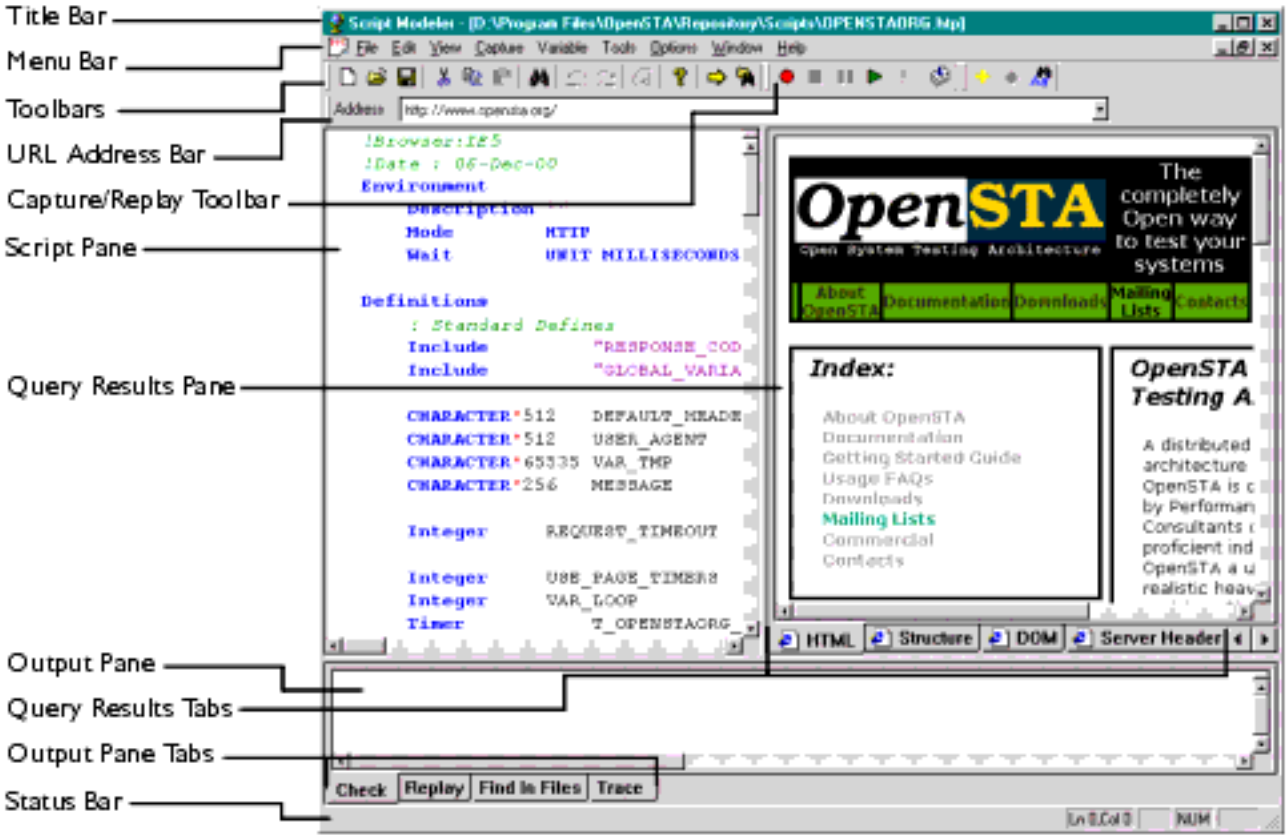
The Script Modeler interface consists of four primary areas:

- Toolbars and Function Bars.
- Script Pane.

- Query Results Pane.
- Output Pane.

Script Modeler Interface Features

The main features of the Script Modeler interface are detailed below:



See also:

Toolbars and Function Bars

Toolbars and Function Bars

Script Modeler Toolbars include:

- **Standard Toolbar**: Use the functions provided here to open, create new and save Scripts, edit text, print, view full screen, search for text and display URL Details.
- **Capture/Replay Toolbar**: Use this toolbar to record, end a recording, replay and compile your Scripts.
- **Variable Toolbar**: Use these options to add and edit variable definitions.
- **URL Address Bar**: Displays the URL GET, POST and HEAD commands,

contained in the current Script.

Click  , to the right of the URL Address Bar text box to display and select from the URL commands displayed.

Script Modeler Function Bars include:

- **Title Bar**: Displays the name of the current Script.

  It also contains the Script Modeler Control Icon  , which gives you access to the Control menu, and incorporates Windows buttons, Minimize, Restore/Maximize and Close.

- **Menu Bar**: Displays Script Modeler menu options. Click on a menu option or use the keyboard shortcuts to access and select the functions you need.
- **Status Bar**: Displays information about the current Script including the cursor position within the .HTP file displayed in the Script Pane.

See also:

Toolbar Display Options

Toolbar Display Options

Script Modeler includes three toolbars which are located below the Menu Bar in the main screen. A fourth toolbar, the URL Address Bar appears when you create a new Script or open an existing one.

You can change the position of the toolbars by clicking on the double bar on the left-hand side of the toolbar and dragging them to a new location. Either within the toolbar area or floated in the Main Window. You can also choose to display or hide the Standard Toolbar by using the **View** menu option.

See also:

Hide/Display the Standard Toolbar

Hide/Display the Standard Toolbar

- Click **View > Toolbar**.

  A tick to the left of the **Toolbar** prompt indicates it is currently displayed.

Script Pane

When you record Web activity using Script Modeler a .HTP file and a .ALL file are produced. The .HTP file, or Script, is displayed in the Script Pane and represents the browser requests recorded during the Web session. It contains the HTTP/S data written in SCL that is designed to be modeled and incorporated into a Web performance Test.

Use the Script Pane to view and model the HTTP/S traffic recorded in your Scripts. The recorded HTTP/S traffic is represented using SCL which gives the recorded data structure and enables you to model the Script if required.

The HTTP/S data displayed in the Script Pane constitutes the Script which determines the behavior of your Virtual Users when replayed as part of a Test. The Script Pane is a window directly into the Web activity you simulate when you run a Test.

A Script records HTTP/S requests issued by a browser during a Web session. It is dynamically linked to the HTML information represented in the Query Results Pane which records the WAE responses, the Web pages that are returned, in a .ALL file.

See also:

Resize the Script Pane

Resize the Script Pane

- You can adjust the size of the Script Pane by clicking on any border it shares with another pane and dragging it to a new position.

Query Results Pane

The Query Results Pane displays HTML and other data relating to the current Script that is stored in a .ALL file. This file is created at the same time and is directly related to, the corresponding Script, which is saved as a .HTP file during the original Web session recording. Some of the HTML information it contains, including Structure, Document Object Model (DOM) and Server Header are dynamically linked to the Script, which enables additional modeling capabilities.

The Query Results Pane remains empty until you populate it by selecting a URL GET, POST and HEAD command from the current Script displayed in the Script Pane, and click  , the URL Details button. The HTML data stored in the .ALL file directly corresponds to the .HTP file. The .ALL file is never updated and only overwritten if the Script is rerecorded.

The data is organized into five categories represented by tabs at the bottom of the pane. Click on a tab to view the data corresponding to a selected URL command. The five categories are:

- The **HTML** tab presents a browser view. It displays the Web page that corresponds to the URL command you selected in the Script Pane.
- The **Structure** tab displays the structure the selected Web page, including links, frames, images and other components which make up the page.
- The **DOM** tab displays Document Object Model, information URL composition and structure of a Web page. It presents a more detailed structural display of the diverse elements that comprise a Web page.
- The **Server Header** tab displays the HTTP/S response header fields sent from a WAE to the browser. Including the date the HTTP/S information was recorded, the kind of connection used and which Web pages were contacted.
- The **Client Header** tab displays the HTTP/S request header fields sent by the browser to a WAE.

See also:

Display Query Results Pane Information

Resize the Query Results Pane

Display Query Results Pane Information

1. In the Script Pane, use the scroll bars or the Find function to locate a URL command, or:

   Click  , to the right of the URL Address Bar text box to display a list of all the URL commands contained in the current Script and select one from the list.

2. Click an insertion point inside the URL command.
3. Click  in the Standard toolbar, or **View > URL Details**, to open the. ALL file.

   The HTML tab is the default view and displays the Web page corresponding to the selected URL command.

Resize the Query Results Pane

- Adjust the size of the Query Results Pane by clicking on any border it shares with another pane and dragging it to the new position.

Output Pane

The Output Pane is used to display results of Script compilation and to report the progress during the replay and compilation of a Script. This includes any errors and other status messages.

Output Pane information is organized into four categories which are represented by the Query Pane Tabs at the bottom of the pane.

Click on the tabs to view the information they contain. The categories are:

- **Check**: This tab displays compilation progress and results.
- **Replay**: Click  , in the Capture/Replay Toolbar to replay a Script. The replay progress is displayed here. Replay data is retained until you close the Script file. During this time you can view the replay data at any time by clicking on the **Replay** tab.
- **Find In Files**: Displays search results generated after using the Find In FIles button  , in the Standard Toolbar.

See also:

Resize the Output Pane

Resize the Output Pane

- You can adjust the size of the Output Pane by clicking on any border it shares with another pane and dragging it to a new position.

HTTP/S Load

Overview of HTTP/S Load

HTTP/S Load supplies flexible software that enables you to quickly develop and run HTTP/S load Tests and production monitoring Tests, to help you assess the performance of Web Application Environments (WAEs).

HTTP/S Load is comprised of several Modules including the OpenSTA Architecture and the Commander GUI which runs within it. Use Commander to initiate and control the Test development process, including Script creation, Collector creation, Test creation, running Tests, monitoring Test-runs and displaying the results data for analysis.

HTTP/S Load combines HTTP/S recording and Script modeling functionality, using the Script Modeler Module, with Test creation and system data collection. It records browser requests issued during a Web session at the HTTP/S level, rather than recording the real time events of a browser, in order to create Scripts. This allows you to create and run load Tests, incorporating Scripts, that use minimum system resources enabling you to carry out large volume load Tests.

In Commander, a Test is represented as a table known as the Test Pane. This is the workspace where you can develop the contents of a Test by adding the Scripts and the Collectors you need from the Repository. Select them individually working from the Repository Window, then drag and drop them into the Test Pane in the required order.

Collectors are used to monitor and record performance data during a Test-run. They contain user-defined data collection queries and monitoring options that control the data collected from Host computers and other target devices during a Test-run.

Scripts are created from the recordings of HTTP/S browser requests issued during a Web session and written in SCL scripting language, which enables you to model their content. They encapsulate the Web activity you want to simulate during a Test-run and enable you to generate the load levels required against target WAEs by controlling the number of Virtual Users who run them.

Tests can be comprised of one or more Collectors, one or more Scripts or a combination of both, depending on whether you are performance testing a system within a development or a production environment. It is possible to modify a load Test to monitor the same target system in a production scenario by disabling the Scripts it includes so that no load is generated when the Test is run.

During a Test-run you can monitor Task Group activity from the Monitoring tab of the Test Pane. The results collected can be displayed as they are returned to the Repository while a Test is running or after a Test-run is complete, to assist you in the analysis of the target WAE performance.

See also:

Core Functions of HTTP/S Load

Core Functions of HTTP/S Load

HTTP/S Load supplies versatile software that caters for the needs of different users and the type of system you are evaluating, by supplying the full range of functions that e-business project managers and system performance testers need in order to develop transparent, easy to maintain Tests.

HTTP/S Load is a modular software system in which the creation of Scripts, Collectors and Tests are separate processes that can be conducted independently. It provides the functionality required to support the tasks you need to conduct, in order to achieve the objectives of your performance tests.

All Test development procedures are initiated from Commander. Use it to create Tests and to coordinate the development process.

Performance Testing Using HTTP/S Load

1. Create Scripts (Script Modeler).
2. Model Scripts if required (Script Modeler).
3. Create data collection Collectors - optional (SNMP, NT Performance).
4. Create Tests, by adding Task Groups containing the Scripts and Collectors required (Commander).
5. Define Task Group settings (Commander), including:

   - Schedule settings to control when Task Groups start and stop during a Test-run.
   - Host computers used to run a Task Group: Script and Collector-based Task Groups.
   - Number of Virtual Users used: Script-based Task Groups only.
   - Task settings control the number of Script iterations and the delay between iterations during a Test-run: Script-based Task Groups only.

6. Run a Test (Commander).
7. Monitor a Test-run (Commander).
8. Display Test results (Commander).

Note: It is not necessary to stick rigidly to this procedural sequence.

HTTP/S Load supplies flexible software that enables you to work in ways that best suit you and the type of Test you are creating.

See also:

Using HTTP/S Load

Using HTTP/S Load

The main areas of procedure supported by HTTP/S Load are summarized below:

- Creating Scripts
- Modeling Scripts
- Creating Collectors
- Creating Tests
- Running and Monitoring Tests

- Displaying Results

See also:

The Commander Interface

Creating Scripts

Creating Scripts involves deciding how you expect clients to use the WAE under test, then recording browser sessions which incorporate this behavior to produce Scripts. Scripts encapsulate the browser requests issued during a Web session at the HTTP/S level and form the basis of your Tests.

Browser requests and WAE responses are recorded using the OpenSTA Gateway. It is launched automatically when you begin recording a Script using the Script Modeler Module. The Gateway records the HTTP/S requests issued by a browser during Web sessions using SCL scripting language, which enables you to model their content.

Creating Scripts is a separate procedure within the Test development process, and can be carried out independently of Test and Collector creation. For more information see Recording Scripts.

See also:

Modeling Scripts

HTTP/S Scripts

Modeling Scripts

Modeling Scripts involves identifying and editing SCL code that represents user input during a browser session, so that the Scripts can be used in Tests to function as one or more Virtual Users during a Test-run. Modeling Scripts enables you to develop Tests that more accurately simulate the Web activity you want to reproduce during a Test-run.

Modeling Scripts is not an essential procedure, particularly if the WAE under test comprises static content only. But it is a useful facility if you need to record the dynamic changes of a WAE during a session. For example, you may need to use a unique user name and password for each Virtual User, so that the Test more accurately simulates real end user activity. You can achieve this by creating a Script then modeling it to include variables that change the user name and password for each Virtual User, every time the Script is run as part of a Test. Using just one modeled Script it is possible to create all the Virtual Users you need, each with unique identities just like real end users.

Script Modeling is enhanced beyond the addition of variables to a Script. The Web pages issued in response to browser requests are recorded at the same time as a Script is created. In HTTP/S Load there is the capability to use objects from these Web pages to model the corresponding Script. This modeling technique is known as DOM Addressing. This technique can be used to verify the results of a Test by checking the validity of WAE responses during Test-run. For more information see Recording Scripts and Modeling Scripts.

See also:

Creating Collectors

HTTP/S Scripts

Creating Collectors

Creating Collectors involves deciding which Host computers or other devices to collect performance data from and the type of data to collect during a Test-run. HTTP/S Load supports the creation of NT Performance for recording performance data from Hosts running Windows NT or Windows 2000, and SNMP Collectors for collecting SNMP data from Hosts and other devices running an SNMP agent or proxy SNMP agent.

Collector-based Task Groups can be monitored during a Test-run. The data collected can be displayed alongside other results to provide information about a Test-run.

Creating Collectors is a separate procedure within the Test development process and can be carried out independently of Test and Script creation. For more information see Creating and Editing Collectors.

See also:

Creating Tests

Creating Tests

Creating Tests first involves creating the Scripts and Collectors you want to include in them. Then select the Scripts and Collectors you need from the Repository Window and add them one at a time to a Test. Scripts and Collectors are included in Tests by reference. This means that you can include them in multiple Tests in which different Task Group settings apply.

The Scripts and Collectors you add are known as Tasks which are structured in Script-based and Collector-based Task Groups. A load Test must contain at least one Script-based Task Group which can include one, or a sequence of Scripts. Collector-based Task Groups are optional.

Create and run Collector-only Tests for performance monitoring and data collection within production scenarios. Or alternatively, use a load Test that includes Collectors and disable the Script-Task Groups it includes, to turn off the load element they supply before running the Test within a production monitoring environment.

The Test scenario you want to simulate during a Test-run can be controlled by adjusting the Task Group settings. Assemble the Scripts and Collectors of your Test then select the Task Group settings you want to apply in order to generate the level of load required. For Script-based Task Groups these settings include the Host used, the number of Virtual Users and the number of Script iterations per Virtual User. For Collector-based Task Groups the Host used to run the Task Group can be defined.

Creating Tests is a separate procedure within the Test development process, and can be carried out independently of Script and Collector creation. For more information see Creating and Editing Tests.

See also:

Running and Monitoring Tests

Running and Monitoring Tests

Running a Test enables you to imitate real end user Web activity and accurately simulate the test conditions you want in order to generate the level of load required against target WAEs.

The Task Groups that comprise a Test can be run on remote Hosts during a Test-run. Distributing Task Groups across a network enables you to run Tests that generate realistic heavy loads simulating the activity of many users.

You can monitor the progress of a Test-run by selecting a Script-based Task Group and tracking the Scripts and the Virtual Users that are currently running from the Monitoring tab of the Test Pane.

You can add Collector-based Task Groups to a Test which can be monitored by selecting the data collection queries defined in the Collector and displaying them in graphs. For more information see Running Tests.

See also:

Displaying Results

Distributed Architecture

Displaying Results

Running a Test then displaying the results enables you to analyze and assess whether WAEs will be able to meet the processing demands that will be placed on them.

HTTP/S Load provides a variety of data collection and display options to assist you in the analysis of Test results. When a Test is run a wide range of data is generated automatically. It enables you to collect and graph both Virtual User response times and resource utilization information from all WAEs under test, along with performance data from the Hosts used to run the Test.

After a Test-run is complete the results can be displayed. Open the Test you want from the Repository Window and click on the Results tab in the Test Pane, then select the results you want to display. For more information see Results Display.

See also:

The Commander Interface

Creating and Editing Collectors

The Commander Interface

Commander combines an intuitive user interface with comprehensive functionality to give you control over the Test development process, enabling you to successfully create and run performance Tests.

Use the menu options or work from the Repository Window to initiate the creation of Collectors, Scripts and Tests. Right-click on the Repository Window folders and choose from the functions available.

Work within the Main Window of Commander to create Collectors and Tests. The Main Window houses the Repository Window and supplies the workspace for Test creation using the Test Pane, and Collector creation using the Collector Pane. Use Script Modeler to create the Scripts you need.

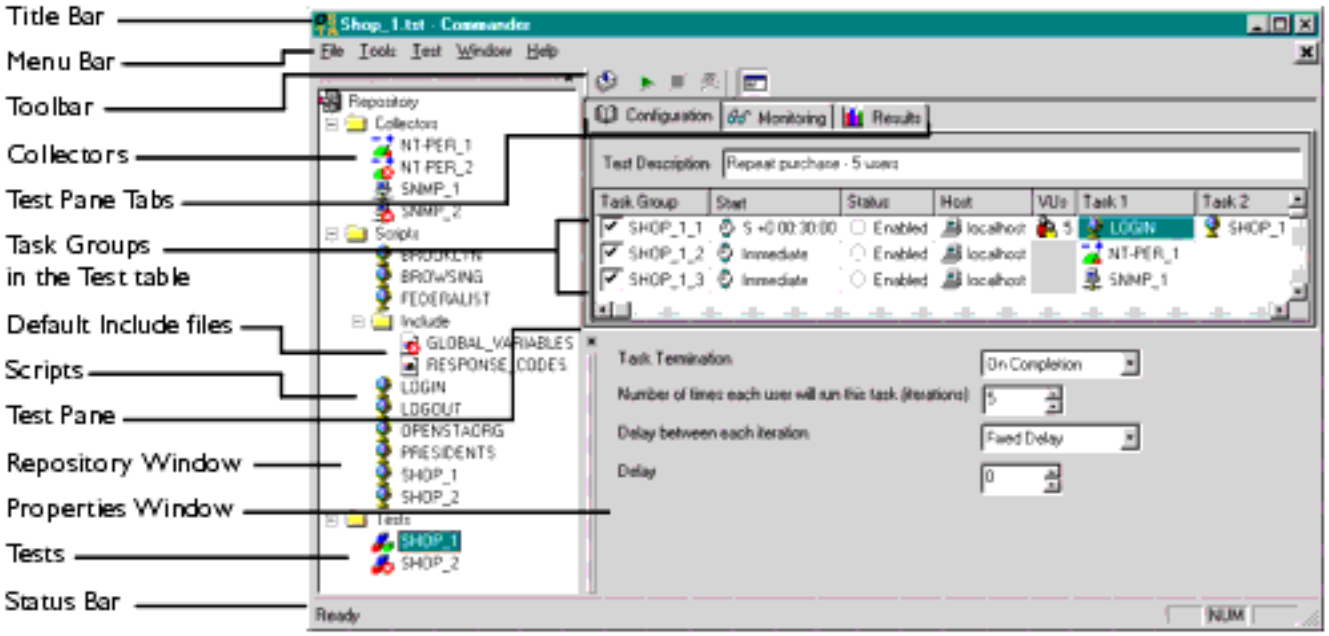
After you have created or edited a Test or Collector in the Main Window they are automatically saved when you switch to another procedure.

The Commander interface is divided up into three primary areas:

- Commander Toolbars and Function Bars.
- The Repository Window.
- The Commander Main Window.

Commander Interface Features

The main features of the Commander interface are detailed below:



Note: The Test illustrated above contains Script-based and Collector-based Task Groups.

See also:

Commander Toolbars and Function Bars

Creating and Editing Collectors

Commander Toolbars and Function Bars

Function Bars

- **Title Bar**: Displays the Commander Control Icon , which gives you access to the Control menu. It also incorporates the standard Windows buttons Minimize, Restore/Maximize and Close.

  Double-click the Title Bar to toggle between a maximized and reduced window display.

- **Menu Bar**: Displays Commander menu options, including the File option from where you can create new Scripts, Collectors and Tests.

  Some of the menu options available vary depending on the procedure you are performing. If you are editing a Test, the Test menu option appears. If you switch to editing a Collector, the Collector menu option appears, replacing the Test menu option.

  Select a menu option using your mouse or use the keyboard shortcuts to

access the functions you need.

- **Status Bar**: Displays information relating to the current file.

Toolbars

When you start-up Commander no toolbars are visible. A toolbar relevant to the procedure you are conducting appears below the Menu Bar when you open a Test or Collector in the Main Window.

You can hide toolbars using the **View** menu option to maximize the workspace available in the Main Window. You can also float toolbars over the Main Window to increase the workspace area.

See also:

Hide/Display Toolbars

The Commander Main Window

Hide/Display Toolbars

- Click **View > Toolbar**.

  A tick to the left of the toolbar listed indicates that it is currently displayed.

The Commander Main Window

The Commander Main Window is located below the Menu Bar and functions as a workspace and container for the creation of Tests and data collection Collectors.

The Test Pane is displayed here when you open a Test by double-clicking a Test icon,  or  , in the Repository Window. Use the Test Pane to create and edit Tests, then run a Test and monitor its progress. When results are returned they can be displayed here for analysis. For more information, see The Test Pane.

The Collector Pane is displayed in the Main Window when you open a Collector by double-clicking a Collector icon,  ,  (NT Performance), and  ,  (SNMP), in the Repository Window. Use this workspace to create and edit Collectors. For more information, see Creating and Editing Collectors.

The Repository Window is displayed in the Commander Main Window. Use it to initiate the creation of the Scripts, Collectors and Tests by right-clicking on the default folders within and selecting the menu options you need.

See also:

Commander Main Window Display Options

- Resize the Main Window to increase your workspace area by adjusting the borders of the Repository Window.

  Click on the right-hand border of the Repository Window then drag the border to the desired position.

- Float the Repository Window over the Main Window to increase the workspace to the full width of the Main Window.

  Click on the double bar at the top of the Repository Window then drag and drop it in the new location.

- Close the Repository Window to maximize the workspace area.

  Click  , in the top right-hand corner of the window, or select **Tools > Show Repository**.

  To display the Repository Window select **Tools > Show Repository**.

- Resize the Main Window by adjusting the borders.

  Click on the border of a window and drag it to the required position.

The Repository Window

After you have planned your performance Test you can work from the Repository Window to initiate Test development procedures, including the creation of Scripts, Collectors and Tests. The Repository Window displays the contents of the Repository which stores all the files that define a Test. Use the Repository Window to manage the contents of the Repository by creating, displaying, editing and deleting Collectors, Scripts and Tests.

The Repository Window is located on the left-hand side of the Main Window by default and displays the contents of the Repository in three predefined folders  **Collectors**,  **Scripts**, and  **Tests**. These folders organize the contents of the Repository into a directory structure which you can browse through to locate the files you need. Double-click on the predefined folders to open them and display the files they contain. These folders have functional options associated with them, which you can access by right-clicking on the folders. They present separate right-click menus which contain options for creating new Collectors, Scripts and Tests.

When you double-click on a Test or Collector in the Repository Window, they are opened in the Commander Main Window, where they can be developed or

edited. Double-click on a Script in the Repository Window to open it using the Script Modeler. This Module is launched in a separate window where you can create and model Scripts.

The Scripts, Collectors and Tests stored in the Repository are organized alphabetically and can be deleted by using the right-click menu option associated with each file or by using the keyboard.

The order and appearance of the predefined folders, Collectors, Scripts and Tests, cannot be modified.

Repository Path

When you first run Commander the Repository that was automatically created in the default location within the program directory structure is displayed, which appears as  **Repository**. Additional Repositories can be created using Commander that can be located on your hard drive or on a networked computer.

We recommend changing the location of the Repository, especially if you expect to generate large volumes of Test related files, so that the performance of your computer is not compromised. Use the Select a New Repository Path option in the Tools menu to create a new Repository or change the path.

See also:

Select a New Repository Path

Repository Window Display Options

The Commander Interface

Collectors Folder

Scripts Folder

Tests Folder

Collectors Folder

The Collectors folder in the Repository Window displays all the Collectors stored in the Repository and has a right-click menu option associated with it that enables you to create new Collectors.

Open the Collectors folder and display the Collectors contained by double-clicking  **Collectors**.

See also:

Collectors Folder and Collectors, Display Options and Functions

- Double-click **Collectors** in the Repository Window, to expand or collapse the directory structure, or click ⊞ , or ⊟ , alongside the folder.
- Right-click **Collectors**, to display a pop-up menu from where you can choose to create a **New Collector**, either **SNMP** or **NT Performance**.
- Double-click on a new Collector (NT Performance) or (SNMP), to open it in the Collector Pane in the Commander Main Window and define the performance data to be collected during a Test-run.

  Note: When you first create a Collector no data collection queries have been defined, so it appears with a small crossed red circle over the Collector icon to indicate this, or . After you have defined your data collection queries, the cross is removed, (NT Performance) or (SNMP).

- Double-click on a Collector or , to open it in the Collector Pane in the Commander Main Window and make any edits required to the files.

  Note: When a Collector is open in the Test Pane, the Collector icon in the Repository Window appears with a small, yellow lock icon overlaid, . This makes it easy to spot which Collector or Test is currently open in Commander. The name of the open Collector or Test is displayed in the Commander Title bar.

- Right-click on a Collector or , to display a pop-up menu which gives you the option to **Rename** or **Delete** the Collector from the Repository.

  Note: If a Collector is open , it cannot be renamed or deleted.

  Note: Collectors are included in Tests by reference so editing their data collection queries affects the type of results recorded during a Test-run for all the Tests that use them.
  Renaming a Collector or deleting one from the Repository means that the Tests using them cannot run. A missing Collector is still referenced in a Task Group and the altered status of the Collector Task is indicated within the Configuration tab of an open Test by highlighting in red the cell it occupies in the Test table. Tests can only run if a missing Collector is recreated, an existing Collector is renamed, or the Collector Task is

deleted from the Task Group.

See also:

Creating and Editing Collectors

Collectors Folder

Scripts Folder

The Scripts folder in the Repository Window displays all the Scripts stored in the Repository and has a right-click menu option associated with it that enables you to create new Scripts.

Open the Scripts folder and display your Scripts by double-clicking **Scripts**.

See also:

Scripts Folder and Scripts, Display Options and Functions

Tests Folder

Collectors Folder

Scripts Folder and Scripts, Display Options and Functions

- Double-click **Scripts** in the Repository Window, to expand or collapse the directory structure or click ⊞ , or ⊟ , alongside the folder.
- Right-click **Scripts**, to display a pop-up menu from where you can create a **New Script > HTTP**.
- Double-click on a new Script , to launch Script Modeler and record a web browser session.

  Note: When you first create a Script in Commander it contains no data because no HTTP/S recording has been performed, so it appears with a small crossed red circle over the Script icon to indicate this .
  After you have recorded the Script, the cross is removed, .

- Double-click on a Script , to launch Script Modeler and model the Script.
- Right-click on a Script , to display a pop-up menu which gives you the option to **Rename** or **Delete** the Script from the Repository.

  Note: Scripts are included in Tests by reference, so modeling them affects the type of web browser activity generated during a Test-run for all the Tests that use them.

Renaming a Script or deleting one from the Repository means that the Tests using them cannot run. A missing Script is still referenced in a Task Group and the altered status of the Script Task is indicated within the Configuration tab of an open Test by highlighting in red the cell it occupies in the Test table. Tests can only run if a missing Script is recreated, an existing Script is renamed, or the Script Task is deleted from the Task Group.

See also:

Scripts Folder

Working With Scripts

Tests Folder

The Tests folder in the Repository Window displays all the Tests stored in the Repository and has a right-click menu option associated with it that enables you to create new Tests.

Open the Tests folder and display your Tests by double-clicking **Tests**.

See also:

Tests Folder and Tests, Display Options and Functions

Scripts Folder

Collectors Folder

Tests Folder and Tests, Display Options and Functions

- Double-click **Tests** in the Repository Window, to expand or collapse the directory structure or click ⊞ , or ⊟ , alongside the folder.
- Right-click **Tests**, to display a pop-up menu from where you can create a **New Test > Test**.
- Double-click on a new Test , to open it in the Test Pane in the Commander Main Window, where you can add and delete Task Groups which contain the Scripts and Collectors you want. Apply the Task Group settings you need using the Properties Window, to control the load generated during a Test-run and which Hosts are used to run each Task Group. .

  Note: When you first create a Test it contains no Task Groups containing Scripts or Collectors, so it appears with a small crossed red circle over the Test icon to indicate this . After you have added one or more Task

Groups, the cross is removed,  .

- Double-click on a Test  , to open it in the Test Pane in the Commander Main Window, where you can add, delete and reorganize Task Groups containing Scripts and Collectors, then edit your Task Group settings.

  Note: When a Test is open in the Test Pane, the Test icon in the Repository Window appears with a small, yellow lock icon overlaid,  . This makes it easy to spot which Test or Collector is currently open in Commander. The name of the open Test or Collector is displayed in the Commander Title Bar.

- Right-click on a Test  , to display a pop-up menu which gives you the option to **Rename** or **Delete** the Test from the Repository.

  Note: If a Test is open  , it cannot be renamed or deleted.

  Note: Scripts and Collectors are included in Tests by reference, so editing, renaming or deleting Tests does not affect the Scripts and Collectors they contain.

See also:

Tests Folder

Creating and Editing Tests

Repository Window Display Options

The Repository window is located on the left-hand side of the Main Window by default. You can hide it to increase the Main Window workspace area available, move it to a new position or float it over the Main Window.

See also:

Hide/Display The Repository Window

Move The Repository Window

Resize The Repository Window

Select a New Repository Path

Hide/Display The Repository Window

- Click  , in the double bar at the top of the Repository Window to close it.
- Select **Tools > Show Repository** to hide and display the Repository

Window.

Move The Repository Window

1. Click on the double bar at the top of the Repository Window.
2. Drag, then drop it in the new position within the Main Window.

   Note: The Repository Window docks with the Main Window's borders if it contacts them.

Resize The Repository Window

1. Move your cursor over part of the window edge.
2. Click and drag, then drop the border in the required position.

Select a New Repository Path

1. In Commander select **Tools > Repository Path**.
2. In the Browse for folder dialog box, select a new Repository path by moving through the directory structure displayed and choose the location you want.
3. Click **OK** to create a new Repository.

   Note: You can create several Repositories and use them to store different performance Test projects. This procedure creates a new Repository if none exists in the location you choose, or switches the Repository Path to reference an existing Repository.

Getting Started

Minimum System Requirements for Installation

Make sure your PC conforms to the following minimum hardware and software requirements:

Hardware Specifications

- Pentium 200 processor
- 80MB RAM
- 20MB free hard disk space required for installation.

Web Browsers Supported for HTTP/S Recording in Script Modeler

- Internet Explorer 4
- Internet Explorer 5
- Netscape 4.7

Supported Protocols

- HTTP 1.0 and 1.1
- HTTPS (SSL)

Software Prerequisites

- Microsoft Windows 2000 or Microsoft Windows NT 4.0, with at least service pack 5
- Windows Installer for Windows NT 1.1, **instmsi.exe**. This is not part of the basic installation of Windows NT 4.0. It can be downloaded from **http://opensta.org/**.
- An up-to-date HTML Help system, the update may be downloaded from msdn.microsoft.com; search for Microsoft HTML Help.
- OpenSTA also requires version 2.5, or later, of Microsoft Data Access Components **MDAC_Typ.exe**. Visit **http://opensta.org/** for download details.

See Also:

Installing HTTP/S Load and OpenSTA

Installing HTTP/S Load and OpenSTA

1. Close down all applications.
2. Locate the OpenSTA Microsoft Windows Installer Package, **.MSI** file, and double-click.

   Or click **Start > Run**. Click **Browse** and locate the executable file or type the path and file name, then click **OK**.

   After the install preparation is complete, the Welcome window appears.

   Note: You may need Administrator rights depending on your computer's configuration.
3. Click **Next**.
4. In the Select Installation Folder window, enter the installation path in the text box.

   Make sure the location you select for the installation has at least 20MB of free space.

   Note: This location is the default location for the automatic creation of the Repository when you first run Commander. We recommend you

Select a New Repository Path after startup.

5. Follow the on-screen instructions until installation is complete.
6. At the end of the installation procedure you must reboot your computer before running Commander.

   Note: When your system has restarted the OpenSTA Name Server should be running automatically. This is indicated by the Name Server Configuration Utility  , which appears in the Task Bar. The OpenSTA Name Server must be running on the Host computers you use to run a Test.

See Also:

Commander Startup Instructions

Overview of HTTP/S Load

Commander Startup Instructions

Commander is the Graphical User Interface that runs within the OpenSTA Architecture and functions as the front end for all Test development activity. It is the program you need to run in order to use HTTP/S Load.

See also:

Launch Commander

Changing the Repository Path

Launch Commander

- Click **Start > Programs > OpenSTA > OpenSTA Commander**.

  Or,

1. Click **Start > Run**.
2. Enter the application path and program file:

   \Program Files\OpenSTA\BaseUI\OSCommander.exe

   or click **Browse**, then locate and double-click the program file.
3. Click **OK** to launch Commander.

   Note: When you startup Commander for the first time an empty Repository is automatically created in the program directory structure.

See also:

Changing the Repository Path

Overview of HTTP/S Load

Changing the Repository Path

The Repository is used to store all the files that define a Test and Test results, which can use up a lot of hard disk space when you create and run Tests.

The OpenSTA program directory structure is the default location for the creation of the Repository when you first startup Commander.

We recommend changing the location of the Repository if you expect to generate large volumes of Test related files, so that the performance of your PC is not compromised.

See also:

Select a New Repository Path

Select a New Repository Path

1. In Commander select **Tools > Repository Path**.
2. In the Browse for folder dialog box, select a new Repository path by moving through the directory structure displayed and choose the location you want.
3. Click **OK** to create a new Repository.

   Note: You can create several Repositories and use them to store different performance test projects.

   This procedure creates a new Repository if none exists in the location you choose, or switches the Repository Path to reference an existing Repository.

Upgrading

You can download the latest version of HTTP/S Load from **http://opensta.org/**.

Before installing new versions of HTTP/S Load you must remove the current version. Then reboot your PC and install HTTP/S Load as previously described.

When you uninstall HTTP/S Load only program files are removed. The Repository is unaffected so your Tests and Test results are available when you

startup the new version.

The OpenSTA program directory structure is the default location for the creation of the Repository when you first startup Commander. So if you install the upgrade in a different location to the previous version, you will need to Select a New Repository Path to locate the Repository and access existing Tests.

See also:

Uninstalling HTTP/S Load and OpenSTA

Installing HTTP/S Load and OpenSTA

Select a New Repository Path

Uninstalling HTTP/S Load and OpenSTA

1. Locate the OpenSTA Microsoft Windows Installer Package, **.MSI** file, and double-click.

   Or click **Start > Run**. Click **Browse** and locate the executable file or type the path and file name, then click **OK**.

   After the install preparation is complete, the Welcome window appears.
2. Click **Remove OpenSTA**.
3. Click **Finish** to uninstall the current version of OpenSTA.
4. Follow the on-screen instructions until the uninstall is complete.

   Note: At the end of the uninstall procedure you must reboot your PC before installing an upgrade version of HTTP/S Load.

   When you uninstall HTTP/S Load only program files are removed. The Repository is unaffected.

See also:

Installing HTTP/S Load and OpenSTA

Getting Help

On-line Help

HTTP/S Load incorporates a comprehensive on-line Help system. Click **Help** in the Menu Bar and select an option from the list.

Web Site Help

HTTP/S Load and OpenSTA are supported by several Web sites which contain a variety of product information. Some of these sites can be accessed through the HTTP/S Load on-line Help system.

http://opensta.org/: Includes a variety of useful information and downloads, including the software, source code, documentation, *HTTP/S Testing, Getting Started Guide* and demonstration Web site for product training.

Download and work through the *HTTP/S Testing, Getting Started Guide* tutorial in combination with the demo Web site to learn basic functionality of HTTP/S Load.

http://opensta.com/: Includes information on how OpenSTA can help you in relation to your specific business circumstances. Provides details on Consultancy, Training and Support services relating to OpenSTA HTTP/S Load software. It also includes FAQs, mailing lists, contact details and product news.

http://cyrano.com/: OpenSTA Support, Consultancy and Training services, are available through CYRANO.

Technical Support

Technical support is available from CYRANO Technical Support Department if you have a valid support license. For details on purchasing a support license contact **http://opensta.com**/.

Technical Support:	France: **support-fr@cyrano.com** USA, Canada & South America: **support-us@cyrano.com** UK: **support-uk@cyrano.com**
Telephone:	France: **+33 (0) 1 56 33 40 00** USA: **+1 (978) 499 3629**, Toll Free: **+1 (800) 714-4900** UK: **+44 (0) 1274 761024**

If you are located in a country which is not listed above please contact us at: **http://cyrano.com/support/**, and complete the form.

Feedback

Use the OpenSTA mailing lists if you have suggestions or questions about HTTP/S Load, or if you want to share your experiences of using the product. You can browse through previously submitted messages in the archives to see if other users may have addressed issues that interest you. For up to date information on all mailing list discussion forums go to **http://opensta.org**/.

If you have questions or comments that you do not feel are appropriate for the mailing groups, launch **http://opensta.org/** and follow the **Contacts** link.

Contact OpenSTA project members at CYRANO

Mailing Lists	There are three mailing lists available, Announce, User and Developer. To find out more and to sign up to the appropriate mailing list visit: **http://opensta.org/**
E-mail:	Further enquiries: **info@opensta.com** Documentation: **docs@opensta.org**
Web sites:	**http://opensta.org/** **http://opensta.com/** **http://cyrano.com/**

Introduction

What is HTTP/S Load?

HTTP/S Load supplies performance testing software for evaluating Web Application Environments (WAEs). It is built on the OpenSTA Architecture which is a distributed testing architecture that enables you to create and run versatile production monitoring and HTTP/S load Tests. It can be employed at all stages of WAE development as well as being used to continuously monitor system performance once an WAE goes live.

Use HTTP/S Load to develop Tests which monitor and collect performance data from live WAEs within a production environment and to create load Tests which include an HTTP/S load element, known as Scripts, to help assess the performance of WAEs during development. HTTP/S Load enables you to run Tests against the same target system within both load testing and production monitoring scenarios. This means that you can directly compare the performance of the target system within these two environments.

Within HTTP/S Load, use Commander to create Collectors and Scripts, then create and run the Tests which incorporate them in order to generate the performance data you need.

HTTP/S Load Test

HTTP/S Load is designed to create and run HTTP/S load Tests to help assess the performance of WAEs. Tests can be developed to simulate realistic Web scenarios by creating and adding Scripts to a Test to reproduce the activity of hundreds to thousands of users. Resource utilization information and response

times from WAEs under test can be monitored and collected during a Test-run and then displayed. This enables you to identify system limitations or faults before launch, or after Web services have been modified, in order to help you create reliable Web sites that meet your load requirements.
Load Tests can also incorporate Collectors which monitor and record the performance of target components that comprise the system under test.

The Scripts used in a Test can be disabled when the WAE goes live allowing you to use the same Test and the Collectors it includes, to monitor and record performance data during a production-based Test-run. Test results can then be directly compared to assess the performance of the target system within a load Test and production environment.

Production Monitoring Test

HTTP/S Load supports the creation of Collector-only Tests. The ability to develop Tests without an HTTP/S load element enables you to create and run Tests which monitor and collect performance data from target systems in a production scenario. In this environment Tests are used to monitor and collect performance data within a production system where the load is generated externally by the normal use of the system.

See also:

Getting Started

HTTP/S Load

Documentation Conventions

This guide uses the following conventions to indicate specific actions. Most of these relate to procedural text and are listed below:

Example	**Describes**
Click **OK**	Pressing the left mouse button to select the **OK** button. Bold text represents menu options, dialog box selections and key presses.
Right-click	Pressing the right mouse button.
Control-click	Holding down the **Control** key and pressing the left mouse button.

Select **File > New Test**	Choosing the **New Test** option from the **File** menu. The **>** character indicates the requirement to select each option in sequence.
Alt + 2	Holding down the **Alt** key and pressing **2**.
`Fixed-Pitch type`	Program code.

Note: Most procedures in Commander are supported by keyboard shortcuts.

Feedback

If you have any comments or suggestions relating to this help system or any other product documentation, please e-mail them to **docs@opensta.org**.

See also:

Getting Started

Welcome to the HTTP/S Load under OpenSTA

Congratulations on choosing the HTTP/S Load under OpenSTA.

HTTP/S Load is an ideal tool for performance testing Web Application Environments (WAEs). It supplies versatile software that enables you to create and run HTTP/S load Tests and production monitoring Tests to help evaluate target systems. Use it to assess the performance of WAEs before launch or after modifications to Web services.

HTTP/S Load enables you to run Tests against the same target system within both load testing and production monitoring scenarios. Tests can include an HTTP/S load element to help you evaluate the performance of WAEs during development. Tests can also be used to monitor and collect performance data from WAEs after they go live within a production scenario. Results data is collected during Test-runs and can be displayed for analysis during or after a Test-run, enabling you to directly compare the performance of a target system within these two environments.

Having a reliable Web site is an absolute requirement to compete in the e-market place and the only way to ensure this is to thoroughly test it before launch. HTTP/S Load provides you with a versatile solution to this requirement.

OpenSTA - Open System Testing Architecture

HTTP/S Load is built on the OpenSTA Architecture, which is an Open Source system testing architecture, built around the Object Management Group's CORBA standard. It provides the foundations for building an integrated and comprehensive modular testing environment for WAEs. OpenSTA performance testing methodology is based on the OpenSTA Architecture and additional OpenSTA Modules, which will encompass the full range of WAE performance testing requirements.

OpenSTA allows you to enhance the scope of your performance tests by

installing additional OpenSTA Modules to test and analyze the components of your WAEs. OpenSTA enables you to integrate all the elements of your performance tests and to develop a coordinated and systematic approach to testing and results analysis. This approach allows the implementation of testing methodologies which enable you to produce accurate results on which to develop strategies for enhancing the performance of your WAEs.

For more information visit **http://opensta.org/**, **http://opensta.com/** and **http://cyrano.com/**.

See also:

Introduction

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

OpenSTA

Mailing Lists Search Advanced

Stats RSS

Mailing lists provided via a SourceForge.net version of GNU Mailman. Thanks to the Mailman and Python crews for excellent software.

Choose a list to browse or search. To post to this list, send mail to listname@lists.sourceforge.net, replacing listname with the name of the list, shown below.

opensta-announce Archives 38 messages
Approximate subscriber count: 611
(go to Subscribe/Unsubscribe/Preferences)
OpenSTA project update announcements

opensta-devel Archives 1256 messages
Approximate subscriber count: 99
(go to Subscribe/Unsubscribe/Preferences)
OpenSTA developer discussions

opensta-users Archives 7063 messages
Approximate subscriber count: 719
(go to Subscribe/Unsubscribe/Preferences)
OpenSTA users discussion and support

Production Monitoring: Getting Started Guide

Introduction

OpenSTA is a distributed testing architecture that enables you to create and run performance Tests to evaluate Web Application Environments (WAEs) and production systems. It can be employed at all stages of WAE development as well as being used to continuously monitor system performance once a WAE goes live.

Use it to develop load Tests that include an HTTP/S load element, known as Scripts, to help assess the performance of WAEs during development, and to create Tests that monitor and collect performance data from live WAEs within a production environment.

OpenSTA enables you to run Tests against the same target system within both load testing and production monitoring scenarios. This means that you can directly compare the performance of the target system within these two environments.

This guide is intended to give new users a practical introduction to OpenSTA by explaining how to create and run a simple production monitoring Test. It is structured according to the procedural sequence for creating a production monitoring Test, from creating a Collector through to running a Test and results display. It documents the key features and procedures you need to use.

Before working through the guide, make sure you have downloaded and installed OpenSTA version 1.2 and that the target Host computers in the system under test are operational.

Launch http://opensta.org/download.html for download and installation instructions.

Contents

Notes

- Make use of the Glossary at the end of this guide if you come across unfamiliar terminology.
- If you have already used OpenSTA and want to know how to perform a specific task, please refer to the appropriate section in the *HTTP/S Load User's Guide*, which you can view or download from OpenSTA.org.
- If you want information on using OpenSTA to performance test WAEs, please refer to the *HTTP/S Testing: Getting Started Guide*, which you can view or download from OpenSTA.org.

Next Section: OpenSTA Overview

OpenSTA Overview

OpenSTA includes versatile Test development software that enables you to create and run Tests tailor-made for the environment you are assessing.

OpenSTA is a distributed software testing architecture based on CORBA which enables you to create then run Tests across a network. The OpenSTA Name Server configuration utility is the component that allows you to control your Test environment.

After installing OpenSTA you will notice that the OpenSTA Name Server is running indicated by  , in the Windows Task Bar. This component must be running before you can run a Test.

If no icon appears click **Start > Programs > OpenSTA > OpenSTA NameServer**.

If the OpenSTA Name Server stops the Name Server Configuration utility icon appears  , in the Task Bar. You can start it by right-clicking  , and selecting **Start Name Server** from the menu.

Commander

Commander is the Graphical User Interface that runs within the OpenSTA Architecture and functions as the front end for all Test development activity. It is the program you need to run in order to use OpenSTA.

Launch Commander

- Click **Start > Programs > OpenSTA > OpenSTA Commander**.

The Commander Interface

Commander combines an intuitive user interface with comprehensive functionality to give you control over the Test development process, enabling you to successfully create and run production monitoring Tests.

Use the menu options or work from the Repository Window to initiate the creation of Collectors and Tests. Right-click on the predefined folders in the Repository Window and choose from the functions available.

Work within the Main Window of Commander to create Collectors and Tests. The Main Window houses the Repository Window and supplies the workspace for Test creation using the Test Pane, and Collector creation using the Collector Pane.

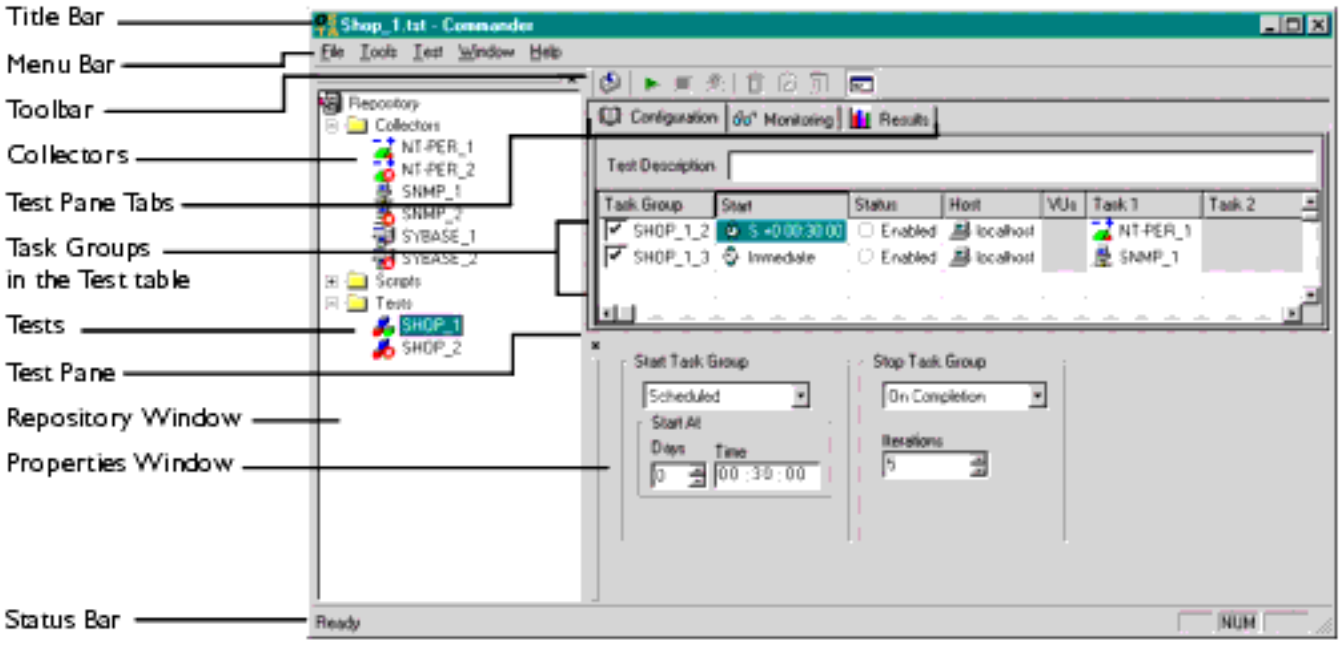
After you have created or edited a Test or Collector in the Main Window it is automatically saved when you switch to another procedure or exit from Commander.

The Commander interface is divided up into three primary areas:

- Commander Toolbars and Function Bars.
- The Repository Window.
- The Commander Main Window.

Commander Interface Features

The main features of the Commander interface are detailed below:



Now you have an overview of OpenSTA and Commander, you are ready to create a Collector to include in a new Test. Move on to the next chapter for information on the Collector creation process.

Next Section: Creating a Collector

Contents

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q - R - S - T - U - V - W - X - Y - Z

Index

C

G

H

L

M

N

O

P

R

S

T

W

Glossary

Collector

An OpenSTA Collector is a collection of user-defined queries which determine the performance data that is monitored and recorded from target Hosts when a Test is run. OpenSTA supports NT Performance and SNMP Collectors.

Collector Pane

The Collector Pane is the workspace used to create and edit Collectors. It is displayed in the Commander Main Window when you open a Collector from the Repository Window.

Commander

OpenSTA Commander is the Graphical User Interface used to develop and run HTTP/S Tests and to display the results of Test-runs for analysis.

Each OpenSTA Module, provides its own Plug-ins and supplies Module-specific Test Configuration, data collection, Test-run monitoring and Results display facilities. All Plug-in functionality is invoked from Commander.

Cookie

A packet of information sent by a Web server to a Web browser that is returned each time the browser accesses the Web server. Cookies can contain any information the server chooses and are used to maintain state between otherwise stateless HTTP transactions.

Typically cookies are used to store user details and to authenticate or identify a registered user of a Web site without requiring them to sign in again every time they access that Web site.

CORBA

Common Object Request Broker Architecture.

A binary standard, which specifies how the implementation of a particular software module can be located remotely from the routine that is using the module. An Object Management Group specification which provides the standard interface definition between OMG-compliant objects. Object Management Group is a consortium aimed at setting standards in object-oriented programming. An OMG-compliant object is a cross-compatible distributed object standard, a common binary object with methods and data that work using all types of development environments on all types of platforms.

CYRANO

http://cyrano.com/

CYRANO is a public company listed on the EuroNM of the Paris Bourse (Reuters: CYRA.LN, Sicovam 3922). Created in 1989 and publicly trading since 1998, CYRANO is headquartered in Paris, France, with regional headquarters in the UK and USA.

CYRANO is a sponsor and lead developer on the OpenSTA™ project. CYRANO is an end-to-end quality assurance provider to its customers, helping them maximize their IT investments and ensure uninterrupted e-business. CYRANO offers integrated solutions, service and support to companies that want to minimize risk, benchmark Service Level Agreements, and enable Capacity Planning for their IT infrastructures.

Document Object Model or DOM

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents (Web pages). It defines the logical structure of documents and the way a document is accessed and manipulated.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.

For more information:

- What is the Document Object Model?

  www.w3.org/TR/1998/REC-DOM-Level-1-19981001/introduction.html

- The Document Object Model (DOM) Level 1 Specification

www.w3.org/TR/REC-DOM-Level-1/

Gateway

The OpenSTA Gateway interfaces directly with the Script Modeler Module and enables you to create Scripts. The Gateway functions as a proxy server which intercepts and records the HTTP/S traffic that passes between browser and Web site during a Web session, using SCL scripting language.

Host

An OpenSTA Host is a networked computer or device used to execute a Task Group during a Test-run. Use the Configuration tab in the Test Pane of Commander to select the Host you want to use a to run Task Group.

Host also refers to a computer or device that houses one or more components of a Web Application Environment under Test, such as a database. Use Collectors to define a Host and the type of performance data you want to monitor and collect during a Test-run

HTML

Hypertext Markup Language. A hypertext document format used on the World-Wide Web. HTML is built on top of SGML. Tags are embedded in the text. A tag consists of a **<**, a case insensitive directive, zero or more parameters and a **>**. Matched pairs of directives, like **<TITLE>** and **</TITLE>** are used to delimit text which is to appear in a special place or style.

.HTP file

See Scripts.

HTTP

HyperText Transfer Protocol. The client-server TCP/IP protocol used on the World-Wide Web for the exchange of HTML documents. HTTP is the protocol which enables the distribution of information over the Web.

HTTPS

HyperText Transmission Protocol, Secure. A variant of HTTP used by Netscape for handling secure transactions. A unique protocol that is simply SSL underneath HTTP. See SSL.

HTTP/S

Reference to HTTP and HTTPS.

Load Test

Using a Web site in a way that would be considered operationally normal with a normal to heavy number of concurrent Virtual Users.

Modules

See OpenSTA Modules.

Name Server

See OpenSTA Name Server.

O.M.G.

Object Management Group. A consortium aimed at setting standards in object-oriented programming. In 1989, this consortium, which included IBM Corporation, Apple Computer Inc. and Sun Microsystems Inc., mobilized to create a cross-compatible distributed object standard. The goal was a common binary object with methods and data that work using all types of development environments on all types of platforms. Using a committee of organizations, OMG set out to create the first Common Object Request Broker Architecture (CORBA) standard which appeared in 1991. The latest standard is CORBA 2.2.

Open Source

A method and philosophy for software licensing and distribution designed to encourage use and improvement of software written by volunteers by ensuring that anyone can copy the source code and modify it freely.

The term Open Source, is now more widely used than the earlier term, free software, but has broadly the same meaning: free of distribution restrictions, not necessarily free of charge.

OpenSTA Dataname

An OpenSTA Dataname comprises between 1 and 16 alphanumeric, underscore or hyphen characters. The first character must be alphabetic.

The following are not allowed:

- Two adjacent underscores or hyphens.
- Adjacent hyphen and underscore, and vice versa.
- Spaces.
- Underscores or hyphens at the end of a dataname.

Note: Where possible avoid using hyphens in the names you give to Tests, Scripts and Collectors. The hyphen character functions as an operator in SCL and conflicts can occur during Test-runs.

OpenSTA Modules

OpenSTA is a modular software system that enables users to add additional functionality to the system by installing new OpenSTA Modules. When a new Module is installed existing functionality is enhanced, enabling users to develop their configuration of OpenSTA in line with their performance Testing requirements. Each Module comes complete with its own user interface and run-time components.

OpenSTA Modules are separate installables that bolt on to the core architecture to add specific functionality, including performance monitoring and data collection for all three layers of Web Application Environment activity:

- Low-level - Hardware/Operating System performance data
- Medium-level - Application Performance Data
- High-level - Transaction Performance Data

OpenSTA Name Server

The OpenSTA Name Server allows the interaction of multiple computers across a variety of platforms in order to run Tests. The Name Server's functionality is built on the Object Management Group's CORBA standard.

Performance Test

One or more Tests designed to investigate the efficiency of Web Application Environments (WAE). Used to identify any weaknesses or limitations of target WAEs using a series of stress Tests or load Tests.

Proxy Server

A proxy server acts as a security barrier between your internal network (intranet) and the Internet, keeping unauthorized external users from gaining access to confidential information on your internal network. This is a function that is often combined with a firewall.

A proxy server is used to access Web pages by the other computers. When another computer requests a Web page, it is retrieved by the proxy server and then sent to the requesting computer. The net effect of this action is that the remote computer hosting the Web page never comes into direct contact with anything on your home network, other than the proxy server.

Proxy servers can also make your Internet access work more efficiently. If you access a page on a Web site, it is cached on the proxy server. This means that the next time you go back to that page, it normally does not have to load again from the Web site. Instead it loads instantaneously from the proxy server.

RDBMS - Relational Database Management System

A relational database allows the definition of data structures, storage and retrieval operations and integrity constraints. In such a database the data and relations between them are organized in tables. A table is a collection of records and each record in a table contains the same fields. Certain fields may be designated as keys, which means that searches for specific values of that field will use indexing to speed them up.

Where fields in two different tables take values from the same set, a join operation can be performed to select related records in the two tables by matching values in those fields. Often, but not always, the fields will have the same name in both tables. For example, an `orders' table might contain (customer-ID, product-code) pairs and a `products' table might contain (product-code, price) pairs so to calculate a given customer's bill you would sum the prices of all products ordered by that customer by joining on the product-code fields of the two tables. This can be extended to joining multiple tables on multiple fields. Because these relationships are only specified at retrieval time, relational databases are classed as dynamic database management system

ifRepository

The OpenSTA Repository is where Scripts, Collectors, Tests and results are stored. The default location is within the OpenSTA program files directory structure. A new Repository is automatically created in this location when you run Commander for the first time.

You can create new Repositories and change the Repository path if required. In Commander click **Tools > Repository Path**.

Manage the Repository using the Repository Window within Commander.

Repository Host

The Host, represented by the name or IP address of the computer, holding the OpenSTA Repository used by the local Host. A Test-run must be started from the Repository Host and the computer must be running the OpenSTA Name Server.

Repository Window

The Repository Window displays the contents of the Repository which stores all the files that define a Test. Use the Repository Window to manage the contents of the Repository by creating, displaying, editing and deleting Collectors, Scripts and Tests.

The Repository Window is located on the left-hand side of the Main Window by default and displays the contents of the Repository in three predefined folders  **Collectors**,  **Scripts**, and  **Tests**. These folders organize the contents of the Repository into a directory structure which you can browse through to

locate the files you need.

Double-click on the predefined folders to open them and display the files they contain.

Right-click on the folders to access the function menus which contain options for creating new Collectors, Scripts and Tests.

SCL

See Script Control Language.

SCL Reference Guide

Hard copy and on-line versions of this guide are available.

In Script Modeler click **Help > SCL Reference**.

Script

Scripts form the basis of HTTP/S load Tests using OpenSTA. Scripts supply the HTTP/S load element used to simulate load against target Web Application Environments (WAE) during their development.

A Script represents the recorded HTTP/S requests issued by a browser to WAEs during a Web session. They are created by passing HTTP/S traffic through a proxy server known as the Gateway, and encoding the recorded data using Script Control Language (SCL). SCL enables you to model the content of Scripts to more accurately generate the Web scenario you need reproduce during a Test.

Scripts encapsulate the Web activity you need to test and enable you to create the required Test conditions. Use Commander to select Scripts and include them in a Test then run the Test against target WAEs in order to accurately simulate the way real end users work and help evaluate their performance.

Scripts are saved as an .HTP file and stored in the Repository.

Script Control Language

SCL, Script Control Language, is a scripting language created by CYRANO used to write Scripts which define the content of your Tests. Use SCL to model Scripts and develop the Test scenarios you need.

Refer to the *SCL Reference Guide* for more information.

Script Modeler

Script Modeler is an OpenSTA Module used to create and model Scripts produced from Web browser session recordings, which are in turn incorporated into performance Tests by reference.

Script Modeler is launched from Commander when you open a Script from the Repository Window.

SNMP

Simple Network Management Protocol. The Internet standard protocol developed to manage nodes on an IP network. SNMP is not limited to TCP/IP. It can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs, toasters and jukeboxes.

For more information visit the NET_SNMP Web site:

- What is it? (SNMP)

  http://net-snmp.sourceforge.net/

SQL

Structured Query Language. An industry-standard language for creating, updating and, querying relational database management systems.

SQL was developed by IBM in the 1970s for use in System R. It is the defacto standard as well as being an ISO and ANSI standard. It is often embedded in general purpose programming languages.

The first SQL standard, in 1986, provided basic language constructs for defining and manipulating tables of data; a revision in 1989 added language extensions for referential integrity and generalized integrity constraints. Another revision in 1992 provided facilities for schema manipulation and data administration, as well as substantial enhancements for data definition and data manipulation.

SSL

Secure Sockets Layer. A protocol designed by Netscape Communications Corporation to provide encrypted communications on the Internet. SSL is layered beneath application protocols such as HTTP, SMTP, Telnet, FTP, Gopher, and NNTP and is layered above the connection protocol TCP/IP. It is used by the HTTPS access method.

Stress Test

Using a WAE in a way that would be considered operationally abnormal. Examples of this could be running a load test with a significantly larger number of Virtual Users than would normally be expected, or running with some infrastructure or systems software facilities restricted. Collector

Task

An OpenSTA Test is comprised of one or more Task Groups which in turn are

composed of Tasks. The Scripts and Collectors included in Task Groups are known as Tasks. Script-based Task Groups can contain one or multiple Tasks. Tasks within a Script-based Task Group can be managed by adjusting the Task Settings which control the number of Script iterations and the delay between iterations when a Test is run.

Collector-based Task Groups contain a single Collector Task.

Task Group

An OpenSTA Test is comprised of one or more Task Groups. Task Groups can be of two types, Script-based or Collector-based. Script-based Task Groups represent one or a sequence of HTTP/S Scripts. Collector-based Task Groups represent a single data collection Collector. Task Groups can contain either Scripts, or a Collector, but not both. The Scripts and Collectors included in Task Groups are known as Tasks.

A Test can include as many Task Groups as necessary.

Task Group Definition

An OpenSTA Task Group definition constitutes the Tasks included in the Task Group and the Task Group settings that you apply.

Task Group Settings

Task Group settings include Schedule settings, Host settings, Virtual User settings and Task settings and are adjusted using the Properties Window of the Test Pane. Use them to control how the Tasks and Task Group that comprise a Test behave when a Test is run.

Schedule settings determine when Task Groups start and stop.

Host settings specify which Host computer is used to run a Task Group.

Virtual User settings control the load generated against target Web Application Environments during specifying the number of Virtual Users running a Task Group. Set Logging levels to determine the amount of performance statistics collected from Virtual Users running the Tasks. You can also select to Generate Timers for each Web page returned during a Test-run.

Task settings control the number of times a Script-based Tasks are run including the delay you want to apply between each iteration of a Script during a Test-run.

Test

An OpenSTA Test is a set of user-controlled definitions that specify which Scripts and Collectors are included and the settings that apply when the Test is run. Scripts define the test conditions that will be simulated when the Test is run. Scripts and Collectors are the building blocks of a Test which can be

incorporated by reference into many different Tests.

Scripts supply the content of a Test and Collectors control the type of results data that is collected during a Test-run. Task Group settings specify the settings that apply when you run the Test, including the number of Virtual Users, the iterations between each Script, the delay between Scripts and which Host computers are used to run a Test.

Commander provides you with a flexible Test development framework in which you can build Test content and structure by selecting the Scripts and Collectors you need. A Test is represented in table format where each row within it represents the HTTP/S replay and data collection Tasks that will be carried out when the Test is run. Test Tasks are known as Task Groups of which there are two types, either Script-based and Collector-based.

Test Pane

The Test Pane is the workspace used to create and edit Tests, then run a Test and monitor its progress. After a Test-run is complete results can be viewed and compared here. The Test Pane is displayed in the Commander Main Window when you open a Test from the Repository Window.

Threshold Value

Anomaly thresholds are performance rules used to determine the type of data collected by a Sybase Monitor Module Collector during a Test-run and to control the results data displayed in new Anomaly Lists after a Test-run is complete.

Threshold values define performance levels to be monitored and data to be collected from a target Sybase Monitor database server during a Test-run. When database transactions meet or exceed a threshold value (for example, excessive transaction duration), an Anomaly is raised and performance data is recorded.

Transaction

A unit of interaction with a RDBMS or similar system.

URI

Uniform Resource Identifier. The generic set of all names and addresses which are short strings which refer to objects (typically on the Internet). The most common kinds of URI are URLs and relative URLs.

URL

Uniform Resource Locator. A standard way of specifying the location of an object, typically a Web page, on the Internet. Other types of object are described below. URLs are the form of address used on the World-Wide Web. They are used in HTML documents to specify the target of a hyperlink which is

often another HTML document (possibly stored on another computer).

Variable

Variables allow you to vary the fixed values recorded in Scripts. A variable is defined within a Script. Refer to the Modeling Scripts section for more information.

Virtual User

A Virtual User is the simulation of a real life user that performs the activity you specify during a Test-run. You control the activity of your Virtual Users by recording and modeling the Scripts that represent the activity you want. When the Test that includes the Script is run, the Script is replayed exactly as the browser made the original requests.

Web Application Environment, WAE

The applications and/or services that comprise a Web application. This includes database servers, Web servers, load balancers, routers, applications servers, authentication/encryption servers and firewalls.

Web Applications Management, WAM

Consists of the entirety of components needed to manage a Web-based IT environment or application. This includes monitoring, performance testing, results display, results analysis and reporting.

Web Site

Any computer on the Internet running a World-Wide Web server process. A particular Web site is identified by the host name part of a URL or URI. See also Web Application Environment.

Displaying Test Results

After a Test-run is complete use Commander to control which results are displayed and how they are presented, in order to help you analyze the performance of target WAEs and the network used to run the Test.

Open the Test you want from the Repository Window and click on the  **Results** tab in the Test Pane, then choose the results you want to display using the Results Window. Depending on the category of results you select, data is displayed in graph or table format. You can choose from a wide range of tables and customizable graphs to display your results which can be filtered and exported for further analysis and print.

Use the Results Window to view multiple graphs and tables simultaneously to compare results from different Test-runs.

When a Test is run a range of results data is collected automatically, including performance data from the Hosts used to run the Test. Results categories include the Test Summary option which presents a brief description of the Test and the Task Groups settings that applied during a Test-run and the Test Audit log records significant events that occur during a Test-run.

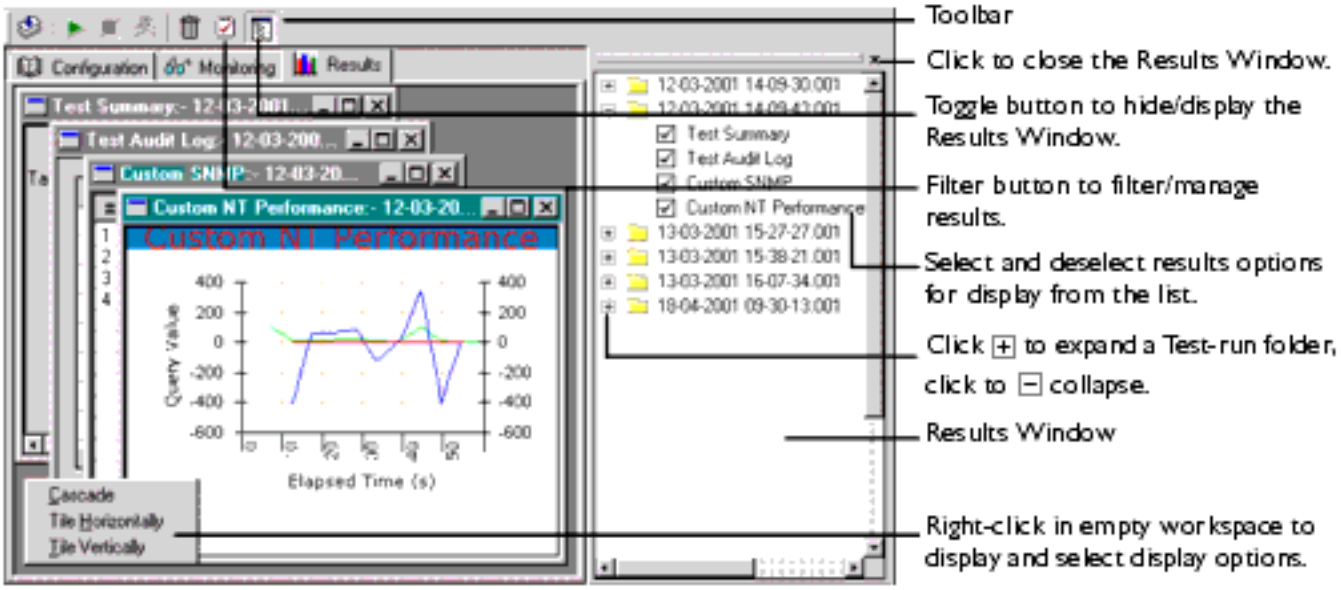
Creating and referencing Collectors in a Test helps to improve the quality and extend the range of the results data produced during a Test-run. NT Performance and SNMP Collectors give you the ability to target the Host computers and devices used to run a Test and the components of WAEs under test, with user-defined data collection queries.

Results Tab

Results are stored in the Repository after a Test-run is complete. You can view them by working from the Repository Window to open the Test you want, then click on the  **Results** tab in the Test Pane.

Use the Results Window to select the results you want to view in the workspace of the Test Pane. You can reposition the Results Window by floating it over the Main Window to give yourself more room for results display, or close it once you have selected the results options you want to view.

The Results Tab of the Test Pane



The Results Window

When you click on the **Results** tab, the Results Window opens automatically. Its default location is on the right-hand side of the Test Pane where it is docked. Use it to select and display results from any of the Test-runs associated with the current Test.

Test-runs are stored in date and time stamped folders which you can double-click on to open, or click ⊞ . When you open a Test-run folder, the available results are listed below. Display the results you want by clicking on the options and ticking the check boxes to the left of the results options. The results you choose are displayed in the Test Pane.

Multiple graphs and tables from different Test-runs associated with the current Test can be displayed concurrently. Use the Results Window to select additional Test-runs and equivalent results options to compare Test results and help evaluate performance.

Display Test Results

1. In the Repository Window, double-click **Tests** to expand the directory structure.
2. Double-click the Test , whose results you want to display.

3. In the Test Pane click the **Results** tab.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.

4. In the Results Window, double-click on a Test-run folder or click , to open it and display the available results.
5. Click on a results option to display your selection in the Test Pane.

   A ticked check box to the left of a results option indicates that it is open in the Test Pane.

   Note: Click , in the title bar of a graph or table to close it or deselect the results option in the Results Window by clicking on the option.

   Tip: All available results have display and output options associated with them, These may include filtering, customizing and exporting. Right-click within a graph or table to display and select from the choices available.

   Use the **Windows** option in the Menu Bar to control the display of graphs and tables. Alternatively, right-click within the empty workspace of the Test Pane to access these functions.

Conclusion

The Test you have created and run whilst working through this guide will hopefully have given you an understanding of the basic techniques involved in successfully developing Tests to monitor and evaluate production environments using OpenSTA, in order to improve their performance.

Contents

Running a Test

Running a Test enables you to monitor and collect data from Host computers according to the data collection queries you have defined in the Collectors referenced by the Test, in order to analyze and assess the performance of a target production environment.

Running a Test is a straightforward procedure, because the Task Group settings have already been specified during Test creation. Open the Test you want to run and click the Start Test button  , in the toolbar.

Dynamic Tests

In OpenSTA Tests are dynamic, which means that the Test contents and settings can be modified while it is running, giving you control over a Test-run and the results that are generated.

New Task Groups can be added and the contents and settings of the existing Task Groups that comprise a Test can be individually edited by temporarily stopping the Task Group, making the changes required then restarting them. These facilities give you control over the load generated and enable you to modify the type of performance data you monitor and record without stopping the Test-run.

Note: It is not possible to remove a Task Group from a Test during a Test-run.

While a Test is running you can:

- Add a new Task Group.

  Scripts and Collectors can be added to a Test and the Task Groups that contain them started.

- View the settings and status of Task Groups using the Properties Window and the Status column of the Configuration tab.
- Modify Task Group settings when the selected Task Group has stopped.

  These settings are:

  Schedule settings

  Host settings

  Virtual User settings (Script-based Task Groups only)

  Task settings (Script-based Task Groups only)

- Stop/Start a Task Group.

  Task Groups can be stopped and started during a Test-run using the **Stop** and **Start** buttons in the new Control column of the Configuration tab. The **Stop** button is displayed if the Task Group is Active and a **Start** button is displayed if the Test is running and the Task Group is stopped, otherwise no button is displayed.

Run a Test

1. In the Repository Window, double-click **Tests** to open the folder and display the Tests contained.
2. Double-click the Test, **PRODUCTION_MONITOR** , you want to run, which launches the Test Pane in the Commander Main Window.
3. Check the Test contains the Collectors you want and that the Task Group settings are correct, then click in the toolbar to run the Test.

   Note: When you click , the Test is automatically compiled. If there is an error during compilation the Compile Errors dialog box appears with a description of the fault(s) to assist you in resolving any problems.

   After your Test has been compiled successfully, the Starting Test dialog box appears which displays a brief status report on the Test-run.

   Tip: Click on the **Monitoring** tab within the Test Pane during a Test-run and select a Collector or Task Group, to monitor the performance of Hosts within the target production system and Hosts used to run the Test, in graph and table format.

Monitoring a Test-run

Task Groups and the Collectors they contain can be monitored using the Monitoring tab of the Test Pane during a Test-run. When you run a Test that

includes Collectors you can monitor:

- A summary of current Test-run activity.
- Collector-based Task Groups: All the data collection queries defined in a Collector.

Monitoring Collectors

1. Make sure the **PRODUCTION_MONITOR** Test is open and running with the **Monitoring** tab of the Test Pane displayed.

   Note: Ensure that the entry in the Status column of the Configuration tab reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click ⊞ , to open a Task Group folder that contains an NT Performance or an SNMP Collector.

   The data collection queries defined in the selected folder are listed below. They represent the display options available.

3. Select one or more of the data collection queries you want to monitor from the Monitoring Window.

   Note: When a Test-run is complete, the entry in the Test Status box at the top of the Monitoring Window reads **INACTIVE** and the display options in the Monitoring Window are cleared.

After you have run your Test, use the results display functions to view the data collected during the Test-run. Move on to the next chapter for details. Click on the **Results** tab within the Test Pane, to display the results generated.

Next Section: Displaying Test Results

Contents

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Creating a Test

After you have planned your Test you need to develop it by adding the Collectors you want to include. Use Commander to coordinate the Test development process by selecting the Collectors you need and combining them into a new Test with the required Task Group settings applied.

Task Group settings control which Host is used to run a Task Group and when it starts and stops during a Test-run. Select a Collector-based Task Group in the Test table then use the Properties Window below to apply your settings.

Tests can be developed and then run using remote Hosts across a network to execute the Task Groups that comprise a Test. In order to do this, OpenSTA must be installed on each Host and the OpenSTA Name Server must be running on each and configured to specify the Repository Host for the Test.

Tasks and Task Groups

Work from the Repository Window, located by default on the left of the Commander Main Window, to create new Tests and to open existing ones.

The Repository Window displays the contents of the Repository and functions as a picking list from where you can select the Collectors and Scripts you want to include in a Test. Use it in conjunction with the Configuration tab of the Test Pane to develop the contents of a Test. Select a Collector or Script from the Repository Window then drag and drop it on to a Task column of a Test to create a new Task within a new Task Group.

The Collectors you add to a Test are referred to as Tasks. A Collector Task is represented by a Collector-based Task Group. When you add a Collector to a Test you can apply the Task Group settings you require, or you can accept the default settings and return later to edit them.

Some of the Task Group cells in the Test table are dynamically linked to the Properties Window below, select them one at a time to display and edit the associated Task Group settings in the Properties Window.

Select the **Start** or **Host** cells in a Task Group row to control the Schedule and Host settings. Script-based Task Groups and the Script Tasks they contain have additional settings associated with them. Select the **VUs** and **Task** cells to control the load levels generated when a Test is run.

Use the Disable/Enable Task Group function to control which Task Groups are executed when a Test is run by clicking the check box in the Task Group column cell. This is a useful feature if you want to disable Script-based Task Groups to turn off the HTTP/S load element. The Test can then be used to monitor a target system within production scenario.

Note: For Production Monitoring you will not need to add a Script. The load element is supplied by the production input to the system.

Collector-based Task Group Settings include:

- **Schedule Settings**: Control when a Task Group starts and stops to determine the period of data collection during a Test-run.
- **Host Settings**: Specify the Host computer you want to use to run a Task Group during a Test-run.

The Test Pane

Use the Test Pane to create and edit a Test, then apply the Task Group settings you require to control how they behave during a Test-run. Run and monitor the Test-run then display your results for analysis.

The Test Pane is displayed in the Main Window when you open a Test by double-clicking a new Test  , or an existing Test  , in the Repository Window.

The Test Pane comprises three sections represented by the following tabs:

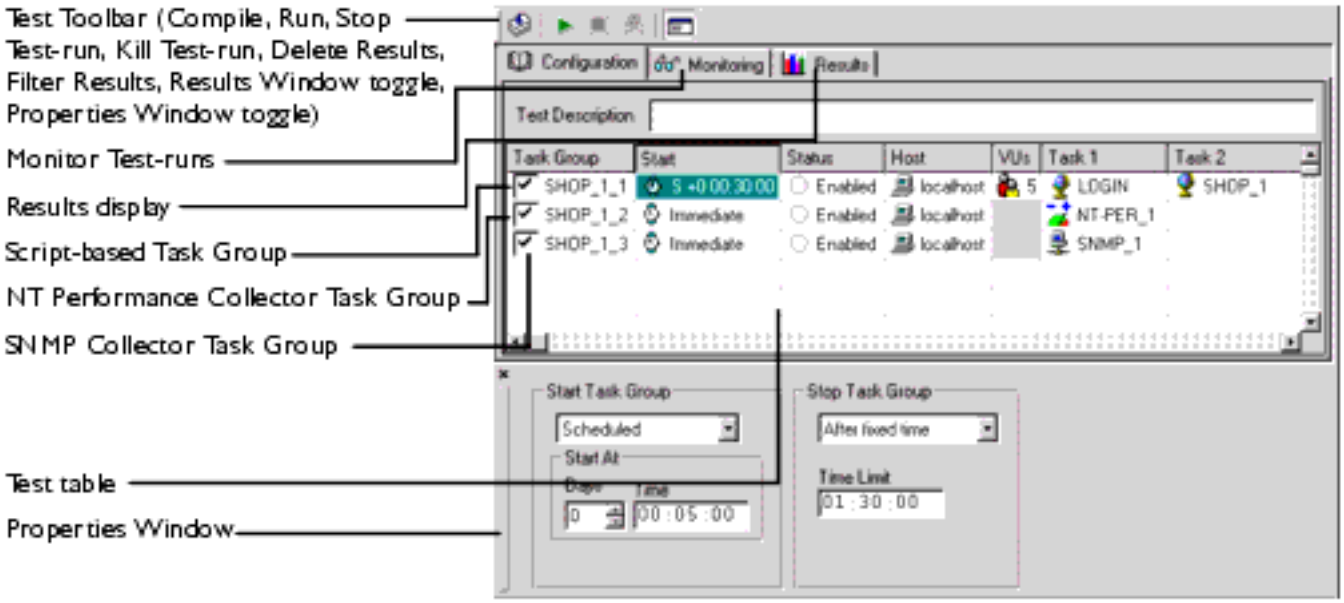
-  **Configuration**: This is the default view when you open a Test and the workspace used to develop a Test. Use it in combination with the Repository Window to select and add Scripts and Collectors. It displays the Test table which has categorized column headings that indicate where Script and Collector Tasks can be placed and the Task Group settings that apply to the contents of the Test.
  Select a Task Group cell to view and edit the associated settings using the Properties Window displayed below the Test table.
-  **Monitoring**: Use this tab to monitor the progress of a Test-run. Select the display options you want from the Monitoring Window,

including a Summary and data for individual Task Groups.

- **Results**: Use this tab to view the results collected during Test-runs in graph and table format. Use the Results Window to select the display options available which are dependent on the type of Test you are running.

Test Pane Features

The Configuration tab view of the Test Pane is displayed below:



The Test Development Process

The Test development process typically includes the following procedures:

- Create a Test
- Add a Collector to a Test
- Define Task Group settings, these include:
- Edit the Task Group Schedule Settings
- Select the Host used to Run a Task Group
- Save and Close a Test

Create a Test

1. In Commander select **File > New Test > Tests**.

   Or: In the Repository Window, right-click **Tests**, and select **New Test**

> Tests.

The Test appears in the Repository Window with a small crossed red circle over the Test icon , indicating that the file has no content. As soon as you open the Test and add a Collector or a Script, the icon changes to reflect this and appears .

2. In the Repository Window give the Test a name, in this example **PRODUCTION_MONITOR**, then press **Return**.

   Note: The new Test is saved automatically in the Repository when you switch to a different function or exit from Commander.

Add a Collector to a Test

1. In the Repository Window, locate your new Test and double-click **PRODUCTION_MONITOR**, to open it with the **Configuration** tab of the Test Pane displayed.

   The Configuration tab displays the Test table where you can add Tasks, and the Properties Window which is used to apply Task Group settings.

2. Double-click **Collectors**, in the Repository Window to open the folder and display the contents.
3. In the Repository Window, click on the **NT_PERFORMANCE** Collector, then drag it across to the Test table and drop it in a new row under the **Task 1** column.

   The selected Collector **NT_PERFORMANCE**, appears in the first empty row under the first Task column, in a new Task Group.

   Note: Collector-based Task Groups can only contain a single Task.

- The Task Group name is taken from the Test name and includes a number suffix which is automatically incremented for each new Task Group added to the Test.

  Use the Task Group cell to disable and enable a Task Group.

  Note: Uniquely naming Task Groups enables you to select and monitor them during a Test-run from the Monitoring tab.

- The Start column indicates the Task Group Schedule settings. For more information on Task Group scheduling, see Edit the Task Group Schedule Settings.
- The Status column displays Task Group activity and status information.
- The Host column defaults to **localhost**, which refers to the computer

you are currently working on.
The Host you select here determines which computer or device will run the Task Group during a Test-run. For more information on selecting a Host, see Select the Host used to Run a Task Group.

4. Repeat steps 1-4 this time add the **SNMP** Collector you created.

   Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

Edit the Task Group Schedule Settings

1. Open a Test with the  **Configuration** tab of the Test Pane displayed.
2. Click on the **Start** cell in a Task Group.

   The current Schedule settings are displayed in the Properties Window at the bottom of the Configuration tab. The default setting is for an **Immediate** start when the Test is run.

3. In the Start Task Group section of the Properties Window, click  to the right of the selection box and choose a Start option:

- **Scheduled**: The Task Group starts after the number of days and at the time you set.
  Enter a time period using the Days and Time text boxes.
- **Immediately**: The Task Group starts when the Test is started.
- **Delayed**: The Task Group starts after the time period you set, (days: hours: minutes: seconds), relative to when the Test was started.
  Enter a time period using the Days and Time text boxes.

  Note: Your settings are displayed in the Test table.

4. In the Stop Task Group section of the Properties Window, click  to the right of the text box and select a stop option:

- **Manually**: The Task Group will run continuously until you click the **Stop** button in the Status column of the Task Group that activates during a Test run.
- **After fixed time**: The Task Group is stopped after a fixed period of time.
  Enter a time period using the Time Limit text box.
- **On Completion**: The Script-based Task Group is stopped after completing a number of iterations.
  Enter the number of Task Group iterations in the Iterations text box.

  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

Note: During a Test-run Schedule settings cannot be edited, but they can be overridden manually using the **Start** and **Stop** buttons in the Status column of each Task Group.

Select the Host used to Run a Task Group

Note: Collector-based Task Groups include a Collector which defines a set of data to be recorded from one or more target Hosts during a Test-run. The Host you select in the Test table determines which computer or device will run the Task Group during a Test-run, not the Host from which data is collected.

1. Make sure the **PRODUCTION_MONITOR** Test is open with the **Configuration** tab of the Test Pane displayed.
2. Click on the **Host** cell , in a Task Group.

   The current Host settings are displayed in the Properties Window at the bottom of the Configuration tab. The default setting is **localhost**, which refers to the computer you are currently using.
3. In the Host Name text box of the Properties Window, enter the name of the Host to run the Task Group. Your settings are then displayed in the Test table.

   Note: The Host you select must have the OpenSTA Name Server installed and running with the Repository Host setting pointing to the local Host.

   Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

Save and Close a Test

- The Test related work you perform is automatically saved in the Repository and the Test is closed when you switch to a different function or exit Commander.

After you have created a Test, by adding Collectors, and applying the Task Group settings required, you are ready to run it against the target production system. Move on to the next chapter for details on how to do this.

Next Section: Running a Test

Contents

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Creating a Collector

A Collector is a user-defined collection of queries which determine the data collection carried out from one or more Host computers or devices during a Test-run. Include them in your Tests to gather the data you need to assess the performance of your target system. Create Collectors and incorporate them into your Tests, then run the Test to generate the results data required.

Collectors give you the flexibility to collect and monitor a wide range of performance data at user defined intervals during a Test-run. A Collector can contain a single data collection query and be used to target a single Host. Or alternatively, they can contain multiple queries and target multiple Hosts.

OpenSTA supplies two Modules which facilitate the creation of Collectors:

- NT Performance Module
- SNMP Module

NT Performance Collectors

NT Performance Collectors are used to monitor and collect performance data from your computer or other networked Hosts running Windows NT or Windows 2000 during a Test-run. Creating and running NT Performance Collectors as part of a Test enables you to collect comprehensive data to help you assess the performance of systems under test.

Use NT Performance Collectors to collect performance data during a Test-run from performance objects such as Processor, Memory, Cache, Thread and Process on the Hosts you specify in the data collection queries. Each performance object has an associated set of performance counters that provide information about device usage, queue lengths, delays, and information used to measure throughput and internal congestion.

NT Performance Collectors can be used to monitor Host performance according to the data collection queries defined in the Collector during a Test-run. Performance counters can be displayed graphically by selecting the Task Group that contains the Collector from the Monitoring Window in the Monitoring tab of the Test Pane.

The results recorded using a Collector can be monitored then viewed after the Test-run is complete. Select a Test and open up the Custom NT Performance graph from the Results tab of the Test Pane to display your results.

Note: If you are using an NT Performance Collector to target a Web server that is running Microsoft IIS (Internet Information Server), you can monitor and collect performance from it by selecting the Web Service object from the Performance Object text box when you set up a new query.

In this example the procedure below takes you through adding two data collection queries targeting the same Host.

Create an NT Performance Collector

1. In Commander, select **File > New Collector > NT Performance**.

   Or: In the Repository Window, right-click **Collectors**, and select **New Collector > NT Performance**.

   The Collector appears in the Repository Window with a small crossed red circle over the Collector icon , indicating that the Collector has no content.

   Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon changes to reflect this .

2. Give the new Collector a name within the Repository Window, in this example **NT_PERFORMANCE**, then press **Return**.

3. In the Repository Window, double-click the new Collector **NT_PERFORMANCE**, to open the Collector Pane in the Commander Main Window, where you can setup your data collection queries.

   The Edit Query dialog box opens automatically when you open a new Collector , or double-click on a row of an open Collector. Use this dialog box to add NT Performance data collection queries.

4. In the Name text box enter a unique title for the data collection query, in this case **Processor**.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring tab of the Test Pane. You can select and monitor queries during a Test-run.

Query names also appear in the Custom SNMP graph with the associated results data. Use the Results Window in the Results tab of the Test Pane to display them.

5. Click the **Browse Queries** button to open the Browse Performance Counters dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit Query dialog box.

6. In the Browse Performance Counters dialog box, select the Host you want to collect data from. You can select to either:

- **Use local computer counters**: Collects data from the computer you are currently using.
- Or, **Select counters from computer**: Enables you to specify a networked computer. Type **** then the name of the computer, or click  and select a computer from the list.

7. In the Performance object selection box select a performance object, in this example **Processor**. Click  , to the right of the selection box and choose an entry from the drop down list.
8. In the Performance counters selection box choose a performance counter, for example **% Processor Time**.

   Note: Click **Explain** to view a description of the currently selected Performance counter.

9. In the Instances text box select an instance of the performance counter you have chosen.
10. Click **OK** to confirm your choices and return to the Edit Query dialog box.
11. In the Interval text box enter a time period in seconds, for example **5**, to control the frequency of data collection, or use  , to set a value.
12. Leave the Delta Value column check box unchecked to record the raw data value, or check the box to record the Delta value.

    Note: Delta value records the difference between the data collected at each interval.

13. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

14. Use  , in the toolbar to add an additional query then repeat steps 4-13. This time select the **Memory** Performance object and **Page Faults/sec** Performance counter.

Note: Double-click on a query to edit it. Select a query then click  , in the toolbar to delete it.

Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

SNMP Collectors

SNMP Collectors (Simple Network Management Protocol) are used to collect SNMP data from Host computers or other devices running an SNMP agent or proxy SNMP agent during a Test-run. Creating then running SNMP Collectors as part of a Test enables you to collect results data to help you assess the performance of production systems under test.

SNMP is a standard protocol developed to manage nodes on an IP network. It can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs and printers. That is, any device capable of running an SNMP management process, known as an SNMP agent. All computers and many peripheral devices meet this requirement, which means you can create and include SNMP Collectors in a Test to collect data from most components used in target production systems.

SNMP data can be displayed graphically during a Test-run and as offline data. Select a Test and open the Custom SNMP graph from the Results tab of the Test Pane after a Test-run, to display your results.

In this example the procedure below takes you through adding two data collection queries targeting the same Host.

Create an SNMP Collector

1. In Commander, select **File > New Collector > SNMP**.

   Or: In the Repository Window, right-click  **Collectors**, and select **New Collector > SNMP**.

   The Collector appears in the Repository Window with a small crossed red circle over the icon  , indicating that the Collector has no content.

   Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon changes to reflect this  .

2. Give the new Collector a name within the Repository Window, in this example **SNMP**, then press **Return**.
3. In the Repository Window, double-click the new Collector  **SNMP**, to open the Collector Pane in the Commander Main Window, where you can

setup your data collection queries.

The Edit Query dialog box opens automatically when you open a new Collector  , or double-click on a row of an open Collector. Use this dialog box to add SNMP data collection queries.

4. In the Name text box enter a unique title for the data collection query, in this example **IP In**.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring tab of the Test Pane. You can select query names to monitor the progress of the Test-run.

   Query names also appear in the Custom SNMP graph with the associated results data. Use the Results Window in the Results tab of the Test Pane to display them.

5. In the SNMP Server text box enter the Host name or the IP address you want to collect data from.

   Tip: You can run the SNMP Server Scan by clicking  in the toolbar, to identify all networked SNMP Servers currently running an SNMP agent, then click  , to the right of the SNMP Server text box to display the list and select an SNMP server.

6. In the Port text box enter the port number used by the target SNMP Server.

   Note: Port 161 is the default port number that an SNMP agent runs from.

7. Click the **Browse Queries** button to open the Select Query dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit Query dialog box.

8. In the Select Query dialog box, click  to the right of the Category selection box and choose a category from the drop down list, in this example **ip**.
9. In the Query selection box below, pick a query associated with the category you have chosen, in this example **ipInReceives.0**.

   Note: The Current Value of the query must contain a numeric counter in order to generate data to populate the results graphs.

10. Click **Select** to confirm your choices and return to the Edit Query dialog box.

    The selected query, **public ip.ipInReceives.0**, records the total number of input datagrams received from interfaces, including those received in

error.

11. In the Edit Query dialog box use the Interval text box to enter a time period in seconds, for example **5**, to control the frequency of data collection, or use  , to set a value.
12. Leave the Delta Value column check box unchecked to record the raw data value, or check the box to record the Delta value.

    Note: Delta value records the difference between the data collected at each interval.

13. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

14. Use  , in the toolbar to add an additional query then repeat steps 4-13. This time select the **ip** category and the **ipOutRequests.0** query.

    Note: **public ipOutRequests.0** records the total number of IP datagrams which local IP user - protocols (including ICMP) supplied to IP in requests for transmission. This counter does not include any datagrams counted in ipForwDatagrams.

    Note: Double-click on a query to edit it. Select a query then click  , in the toolbar to delete it.
    Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

It is also possible to create new SNMP data collection categories which can then be selected during the Collector creation process. Follow the procedure below for details.

Create New SNMP Data Collection Categories

Use this option to create new SNMP data collection categories which you can select when you define a new query in the Select Query dialog box.

1. In the Repository Window double-click on an SNMP Collector to open the Collector Pane in the Main Window.
2. Click  , in the toolbar.
3. In the Category Definition dialog box, click in the Name text box and enter the title of the new data collection category.

   Note: The new category can be chosen from the Category text box of the Select Query dialog box when you are defining a query.

4. In the Walk Point text box enter the query definition.

   Note: The Walk Point you define can be selected in the Query text box of the Edit Query dialog box and the Category text box of the Select Query dialog box when you are choosing a query.

5. Click **Apply** to make the new category available for selection. Click **Close** to cancel.

   Note: Edit the Walk Point of a category by clicking  , to the right of the Name text box to display and select a category, then enter the new query definition.

After you have created the Collector, the next step is to add it to a Test. Move on to the next section for details on how to create a Test.

Next Section: Creating a Test

Contents

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

OpenSTA SCL Reference

License and Contents Info

License and Contents Info

Table of Contents

OpenSTA SCL Reference

Alphabetic Index

Alphabetic Index

OpenSTA SCL Reference

Continuation Lines

Continuation Lines

It is not always possible to fit a complete SCL command onto one line due to SCL's **maximum line length**, so SCL allows you to use *continuation lines*.

An SCL command may be split over two or more lines by ending split lines with a *continuation character*. This shows more of the command is given on the next line. The *continuation character* may be an ampersand or hyphen (**&** or **-**). To avoid possible confusion with the minus character, it is recommended that the hyphen not be used for line continuation. It is good practice to separate the *continuation character* character from the preceding characters on the line by at least one space.

The only characters that may follow a line continuation character on a line are **whitespace** and **comments**. Here is an example:

```
GET URL "http://osta.lan/" &
        ON Conid &
        HEADER Sub-Heads & ! default headers
        WITHOUT "Referer"
```

OpenSTA SCL Reference

Maximum Values in Scripts

Maximum Values in Scripts

The SCL compiler, the OpenSTA run-time environment, and host system resources impose many limitations on Scripts. Exceeding these limits will be flagged by the compiler or the run-time environment. The maximum values (number, size, level, etc.) allowed in SCL source files are given in the following table:

attribute	maximum value
source line length (characters)	132
number of **labels** (per subroutine/main code)	255
number of **timers**	1020
number of **variables**	8000
number of **global variables**	8000
number of **subroutines**	255
number of **parameters** passed between Scripts	8
number of **external data files referenced** in Script	256
number of **external data files open** concurrently	10
character variable size (bytes)	65535
character constant/literal size (bytes)	65535
space available for Script values (Kbytes)	128
nesting level of **array expressions**	10
nesting level of **conditional compilations**	10
nesting level of **IF/DO** commands	100
nesting level of **subroutines**	10

OpenSTA SCL Reference

Labels

Labels

Labels are names used to identify SCL commands during some **flow control** commands. A label starts at the beginning of a line and consists of the name followed by a colon. For example here is the definition of the label **LABELEX** on a log statement:

```
LABELEX: LOG "Just branched to LABELEX"
```

A label name must be a valid **OpenSTA Dataname** and is not **case sensitive**.

Labels are local to the module within which they are defined; this means that labels defined within a **subroutine** may not be reference in other sections of the code.

OpenSTA SCL Reference

Flow Control Commands

Flow Control Commands

Flow control commands determine how sections of a Script are **processed**, and in what sequence. These are the available commands:

- **ENTRY Command**
- **CALL SCRIPT Command**
- **IF Command**
- **DO Command**
- **GOTO Command**
- **ON ERROR Command**
- **CANCEL ON Command**
- **WAIT Command**
- **SUBROUTINE Command**
- **RETURN Command**
- **END SUBROUTINE Command**
- **CALL Command**
- **DETACH Command**
- **EXIT Command**

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

Script Processing

Script Processing

When a Script is executed, the first command in the **CODE section** of the Script is selected and executed.

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

A Script terminates when the end of the Script is reached, when an **EXIT**, or **DETACH** command is executed, or when an error is detected and **error trapping** is not enabled for the Script.

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

CODE Section

CODE Section

The mandatory **CODE** section of the SCL source file contains all the commands that define the **Script's behavior**.

A valid Script file will always contain one **CODE** section. It is introduced by the mandatory **CODE** command and continues to the end of the file. Within this section there are two types of processed text:

- **Labels** - used during some **flow control** commands.
- **Commands** - the definition of a Scripts behavior when run.

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

OpenSTA SCL Reference

ENTRY Command

ENTRY Command

If an optional **ENTRY** command is provided it must be the first real command in the **Code section** of the Script. There can only be one **ENTRY** command per Script and it identifies which variables are to receive values passed as parameters when the script is called.

Variables declared in the **ENTRY** command should not have an associated value **list or range** or **file**. Values passed in this way will be overwritten when Script initialization takes place following the **ENTRY** command.

Command Definition:

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

parameter

A **character** or **integer** declared in the **Definitions section** of the Script. Up to 8 parameters may be declared in the **ENTRY** command. There must be the same number of parameters in this list as are passed to the Script, and the data types of corresponding parameters must match.

Example:

```
ENTRY [Date-Param, Time-Param, Code-Param]
```

Related:

- **CALL SCRIPT**

Variable Values

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

A value *list* has the following format:

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

The values must be of the same data type as the variable; that is **integer values** for integer variables, and **character values** for character variables. They may also be **constants** which have previously been defined.

A *range* provides a shorthand method for defining a list of adjacent integer values and has the following format:

```
(start-value - end-value)
```

If the *start-value* is less than the *end-value*, the variable is increased by 1 on each execution of the **NEXT** command, until the end value is reached. If the *start-value* is greater than the *end-value*, the variable is decreased by 1 on each execution of the **NEXT** command, until the *end-value* is reached.

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

In the following list of example variable definitions including values, the definitions of **A** and **B** are equivalent:

```
Integer A (4,3,2,1,0,-1)
Integer B (4 - -1)
Integer C (100 - 999)
Integer D (100,200,300,400)
Character*10 Lang ("en", 'fr', 'de', "es")
Character Control ("~<CR>", "~<LF>", "^Z", "^X", "^U")
```

Note: Lists may contain only individual values and not ranges.

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

Note: Variables which have been declared as an array may not have an associated value list or range.

Note: Referencing a variable with a value list or range without first calling **NEXT** will result in a runtime error. The compiler cannot spot this type of problem.

Related:

- **GENERATE Command**
- **NEXT Command**
- **RESET Command**

<<< **prev page** **^^^** **section start** **>>>** **next page**

GENERATE Command

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

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

Command Definition:

```
GENERATE variable
```

variable

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

Example:

```
GENERATE Part-Number
```

Related:

- **Variable Values**

Variable Random Options

The random options are only valid for variables which have an associated **set of values**. Only one of the two possible options may be used in a declaration; these option syntaxes are:

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

These syntax elements are described below:

RANDOM

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

This option is particularly useful when load testing a system.

This is the default if no random option is specified.

REPEATABLE {RANDOM}

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

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

SEED = *n*

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

OpenSTA SCL Reference

Variable Scope Options

Variable Scope Options

The variable scope options define how widely accessible the variable is; a variable may only have one scope value, these values are mutually exclusive. The variable scope option values are:

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

These option values are described below:

LOCAL

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

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

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

SCRIPT

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

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

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

THREAD

Thread variables are accessible from any Script executed by the thread (Virtual User) which declares an instance of them.

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

Thread variables cannot have associated **value lists or ranges**.

GLOBAL

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

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

Global variables cannot have associated **value lists or ranges**.

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

EXIT Command

This command causes execution of the current Script to terminate immediately, no further Script commands will be executed.

An optional status value can be returned when the Script in question has been called from another Script. This is achieved by using the *status* variable to place a value into the return status variable specified on the call to this Script. If no status is specified, but the caller is expecting one, then the status returned will be that returned by the last Script which exited with a status. This allows a status to be retrieved from a deeply nested Script where no explicit status returning has been employed. This feature is broken - see bug#573365 for updates.

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

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

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

Command Definition:

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

status

An integer variable or value to be returned as the status from this Script to the caller. The status will be returned into the integer variable specified on the **CALL SCRIPT** commands **RETURNING** clause. This feature is broken - see bug# 573365 for updates.

Example:

```
EXIT Return-Status, KEEPALIVE
```

Related:

- **Broken & Useless SCL Features**

OpenSTA SCL Reference

CALL SCRIPT Command

CALL SCRIPT Command

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

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

For Scripts, a maximum of 8 parameters may be passed from the calling Script to the called Script. An omitted parameter is specified by two consecutive commas (**,,**). The calling Script must pass exactly the same number of parameters to the called Script as the called Script has defined in its **ENTRY** statement (accounting for any omitted parameters). In addition, the data types of each of the parameters must match. Failure to comply with these conditions will result in a Script error being generated.

The values of the parameters are passed from the caller into the variables defined within the **ENTRY** statement of the called Script. Any modifications to the values of the variables should be copied back to the caller on return from the called Script - except this feature is broken (see bug# 573365).

An optional status value can be returned from the called Script by using the **RETURNING** clause to specify the integer variable which is to hold the return status value. The called Script passes a value back using the **EXIT** command. This feature is also broken, check bug# 573365 for any updates.

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

The **ON ERROR GOTO *err-label*** clause can be specified to define a label to which control should be transferred in the event of an error while attempting to call the Script.

Command Definition:

```
CALL SCRIPT name {[parameter{, parameter ...}]}
         {RETURNING status} {ON ERROR GOTO err-label}
```

name

A character variable or quoted character string defining the name of the Script to be called. The name must be a valid **OpenSTA Dataname**.

parameter

A character or integer variable, quoted character string, integer value or file ID to be passed to the called Script. A maximum of 8 parameters may be passed between Scripts.

status

An integer variable to receive the returned status from the called Script. If no status is returned from the called Script, then this variable will contain the last status returned from any called Script. This feature is broken, check bug# 573365 for any updates.

err-label

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

Examples:

```
CALL SCRIPT Script-Name
CALL SCRIPT "TEST"
CALL SCRIPT "CALC_TAX" [Cost, Rate, Tax]
CALL SCRIPT "GET_RESP" RETURNING Response &
         ON ERROR GOTO ERR_LABEL
```

Related:

- **Broken & Useless SCL Features**

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

OpenSTA SCL Reference

ON ERROR Command

ON ERROR Command

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

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

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

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

When **ON ERROR** checking is established, it can be disabled by using the **CANCEL** command.

Command Definition:

```
ON ERROR GOTO label
```

label

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

Example:

```
ON ERROR GOTO SCRIPT_ERR
```

SUBROUTINE Command

This command defines the start of a discrete section of code which is bounded by the **SUBROUTINE** and **END SUBROUTINE** commands. Subroutines are *callable* from the main code section or from within other subroutines, this has the effect of temporarily causing the **script processing** to run the code in the subroutine.

Subroutines are called from other code with a command of the format **CALL *name***. They return control to the code from where they were called by use of the **RETURN** command. A maximum of 255 subroutines may be defined within a Script.

Subroutines share the same variable definitions as the main code but have their own **labels**. A label may not be referenced outside the main module or outside the subroutine in which it occurs. This has the effect of disabling branching in and out of subroutines, and means that each subroutine has its own 255 label **symbol table**.

Command Definition:

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

name

The name of the subroutine. This must be a valid **OpenSTA Dataname**, and must be unique within the Script.

parameter

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

Examples:

```
SUBROUTINE CREATE_FULL_NAME [First-Name, Middle-Initial, Last-Name]
        SET Full-Name = First-Name + "  " + Middle-Initial + " " + Last-
Name
        RETURN
END SUBROUTINE
```

OpenSTA SCL Reference

END SUBROUTINE Command

END SUBROUTINE Command

This command marks the end of a **SUBROUTINE**. The only statements that may follow an **END SUBROUTINE** command are a **comment**, or a **SUBROUTINE** command to start a new subroutine

If command execution reaches the **END SUBROUTINE** command in a subroutine then execution continues at the command after the subroutine **CALL**. This may be achieved at any point within the subroutine code by using the **RETURN** command.

Command Definition:

```
END SUBROUTINE
```

Comments

Scripts may incorporate author comments; these are usually bits of text to make the code purpose easier to understand and are ignored by the compiler. Comment text is identified by the comment character (**!**), and terminated by the end of the line. Comments can occur either on lines by themselves or embedded in statements or commands. For example comments here are shown in bold:

```
! Get next page.
SET Conid = Conid + 1 ! Update connection ID
GET URL "http://osta.lan/" & ! Get this URL
        ON Conid & ! use this TCP connection
        HEADER Sub-Heads & ! default headers
        WITHOUT "Referer" ! no referer
```

OpenSTA SCL Reference

General Rules

General Rules

SCL scripts are ASCII text files containing a source code syntax as defined in this documentation. Compilation of the source code will fail if the syntax is not completely correct. Scripts must compile successfully to be usable within the OpenSTA toolset to run Tests.

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

- **Comments and Whitespace** - used only to improve Script legibility and aid maintenance.
- **SCL Commands and Labels** - these are the language keywords that are defined in the following sections of this document.
- **Arguments to SCL commands** - the **variable names**, **integer values** or **character strings**.

Commands, Labels and other SCL keywords are not **Case Sensitive**.

The layout of an SCL Script file is split into three distinct sections and the SCL Commands that can be used in each of these sections are defined within a distinct part of this document describing that section. These sections are:

- **ENVIRONMENT** - This first section is mandatory and is where the global attributes of the Script are defined. It is introduced by the **ENVIRONMENT** command, and continues until a **DEFINITIONS** or **CODE** command is encountered.
- **DEFINITIONS** -This second section is optional and is where the **variables** for the Script are defined. It starts with the **DEFINITIONS** command, and continues until the **CODE** command.
- **CODE** - This last section is mandatory and contains the main Script commands. The start of this section is marked by the **CODE** command; it continues until the end of the Script file.

In this Section:

- **Including Text from Other Source Files**

<<< **prev page** **^^^** **section start** **>>>** **next page**

Whitespace

Just like **comments**, whitespace characters (spaces and tabs) can be used to aid code readibility and maintenance with no effect on the resulting compiled code. Whitespace characters may be incorporated within SCL commands (outside **quoted character strings**) to align keywords and generally aid legibility. Commands may also be broken up onto multiple lines using the **line continuation character** to aid readibility.

The Script compiler allows some ASCII control characters, the *non-printing* characters with an ASCII value in the range 0x00 to 0x20, or 0x81 to 0x8F inclusive, to appear at the start or end of a line. These characters are ignored allowing characters such as the form-feed to be used to aid legibility. Otherwise, if any ASCII character that has a value in the ranges 0x00 to 0x20, 0x7F to 0xA0, or the value 0xFF appears anywhere in the Script the compiler will generate a compilation error.

Character Strings

SCL character strings provide the ability of specifying text data of an undetermined length - that is the actual storage contains information of how long the specified text is. When literally specified in the code a character string is referred to as a *quoted character string*, these may be used as parameters to commands or functions, or assigned to **character variables** or **constants**. Some commands can take parameters as *character expressions* allowing limited character string manipulation *on the fly*.

Any ASCII character can be contained within a character string but special syntax may be required to **represent certain characters**, (ie. *non-printing* ones). For example, the character string **~<07>** always represents a single character (namely the character with a value of 0x07), not five characters.

Quoted Character Strings are sequences of ASCII characters surrounded by quote characters. The quote character used can be the single (**'**) or double (**"**) quote but the string is only terminated by a quote character matching the starting quote character.

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

```
LOG "This string of text is continued " &
        'over two lines.'
LOG "This message contains a variable ", Var1, &
        ' and is continued on this line ', &
        Var2, ' and this line' &
        ' and this line'
```

Some commands allow *Character Expressions* to be used as parameters. This means a single parameter can be represented with character **variables** and/or *quoted strings*, these are combined using **+** and **-** operators like their use in the **SET** command. The example below illustrates one possible use of this technique:

```
GET URI "http://" + Host-Name + "/~~dansut/ HTTP/1.0" ON 1 &
        HEADER Sub-Head &
        ,WITH {"Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg",
&
                "Host: " + Host-Name, &
                "Connection: Keep-Alive"}
```

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

OpenSTA SCL Reference

CHARACTER Command

Table of Contents | Alphabetical Index | Documentation Index | Frequently Asked Questions | OpenSTA Home Page

CHARACTER Command

This command defines a character variable capable of holding everything that can be contained in a **quoted character string**. SCL character variables can be defined to contain up to 65535 characters.

Arrays of character variables can be defined, with a maximum of 3 dimensions.

Command Definition:

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

n

An unsigned **integer value** in the range 1-65535, representing the size of the variable in characters. If unspecified the default is 1.

name

The **name** of the variable. This must be a valid **OpenSTA Dataname**.

dimensions

The dimensions of the **array** to be allocated for this variable. Up to three dimensions can be specified, each separated by comma. If *dimensions* are specified, *values* may not be.

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

values

A list of character **values** to be associated with the variable. If *values* are specified, *dimensions* may not be.

options

A list of **variable options**.

Note: Only one of *dimensions* or *values* may be specified for any one definition.

Examples:

```
CHARACTER:15 Dept
CHARACTER:20 Names ('TOM','JOHN','DICK'), SCRIPT
CHARACTER:9 Months [12]
CHARACTER*20 Staff-By-Dept [8,101:150]
```

Related:

- **Variables**

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

Variable Arrays

Variable Arrays

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

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

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

For example both of the variable declarations below declare an array of character variables each with 12 elements. The elements in the array are both numbered 1 to 12.

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

Compare these with the following example which also declares an array of 12 elements, but the array elements are numbered from 0 to 11.

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

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

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

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

OpenSTA SCL Reference

INTEGER Command

INTEGER Command

This command defines a variable which can hold an **integer value**. Integers variables have a possible value range of -2147483648 to +2147483647.

Arrays of integer variables can be defined, with a maximum of three dimensions.

Command Definition:

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

name

The **name** of the variable. This must be a valid **OpenSTA Dataname**.

dimensions

The dimensions of the **array** to be allocated for this variable. Up to 3 dimensions can be specified, each separated by comma. If *dimensions* are specified, *values* may not be.

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

values

A list or range of integer **values** to be associated with the variable. If *values* are specified, *dimensions* may not be.

options

A list of **variable options**.

Examples:

```
INTEGER Loop-Count
INTEGER Fred (1-99), SCRIPT
INTEGER Values [50:100,20]
```

Related:

- **Variables**

OpenSTA SCL Reference

Integer Values

Integer Values

Integer values are used as parameters to many commands and can be assigned to **integer variables** and **constants**. They are always specified in decimal and may take value in the range -2147483648 to +2147483647.

OpenSTAs SCL has no floating point capabilities.

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

OpenSTA SCL Reference

CONSTANT Command

CONSTANT Command

This command defines a **variable** like object which has a static value in a Script. If you want to define a **named** value for use in your Script(s) then using a **CONSTANT** is much more efficient than using a **variable**.

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

Constants can be used in any situation where a literal of the same type (i.e. **character** or **integer**) can be used, for example in a **value list**.

Command Definition:

```
CONSTANT name = value
```

name

The **name** of the constant. This must be a valid **OpenSTA Dataname**.

value

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

Examples:

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

OpenSTA SCL Reference

Variables

Variables

The SCL language makes use of *variables* for temporary storage of data at script run time. Variables are given **unique names** that fulfill the requirements of an **OpenSTA Dataname**.

All variables accessed by a Script must be predefined in the **DEFINITIONS** section of the Script. If a variable that has not been defined is used in the **CODE** section, then a compilation error will be given.

Part of the definition of a variable assigns that variable a *type* of data it may contain. OpenSTAs SCL has just 2 basic types of data: **integers** and **characters**. All **integer** variables are initially set to zero, and **character** variables are empty, unless otherwise specified in their definition.

Both of the above data types may be specified as **arrays** allowing a single variable name to contain many indexed items of that type. A variable can also be pre-assigned **values** such that those values can be cycled through or **randomly** chosen - the values can be specified in the defintions section or be pulled from a **file**.

A variable may be accessed by more than just the specific instance of the Script it is running in, this is controlled by specifying a variable **scope** in its definition.

There is a further type of variable available that has a specific use, this is a **TIMER** and is purely used for making named timings of specific areas of code.

To allow the use of named bits of data, that don't need to change throughout the script run, there exist **CONSTANT** definitions. These are similar to variables but may not have their contents altered and can therefore be handled more efficiently.

OpenSTA SCL Reference

Symbols

Symbols

An SCL symbol is a user-defined name for an item such as a **variable**, **timer**, or **label**. During compilation, the compiler maintains tables of all the defined symbols it has encountered, so that it may resolve references to them.

There are separate symbol tables for **variables**, **timers**, and **labels**. All symbols within an individual symbol table must be unique. Symbols can be duplicated across separate symbol tables though. This allows the same symbol name to be used for a label, variable, or timer.

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

TIMER Command

The **TIMER** command declares the name of a special variable used only for reporting the time taken to run an area of code. These timers are be used in conjunction with the **START TIMER** and **END TIMER** statements in the **CODE section** of the Script.

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

Command Definition:

```
TIMER name
```

name

The **name** of the timer. This must be a valid **OpenSTA Dataname**.

Examples:

```
TIMER Log-In
TIMER Check-Out
```

Related:

- **LOAD Commands**
- **Logging and Results Commands**

START TIMER Command

This command starts the named stopwatch timer and writes a *start timer* record to the statistics log.

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

A stopwatch timer is stopped by the **END TIMER** command.

Command Definition:

```
START TIMER name
```

name

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

OpenSTA SCL Reference

END TIMER Command

END TIMER Command

This command ends the named stopwatch timer and writes an *end timer* record to the statistics log, even if the timer is already ended.

A stopwatch timer is started by the **START TIMER** command.

Command Definition:

```
END TIMER name
```

name

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

OpenSTA SCL Reference

OpenSTA Datanames

OpenSTA Datanames

The names of many items within Scripts must be defined as an *OpenSTA Dataname*. For example **labels**, **variable** names, and **subroutine** names must all be *OpenSTA Datanames*.

An OpenSTA Dataname has between 1 and 16 characters. These characters may only be alphanumerics, underscores, or hyphens. The first character must be alphabetic, no spaces, no double underscores or hyphens, and no trailing underscore or hyphen.

OpenSTA Datanames are **not case sensitive**.

Note: The compiler currently only seems to care about trailing underscores or hyphens in labels. This behavior may change so our advice is to avoid using trailing underscores or hyphens everywhere.

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

Case Sensitivity

In general, SCL code compilation is not case sensitive. That is to say that the character ranges A-Z and a-z are viewed as being equivalent by the compiler. This does not apply to any **quoted character strings** within the code where case is maintained. So for example:

```
ENVIRONMENT
    DESCRIPTION "A very short example script"
    MODE HTTP
DEFINITIONS
    CHARACTER*64 Test-Str
CODE
    SET Test-Str = "Nothing useful"
    EXIT
```

Is the way code is being presented in this document. But the following code is veiwed the same by the compiler:

```
Environment
    description "A very short example script"
    Mode HTTP
Definitions
    CHARACTER*64 TEST-str
Code
    set test-STR = "Nothing useful"
    ExiT
```

The contents of character strings are case sensitive by default. Commands that do character comparison can usually be made to be case independent by specifying a **CASE BLIND** modifier.

OpenSTA SCL Reference

Character Representation

Character Representation

Within **quoted character strings** and **character variables** SCL supports the use of any character with an ASCII value in the range 0x00 to 0xFF inclusive. However, direct specification of these characters is not always possible, for two reasons:

- Characters with values in the ranges 0x00 to 0x20 and 0x7F to 0xA0, and the value 0xFF, are *non-printing* characters.
- Two characters are special to SCL because of the notation used to represent the above - the **command character**(**~**) and the **control character** (**^**).

The **command** and **control** characters can be represented by placing the command character in front of them (**~~** and **~^**). Other characters can be represented using the command and control characters as follows:

Using ASCII Mnemonic Notation

The commonly used characters have a mnemonic notation giving an easily identifiable representation of control characters. These use the ASCII mnemonic of the control character in question. The following notations are available:

mnemonic	meaning	hex.	control
~<BEL>	Bell	0x07	^G
~<BS>	Backspace	0x08	^H
~<CR>	Carriage return	0x0D	^M
~<DEL>	Delete	0x7F	
~<ESC>	Escape	0x1B	^[
~<FF>	Form feed	0x0C	^L
~<HT>	Horizontal tab	0x09	^I
~<LF>	Line feed	0x0A	^J
~<VT>	Vertical tab	0x0B	^K

Using Hexadecimal Notation

All characters can be represented by hexadecimal ASCII code, using this syntax:

```
~<hh>
```

Where *hh* is the hexadecimal ASCII code (0x*hh*) of the required character.

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

Note: As an exception to this notation, **~<FF>** represents the form feed character as covered above, so to represent the character with the ASCII code of 255 (0xFF) you must use **~<0FF>**.

Using Control Character Notation

All 7-bit control characters, i.e. characters with ASCII codes in the range 0x00 to 0x1F inclusive, may be represented using a *control character syntax*. This syntax has the following format:

```
^c
```

Where *c* is the control character specifier. The control character specifier is an ASCII graphics character with an ASCII code in the range 0x40 (ASCII @) to 0x5F (ASCII _). The compiler will apply the bottom 6 bits only, to generate an ASCII code in the range 0x00 to 0x1F.

For example, the ASCII bell character (ASCII code 0x07), is represented by **^G**.

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

Command Character

Command Character

The command character tilda (**~**) is used to introduce a command within a **SET** and therefore cannot be used to represent the command character itself. The command character needs to be duplicated to represent itself, like this:

```
~~
```

Note: the tilda character can also be supplied using **Hex ASCII Code ~<7E>**.

Related:

- **Broken & Useless SCL Features**

SET Command

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

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

operator	meaning
+	addition
-	subtraction
*	for multiplication
/	division
%	modulo
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR

The value resulting from a division operation will be an integer, any remainder will be discarded. The modulo calculation is the opposite of this operation, the variable will be set to the value of the remainder. For example:

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

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

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

When using arithmetic expressions integer overflows will cause run-time Script errors.

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

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

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

variable.

The **ON ERROR GOTO *err-label*** clause can be specified to define a label to which control should be transferred in the event of an error. Example errors are an **~EXTRACT** function is specified with an invalid offset, or an attempt is made to divide by zero.

Command Definition:

```
SET variable = operand
        {operator operand {operator operand ...}}
        {ON ERROR GOTO err-label}
```

variable

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

operand

For **SET** commands with a **CHARACTER** *variable*, the operand may be a **CHARACTER variable**, **quoted character string**, or character function reference (see below). For **SET** commands with an **INTEGER** *variable*, the operand may be an integer function reference (see below), literal, or variable.

operator

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

The following character and integer funtions are available for use as operands within a **SET** command:

- **~LENGTH Integer Function**
- **~LOCATE Integer Function**
- **~EXTRACT Character Function**
- **~RIGHTSTR Character Function**
- **~LEFTSTR Character Function**
- **~LTRIM Character Function**
- **~RTRIM Character Function**

err-label

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

Examples:

```
SET String1 = String2 - "ERROR"
SET String1 = String2 + String3 + String4
SET String1 = String2 - '"END MARKER"' &
        ON ERROR GOTO ERROR_REPORT
```

EXTRACT Character Function

This a Character Function and can only be referenced within a **SET** command. It returns the portion of the *string* identified by the specified *offset* and *length*.

Function Definition:

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

Returns:

The **character** substring extracted from the source *string*.

offset

An **integer** variable or **value** defining the *offset* in the *string* of the first character that is to be extracted. The first character of the source *string* is at offset zero. If the *offset* is not within the bounds of the source *string* then a message will be written to the audit log, indicating that a bad parameter value has been specified. Script execution will then be aborted, or the specified action taken if error trapping is enabled via the **ON ERROR** command.

length

An **integer** variable or **value** defining the number of characters to extract to form the returned string. If *length* specified causes the specified area to overrun the end of the *string*, only the characters up to the end of the *string* will be returned.

string

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

Example:

```
SET Name-Code = ~EXTRACT(0, 4, Name) + Running-No
```

OpenSTA SCL Reference

CANCEL ON Command

CANCEL ON Command

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

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

Command Definition:

```
CANCEL ON {ERROR}
```

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

WAIT Command

WAIT Command

This command suspends the Script execution for the specified time interval. The unit of the specified time interval is either seconds or milliseconds depending upon the value of the **ENVIRONMENT** statement **WAIT UNIT**.

Command Definition:

```
WAIT period
```

period

An integer variable or value defining the number of seconds for which Script execution is to be suspended. The valid range is 0-2147483647.

Examples:

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

Related:

- **WAIT UNIT Statement**
- **WAIT FOR SEMAPHORE Command**

OpenSTA SCL Reference

ENVIRONMENT Section

ENVIRONMENT Section

The purpose of the **ENVIRONMENT** section is to define attributes specific to the Script.

The **ENVIRONMENT** section of an SCL Script is introduced by the mandatory **ENVIRONMENT** command. Only **comments** and **whitespace** may come before the **ENVIRONMENT** command in a Script source file. The **ENVIRONMENT** section is closed by the start of a **DEFINITIONS**(optional) or **CODE** section.

The following commands are the only ones valid in the **ENVIRONMENT** section:

- **DESCRIPTION Command**
- **MODE HTTP Command**
- **WAIT UNIT Command**

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

DEFINITIONS Section

DEFINITIONS Section

The **DEFINITIONS** section of an SCL Script is the part of the source code where the Scripts **variables**, **timers**, and **constants** used by the Script are defined. The section is optional and is specified by a **DEFINITIONS** command immediately after the **ENVIRONMENT** section. The **DEFINITIONS** section is ended by the start of the **CODE** section.

Only one **DEFINITIONS** section may appear in a Script; if it is present, it must follow the **ENVIRONMENT** section and precede the **CODE** section.

These Commands can appear in this section Section:

- **INTEGER Command**
- **CHARACTER Command**
- **CONSTANT Command**
- **TIMER Command**

And some of these Commands can take these modifiers:

- **Variable Arrays**
- **Variable Values**
- **Variable Options**
  - **Variable Scope Options**
  - **Variable Random Options**
  - **Variable File Option**

Related:

- **Variables**

OpenSTA SCL Reference

Broken and Useless SCL Features

Broken and Useless SCL Features

This section is here to document those features of SCL which are broken or just useless within OpenSTA. SCL is a relatively old language, it is much older than OpenSTA itself and with this it has inherited some *baggage*. Although OpenSTAs SCL is considerably different than the earlier SCLs (in implementation and features) it still shares a reasonable amount with its ancestors; some of these things don't make a whole lot of sense in OpenSTA or were just never finished properly.

Within the **ENVIRONMENT** section there are a couple of little gotchas that technically work but don't seem to make any sense. The **DESCRIPTION** is mandatory, that is, if you delete it then your Script will not compile. You can provide an empty description string but you can't delete the command, this doesn't make much sense. The **MODE HTTP** could probably also be defaulted as OpenSTA scripts are always used for HTTP.

The **command** and **control** characters played much more of a part in the previous generations of SCL. It was possible to change what these characters were and all commands were actually introduced by the command character. This fact means that the documentation and use of these characters is perhaps a little *weird*.

The ability to pass data back from **called sub-scripts** is not present. Any variables used as parameters are not updated on return and the variable provided in the **RETURNING** clause seems to be cleared rather than updated.

Totally Broken Features:

- **Conditional Compilation**
- **File Handling**

OpenSTA SCL Reference

Variable Options

Variable Options

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

The following sections describe the types of variable option clause:

- **Variable Scope Options**
- **Variable Random Options**
- **Variable File Option**

<<< prev page | ^^^ section start | >>> next page

Variable File Option

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

```
, FILE = filename
```

where *filename* is a quoted character string which defines the name of the ASCII text file, excluding the path name and file extension. The file must reside in the data directory of the Repository and have the file extension **.FVR**.

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

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

```
-1
0
1
2
```

A file for a **character** variable could contain the values:

```
Cat
Dog
27
Dinosaur
```

Note: SCL **character representation** is not recognized within the file variable files - the file should contain raw ASCII characters only.

Values are retrieved from the file associated with a variable using the **NEXT** command. This command retrieves the next sequential value from the file. When the **NEXT** command is first executed, it will retrieve the first value from the file. If the variable is set to the last value in the file when the **NEXT** command is executed, the variable will be reset to the first value in the file. You can also reset the variable explicitly, by using the **RESET** command.

The file option is not valid for variables which:

- are defined as an **array**
- have an associated **value list**

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

NEXT Command

NEXT Command

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

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

The first **NEXT** command to be executed after the **RESET** retrieves the first value in the set.

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

Command Definition:

```
NEXT variable
```

variable

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

Example:

```
NEXT Emp-Name
```

Related:

- **Variable Values**
- **Variable File Option**

OpenSTA SCL Reference

RESET Command

RESET Command

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

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

Command Definition:

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

variable

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

value

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

Examples:

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

Related:

- **Variable Values**
- **Variable File Option**

OpenSTA SCL Reference

Variable Manipulation Commands

Variable Manipulation Commands

The commands described in this section are used to manipulate Script **variables**:

- **SET Command**
  - **~LENGTH Integer Function**
  - **~LOCATE Integer Function**
  - **~EXTRACT Character Function**
  - **~LEFTSTR Character Function**
  - **~RIGHTSTR Character Function**
  - **~LTRIM Character Function**
  - **~RTRIM Character Function**
- **CONVERT Command**
- **FORMAT Command**
- **LOAD Command**
- **GENERATE Command**
- **NEXT Command**
- **RESET Command**

Related:

- **Variables**

OpenSTA SCL Reference

LENGTH Integer Function

LENGTH Integer Function

This command is an Integer Function and can only be referenced within a **SET** command. It returns the length of the source *string*.

Function Definition:

```
~LENGTH(string)
```

Returns:

The number of characters within the *string* parameter.

string

A **quoted character string** or **CHARACTER** variable whose contents will have the number of characters counted and returned.

Example:

```
SET Str-Length = ~LENGTH(Name)
```

OpenSTA SCL Reference

LOCATE Integer Function

LOCATE Integer Function

This command is an Integer Function and can only be referenced within a **SET** command. It returns an **integer value**, corresponding to the *offset* of the specified substring in the source string.

By default, the matching is **case sensitive**. The strings "OpenSTA" and "opensta", for example, would not produce a match, because the case of the characters is not the same. This can be overridden by specifying the **CASE_BLIND** clause.

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

Function Definition:

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

Returns:

The **integer** offset of the *substring* in the source *string*. The offset of the first character in the *string* is zero. If the *substring* is not found a value of -1 is returned.

substring

The **character** variable or **quoted character string** defining the string to be scanned for in the *string*.

string

The **character** variable or **quoted character string** to be scanned for the specified *substring*.

Example:

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

OpenSTA SCL Reference

LEFTSTR Character Function

LEFTSTR Character Function

This is a Character Function and can only be referenced within a **SET** command. It returns a character string containing the first *length* characters of the source *string*.

Function Definition:

```
~LEFTSTR(length, string)
```

Returns:

A **character** expression.

length

An integer **variable** or **value** defining the number of characters to extract from the beginning of *string* to form the returned value.

string

A character **variable** or **value** to have the return string extracted from the beginning of.

Example:

```
SET New-Str = ~LEFTSTR(Length, Name)
```

OpenSTA SCL Reference

RIGHTSTR Character Function

RIGHTSTR Character Function

This is a Character Function and can only be referenced within a **SET** command. It returns a character string containing the last *length* characters of the *string*.

Function Definition:

```
~RIGHTSTR(length, string)
```

Returns:

A **character** expression.

length

An integer **variable** or **value** defining the number of characters to extract from the end of *string* to form the returned value.

string

The character **variable** or **value** to extract the last *length* characters of.

Example:

```
SET New-Str = ~RIGHTSTR(Length, Name)
```

OpenSTA SCL Reference

LTRIM Character Function

LTRIM Character Function

This is a Character function and can only be referenced within a **SET** command. It returns a character string after removing any leading blanks from the given *string*.

Function Definition:

```
~LTRIM(string)
```

Returns:

A **character** expression after removing leading blanks from the *string*.

string

A character **variable** or **value** to strip the leading blanks from.

Example:

```
SET New-Str = ~LTRIM(Name)
```

RTRIM Character Function

This is a Character Function and can only be referenced within a **SET** command. It returns a character string after truncating all trailing blanks.

Function Definition:

```
~RTRIM(string)
```

Returns:

A **character** expression after truncating all trailing blanks.

string

A character **variable** or **value** to strip the trailing blanks from.

Example:

```
SET New-Str = ~RTRIM(Name)
```

OpenSTA SCL Reference

CONVERT Command

CONVERT Command

This command allows the value in an **integer variable** to be converted to a **string**, or vice versa. The default radix for the conversion is 10, but this may be overridden by including a **RADIX** clause in the command.

For integer-to-character conversions, format options may be specified. These options can cause the string to be left or right justified within the output buffer, or to have leading zeros or spaces, or cause the conversion to be signed or unsigned.

The default options are **SIGNED** and **LEFT JUSTIFY**. If **RIGHT JUSTIFY** is in operation, the default filling is **LEADING ZEROS**.

If the output buffer is too small to hold the output string, it will be filled with asterisk (*****) characters.

For character-to-integer conversions, leading and trailing spaces are removed form the ASCII string before the conversion. Specification of a non-numeric string, or of a string which is converted to a numeric outside the range of an **integer value**, will cause a message to be logged to the audit file indicating an invalid character string to convert. The thread will be aborted.

The **ON ERROR GOTO *err-label*** clause can be specified to define a **label** to which control should be transferred in the event of an error.

Command Definition:

```
CONVERT variable1 TO variable2
        {,[SIGNED|UNSIGNED]} {,LEADING [ZEROS|SPACES]}
        {,[LEFT|RIGHT] JUSTIFY} {,RADIX=radix}
        {,ON ERROR GOTO err-label}
```

variable1

A variable containing the value to be converted.

variable2

A variable into which the converted *variable1* is to be placed.

radix

An integer **variable** or **value** in the range 2 to 36.

err-label

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

Examples:

```
CONVERT Number To String
```

```
CONVERT Number To Employee-Code, RIGHT JUSTIFY
CONVERT Ascii-Code To Numeric-Code
CONVERT Ascii-Code To Hex-Code, RADIX=16, &
        ON ERROR GOTO CONV_ERROR
```

Note: This command is known to have multiple issues - see bug# 460324 for current status.

<<< **prev page** **^^^** **section start** **>>>** **next page**

FORMAT Command

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

In all translations, the command requires three elements:

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

The following types of translation are supported:

- **%U** - Translate each alphabetic character in the input string into its uppercase equivalent. Both source and target variables must be character variables. The source string if necessary is truncated to fit the target variable.
- **%L** - Translate each alphabetic character in the input string into its lowercase equivalent. Both source and target variables must be character variables. The source string if necessary is truncated to fit the target variable.
- **%D** - Convert a character string date value into numeric format (representing the number of days since the Smithsonian base date of 17-Nov-1858). The target variable must be an integer variable, and the source variable a character string containing a valid date; this can be either in the default style for the platform on which the Script is running or in the fixed format **DD-MMM-CCYY** (where **CC** is optional).

  This format identifier may also be used to convert a numeric date value (representing the number of days since the Smithsonian base date of 17-Nov-1858) into a character string in the fixed format **DD-MMM-CCYY**. The source must be an **integer variable** and the target **character variable**, which will be truncated if necessary.
- **%T** - Convert a character string time value into a numeric format (representing the number of 10 milli-second 'ticks' since midnight). The target variable must be an integer variable, and the source variable a character string containing a valid time; this can be either in the default style for the platform on which the Script is running or in the form **HH:MM:SS.MMM** (where **.MMM** is optional).

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

Command Definition:

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

target-variable

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

format-string

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

variable

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

err-label

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

Examples:

```
FORMAT(Date-Str, &
        "The date is %D today, and the time is %T", &
        Int-Date, Int-Time), ON ERROR GOTO THE_END
FORMAT(Date-Value, "%D", Char-Date), ON ERROR GOTO FRM_ERR
FORMAT(UC-String, "Name in uppercase is %U", LC-String)
```

LOAD Commands

The LOAD commands provide a means to get data from the currently running test into the variables of your script.

Command Definition:

```
LOAD test-data INTO variable
```

test-data

This parameter provides the item that is to be retrieved and placed into the *variable*. Its possible values and their meaning are covered in the table below:

test-data	***variable* type**	**meaning**
ACTIVE_USERS	integer	The number of threads which are currently active on the current Test Manager.
DATE	integer	The number of days since the system base date.
	character	The system date in the system default format (for example, "DD-MMM-CCYY").
NODENAME	character	The node(host) name of the machine running the test.
SCRIPT	character	The name of the Script the command is executed as part of.
TEST	character	The name of the Test the command is executed as part of.
THREAD	character	The name of the Thread(virtual user) the command is executed as part of.
TIME	integer	The number of 10ms *ticks* since midnight.
	character	The system time in the host systems default format.
TIMER *name*	integer	The number of 10ms *ticks* of the timer specified by *name*. The current value of a timer is calculated by taking the time for the latest **END TIMER** and subtracting from it the time for the preceding **START TIMER**. If no **START**/**END TIMER** commands have been executed for the specified timer by the current thread an error will occur. This will either abort Script execution, or take the specified action if error trapping is enabled via the **ON ERROR** command.

variable

The variable which will contain the result of the **LOAD** query after the call. The possible type of the *variable* and its resulting content are determined by the value of the *test-data* parameter.

Related:

- **LOAD RESPONSE_INFO BODY Command**
- **LOAD RESPONSE_INFO HEADER Command**

<<< **prev page** **^^^** **section start** **>>>** **next page**

*Proud to be **Open**, prouder to be **Free***

OpenSTA SCL Reference, version 2.0.6
Copyright & License Info on ToC page

Last Updated: 2005-05-11

LOAD RESPONSE_INFO BODY Command

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

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

If the character data requested is too big to fit into the target variable, it will be truncated. For a response message body containing an HTML document, the optional **WITH *identifier*** clause can be used to specify part of the structured document using a **special syntax**.

The optional **RETURNING STATUS *load-status*** clause can be used to specify the integer variable to hold one of two values indicating whether the command succeeded or failed. When **RETURNING STATUS** is specified, any current **ON ERROR** action is disabled.

By default, if an error occurs, an error message is written in the audit log and the virtual user will continue. However, if **error trapping** is enabled, control will be transferred to the error-handling code.

Command Definition:

```
LOAD RESPONSE_INFO BODY ON conid INTO variable
        {,WITH identifier}
        {,RETURNING STATUS load-status}
```

conid

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

variable

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

identifier

A **CHARACTER** variable, quoted character string or expression identifying the data to be retrieved from the response message body. A full a definition of the identifier format is covered in the **Identifier** section.

load-status

The name of an **INTEGER** variable into which the status of the LOAD RESPONSE_INFO

execution is loaded. Failure returns a negative value.

Example:

```
LOAD RESPONSE_INFO BODY ON 1 INTO Post-Body
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

GET Command

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

The optional **PRIMARY** keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example: If a request returns a HTML page from a Web server this can be followed by requests for the images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or name using the **CERTIFICATE FILE** and **CERTIFICATE NAME** clauses.

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

The response code in the response message can be retrieved by using the optional **RETURNING CODE *http-code*** clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional **RETURNING STATUS *get-status*** clause can be used to specify the integer variable to hold a value indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with the OpenSTA toolset, which defines SCL integer constants for both the response code and response status values. When **RETURNING STATUS** is specified, the **ON ERROR** action is disabled.

The size of the response message can be retrieved by using the optional **RETURNING BODYSIZE *body-size*** clause to specify the integer variable to hold the message size. The variable is loaded when the response message is received from the server.

On failure, the HTTP GET request can be retried by using the optional **WITH RETRY *retry-number***.

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

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

Command Definition:

```
{PRIMARY} GET [ URI | URL ] uri-httpversion ON conid
         HEADER http-header
        {,WITH header-value}
        {,WITHOUT header-field}
        {,CERTIFICATE FILE cert-filename}
        {,CERTIFICATE NAME cert-name}
        {,RESPONSE TIMER timer-name}
        {,RETURNING STATUS get-status}
        {,RETURNING CODE http-code}
        {,RETURNING BODYSIZE body-size}
        {,WITH RETRY retry-number}
```

uri-httpversion

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

conid

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

http-header

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

header-value

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

header-field

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

cert-filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert-name

A character variable, quoted character string, character expression, containing a client certificate name.

timer-name

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

get-status

An integer variable into which the status of the SCL GET command is loaded when the request completes. Success returns zero.

http-code

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

body-size

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

retry-number

An integer variable containing the number of times the request should be retried.

Examples:

```
GET URL "http://osta.lan/~~dansut/test.html HTTP/1.0" ON Conid &
        HEADER Sub-Head &
        ,WITH ("Host: osta.lan", "Referer: http://osta.lan/")

GET URI "http://osta.lan/~~dansut/test.html HTTP/1.0" ON 2 &
        HEADER Sub-Head &
        ,WITH "Host: osta.lan" &
        ,WITHOUT "Referer Accept-Language"
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

MODE HTTP Command

MODE HTTP Command

This optional command defines the Script as an HTTP mode Script. This enables the **HTTP commands** enabling testing of Web servers.

Command Definition:

```
MODE HTTP
```

Note: **MODE HTTP is currently the only mode available.**

<<< prev page | **^^^** section start | **>>>** next page

OpenSTA SCL Reference

HTTP Commands

HTTP Commands

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

- **GET**
- **HEAD**
- **POST**
- **LOAD RESPONSE_INFO BODY**
- **Identifiers used in LOAD RESPONSE_INFO BODY**
- **LOAD RESPONSE_INFO HEADER**
- **CONNECT**
- **DISCONNECT**
- **SYNCHRONIZE REQUESTS**
- **BUILD AUTHENTICATION BLOB**

CONNECT Command

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

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

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

The optional **RETURNING STATUS *connect-status*** clause can be used to specify the integer variable to hold a value indicating whether the **CONNECT** succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer **constants** for the response status values. When **RETURNING STATUS** is specified, the **ON ERROR** action is disabled.

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

If a **GET**, **HEAD**, or **POST** uses a connection id that has not been CONNECTed then the TCP connect is done implicitally to the host and port specified in the URL. If you wish to use a proxy for scripted HTTP requests then you must use a CONNECT first to that proxy.

Command Definition:

```
CONNECT TO host ON conid
        {,RETURNING STATUS connect-status}
```

host

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

conid

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

connect-status

An integer variable into which the connect status is loaded.

Examples:

```
CONNECT TO "proxy.osta.lan:3128" ON 1
CONNECT TO Test-Host ON 2
CONNECT TO 'osta.lan' ON Conid
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

HEAD Command

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

The optional **PRIMARY** keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example:

A request pulling back an HTML page from a Web server can be followed by requests pulling back some GIF images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or by name using the **CERTIFICATE FILE** and **CERTIFICATE NAME** clauses.

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

The response code in the response message can be retrieved by using the optional **RETURNING CODE *response-code*** clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional **RETURNING STATUS *response-status*** clause can be used to specify the integer variable to hold a value indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values. When **RETURNING STATUS** is specified, the **ON ERROR** action is disabled.

On failure, the HTTP **HEAD** request can be retried by using the optional **WITH RETRY *retry-number***.

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

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

Command Definition:

```
{PRIMARY} HEAD [ URI | URL ] uri-httpversion ON conid
```

```
        HEADER http-header
        {,WITH header-value}
        {,WITHOUT header-field}
        {,CERTIFICATE FILE cert-filename}
        {,CERTIFICATE NAME cert-name}
        {,RESPONSE TIMER timer-name}
        {,RETURNING STATUS response-status}
        {,RETURNING CODE response-code}
        {,WITH RETRY retry-number}
```

uri-httpversion

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

conid

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

http-header

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

header-value

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

header-field

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

cert-filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert-name

A character variable, quoted character string, character expression, containing a client certificate name.

timer-name

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

response-status

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

response-code

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

retry-number

An integer variable containing the number of time the request should be retried.

Examples:

```
HEAD URL "http://osta.lan/~~dansut/test.html HTTP/1.0" ON Conid &
        HEADER Sub-Head &
        ,WITH ("Host: osta.lan", "Referer: http://osta.lan/")

HEAD URL "http://osta.lan/~~dansut/test.html HTTP/1.0" ON 2 &
        HEADER Sub-Head &
        ,WITH "Host: osta.lan" &
        ,WITHOUT "Referer Accept-Language"
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

POST Command

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

The optional **PRIMARY** keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example: If a request returns a HTML page from a Web server this can be followed by requests for the images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or by name using the **CERTIFICATE FILE** and **CERTIFICATE NAME** clauses.

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

The response code in the response message can be retrieved by using the optional **RETURNING CODE *response-code*** clause to specify the integer variable to hold the response code. The variable is loaded when the response message is received from the server. In addition, the optional **RETURNING STATUS *response-status*** clause can be used to specify the integer variable to hold a value indicating whether the request succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for both the response code and response status values. When **RETURNING STATUS** is specified, the **ON ERROR** action is disabled.

The size of the response message can be retieved by using the optional **RETURNING BODYSIZE *body-size*** clause to specify the integer variable to hold the message size. The variable is loaded when the response message is received from the server.

On failure, the HTTP POST request can be retried by using the optional **WITH RETRY *retry-number***.

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

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

Command Definition:

```
{PRIMARY} POST [URI|URL] uri-httpversion ON conid
        HEADER http-header
        {,{BINARY} BODY http-body}
        {,WITH header-value}
        {,WITHOUT header-field}
        {,CERTIFICATE FILE cert-filename}
        {,CERTIFICATE NAME cert-name}
        {,RESPONSE TIMER timer-name}
        {,RETURNING STATUS response-status}
        {,RETURNING CODE response-code}
        {,RETURNING BODYSIZE body-size}
        {,WITH RETRY retry-number}
```

uri-httpversion

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

conid

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

http-header

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

http-body

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

header-value

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

header-field

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

cert-filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert-name

A character variable, quoted character string, character expression, containing a client certificate name.

timer-name

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

response-status

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

response-code

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

body-size

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

retry-number

An integer variable containing the number of times the request should be retried.

Examples:

```
POST URL "http://osta.lan/~~dansut/test.php HTTP/1.0" ON Conid &
         HEADER Sub-Header &
         ,WITH ("Host: osta.lan", "Referer: http://osta.lan/")

POST URL "http://osta.lan/~~dansut/test.php HTTP/1.0" ON 2 &
         HEADER Post-Head &
         ,WITH ("Host: osta.lan", &
                 "Referer: http://osta.lan/~~dansut/test.php") &
         ,BODY "data=nonsense"
```

Identifiers used in LOAD RESPONSE_INFO BODY

The **LOAD RESPONSE_INFO BODY** command loads a character variable with all or part of the data from an HTTP response message body for a specified TCP connection. For a response body containing an HTML document, the **WITH** clause may be used to load a character variable with an element or part of an element from the document.

The **WITH** clause has the following format:

```
,WITH identifier
```

Note: *identifier* is a character variable, quoted character string, or character expression identifying the data to be retrieved from the HTML document in the response message body. The following sections describe the format of this identifier:

HTML Element Addressing

An element within an HTML document is identified by an element address string.

Format Definition:

```
tag(tagnum){/tag(tagnum)}:element-type:{attribute}(element-num)
```

tag

The HTML tag name.

tagnum

A number identifying the tag relative to its parent tag or the document root:

- 0 = First child tag
- 1 = Second child tag
- n = nth child tag

element-type

The HTML element type. This must be one of the following:

- ANONYMOUS ATTRIBUTE
- ATTRIBUTE
- COMMENT
- SCRIPT
- TEXT

attribute

For element-type ATTRIBUTE, specifies the name of the HTML attribute.

element-num

A number identifying the element. For element type ATTRIBUTE, the number identifies the attribute relative to its associated tag:

- 0 = First attribute
- 1 = Second attribute
- n = nth attribute

Examples:

```
HTML(0)/BODY(1)/TABLE(1)/TBODY(0)/TR(0)/TD(0):TEXT:(0)
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)
```

Note: There must be no whitespace between any of the components of an identifier.

Note: Identifiers are not validated at compile time.

Qualifying an HTML Element Address

A complete HTML element string may be retrieved from an HTML document using an identifier containing only an HTML element address. However, a substring may be selected from it using a variety of qualifiers. These qualifiers immediately follow the HTML element address and are described below.

Selecting a Substring by Position and Length

An HTML element substring may be selected using an identifier specifying the offset of the substring and its length.

Format Definition:

```
element-addr[offset, length]
```

element-addr

The HTML element address in the format described above.

offset

The offset of the first character of the substring from the start of the element string.

length

The number of characters in the substring.

Note: If the offset is invalid, an empty string is returned.

Note: If the length is zero, or is invalid, all characters from the start offset to the end of the element string are returned.

Example:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,5]
```

Selecting a Substring using Delimiters

An HTML element substring may be selected by specifying an identifier containing two string delimiters. The substring returned contains all the characters between the first occurrence of the first delimiter and the first occurrence of the second. The string will also include both delimiter strings.

Format Definition:

```
element-addr[delimiter1, delimiter2]
```

element-addr

The HTML element address in the format described above.

delimiter1

A string - enclosed in single quotes - identifying the characters at the beginning of the substring.

delimiter2

A string - enclosed in single quotes - identifying the characters at the end of the substring.

Note: If *delimiter1* cannot be found, an empty string is returned.

Note: If *delimiter2* cannot be found, all characters from and including *delimiter1* to the end of the

element string are returned.

Example:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['document.cookie=',';']
```

Selecting a Substring Using Position, Length and Delimiter String

The above two methods of substring selection can be combined, allowing an HTML element substring to be identified by a start string and a length or an offset and a termination string.

Format Definition:

```
element-addr[delimiter1, length]
   or
element-addr[offset, delimeter2]
```

element-addr

The HTML element address in the format described above.

delimiter1

A string - enclosed in single quotes - identifying the characters at the beginning of the substring.

length

The number of characters in the substring.

offset

The offset of the first character of the substring from the start of the element string.

delimiter2

A string - enclosed in single quotes - identifying the characters at the end of the substring.

Note: If *delimiter1* cannot be found, an empty string is returned.

Note: If the *offset* is invalid, an empty string is returned.

Note: If *delimiter2* cannot be found, all characters after, and including, *delimiter1* to the end of the element string are returned.

Note: If the *length* is zero, or is invalid, all characters from the specified *offset* to the end of the element string are returned.

Examples:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['cookie=',3]
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';']
```

Excluding Delimiters from Selection

With the syntax described above, any delimiter strings specified are included in the returned substring. Either or both delimiters may be excluded from the returned substring by inverting the square bracket nearest to the delimiter, i.e. using an opening square bracket in place of a closing square bracket and vice versa.

This method can also be used with offset parameters. Instead of identifying the offset of the first character of the substring to be selected, using this alternative syntax, the offset becomes the offset of the character immediately before the first character to be selected.

The following examples illustrate how a substring may be selected from the CONTENT attribute string of an HTML META tag.

This example selects the substring that starts at offset 3 from the beginning of the attribute string and that is terminated by the next semicolon (included).

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';']
```

This example selects the substring that starts at offset 2 from the beginning of the attribute string and that is terminated by the next semicolon (not included).

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';'[
```

This example selects the substring that starts at offset 3 from the beginning of the attribute string and that is terminated by the next semicolon (not included).

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';'[
```

Ignoring the Characters at the Beginning of an HTML Element

There are occasions when it is useful to use the above facilities starting from some point within the element string, rather than at the beginning of the string. This can be achieved by resetting the selection base. This can be done by specifying the selection base as an offset from the beginning of the element string, or by specifying a substring that identifies the characters at the beginning of the substring to be examined. The offset or substring is preceded by one of two operators **>** or **>=**:

format	meaning
>*offset*	The *offset* is that of the character immediately before the substring to be examined.
>*substring*	The *substring* identifies the characters at the end of the string to be ignored. The substring starts with the first character after the *substring*.
>=*offset*	The *offset* is that of the first character in the substring to be examined.
>=*substring*	The *substring* identifies the characters at the beginning of the substring to be examined.

Note: If the *offset* or *substring* cannot be found, an empty string is returned.

The following examples illustrate how the selection base is reset for a selection from the CONTENT attribute string of an HTML META tag.

In this example the selection base offset is set to the offset of the first character after the first occurrence of the string **// Cookie** in the element string. The selected substring starts with the character after **document.cookie=** and ends with the next semicolon (included).

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>'// Cookie','document.cookie=',';']
```

Same as above, except that the selection base offset is now the first character of **// Cookie**.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>='// Cookie','document.cookie=',';']
```

Same as above, except that selection base offset is now 50 characters from the start of the element string.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>=50,'document.cookie=',';']
```

Ignoring the Case of Characters

All string comparisons specified by **LOAD RESPONSE_INFO BODY** identifiers are by default case sensitive. The case of characters can be ignored in comparisons by prefixing the search string or delimiter string by **I**.

In the example below the selection base is reset by searching the element string for **// Cookie**; the case of characters is ignored in the search.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>I'// Cookie',I'document.
cookie=',';']
```

Specifying Quotes Within Identifiers

Quoted character strings within SCL are delimited, either by single quotes or by double quotes. Since the syntax of a **LOAD RESPONSE_INFO BODY** identifier includes single quotes, it is recommended that double quotes are used to delimit a quoted character string containing such an identifier.

A literal single quote character can be included within an identifier string by preceding it with a

backslash. For example, this selects a substring terminated by a single quote:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'\'']
```

A literal double quote character can be specified within an identifier string, using the SCL character command, **~<22>**. For example, this selects a substring terminated by a double quote:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'~<22>']
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

LOAD RESPONSE_INFO HEADER Command

LOAD RESPONSE_INFO HEADER Command

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

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

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

The **WITH** clause can be used to specify the names of a header field whose value is to be retrieved from the HTTP response message. If this clause is omitted, all the response message header fields are retrieved.

The optional **RETURNING STATUS** ***load_status*** clause can be used to specify the integer variable to hold one of two values indicating whether the command succeeded or failed. There is an SCL include file "response_codes.inc" supplied with OpenSTA, which defines SCL integer constants for response status values. When **RETURNING STATUS** is specified, the **ON ERROR** action is disabled.

Command Definition:

```
LOAD RESPONSE_INFO HEADER ON conid INTO variable
        {,WITH identifier}
        {,RETURNING STATUS load-status}
```

conid

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

variable

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

identifier

A character variable, quoted character string or character expression containing the name of the response message header field to be retrieved. If the header in question is "Set-Cookie" then the identifier syntax is further extended so that "Set-Cookie,*name*" can be specified. This will retrieve just the "*name*=*value*" part of the Set-Cookie header that matches the *name*. The *name* may also contain the wildcard character * and the first matching cookie name will be retrieved.

load-status

An **INTEGER** variable into which the status of the LOAD RESPONSE_INFO execution is

loaded.

Example:

```
LOAD RESPONSE_INFO HEADER ON 4 INTO Resp-Heads
```

<<< **^^^** **>>>**
prev page **section start** **next page**

OpenSTA SCL Reference

DISCONNECT Command

DISCONNECT Command

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

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

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

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

Command Definition:

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

conid

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

Examples:

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

OpenSTA SCL Reference

SYNCHRONIZE REQUESTS Command

SYNCHRONIZE REQUESTS Command

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

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

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

Command Definition:

```
[SYNCHRONIZE|SYNCHRONISE] REQUESTS
        {,WITH TIMEOUT period {,ON TIMEOUT GOTO tmo-label}}
```

period

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

tmo-label

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

Examples:

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

BUILD AUTHENTICATION BLOB Command

This command generates a character string containing user authentication data and loads it into a character variable. This variable may be used to supply a value to an **Authorization** HTTP request-header field in a **GET**, **HEAD**, or **POST** command. This command is only valid within a script that has been defined as **MODE HTTP**.

This command allows scripts to support Basic, NTLM, and Negotiate client authentication over HTTP. Negotiate client authentication applies to Windows 2000 clients only.

Basic authentication is the simplest **user:password** scheme, defined in RFC 2617. NTLM is Microsoft's NT Lan Manager, a security package available on all Windows platforms and used for authentication of Windows users.

The Negotiate security package was introduced in Windows 2000 and allows a client and server to negotiate the actual authentication protocol. OpenSTA supports Negotiate authentication only when NTLM is selected as the underlying package.

A character string for use in Basic authentication, may be generated by specifying **FOR BASIC** and supplying a username, password and, optionally, a domain name.

A character string for use in NTLM authentication, may be generated by specifying **FOR NTLM** and user authorization data in one of three forms:

- An explicit username, password, and domain name.
- Current user data.
- A value returned in a **WWW-Authenticate** HTTP response-header field.

The value from a **WWW-Authenticate** HTTP response-header field may be obtained using the **LOAD RESPONSE_INFO HEADER** command, like this:

```
LOAD RESPONSE-INFO HEADER ON 1 INTO Blob-Var, &
        WITH "WWW-Authenticate"
```

A character string for use with the Negotiate security package, may be generated by specifying **FOR NEGOTIATE** and user authorization data, as described above for NTLM authentication.

Command Definition:

```
BUILD AUTHENTICATION BLOB FOR BASIC
        FROM USER username PASSWORD password {DOMAIN domain}
        INTO variable
```

or

```
BUILD AUTHENTICATION BLOB FOR [NTLM | NEGOTIATE]
```

```
        FROM [CURRENT USER |
        USER username PASSWORD password DOMAIN domain |
        BLOB blob-variable]
        INTO variable
```

username

A character variable, quoted character string or character expression, containing a username.

password

A character variable, quoted character string or character expression, containing a password.

domain

A character variable, quoted character string or character expression, containing a domain name.

blob-variable

A character variable containing the value returned in a "WWW-Authenticate" HTTP response-header field.

variable

A character variable into which the authentication value is loaded.

Examples:

```
BUILD AUTHENTICATION BLOB FOR BASIC &
         FROM USER "Smith" PASSWORD "John" &
         INTO Auth-Val

BUILD AUTHENTICATION BLOB FOR NTLM &
         FROM USER "Smith" PASSWORD "John" DOMAIN "Ostadom" &
         INTO Auth-Val

BUILD AUTHENTICATION BLOB FOR NTLM &
         FROM CURRENT USER &
         INTO Auth-Val

BUILD AUTHENTICATION BLOB FOR NTLM &
         FROM BLOB Auth-Head &
         INTO Auth-Val
```

OpenSTA SCL Reference

Formal Test Case Commands

Formal Test Case Commands

Formal test case commands provide support for tracking the results of each test, so that it is possible to see easily how well the testing is going. The following commands provide support for these features:

- **START TEST-CASE Command**
- **PASS TEST-CASE Command**
- **FAIL TEST-CASE Command**
- **END TEST-CASE Command**

OpenSTA SCL Reference

START TEST-CASE Command

START TEST-CASE Command

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

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

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

Command Definition:

```
START TEST-CASE description
```

description

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

Examples:

```
START TEST-CASE "Checking for valid input rate"
        IF (Inp-Rate = 0) THEN
                FAIL TEST-CASE
        ENDIF
END TEST-CASE
```

OpenSTA SCL Reference

END TEST-CASE Command

END TEST-CASE Command

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

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

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

Command Definition:

```
END TEST-CASE
```

Example:

```
START TEST-CASE "Checking input rate"
        IF (Inp-Rate < Min-Rate) THEN
                FAIL TEST-CASE
        ENDIF
END TEST-CASE
```

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

REPORT Command

REPORT Command

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

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

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

Any nonprintable ASCII characters in character values are replaced with periods (**.**). Integer values are written as signed values, and use only as many characters as are necessary.

Command Definition:

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

value

The **quoted character string** or **variable** to be written to the report log.

Examples:

```
REPORT "Login succeeded after ", Attempt, ' Trys'
REPORT "This is a long log message ", &
         "that is continued in this string "
REPORT "Message containing a representation of the tilde char ~~"
REPORT "One way to log a 'single quoted section'" &
         'and "a double quoted one".'
```

TRACE Command

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

Command Definition:

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

value

The **quoted character string** or **variable** to be written to the trace log.

OpenSTA SCL Reference

NOTE Command

NOTE Command

This command associates a list of variables or quoted character strings with the current thread. The current value can be viewed in the Monitoring tab of the Active Test Pane in Commander.

Command Definition:

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

value

The **quoted character string** or **variable** to be written to the Monitoring Tab.

LOG Command

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

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

Any non-printable ASCII characters in character values are shown as periods (**.**) in the log. Integer values are written as signed values, using only as many characters as are necessary.

Command Definition:

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

value

The **quoted character string** or **variable** to be written to the audit log.

Examples:

```
LOG "Customer Name = ", Cust-Name, &
        ' Customer Code = ', Cust-Code
LOG "A long message ", &
        "that is continued in this string " &
        "and on this line"
LOG "One way to log a 'single quoted section'" &
        'and "a double quoted one".'
```

OpenSTA SCL Reference

Logging and Results Commands

Logging and Results Commands

These commands are all specifically to allow the Script to provide data about its progress and status to the person performing the testing. This data can be viewed as part of the results or monitored during the Test run. The following commands provide support for these features:

- **LOG Command**
- **NOTE Command**
- **TRACE Command**
- **REPORT Command**
- **HISTORY Command**
- **START TIMER Command**
- **END TIMER Command**

<<<
prev page

^^^
section start

>>>
next page

HISTORY Command

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

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

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

Command Definition:

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

value

The **quoted character string** or **variable** to be written to the history log.

OpenSTA SCL Reference

Code Section Commands

Code Section Commands

SCL provides a wide range of commands that control the behavior and **processing** of the Script.

A command is normally terminated by the end of the source line, but may be continued onto a subsequent line by using the **continuation character**.

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

This section describes the commands that can be included in the **CODE** section of a Script file:

- **Variable Manipulation Commands**
- **Flow Control Commands**
- **Logging and Results Commands**
- **Inter-Script Synchronization Commands**
- **HTTP Commands**
- **Formal Test Case Commands**

<<< prev page | ^^^ section start | >>> next page

OpenSTA SCL Reference

Inter-Script Synchronization Commands

Inter-Script Synchronization Commands

During an OpenSTA Test many Scripts may be running at the same time (concurrently), information can be shared between these Scripts using **variable scope**. These commands allow scripts to coordinate access to shared variables and concurrent Script actions:

- **ACQUIRE MUTEX Command**
- **RELEASE MUTEX Command**
- **SET SEMAPHORE Command**
- **CLEAR SEMAPHORE Command**
- **WAIT FOR SEMAPHORE Command**

<<< prev page | ^^^ section start | >>> next page

ACQUIRE MUTEX Command

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

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

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

Command Definition:

```
ACQUIRE {scope} MUTEX mutex-name
        {,WITH TIMEOUT period {,ON TIMEOUT GOTO tmo-label}}
        {,ON ERROR GOTO err-label}
```

scope

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

mutex-name

A character variable, or quoted character string, containing the name of the mutex which is to be acquired. *mutex-name* must be a valid **OpenSTA Dataname**.

period

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

tmo-label

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

err-label

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

Example:

```
ACQUIRE LOCAL MUTEX "USERMUT", ON ERROR GOTO USERMUT_ERR
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

RELEASE MUTEX Command

RELEASE MUTEX Command

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

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

Command Definition:

```
RELEASE {scope} MUTEX mutex-name
        {,ON ERROR GOTO err-label}
```

scope

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

mutex-name

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

err-label

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

Example:

```
RELEASE LOCAL MUTEX "USERMUT"
```

OpenSTA SCL Reference

SET SEMAPHORE Command

SET SEMAPHORE Command

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

The **ON ERROR GOTO *err-label*** clause can be specified to define a label to which control should be transferred in the event of an error.

Command Definition:

```
SET {scope} SEMAPHORE semaphore-name
        {,ON ERROR GOTO err-label}
```

scope

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

semaphore-name

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

err-label

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

Example:

```
SET LOCAL SEMAPHORE "USERSEM"
```

OpenSTA SCL Reference

CLEAR SEMAPHORE Command

CLEAR SEMAPHORE Command

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

The **ON ERROR GOTO *err-label*** clause can be specified to define a label to which control should be transferred in the event of an error.

Command Definition:

```
CLEAR {scope} SEMAPHORE semaphore-name
        {,ON ERROR GOTO err-label}
```

scope

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

semaphore-name

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

err-label

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

Example:

```
CLEAR LOCAL SEMAPHORE "USERSEM"
```

WAIT FOR SEMAPHORE Command

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

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

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

Command Definition:

```
WAIT {period} FOR {scope} SEMAPHORE semaphore-name
        {,ON TIMEOUT GOTO tmo-label}
        {,ON ERROR GOTO err-label}
```

period

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

scope

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

semaphore-name

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

tmo-label

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

err-label

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

Example:

```
WAIT 10 FOR SEMAPHORE "USERSEM"
```

Note: The **WAIT UNIT statement** does not effect this command - the *period* is **always** specified in seconds in this command.

Related:

- **WAIT Command**

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

OpenSTA SCL Reference

WAIT UNIT Command

WAIT UNIT Command

This optional command defines the unit of the wait period specified in **WAIT** commands within a Script. If this command is omitted, the default wait unit is seconds.

Command Definition:

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

Note: This does not apply to the wait period in the **WAIT FOR SEMAPHORE** command - this is always specified in seconds.

Related:

- **WAIT command**

<<< prev page | ^^^ section start | >>> next page

PASS TEST-CASE Command

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

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

This command is only valid within a test case section of a Script. It can be repeated as often as required within a test case.

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

Command Definition:

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

label

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

Example:

```
START TEST-CASE "Checking input rate"
        IF (Inp-Rate >= Min-Rate) THEN
                PASS TEST-CASE
        ELSE
                FAIL TEST-CASE
        ENDIF
END TEST-CASE
```

OpenSTA SCL Reference

FAIL TEST-CASE Command

FAIL TEST-CASE Command

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

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

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

Command Definition:

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

label

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

Example:

```
START TEST-CASE "Checking input rate"
        IF (Inp-Rate < Min-Rate) THEN
                FAIL TEST-CASE
        ELSEIF (Inp-Rate > Max-Rate) THEN
                FAIL TEST-CASE
        ENDIF
END TEST-CASE
```

OpenSTA SCL Reference

GOTO Command

GOTO Command

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

Conditional branches may be made using the **IF** command.

Command Definition:

```
GOTO label
```

label

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

IF Command

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

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

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

Command Definition:

```
IF condition GOTO label

IF condition THEN commands{s}
        {ELSEIF condition THEN command{s}}
        {ELSEIF condition THEN command{s}}
        {ELSE command{s}}
ENDIF
```

condition

A condition of the following format:

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

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

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

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

The binary operators are:

operator	meaning
=	operand1 equals operand2
< >	operand1 does not equal operand2
<	operand1 is less than operand2
< =	operand1 is less than or equal to operand2
>	operand1 is greater than operand2
> =	operand1 is greater than or equal to operand2
^	operand1 contains operand2

CONTAINS	operand1 contains operand2
< ^ >	operand1 does not contain operand2
NOT CONTAINS	operand1 does not contain operand2
NOT_CONTAINS	operand1 does not contain operand2

All conditions are evaluated from left to right.

label

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

command

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

Example:

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

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

DO Command

DO Command

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

Command Definition:

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

index

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

value1

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

value2

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

step

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

Examples:

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

DO Empno = Start, End, 10
         NEXT Name
         LOG 'Employee number: ', Empno, '; Name: ', Name
ENDDO
```

<<<
prev page

^^^
section start

>>>
next page

---

*Proud to be **Open**,*
*prouder to be **Free***

OpenSTA SCL Reference, version 2.0.6
Copyright & License Info on ToC page

Last Updated:
2005-05-11

OpenSTA SCL Reference

DESCRIPTION Command

DESCRIPTION Command

This mandatory command assigns a descriptive character string to a Script. This descriptive text is currently unused in OpenSTA.

Command Definition:

```
DESCRIPTION string
```

string

A **quoted character string** with a maximum length of 50 characters.

Examples:

```
DESCRIPTION 'Create Customer Details'
DESCRIPTION "Update Customer's Details"
DESCRIPTION "osta.lan exercise pages"
```

Note: **This is an unused mandatory command!** You can leave the string empty but don't delete the command or compilation will fail.

Control Character

The control character (**^**) is used in **control character notation** with the character following it in a **quoted character string**. It therefore cannot be used to represent the control character itself. To represent the control character the command character needs to be given immediately before it:

```
~^
```

Note: the control character could also be supplied using **Hex ASCII Code ~<5E>**.

Related:

- **Broken & Useless SCL Features**

OpenSTA SCL Reference

Conditional Compilation

Conditional Compilation

Previous versions of SCL provided commands that enable you to define the circumstances for the compilation of a section of code. Conditional sections of code were marked with 'variants', these were specifiable on the '-V' option on the SCL compiler command line. In OpenSTA, SCL compilation happens automatically at test run time and there is currently no way of specifying what variants to supply - this renders this feature useless in the current OpenSTA. The documentation below shows the syntax that will not cause errors at compile but is otherwise useless, just in case we choose to resurrect this feature.

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

Command Definition:

```
#condition variant
```

condition

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

condition	meaning
IFDEF	Compile next section if *variant* requested
IFNDEF	Compile next section if *variant* not requested
ELIF	Otherwise compile next section if *variant* requested
ELSE	Otherwise compile the next section
ENDIF	End of variant section

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

variant

An **OpenSTA Dataname** which identifies a section of code that is only compiled under certain conditions. The compiler processes this variant in conjunction with the **-V** option on the SCL command line.

Examples:

```
#IFDEF variant1
        log "Only compiled if -V=variant1 is specified"
#ELIF variant2
```

```
        log "Only compiled if -V=variant2 is specified"
#ELSE
        log "Only compiled if neither variant is specified"
#ENDIF
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

OpenSTA SCL Reference

File Handling Commands

File Handling Commands

File handling commands would help Scripts and external data files exchange data - except they don't work properly in the current version of OpenSTA. To discourage their use documentation for them has been removed from here.

See instead:

- **Variable File Option**

OpenSTA SCL Reference

Script Control Language Introduction

Script Control Language Introduction

The Script Control Language (SCL) is used to write OpenSTA Scripts. Scripts define the behaviour of virtual users, these are used in exercising a system under test using the OpenSTA toolset. More details on the use of OpenSTA can be found at **OpenSTA.org**.

SCL is a compiled programming language designed specifically for scripting the actions of virtual users during application testing. The language is much older than OpenSTA itself, although the version used in OpenSTA is not inteded to be compatible with any previous generation of the language. It does however have some **legacy issues** because of this history. The syntax of this simple language was clearly influenced by Fortran and Digital's DCL, but it shares few identical features with either.

This document is meant to provide a general overview and reference for the SCL Language used within OpenSTA. It is not intended to be a tutorial of SCL use within OpenSTA or any sort of user guide with details of when, how and where the specific language features are best used - this type of information is currently best found in the **OpenSTA FAQ** or by asking questions on the **OpenSTA Users Mailing List**.

The document is divided into the following sections:

- **Document Conventions** - how type and format is used to try to make this document easier to understand.
- **General Rules** - the basic features, restrictions, and structure that applies throughout the source code of SCL Script files.
- **The ENVIRONMENT Section** - the available commands and layout of the first section of every SCL Script.
- **The DEFINITIONS Section** - the available commands and layout of the optional second section to SCL Scripts. This section is where **variables** are defined.
- **The CODE Section** - the available commands and layout of the required last section of SCL Scripts. This section is where the language **steps and actions** are described.

A final section has been provided to cover some of the **features of SCL that are broken or useless**. You might find some workarounds and fixes to common problems here.

OpenSTA SCL Reference

Document Conventions

Document Conventions

To aid in reader understanding, this document uses specific formatting and typographical techniques to represent different types of data. This page describes those conventions below:

Typographical Conventions

bold - bold text is used to represent keywords and text that would be used within an SCL file within normal descriptive text.

italics - italicized text is used to represent: (1) certain concepts, or (2) parts within SCL that you need to replace with your own values.

UPPERCASE - SCL is **not case sensitive** but within this document SCL keywords are all used in full uppercase (capital) letter form to emphasize them within descriptive text. You may use character case for these keywords as you wish in your own scripts.

Command Definition Syntax

Each SCL command and statement has its format defined in this document using a simple syntax. Definitions will appear in this form:

```
COMMAND [OPTION1|OPTION2] param1
        {WITH param2}
```

A command's definition may be split over multiple lines, this is only done for legibility and splitting an actual command over multiple lines requires use of the **continuation character**.

Within the definition the SCL keywords are all listed in uppercase. SCL is **not case sensitive** in its keyword use though, the keywords used in actual Scripts can have any use of case.

The syntax **[OPTION1|OPTION2]** means that either **OPTION1** or **OPTION2** but not both should be given. Any number of options may be given surrounded by square brackets and seperated by the vertical bar symbol.

The syntax **{optional}** means that the items enclosed in the curly brackets are optional.

Items in *italics* are parts of the command that are replaced when used - what they can be replaced with is described following the definition.

In the definition Ellipsis (**...**) or **{s}** can be used in combintation with optional section brackets to define where parts of the command can be repeated.

If square brackets or other characters that may be used in the definition syntax are actually required as part of the command then they will be shown in **bold**.

Example Format

SCL usage examples will appear in this form:

```
COMMAND "Quoted string" ON Variable-Name &
        WITH Another-Var
```

Within all examples, the SCL keywords are listed in uppercase. SCL is **not case sensitive** in its keyword use though, the keywords used in actual Scripts can have any use of case. The **variable** names will all be given with words capitalized and seperated by dashes (-), in real use these are case independent as well.

Hexadecimal values

The integer values that ASCII characters are traditionally represented using hexadecimal (base 16), this document and SCL follow this convention. Within this document, descriptive text hexadecimal integers are preceded by the 2 characters *0x*, so decimal 255 would be 0xFF.

<<< **prev page** | **^^^** **section start** | **>>>** **next page**

OpenSTA SCL Reference

DETACH Command

DETACH Command

This command causes the current Thread (Virtual User) to exit. The playback exits from any Scripts or **subroutines** that have been called (including nested calls) until control returns to the primary Script. The Thread is then detached from the Test Executer.

Command Definition:

```
DETACH {THREAD}
```

OpenSTA SCL Reference

CALL Command

CALL Command

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

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

A maximum of 8 parameters may be passed from the calling code to the called subroutine. The parameters passed may be character or integer variables, literals, or quoted character strings. The calling code must pass exactly the same number of parameters to the called subroutine as the called subroutine has defined in its **SUBROUTINE** statement. The names of the variables in the call need not be the same as in the subroutine parameter list, but the data types of each of the parameters must match. Failure to comply with these conditions will result in a Script error being generated.

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

Command Definition:

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

subroutine

The name of the called subroutine. The name must be a valid **OpenSTA Dataname**.

parameter

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

Examples:

```
CALL DATE_CHECK
CALL CREATE_FULL_NAME[First-Name, Middle-Initial, Last-Name]
```

<<< **prev page** **^^^** **section start** **>>>** **next page**

*Proud to be **Open**, prouder to be **Free***

OpenSTA SCL Reference, version 2.0.6
Copyright & License Info on ToC page

Last Updated: 2005-05-11

OpenSTA SCL Reference

Including Text from Other Source Files

Including Text from Other Source Files

The **INCLUDE** command allows you to combine several source files into a single source file at compilation time. The included files may contain any valid SCL syntax that would make sense at the point of the **INCLUDE** statement. **INCLUDE**s may be nested up to a depth of 10, including the original file.

The **INCLUDE** command can appear at any point within a Script source file, the only caveat is that the *include file contents* must be valid in their expanded entirety at that point in the script. The normal usage of **INCLUDE** files is to make variables or **subroutines** available across mutliple Scripts whilst only defining them within a single file.

Command Definition:

```
INCLUDE filename
```

filename

A quoted character string which defines the name of the source file to be included. The file must be located in the **Scripts\Include** directory within the OpenSTA **Repository**.

Example:

```
INCLUDE 'globals.inc'
```

OpenSTA SCL Reference

RETURN Command

RETURN Command

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

Command Definition:

```
RETURN
```

<<< prev page | ^^^ section start | >>> next page

Contents

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Script Control Language Reference Guide

Version 1.0.1

Copyright

This document is published May, 2001.

Manual reference number: OS-SCL-10-301

A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q - R - S - T - U - V - W - X - Y - Z

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

LOAD TIMER Command

Description:

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

Format:

```
LOAD TIMER name INTO variable
```

Parameters:

name

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

variable

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

Example:

```
LOAD TIMER Transaction INTO Timval
```

See also:

Miscellaneous Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD TIME Command

Description:

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

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

Format:

```
LOAD TIME INTO variable
```

Parameter:

variable

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

Examples:

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

See also:

Miscellaneous Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD THREAD Command

Description:

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

Declare the character variable at 32 bytes long, using the **CHARACTER*32** command. 32 bytes should be long enough to handle most thread names.

The thread name will be truncated as required to fill the target variable if you do not declare a value large enough to cope with the thread names.

Format:

```
LOAD THREAD INTO variable
```

Parameter:

variable

A character variable into which the thread name is loaded.

Example:

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

See also:

Miscellaneous Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD TEST Command

Description:

This command loads the name of the test of which the script is a part, into a variable. The test name will be truncated as required to fit into the target variable. The maximum size of the string returned by this command is 64 characters.

Format:

```
LOAD TEST INTO variable
```

Parameter:

variable

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

Example:

```
LOAD TEST INTO Testname
```

See also:

Miscellaneous Commands

LOAD SCRIPT Command

Description:

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

Format:

```
LOAD SCRIPT INTO variable
```

Parameter:

variable

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

Example:

```
LOAD SCRIPT INTO Scriptname
```

See also:

Miscellaneous Commands

LOAD NODENAME Command

Description:

This command loads the current node name into a variable.

Format:

```
LOAD NODENAME INTO variable
```

Parameter:

variable

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

Example:

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

See also:

Miscellaneous Commands

LOAD DATE Command

Description:

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

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

Format:

```
LOAD DATE INTO variable
```

Parameter:

variable

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

Examples:

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

See also:

Miscellaneous Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD ACTIVE_USERS Command

Description:

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

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

Format:

```
LOAD ACTIVE_THREADS INTO variable
```

Parameter:

variable

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

Example:

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

See also:

Miscellaneous Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

DISCONNECT Command

Description:

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

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

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

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

Format:

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

Parameters:

conid

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

Examples:

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

```
DISCONNECT ALL
DISCONNECT ALL, WITH CANCEL
```

See also:

Miscellaneous Commands

CONNECT Command

Description:

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

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

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

Format:

```
CONNECT TO host ON conid
```

Parameters:

host

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

conid

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

Examples:

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

See also:

Miscellaneous Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Miscellaneous Commands

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

See also:

TRACE Command

Description:

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

Format:

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

Parameters:

value

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

Examples:

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

See also:

Diagnostic Commands

NOTE Command

Description:

This command associates a list of variables or quoted character strings with the current thread. The current value can be viewed in the Monitoring tab of the Active Test Pane in Commander.

Format:

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

Parameters:

value

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

Examples:

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

See also:

Diagnostic Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOG Command

Description:

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

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

Any nonprintable ASCII characters in character values are replaced with periods ("."). Integer values are written as signed values, using only as many characters as are necessary.

Format:

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

Parameters:

value

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

Examples:

```
LOG "Customer Name = ", Cust-Name, &
     ' Customer Code = ', Cust-Code
LOG "This is a long message " &
     "that is continued on this line " &
     "and this line"
LOG "This message contains a 'single quoted section'" &
```

```
        'and "a double one here".'
```

See also:

Diagnostic Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Diagnostic Commands

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

See also:

LOG Command

NOTE Command

START TIMER Command

Description:

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

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

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

Format:

```
START TIMER name
```

Parameter:

name

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

Example:

```
START TIMER Transaction
```

See also:

Statistical Data Logging Commands

END TIMER Command

Description:

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

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

Format:

```
END TIMER name
```

Parameter:

name

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

Example:

```
END TIMER Transaction
```

See also:

Statistical Data Logging Commands

Statistical Data Logging Commands

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

See also:

END TIMER Command

START TIMER Command

WAIT FOR SEMAPHORE Command

Description:

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

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

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

Format:

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

Parameters:

period

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

scope

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

semaphore-name

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

tmo_label

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

err_label

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

Example:

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

See also:

Input Stream Entry Commands

WAIT Command

Description:

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

Format:

```
WAIT period
```

Parameter:

period

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

Examples:

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

See also:

Input Stream Entry Commands

Input Stream Entry Commands

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

See also:

WAIT Command

WAIT FOR SEMAPHORE Command

SYNCHRONIZE REQUESTS Command

Description:

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

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

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

Format:

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

Parameters

period

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

tmo_label

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

Examples:

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

See also:

Synchronization Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

SET SEMAPHORE Command

Description:

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

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

Format:

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

Parameters:

scope

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

semaphore-name

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

err_label

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

Example:

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

See also:

Synchronization Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

RELEASE MUTEX Command

Description:

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

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

Format:

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

Parameters:

scope

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

mutex-name

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

err_label

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

Example:

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

See also:

Synchronization Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

CLEAR SEMAPHORE Command

Description:

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

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

Format:

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

Parameters:

scope

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

semaphore-name

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

err_label

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

Example:

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

See also:

Synchronization Commands

ACQUIRE MUTEX Command

Description:

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

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

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

Format:

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

Parameters:

scope

The scope of the mutex to be acquired. This must be either "LOCAL" or "TEST-

WIDE", and defaults to "LOCAL".

mutex-name

A character variable, or quoted character string, containing the name of the mutex which is to be acquired. "mutex-name" must be a valid OpenSTA Dataname.

period

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

tmo_label

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

err_label

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

Example:

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

See also:

Synchronization Commands

Synchronization Commands

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

See also:

ACQUIRE MUTEX Command

CLEAR SEMAPHORE Command

RELEASE MUTEX Command

SET SEMAPHORE Command

SYNCHRONIZE REQUESTS Command

START TEST-CASE Command

Description:

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

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

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

Format:

```
START TEST-CASE description
```

Parameter:

description

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

Examples:

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

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

See also:

Formal Test Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

REPORT Command

Description:

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

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

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

Any nonprintable ASCII characters in character values are replaced with periods ("."). Integer values are written as signed values, and use only as many characters as are necessary.

Format:

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

Parameters:

value

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

Examples:

```
REPORT "Section 1 Completed after ", loops, &
        ' Iterations'
REPORT "This is a long message ", &
        "that is continued on this line ", "and this line"
REPORT "This message contains a character command " &
```

```
        "to represent the tilde character ~~"
REPORT "This message contains a 'single quoted section'" &
        'and "a double one here".'
```

See also:

Formal Test Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

PASS TEST-CASE Command

Description:

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

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

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

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

Format:

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

Parameter:

label

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

Example:

```
START TEST-CASE "Checking distribution rate"
```

```
    IF (dist_rate >= minimum) THEN
        PASS TEST-CASE
    ELSE
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

See also:

Formal Test Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

HISTORY Command

Description:

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

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

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

Format:

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

Parameters:

value

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

Examples:

```
HISTORY "Test Run Completed." &
        ' Actions = ', action_count
HISTORY "This message contains a character command " &
        "to represent the tilde character ~~"
HISTORY "This message contains a 'single quoted section'" &
```

```
'and "a double one here".'
```

See also:

Formal Test Control Commands

FAIL TEST-CASE Command

Description:

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

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

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

Format:

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

Parameter:

label

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

Example:

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

```
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

See also:

Formal Test Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

END TEST-CASE Command

Description:

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

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

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

Format:

```
END TEST-CASE
```

Parameters:

None

Example:

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

See also:

Formal Test Control Commands

Formal Test Control Commands

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

See also:

END TEST-CASE Command

FAIL TEST-CASE Command

HISTORY Command

PASS TEST-CASE Command

REPORT Command

START TEST-CASE Command

REWIND Command

Description:

This command causes an external data file to be rewound. The file must already have been opened by the OPEN command.

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

Format:

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

Parameters:

fileid

The name associated with the file when it was opened.

err_label

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

Example:

```
REWIND datafile
```

See also:

File Handling Commands

READ Command

Description:

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

The record read will be delimited by a newline character in the file. This newline character is used purely as a record delimiter and does not form part of the record.

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

The file is read sequentially.

Format:

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

Parameters:

variable

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

fileid

The name associated with the file when it was opened.

label

A label within the current scope of the script, to which script execution will

branch if the "End-of-File" status is encountered.

err_label

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

Examples:

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

See also:

File Handling Commands

OPEN Command

Description:

This command opens an external data file for input access and associates an OpenSTA Dataname with it, for future reference.

When reading records from a file, data will be read up to but not including a newline character. The newline character will be skipped over to position the file at the start of the next record to be read.

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

Reads are independent for each thread.

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

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

Format:

```
OPEN filename AS fileid {ON ERROR GOTO err_label}
```

Parameters:

filename

A character variable or quoted character string containing the filename (excluding the path name) of the file to open. The file must reside in the data directory of the Repository.

fileid

An OpenSTA Dataname associated with the file when it is opened; it is used to identify the file in future references. The "fileid" must be declared in a FILE statement in the Definitions section of the script.

err_label

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

Examples:

```
OPEN "Usernames" AS datafile ON ERROR GOTO file-error
OPEN myfile AS datafile ON ERROR GOTO file-error
```

See also:

File Handling Commands

CLOSE Command

Description:

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

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

Format:

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

Parameters:

fileid

The name associated with the file when it was opened.

err_label

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

Example:

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

See also:

File Handling Commands

File Handling Commands

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

See also:

CLOSE Command

OPEN Command

READ Command

REWIND Command

SUBROUTINE Command

Description:

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

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

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

Format:

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

Parameters:

name

The name of the subroutine. This must be a valid OpenSTA Dataname, and must be unique within the script.

parameter

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

Examples:

```
SUBROUTINE GET_NEXT_VALUE
SUBROUTINE CREATE_FULL_NAME [subchr_1, subchr_2, subchr_3]
        SET full_name = subchr_3 + subchr_1 + subchr_2
        RETURN
END SUBROUTINE
```

See also:

Flow Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

RETURN Command

Description:

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

Format:

```
RETURN
```

Parameters:

None

Example:

```
RETURN
```

See also:

Flow Control Commands

ON ERROR Command

Description:

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

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

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

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

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

```
CANCEL ON ERROR
```

Format:

```
ON ERROR GOTO label
```

Parameter:

label

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

Example:

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

See also:

Flow Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

IF Command

Description:

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

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

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

Format:

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

Parameters:

condition

A condition of the following format:

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

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

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

"NOT" inverts the result of the bracketed condition that it precedes.

The binary operators are:

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

All conditions are evaluated from left to right.

label

A label defined in the current scope of the script.

command

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

Example:

```
IF ( NOT(isub=10) AND (NOT(isub=99)) ) THEN
```

```
      LOG "...continued"
ELSE
      LOG " Completed loop"
ENDIF
```

See also:

Flow Control Commands

GOTO Command

Description:

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

Conditional branches may be made using the IF command.

Format:

```
GOTO label
```

Parameter:

label

A label defined within the current scope of the script.

Examples:

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

See also:

Flow Control Commands

EXIT Command

Description:

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

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

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

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

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

Format:

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

Parameter:

status

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

Examples:

```
EXIT
EXIT RETURN-STATUS
```

See also:

Flow Control Commands

ENTRY Command

Description:

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

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

Format:

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

Parameter:

parameter

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

Example:

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

See also:

Flow Control Commands

END SUBROUTINE Command

Description:

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

A subroutine is initiated by the SUBROUTINE command.

Format:

```
END SUBROUTINE
```

Parameters:

None

Example:

```
END SUBROUTINE
```

See also:

Flow Control Commands

DO Command

Description:

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

Format:

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

Parameters:

variable

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

value1

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

value2

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

step

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

Examples:

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

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

See also:

Flow Control Commands

DETACH Command

Description:

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

Format:

```
DETACH {THREAD}
```

Parameters:

None

Examples:

```
DETACH
DETACH THREAD
```

See also:

Flow Control Commands

CANCEL ON Command

Description:

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

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

Format:

```
CANCEL ON {ERROR}
```

Parameters:

None

Examples:

```
CANCEL ON
CANCEL ON ERROR
```

See also:

Flow Control Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

CALL SCRIPT Command

Description:

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

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

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

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

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

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

code.

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

Format:

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

Parameters:

name

A character variable or quoted character string defining the name of the script to be called. The name must be a valid OpenSTA Dataname.

parameter

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

status

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

err_label

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

Examples:

```
CALL SCRIPT Script-Name
CALL SCRIPT "TEST"
CALL SCRIPT "CALC_TAX" [COST, RATE, TAX]
CALL SCRIPT "GET_RESPONS" returning Response &
        ON ERROR GOTO Problem
```

See also:

Flow Control Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

CALL Command

Description:

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

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

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

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

Format:

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

Parameters:

subroutine

The name of the called subroutine. The name must be a valid OpenSTA Dataname.

parameter

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

Examples:

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

See also:

Flow Control Commands

[TOC] [PREV] [NEXT] [INDEX]

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Flow Control Commands

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

See also:

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

~LOCATE Command

Description:

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

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

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

Format:

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

Return Value:

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

Parameters:

substring

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

string

The character value to be searched for the specified substring. This may be a

character variable or quoted character string.

Example:

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

See also:

Output Stream Handling Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD RESPONSE_INFO HEADER Command

Description:

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

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

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

The "WITH" clause can be used to specify the names of a header field whose value is to be retrieved from the HTTP response message. If this clause is omitted, all the response message header fields are retrieved.

Format:

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

Parameters:

conid

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

variable

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

identifier

A character variable, quoted character string or character expression containing the name of the response message header field to be retrieved.

Example:

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

See also:

Output Stream Handling Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

LOAD RESPONSE_INFO BODY Command

Description:

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

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

If the data string is too long to fit into the target variable, it will be truncated. For a response message body containing an HTML document, the "WITH" clause may be used to load a character variable with an element or part of an element from the document.

Format:

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

Parameters:

conid

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

variable

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

identifier

A character variable, quoted character string or character expression identifying the data to be retrieved from the response message body. For a definition of the identifier format see LOAD RESPONSE_INFO BODY Identifiers.

Example:

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

See also:

Output Stream Handling Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

FORMAT Command

Description:

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

In all translations, the command requires three elements:

The target variable that will receive the translated value. This may be either a character variable or an integer variable.

A format string defining the type of translation required. For an integer target variable, the format string must only contain a single format identifier; for a character variable, the format string may contain multiple identifiers and/or ordinary characters that are to be copied unchanged to the target variable.

One or more values to be translated; these may be specified as variables or as literal text. A single value must be specified for each of the format identifiers in the format string; the data type of each must agree with the associated format identifier and the data type of the target variable, as discussed below. Note that any discrepancies in this respect are detected at run-time and are not picked up by the compiler.

The following types of translation are supported:

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

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

%D - Convert a character string date value into numeric format (representing the number of days since the Smithsonian base date of 17-Nov-1858). The target variable must be an integer variable, and the source variable a character

string containing a valid date; this can be either in the default style for the platform on which the script is running or in the fixed format "DD-MMM-CCYY" (where "CC" is optional).

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

%T - Convert a character string time value into a numeric format (representing the number of 10 milli-second `ticks' since midnight). The target variable must be an integer variable, and the source variable a character string containing a valid time; this can be either in the default style for the platform on which the script is running or in the form "HH:MM:SS.MMM" (where ".MMM" is optional).

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

Format:

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

Parameters:

target-variable

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

format-string

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

variable

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

err_label

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

Examples:

```
FORMAT (date_string, &
        "The date is %D today, and the time is %T", &
        int-date, int-time), ON ERROR GOTO end
FORMAT (date_value, "%D", char-date), ON ERROR GOTO frm_err
FORMAT (uc_string, "Name in uppercase is %U", lc_string)
```

See also:

Output Stream Handling Commands

~EXTRACT Command

Description:

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

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

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

Format:

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

Return Value:

The character substring extracted from the source string.

Parameters:

offset

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

length

An integer variable or value defining the number of characters to extract to

form the substring.

string

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

Example:

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

See also:

Output Stream Handling Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

CONVERT Command

Description:

This command allows the value in an integer variable to be converted to an ASCII string, or vice versa. The default radix for the conversion is 10, but this may be overridden by including a "RADIX" clause in the command.

For integer-to-character conversions, format options may be specified. These options can cause the ASCII string to be left- or right-justified within the output buffer, or to have leading zeros or spaces, or cause the conversion to be signed or unsigned.

In the format description below, the "|" characters indicate mutually exclusive options.

The default options are SIGNED and LEFT JUSTIFY. If RIGHT JUSTIFY is in operation, the default filling is LEADING ZEROS.

If the output buffer is too small to hold the output string, it will be filled with asterisk ("*") characters.

For character-to-integer conversions, leading and trailing spaces are removed form the ASCII string before the conversion. Specification of a non-numeric ASCII string, or of an ASCII string which is converted to a numeric outside the range of an interger*4, will cause a message to be logged to the audit file indicating an invalid character string to convert. The thread will be aborted.

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

Format:

```
CONVERT variable1 TO variable2 {&}
        {,[SIGNED][UNSIGNED} {,LEADING [ZEROS]|[SPACES]} {&}
        {,[LEFT]|[RIGHT] JUSTIFY} {,RADIX=radix} {&}
        {,ON ERROR GOTO err_label}
```

Parameters:

variable1

A variable containing the variable to be converted.

variable2

A variable into which the converted variable is to be placed.

radix

An integer variable or literal in the range 2 to 36.

err_label

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

Examples:

```
CONVERT Number To String
CONVERT Number To Employee-Code, RIGHT JUSTIFY
CONVERT Ascii-code To Numeric_code
CONVERT Ascii-code To Hex_code, RADIX=16, &
        ON ERROR GOTO Conv_error
```

See also:

Output Stream Handling Commands

Output Stream Handling Commands

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

See also:

CONVERT Command

~EXTRACT Command

FORMAT Command

LOAD RESPONSE_INFO BODY Command

LOAD RESPONSE_INFO HEADER Command

~LOCATE Command

SET Command

Description:

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

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

+	for addition
-	for subtraction
*	for multiplication
/	for division
%	for modulo
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR

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

```
SET A = B / C
```

```
SET D = B % C
```

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

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

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

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

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

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

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

Format:

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

Parameters:

variable

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

operand1

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

operator

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

operand

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

err_label

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

Examples:

```
SET STRING1 = STRING2 - "ERROR"
SET STRING1 = STRING2 + STRING3 + STRING4
SET STRING1 = STRING2 - '"END MARKER"' &
            ON ERROR GOTO Error_report
```

See also:

Input Stream Entry Commands

RESET Command

Description:

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

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

Format:

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

Parameters:

variable

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

value

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

Examples:

```
RESET Emp-Name
```

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

See also:

Input Stream Entry Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

POST Command

Description:

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

The optional PRIMARY keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example:

A request pulling back an HTML page from a Web server can be followed by requests pulling back some GIF images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or by name using the "CERTIFICATE FILE" and "CERTIFICATE NAME" clauses.

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

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

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

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

Format:

```
 {PRIMARY}               POST [ URI | URL ] uri-httpversion {&}
                 ON conid {&}
                         HEADER http_header {&}
                 {,{BINARY} BODY http_body} {&}
                 {,WITH header_value} {&}
                 {,WITHOUT header_field} {&}
                                 {,CERTIFICATE FILE cert_filename}
{&}
                 {,CERTIFICATE NAME cert_name} {&}
                 {,RESPONSE TIMER timer_name} {&}
                 {,RETURNING STATUS response_status} {&}
                 {,RETURNING CODE response_code}
```

Parameters:

uri-httpversion

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

conid

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

http_header

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

http_body

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

header_value

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

header_field

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

cert_filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert_name

A character variable, quoted character string, character expression, containing a client certificate name.

timer_name

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

response_status

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

response_code

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

Examples:

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

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

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

See also:

Input Stream Entry Commands

NEXT Command

Description:

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

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

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

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

Format:

```
NEXT variable
```

Parameter:

variable

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

Example:

```
NEXT Emp-Name
```

See also:

Input Stream Entry Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

HEAD Command

Description:

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

The optional PRIMARY keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example:

A request pulling back an HTML page from a Web server can be followed by requests pulling back some GIF images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or by name using the "CERTIFICATE FILE" and "CERTIFICATE NAME" clauses.

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

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

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

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

Format:

```
{PRIMARY} HEAD [ URI | URL ] uri-httpversion {&}
        ON conid {&}
                HEADER http_header {&}
                {,WITH header_value} {&}
                {,WITHOUT header_field} {&}
                {,CERTIFICATE FILE cert_filename} {&}
                {,CERTIFICATE NAME cert_name} {&}
                {,RESPONSE TIMER timer_name} {&}
                {,RETURNING STATUS response_status} {&}
                {,RETURNING CODE response_code}
```

Parameters:

uri-httpversion

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

conid

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

http_header

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

header_value

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

fields are added to those specified in "http_header". If a request header field appears in both "http_header" and "http_value", the field specified here overrides that specified in "http_header".

header_field

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

cert_filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert_name

A character variable, quoted character string, character expression, containing a client certificate name.

timer_name

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

response_status

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

response_code

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

Examples:

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

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

See also:

Input Stream Entry Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

GET Command

Description:

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

The optional PRIMARY keyword denotes primary HTTP requests such as those referred to by the "referer" header in secondary requests. For example:

A request pulling back an HTML page from a Web server can be followed by requests pulling back some GIF images whose URLs are contained in the specified page.

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

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

A client certificate may be specified in a request either by file or by name using the "CERTIFICATE FILE" and "CERTIFICATE NAME" clauses.

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

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

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

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

Format:

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

Parameters:

uri-httpversion

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

conid

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

http_header

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

header_value

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

fields are added to those specified in "http_header". If a request-header field appears in both "http_header" and "header_value", the field specified here overrides that specified in "http_header".

header_field

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

cert_filename

A character variable, quoted character string, character expression, containing the name of a file. The file contains a client certificate.

cert_name

A character variable, quoted character string, character expression, containing a client certificate name.

timer_name

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

response_status

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

response_code

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

Examples:

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

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

See also:

Input Stream Entry Commands

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

GENERATE Command

Description:

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

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

Format:

```
GENERATE variable
```

Parameter:

variable

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

Example:

```
GENERATE Part-Number
```

See also:

Input Stream Entry Commands

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Input Stream Entry Commands

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

See also:

GENERATE Command

GET Command

HEAD Command

NEXT Command

POST Command

RESET Command

SET Command

HTTP Commands

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

The HTTP commands are as follows:

Code Section Commands

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

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

Refer to the HTTP Commands section for information relating to the commands that can be used with HTTP.

See also:

HTTP Commands

Input Stream Entry Commands

Output Stream Handling Commands

Flow Control Commands

File Handling Commands

Formal Test Control Commands

Synchronization Commands

Input Stream Entry Commands

Statistical Data Logging Commands

Diagnostic Commands

Miscellaneous Commands

The CODE Section

LOAD RESPONSE_INFO BODY Identifiers

The LOAD RESPONSE_INFO BODY command loads a character variable with all or part of the data from an HTTP response message body for a specified TCP connection. For a response body containing an HTML document, the "WITH" clause may be used to load a character variable with an element or part of an element from the document.

The "WITH" clause has the following format:

,WITH identifier

Note: identifier is a character variable, quoted character string or character expression identifying the data to be retrieved from the HTML document in the response message body. The following sections describe the format of this identifier.

HTML Element Addressing

An element within an HTML document is identified by an element address string.

Format:

tag(tagnum){/tag(tagnum)}:element_type:{attribute}(element_num)

Parameters:

tag

The HTML tag name.

tagnum

A number identifying the tag relative to its parent tag or the document root.

0 = First child tag

1 = Second child tag
n = nth child tag

element_type

The HTML element type. This must be one of the following:

ANONYMOUS ATTRIBUTE

ATTRIBUTE

COMMENT

SCRIPT

TEXT

attribute

For element_type ATTRIBUTE, specifies the name of the HTML attribute.

element_num

A number identifying the element. For element type ATTRIBUTE, the number identifies the attribute relative to its associated tag.

0 = First attribute

1 = Second attribute

n = nth attribute

Examples:

```
HTML(0)/BODY(1)/TABLE(1)/TBODY(0)/TR(0)/TD(0):TEXT:(0)
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)
```

Notes:

- There must be no whitespace between any of the components of an identifier.
- Identifiers are not validated at compile time.

Qualifying an HTML Element Address

A complete HTML element string may be retrieved from an HTML document using an identifier containing only an HTML element address. However, a substring may be selected from it using a variety of qualifiers. These qualifiers immediately follow the HTML element address and are described below.

Selecting a Substring by Position and Length

An HTML element substring may be selected using an identifier specifying the offset of the substring and its length.

Format:

element_addr[offset,length]

where "[" and "]" are literal characters and part of the required syntax.

Parameters:

element_addr

The HTML element address in the format described above.

offset

The offset of the first character of the substring from the start of the element string.

length

The number of characters in the substring.

Notes:

- If the offset is invalid, an empty string is returned.
- If the length is zero, or is invalid, all characters from the start offset to the end of the element string are returned.

Example:

HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,5]

Selecting a Substring using Delimiters

An HTML element substring may be selected by specifying an identifier containing two string delimiters. The substring returned contains all the characters between the first occurrence of the first delimiter and the first occurrence of the second. The string will also include both delimiter strings.

Format:

element_addr[delimiter1,delimiter2]

where "[" and "]" are literal characters and part of the required syntax.

Parameters:

element_addr

The HTML element address in the format described above.

delimiter1

A string - enclosed in single quotes - identifying the characters at the beginning of the substring.

delimiter2

A string - enclosed in single quotes - identifying the characters at the end of the

substring.

Notes:

- If delimiter1 cannot be found, an empty string is returned.
- If delimiter2 cannot be found, all characters from and including delimiter1 to the end of the element string are returned.

Example:

HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['document.cookie=',';']

Selecting a Substring Using Position, Length and Delimiter String

The above two methods of substring selection can be combined, allowing an HTML element substring to be identified by a start string and a length or an offset and a termination string.

Format:

element_addr[delimiter1,length]

or

element_addr[offset,delimeter2]

where "[" and "]" are literal characters and part of the required syntax.

Parameters:

element_addr

The HTML element address in the format described above.

delimiter1

A string - enclosed in single quotes - identifying the characters at the beginning of the substring.

length

The number of characters in the substring.

offset

The offset of the first character of the substring from the start of the element string.

delimiter2

A string - enclosed in single quotes - identifying the characters at the end of the substring.

Notes:

- If delimiter1 cannot be found, an empty string is returned.
- If the offset is invalid, an empty string is returned.
- If delimiter2 cannot be found, all characters after, and including, delimiter1 to the end of the element string are returned.
- If the length is zero, or is invalid, all characters from the specified offset to the end of the element string are returned.

Examples:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['cookie=',3]

HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';']
```

Excluding Delimiters from Selection

With the syntax described above, any delimiter strings specified are included in the returned substring. Either or both delimiters may be excluded from the returned substring by inverting the square bracket nearest to the delimiter, i.e. using an opening square bracket in place of a closing square bracket and vice versa.

This method can also be used with offset parameters. Instead of identifying the offset of the first character of the substring to be selected, using this alternative syntax, the offset becomes the offset of the character immediately before the first character to be selected.

The following examples illustrate how a substring may be selected from the CONTENT attribute string of an HTML META tag.

Examples:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';']
```

Selects the substring that starts at offset 3 from the beginning of the attribute string and that is terminated by - and includes - the next semicolon in the string.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';'[
```

Selects the substring that starts at offset 2 from the beginning of the attribute string and that is terminated by - but does not include - the next semicolon in the string.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';'[
```

Selects the substring that starts at offset 3 from the beginning of the attribute string and that is terminated by - but does not include - the next semicolon in the string.

Ignoring the Characters at the Beginning of an HTML Element

There are occasions when it is useful to use the above facilities starting from some point within the element string, rather than at the beginning of the string. This can be achieved by resetting the selection base. This can be done by specifying the selection base as an offset from the beginning of the element string, or by specifying a substring that identifies the characters at the beginning of the substring to be examined. The offset or substring is preceded by one of two operators ">" or ">=":

>offset

The offset is that of the character immediately before the substring to be examined.

>substring

The substring identifies the characters at the end of the string to be ignored. The substring starts with the first character after the substring.

>=offset

The offset is that of the first character in the substring to be examined.

>=substring

The substring identifies the characters at the beginning of the substring to be examined.

Note:

If the offset or substring cannot be found, an empty string is returned.

The following examples illustrate how the selection base is reset for a selection from the CONTENT attribute string of an HTML META tag.

Examples:

```
 HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>'//
Cookie','document.cookie=',';']
```

The selection base offset is set to the offset of the first character after the first occurrence of the string "// Cookie" in the element string. The selected substring starts with the character after "document.cookie=" and ends with - and includes - the next semicolon.

```
 HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>='//
Cookie','document.cookie=',';']
```

Same as above, except that the selection base offset is now the first character of "// Cookie".

```
 HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>=50,'document.
cookie=','
;']
```

Same as above, except that selection base offset is now 50 characters from the start of the element string.

Ignoring the Case of Characters

All string comparisons specified by LOAD RESPONSE_INFO BODY identifiers are by default case sensitive. The case of characters can be ignored in comparisons by prefixing the search string or delimiter string by "I".

Example:

```
 HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>I'//
Cookie',I'document.cookie=',';']
```

The selection base is reset by searching the element string for "// Cookie"; the case of characters is ignored in the search.

Specifying Quotes Within Identifiers

Quoted character strings within SCL are delimited, either by single quotes or by double quotes. Since the syntax of a LOAD RESPONSE_INFO BODY identifier includes single quotes, it is recommended that double quotes are used to delimit a quoted character string containing such an identifier.

A literal single quote character can be included within an identifier string by preceding it with a backslash. For example:

```
"HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'\'']"
```

This selects a substring terminated by a single quote.

A literal double quote character can be specified within an identifier string, using the SCL character command, ~<22>. For example,

```
"HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'~<22>']"
```

This selects a substring terminated by a double quote.

See also:

The CODE Section

Symbols

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

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

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

See also:

The CODE Section

Labels

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

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

A label name must be a valid OpenSTA Dataname.

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

See also:

The CODE Section

Variables

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

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

See also:

The CODE Section

Script Processing

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

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

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

See also:

The CODE Section

Command Types

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

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

See also:

The CODE Section

Code Section Structure

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

Commands

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

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

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

Characters Ignored by the Compiler

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

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

See also:

The CODE Section

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

The CODE Section

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

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

See also:

Code Section Structure

Command Types

Script Processing

Variables

Labels

Symbols

LOAD RESPONSE_INFO BODY Identifiers

Code Section Commands

Example Variable Definitions

This section shows a number of example variable definitions:

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

See also:

The DEFINITIONS Section

File Option

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

```
, FILE = filename
```

where "filename" is a quoted character string which defines the name of the ASCII text file, excluding the path name and file extension. The file must reside in the data directory of the Repository and have the file extension .FVR.

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

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

-100
0
100

A file for a character variable could contain the values:

Pele
10
Cruyff
14

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

Values are retrieved from the file associated with a variable using the NEXT

command. This command retrieves the next sequential value from the file. When the NEXT command is first executed, it will retrieve the first value from the file. If the variable is set to the last value in the file when the NEXT command is executed, the variable will be reset to the first value in the file. You can also reset the variable explicitly, by using the RESET command.

The file option is not valid for variables which:

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

See also:

Variable Options

Random Variable Options

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

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

These options function as follows:

RANDOM

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

This option is particularly useful when load testing a system.

This is the default if no random option is specified.

See also:

Variable Options

REPEATABLE {RANDOM}

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

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

SEED = n

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

See also:

Variable Options

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Variable Scope Options

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

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

These options are described below:

LOCAL

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

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

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

SCRIPT

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

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

The space for script variables is normally deallocated when the execution of a script terminates, and no other threads are running the script. In some cases,

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

THREAD

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

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

Thread variables cannot have associated value lists or ranges.

GLOBAL

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

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

Global variables cannot have associated value lists or ranges.

See also:

Variable Options

Variable Options

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

The following sections describe the types of variable option clause.

See also:

Variable Scope Options

Random Variable Options

File Option

The DEFINITIONS Section

Variable Values

A set of values may be associated with a variable, using a value clause in the variable definition. They are used by the GENERATE and NEXT commands, which allow the variable to be assigned a value from the list or range, either randomly (using GENERATE) or sequentially (using NEXT). Values may be specified as a list (integer and character variables) or as a range (integers only). Note: Lists may contain only individual values, and not ranges.

Variables which have been declared as an array may not have an associated value list or range. A value list has the following format:

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

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

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

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

```
(start_value - end_value)
```

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

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

In the following list of example variable definitions including values, the first

two definitions are equivalent:

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

See also:

The DEFINITIONS Section

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Variable Arrays

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

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

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

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

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

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

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

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

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

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

See also:

The DEFINITIONS Section

TIMER Statement

Description:

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

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

Format:

```
TIMER name
```

Parameter:

name

The name of the timer. This must be a valid OpenSTA Dataname.

Examples:

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

See also:

The DEFINITIONS Section

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

INTEGER Statement

Description:

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

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

Format:

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

Parameters:

name

The name of the variable. This must be a valid OpenSTA Dataname.

dimensions

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

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

values

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

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

options

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

Examples:

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

See also:

The DEFINITIONS Section

FILE Statement

Description:

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

Format:

```
FILE input_fileid
```

Parameter:

input_fileid

An OpenSTA Dataname used to identify a file that is passed as a parameter to the script.

Example:

```
FILE datafile
```

See also:

The DEFINITIONS Section

CONSTANT Statement

Description:

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

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

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

Format:

```
CONSTANT name = value
```

Parameters:

name

The name of the constant. This must be a valid OpenSTA Dataname.

value

A quoted character string or an integer value.

Examples:

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

See also:

The DEFINITIONS Section

CHARACTER Statement

Description:

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

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

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

Format:

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

Parameters:

n

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

name

The name of the variable. This must be a valid OpenSTA Dataname.

dimensions

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

If a dimension has only one number, the elements in that dimension range from 1 to the number specified. If two numbers are specified, they must be

separated by a colon (":"); the elements in this dimension range from the first number to the second.

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

values

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

options

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

Examples:

```
CHARACTER:15 dept
CHARACTER:20 names ('TOM','JOHN','DICK'), SCRIPT
CHARACTER:9 months [12]
CHARACTER*20 staff-by-dept [8,101:150]
```

See also:

The DEFINITIONS Section

The DEFINITIONS Section

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

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

See also:

CHARACTER Statement

CONSTANT Statement

FILE Statement

INTEGER Statement

TIMER Statement

Variable Arrays

Variable Values

Variable Options

Example Variable Definitions

WAIT UNIT Statement

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

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

Format:

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

Parameters:

None

See also:

The ENVIRONMENT Section

MODE HTTP Statement

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

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

Format:

```
MODE HTTP
```

Parameters:

None

See also:

The ENVIRONMENT Section

DESCRIPTION Statement

Description:

This mandatory statement assigns a descriptive character string to a script.

Format:

```
DESCRIPTION string
```

Parameter:

string

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

Examples:

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

See also:

The ENVIRONMENT Section

The ENVIRONMENT Section

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

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

See also:

DESCRIPTION Statement

MODE HTTP Statement

WAIT UNIT Statement

Conditional Compilation of Source Code

SCL provides commands that enable you to define the circumstances for the compilation of a section of code. Conditional sections of code are marked with `variants', which are specified on the `-V' option on the SCL compiler command line when you compile the source file.

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

Format:

condition variant

Parameters:

condition

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

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

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

variant

An OpenSTA Dataname which identifies a section of code that is only compiled under certain conditions. The compiler processes this variant in conjunction with the `-V' option on the SCL command line.

Examples:

```
#IFDEF variant1
        log "Only compiled if /VARIANT=variant1 is specified"
#ELIF variant2
        log "Only compiled if /VARIANT=variant2 is specified"
#ELSE
        log "Only compiled if neither variant is specified"
#ENDIF
```

See also:

Overview of Script Control Language Syntax

Including Text from Other Source Files

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

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

Format:

```
INCLUDE filename
```

Parameter:

filename

A quoted character string which defines the name of the source file to be included. The location of the file will default to the Scripts\Include directory within the Repository.

Example:

```
INCLUDE 'mydefs.inc'
```

See also:

Overview of Script Control Language Syntax

Maximum Values in Scripts

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

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

Max. **nesting level** for **IF/DO commands**	**100**
Max. **nesting level** for **subroutines**	**10**

See also:

Overview of Script Control Language Syntax

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

OpenSTA Datanames

The names of many items within scripts must be defined as an OpenSTA Dataname. For example labels, variable names and subroutine names must all be OpenSTA Datanames.

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

See also:

Overview of Script Control Language Syntax

Comments

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

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

See also:

Overview of Script Control Language Syntax

Continuation Lines

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

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

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

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

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

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

See also:

Overview of Script Control Language Syntax

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Representing the Control Character

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

```
~^
```

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

See also:

Character Representation

Representing the Command Character

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

```
~~
```

"**~**" is the defined command character.

See also:

Character Representation

Control Command

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

```
^c
```

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

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

See also:

Character Representation

Character Command Using ASCII Mnemonic

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

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

See also:

Character Representation

Character Command Using Hexadecimal ASCII Code

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

~<hh>

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

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

See also:

Character Representation

Character Representation

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

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

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

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

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

Character commands are recognized within all SCL character strings (except for a small number of exceptions that are explicitly stated). Thus, for example, the character string "~<07>" always represents a single character (namely the

character with a hexadecimal value of 7), not five characters.

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

See also:

Character Command Using Hexadecimal ASCII Code

Character Command Using ASCII Mnemonic

Control Command

Representing the Command Character

Representing the Control Character

Overview of Script Control Language Syntax

Overview of Script Control Language Syntax

The Script Control Language (SCL) is used to write scripts. Scripts define and control the test cases and input that are to be used to test the target system.

Script files consist of up to three sections which must appear in the following order if present:

- **Environment** section
- **Definitions** section
- **Code** section

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

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

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

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

See also:

Character Representation

Continuation Lines

Comments

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Further enquiries
Documentation feedback
CYRANO.com

Vue d'ensemble de syntaxe d'ordres de gestion de script

Ordres de gestion de script (SCL) est employé pour écrire des scripts. Les scripts définissent et commandent les tests et les entrent qui doivent être employés pour examiner le système de cible.

Les dossiers de script se composent de jusqu'à trois sections qui doivent apparaître dans l'ordre suivant si présent:

- **section** d'environnement
- **Section** de définitions
- **Section** code

La première section est la section obligatoire d'**environnement**. Cette section définit les attributs globaux du script, c.-à-d. la description de script, le mode de script et les unités de commande d'attente. Elle est présentée par la commande d'Environnement, et continue jusqu'aux DÉFINITIONS ou la commande de CODE est produite.

La deuxième section est la section facultative de **définitions**. Cette section contient les définitions variables, constantes, de temporisateur et de dossier pour le script. Elle commence par les DÉFINITIONS commandent, et continuent jusqu'à la commande de CODE.

La dernière section est la section obligatoire de **code**, qui contient les commandes principales de script. Le début de cette section est marqué par la commande de CODE; il continue jusqu'à l'extrémité du dossier de script.

Des tab , les espaces et les chargements de page peuvent être incorporés au code pour aligner des mots-clés et généralement la lisibilité d'aide; elles n'ont aucun autre effet sur la compilation.

Voyez également:

Représentation De Caractère (Character Representation)

Lignes suite (Continuation lines)

Commentaires (Comments)

OpenSTA Datanames (OpenSTA Datanames)

Représentation De Caractère(Character Representation)

Le texte dans un fichier source de SCL entre dans trois larges catégories:

1. Commandes de SCL.
2. Arguments aux commandes de SCL - noms variables, valeurs de nombre entier ou chaînes de caractères citées, par exemple.
3. Commentaires, pour améliorer la lisibilité et l'entretien.

Dans des arguments de chaîne de caractères, le SCL soutient l'utilisation de n'importe quel caractère avec une valeur d'Ascii dans le gamme HEX 00 à FF inclus. Cependant, les spécifications directes de ces caractères ne sont pas toujours possibles, à deux raisons:

1. Les caractères avec des valeurs dans le HEX de gammes 00 à 20 et le HEX 7F à A0, et le HEX FF de valeur, sont caractères « non imprimable » de , et ne peuvent pas facilement être indiqués dans un fichier source de SCL.
2. Deux caractères sont réservés à l'usage du SCL - un comme caractère de commande et l'autre comme caractère de control. Les caractères utilisés dans ces buts ne peuvent pas être employés en tant que caractères littéraux dans une chaîne de caractères. Les valeurs par défaut sont "~" pour le caractère de commande et "^" pour le caractère de control; ces valeurs sont employées dans toutes ces instructions. Elles peuvent, cependant, être changées dans le script.

Pour résoudre ces problèmes, le SCL fournit un ensemble de « commandes de caractère de » , comme décrit en représentant le caractère de commande et en représentant le caractère de commande . En outre, s'assurer là n'est aucune ambiguïté dans le fichier source, des caractères sont rejetés qui ont des valeurs dans, du HEX de gammes 00 à 20 ou le HEX 7F à A0, ou le HEX FF de valeur, à moins que comme décrit en caractères ignorés par le compilateur .

Des commandes de caractère sont identifiées dans toutes les chaînes de caractères de SCL (excepté un nombre restreint d'exceptions qui sont explicitement énoncées). Ainsi, par exemple, la chaîne de caractères "~<07 >" représente toujours un caractère simple (notamment le caractère avec une valeur hexadécimale de 7), non cinq caractères.

Note: Des citations simples peuvent être incluses dans des chaînes de caractères en employant de doubles citations pour les délimiteurs de chaîne, et vice versa.

Voyez également:

Commande De Caractère En utilisant Le Code Hexadécimal d'cAscii (Charactere Command Using ASCII Code)

Commande De Caractère En utilisant La Mnémonique d'cAscii (Charactere Command Using ASCII Mnemonic)

Commande De Commande (Cintrol Command)

Représentation du caractère de commande (Representing the command Character)

Représentation du caractère de commande (Representing the Control Character)

Vue d'ensemble de syntaxe d'ordres de gestion de script (overview of script Control Language Syntax)

Commande De Caractère En utilisant Le Code Hexadécimal d'Ascii

Tous les caractères peuvent être représentés par code hexadécimal d'Ascii, commande de caractère. Le format de commande est:

```
~<hh >
```

le "~" est le caractère actuellement défini de commande et le "hh" est le code hexadécimal Ascii du caractère exigé. Cette forme de commande de caractère est principalement prévue pour représenter les caractères qui ne peuvent être représentés par aucune de ces autres formes de commande de caractère.

Par exemple, le caractère de tabulation horizontale Ascii est représenté par "~<09 >" et le caractère nul par "~<00 >".

Voyez également:

Représentation De Caractère

Commande De Caractère En utilisant La Mnémonique d'cAscii

Le SCL fournit un certain nombre de commandes de caractère qui donnent une représentation facilement identifiable des caractères de commande communs. Ceux-ci emploient la mnémonique Ascii du caractère de commande en question. Les commandes suivantes sont disponibles:

~<**bel** >	Bell	~<**BEL**>	Bell
~<**bs** >	Espacement arrière	~<**BS**>	Backspace
~<**cr** >	Retour de chariot	~<**CR**>	Carriage return
~<**del** >	Effacement	~<**DEL**>	Delete
~<**esc** >	Évasion	~<**ESC**>	Escape
~<**ff** >	Avance d'état ordinateur	~<**FF**>	Form feed
~<**ht** >	Étiquette horizontale	~<**HT**>	Horizontal tab
~<**lf** >	Retour à la ligne	~<**LF**>	Line feed
~<**vt** >	Étiquette verticale	~<**VT**>	Vertical tab

Voyez également:

Représentation De Caractère

Commande De Contrôle (Control Command)

Tous les caractères de control 7-bit, c.-à-d. caractères avec des codes Ascii dans de la gamme HEX 00 à 1F inclus, peuvent être représentés en utilisant une commande de contrôle . La commande de contrôle a le format suivant:

^c

"^" est le caractère de contrôle par défaut et "c" est le spécificateur de caractère de contrôle . Le spécificateur de caractère de contrôle est un caractère graphique Ascii avec un code Ascii dans la gamme HEX 40 (ASCII "@") à 5F (ASCII"_"). le compilateur appliquera les bits du fond 6 seulement, pour produire d'un code Ascii dans la gamme HEX 00 au 1F.

Par exemple, le caractère d'appel Ascii (HEX 07 de code Ascii), est représenté par le "^G".

Voyez également:

Représentation De Caractère

Représentation du caractère de commande

Le caractère de commande présente toujours une commande et ne peut pas donc être employé pour représenter le caractère de commande lui-même. Le caractère de commande est à la place représenté par une commande du format suivant:

~~

"**~** "est le caractère de commande défini.

Voyez également:

Représentation De Caractère

Représentation du caractère de contrôle

Le caractère de contrôle est toujours employé pour représenter la clef de **Ctrl**, en combinan avec le caractère suivant. Il ne peut pas donc être employé pour représenter le caractère de contrôle lui-même. Le caractère de contrôle est à la place représenté par une commande du format suivant:

~^

le "~" est le caractère de commande défini et "^" est le caractère de contrôle défini.

Voyez également:

Représentation De Caractère

Lignes suite

Il n'est pas toujours possible d'adapter un rapport ou une commande de script sur une ligne, ainsi le SCL vous permet d'employer des « lignes suite de » « continuation lines ».

Une déclaration ou la commande de SCL peut être dédoublé plus de deux lignes ou plus en terminant tout sauf la dernière ligne de la déclaration avec un caractère d'esperluette ou de trait d'union ("et "ou"-"). pour éviter la confusion possible avec le caractère moindre, on lui recommande que l'esperluette soit employée, et qu'elle soit séparée des caractères précédents sur la ligne par au moins un espace.

Les seules choses qui peuvent suivre un caractère de continuation sont des caractères d'espace, des caractères de TAB et des commentaires (voir la prochaine section).

Une chaîne de caractères citée est continuée sur une autre ligne en la fermant à l'extrémité de la ligne et en la rouvrant sur le prochain. Les citations s'ouvrantes et se fermantes doivent s'assortir sur n'importe quelle une ligne, comme montré dans l'exemple suivant:

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

Note: Une ligne qui finit avec une commande SCL ou la déclaration terminé "et" ou "-" impliquent que la prochaine ligne produite sera considérée comme une suite de la commande ou du rapport originale.

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

Commentaires

Les scripts peuvent incorporer des commentaires, sur des lignes par eux-mêmes ou incluses dans les déclaration ou les commandes. En les deux cas, le commentaire est identifié par la commande de commentaire ("!"),et terminé vers la fin de la ligne. Par exemple:

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

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

OpenSTA Datanames

Les noms de beaucoup d'articles dans des scripts doivent être définis comme OpenSTA Dataname. Par exemple les labels, les noms variables et les noms de sous-programme doivent tout être OpenSTA Datanames.

Un OpenSTA Dataname comporte entre 1 et 16 caractères alphanumériques, de soulignage ou de trait d'union. Le premier caractère doit être alphabétique; on ne permet pas les espaces; on ne permet pas deux soulignages ou traits d'union adjacents; et ni l'un ni l'autre n'est un soulignage ou un trait d'union de remorquage.

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

Valeurs maximum en scripts

Le compilateur SCL et le système ressource imposent des limitations au temps d'exécution à la valeur maximum (nombre, taille, niveau etc...) compte tenu d'un certain nombre d'articles qui peuvent être indiqués dans un fichier source de SCL.

Description	Valeur
Maximum. **longueur de ligne de source** (caractères)	132
Numéro de maximum **des étiquettes** (par code de subroutine/main)	255
Numéro de maximum **des temporisateurs**	1020
Numéro de maximum **des variables**	8000
Numéro de maximum **des variables globales**	8000
Numéro de maximum **des sous-programmes**	255
Le numéro de maximum **des paramètres** a passé entre les manuscrits	8
Numéro de maximum **des fichiers de données externes référencés** en manuscrit	256
Le numéro de maximum **des fichiers de données externes s'ouvrent** concurremment	10
Maximum. **taille variable de caractère** (bytes)	65535
Maximum. **taille du caractère constant/literal** (bytes)	65535
Maximum. **l'espace disponible pour des valeurs de manuscrit** (K bytes)	128
Maximum. **niveau d'emboîtement** pour **des conditions**	10
Maximum. **niveau d'emboîtement** pour **des expressions de rangée**	10
Maximum. **niveau d'emboîtement** pour **les compilations conditionnelles**	10
Maximum. **niveau d'emboîtement** pour **des commandes d'cIf/do**	100
Maximum. **niveau d'emboîtement** pour **des sous-programmes**	10

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

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

Inclure des texte à partir d'autres fichiers source

La commande d'Inclure vous permet de combiner plusieurs fichiers source dans un fichier source simple au temps de compilation. Ces fichiers inclus peuvent contenir des commandes de n'importe laquelle de ces sections de script et peuvent enjamber ces sections. Des scripts peuvent être nichés jusqu'à une profondeur de 10, y compris le script principal. Le soin devrait être pris pour éviter de reproduire n'importe laquelle de ces commandes de section de script (par exemple, *ENVIRONMENT*).

Cette commande peut apparaître à un point quelconque dans le script, incluant avant la commande

d'Environment.

Format:

```
INCLUDE filename
```

Paramètre:

Filename

Une chaîne de caractères citée qui définit le nom du fichier source à inclure. La location du fichier se transférera sur l'annuaire de *Scripts\Include* dans le dossier.

Exemple:

```
INCLUDE 'mydefs.inc '
```

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

Compilation conditionnelle de code source

Le SCL fournit les commandes qui vous permettent de définir les circonstances pour la compilation d'une section de code. Des sections conditionnelles du code sont identifiées par « les variantes de », qui sont indiquées option sur `-V 'sur la ligne de commande de compilateur SCL quand vous compilez le fichier source.

Les commandes conditionnelles de compilation peuvent apparaître à un point quelconque dans les sections d'environnement, de définitions et de code, incluant avant la commande *ENTRY* et entre les sous-programmes. Elles ne peuvent pas apparaître voie de partie par une commande ou un rapport. Elles peuvent être nichées à une profondeur de 10.

Format:

Condition variant

Paramètres:

condition

Une commande conditionnelle de compilation qui commence ou finit une section de code. Ceci peut être l'un de ce qui suit:

# IFDEF	Compilez la prochaine section si la "variante" demandait
# IFNDEF	Compilez la prochaine section si "variante" non demandée
# ELIF	Autrement compilez la prochaine section si la "variante" demandait
# AUTREMENT	Autrement compilez la prochaine section
# ENDIF	Extrémité de section variable

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

IFDEF, # IFNDEF et des commandes # ELIF exigent de paramétrer une "variable", d'indiquer la condition dans laquelle la section suivante du code sera compilée. # ELSE et # les commandes # ENDIF se relient à la variante le plus récemment indiquée.

variante

Un OpenSTA Dataname ce qui identifie une section du code qui est seulement compilé dans certaines conditions. Le compilateur traite cette variante option en même temps que `-V 'sur le SCL ligne de commande.

Exemples:

```
#IFDEF variant1
        log "Only compiled if /VARIANT=variant1 is specified"
#ELIF variant2
        log "Only compiled if /VARIANT=variant2 is specified"
#ELSE
        log "Only compiled if neither variant is specified"
#ENDIF
```

Voyez également:

Vue d'ensemble de syntaxe d'ordres de gestion de script

La Section d'Environnement (ENVIRONMENT SECTION)

La section d'environnement du code source de SCL est présentée par la commande obligatoire d'Environnement. Elle définit les attributs globaux du script, c.-à-d. la description de script, le mode de script et unités de commande d'attente.

La section d'environnement doit être la première section du script, précédant la section de définitions (si présent) et la section de code. Elle peut, cependant, être précédée par une déclaration d'Inclusion . Pour de plus amples informations, voyez inclure le texte à partir d'autres fichiers source .

Voyez également:

Rapport de DESCRIPTION

Rapport de HTTP de MODE

Rapport d'cUnité d'cAttente

DESCRIPTION Statement

Description:

Cette déclaration obligatoire assigne une chaîne de caractères descriptive à un script.

Format:

```
    DESCRIPTION  String
```

Paramètre:

String

Une chaîne de caractères citée, entre 1 et 50 caractères de long, utilisée comme description.

Exemples:

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

Voyez également:

La Section d'cEnvironnement

La déclaration du HTTP MODE

Cette déclaration facultatif définit le script comme script en mode HTTP. Ces scripts sont employés pour publier des requêtes HTTP à un serveur HTTP.

Cette déclaration doit être indiqué afin que les commandes HTTP-spécifiques soient disponible à un script.

Format:

```
Mode http
```

Paramètres:

Aucun

Voyez également:

La Section d'cEnvironnement

Déclaration d'Unité d'Attente(WAIT UNIT)

Cette déclaration facultatif définit l'unité de la période d'attente indiquée dans des commandes d'Attente dans un script. Ceci ne s'applique pas à la période d'attente dans la commande de *WAIT FOR SÉMAPHORE* - la période d'attente dans cette commande est toujours indiquée en secondes.

Si ce rapport est omis, l'unité d'attente est des secondes.

Format:

```
WAIT UNIT [ SECONDES|MILLISECONDES ]
```

Paramètres:

Aucun

Voyez également:

La Section d'cEnvironnement

La Section de DÉFINITIONS

La section de définitions du code source SCL définit les variables et les constantes employées par le script. Elle peut également contenir des déclarations des temporisateurs et des fichiers. Elle est facultative et est présenté par les commandes *DÉFINITIONS*.

Seulement une section de définitions peut apparaître dans un script; si elle est présente, elle doit suivre la section d'environnement et précéder la section de code.

Voyez également:

Rapport de CARACTÈRE

Rapport CONSTANT

Rapport de DOSSIER

Rapport de NOMBRE ENTIER

Rapport de TEMPORISATEUR

Rangées Variables

Valeurs Variables

Options Variables

Définitions De Variable D'Exemple

Rapport de CARACTÈRE

Description:

Ce rapport définit une variable de chaîne de caractères se composant des caractères Ascii, y compris des caractères de commande. Le SCL soutient des variables de caractère de entre 1 et 65535 bytes de long.

Des choix de variables de caractère peuvent être définis, avec un maximum de trois dimensions. Pour de plus amples informations au sujet des rangées, voir les rangées variables .

Un astérisque peut être employé au lieu des deux points pour délimiter la taille.

Format:

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

Paramètres:

n

Une valeur de nombre entier non signé dans la gamme 1-65535, représentant la taille de la variable dans les bytes. Le défaut est 1.

name

Le nom de la variable. Ceci doit être un OpenSTA valide Dataname .

dimensions

Les dimensions de la rangée à assigner pour cette variable. Jusqu'à trois dimensions peuvent être indiquées, séparé par des virgules, chacune comportant un ou deux nombres.

Si une dimension a seulement un nombre, les éléments de cette gamme de dimension de 1 au nombre a indiqué. Si deux nombres sont indiqués, ils doivent être séparés par des deux points (":"); les éléments dans cette gamme de dimension du premier nombre au second.

Notez que si des "dimensions" sont indiquées, les "valeurs" peuvent ne pas être.

Values

Une liste de valeurs de caractère à associer à la variable. Notez que si des "valeurs" sont indiquées, les "dimensions" peuvent ne pas être. Voir les valeurs variables pour de plus amples informations sur des valeurs variables.

options

Une liste d'options variables. Voir les options variables pour de plus amples informations en des options variables.

Exemples:

```
CHARACTER:15 dept
CHARACTER:20 names ('TOM','JOHN','DICK'), SCRIPT
CHARACTER:9 months [12]
CHARACTER*20 staff-by-dept [8,101:150]
```

Voyez également:

La Section de DÉFINITIONS

Rapport CONSTANT

Description:

Ce rapport définit une variable qui a une valeur statique dans un script. Ils peuvent être traduits ainsi au temps de compilation, et ne pas consommer la mémoire au temps d'exécution.

La valeur d'une constante peut être une valeur de nombre entier ou une chaîne de caractères citée.

Des constantes peuvent être employées dans n'importe quelle situation où une coquille du même type (c.-à-d. caractère ou nombre entier) peut être employée, par exemple dans une liste de valeur. La seule contrainte est que la constante doit avoir été définie avant qu'elle soit employée.

Format:

```
CONSTANT name = value
```

Paramètres:

name

Le nom de la constante. Ceci doit être un OpenSTA valide Dataname .

valeur

Une chaîne de caractères citée ou une valeur de nombre entier.

Exemples:

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

Voyez également:

La Section de DÉFINITIONS

La déclaration des fichiers

Description:

Cette déclaration déclare une marque (ID) pour tous les fichiers externes qui sont consultés par ce script. La déclaration des FICHIERS est obligatoire pour tous les fichiers qui sont passés comme paramètre au script, et facultatif autrement. Il est dans de bonnes habitudes, cependant, de déclarer formellement toutes les identifications de dossier de cette façon avant l'emploi.

Format:

```
        FILE input_fileid
```

Paramètre:

input_fileid

Un OpenSTA Dataname est utilise pour identifier un fichier qui est passé comme paramètre au script.

Exemple:

```
FILE datafile
```

Voyez également:

La Section de DÉFINITIONS

La déclaration des NOMBRE ENTIER

Description:

Ce rapport définit une variable avec une valeur intégrale positive ou négative. Dans le SCL, des nombres entiers sont définis en tant qu'étant 4 bytes de long, donnant une gamme de -2147483648 à +2147483647.

Des choix de variables de nombre entier peuvent être définis, avec un maximum de trois dimensions. Pour de plus amples informations au sujet des rangées, voir les rangées variables .

Format:

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

Paramètres:

Name

Le nom de la variable. Ceci doit être un OpenSTA valide Dataname .

dimensions

Les dimensions de la rangée à assigner pour cette variable. Jusqu'à trois dimensions peuvent être indiquées, séparé par des virgules, chacune comportant un ou duex nombres .

Si une dimension a seulement un nombre, les éléments du fait la gamme de dimension de 1 au nombre a indiqué. Si deux nombres sont indiqués, ils doivent être séparés par des deux points (":"); les éléments dans cette gamme de dimension du premier nombre à la seconde. Notez que si des "dimensions" sont indiquées, les "valeurs" peuvent ne pas être.

valeurs

Une liste ou une gamme des valeurs de nombre entier à associer à la variable.

Notez que si des "valeurs" sont indiquées, les "dimensions" peuvent ne pas être. Pour de plus amples informations sur des valeurs variables, voir les valeurs variables .

options

Une liste d'options variables. Pour de plus amples informations en des options variables, voir les options variables .

Exemples:

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

Voyez également:

La Section de DÉFINITIONS

La déclaration du TEMPORISATEUR(TIMER)

Description:

La déclaration de TEMPORISATEUR déclare le nom d'un temporisateur de chronomètre. Ces temporisateurs peuvent être employés en même temps que les rapports de TEMPORISATEUR de DÉBUT(STARTER TIMER) et de TEMPORISATEUR de fin (END STARTER) dans la section de code du script.

Jusqu'à 1020 temporisateurs peuvent être déclarés et employés dans un script.

Format:

```
        TIMER name
Paramètre:
```

Name

Le nom du temporisateur. Ceci doit être un OpenSTA valide Dataname .

Exemples:

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

Voyez également:

La Section de DÉFINITIONS

Rangées Variables

Des variables de caractère et de nombre entier avouées dans la section de définitions d'un script peuvent être définies en tant que rangées. Choix de soutiens de SCL jusqu'à trois dimensions. Il n'y a aucune limite définie au nombre d'éléments qui peuvent être déclarés dans une dimension de rangée.

Si un choix de deux ou trois dimensions est indiqué, chaque dimension doit être séparée de la dimension suivante par une virgule. Quand une rangée est mise en référence, des indices inférieurs de rangée doivent être indiqués pour chacune de ses dimensions.

La numérotation des éléments de rangée dépend de la façon dont la rangée a été déclarée. Le S CL soutient les valeurs souscrites de rangée de début et de fin dans la convention de matrice elle-même. Par exemple:

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

Tous les deux déclarations variables déclarent un choix de variables de caractère chacune avec 12 éléments. Les éléments dans la rangée sont tous les deux des numéros 1 à 12. Comparez-les à l'exemple suivant:

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

Cet exemple déclare également un choix de 12 éléments, mais les éléments de rangée sont numérotés de 0 à 11.

Seulement des valeurs positives peuvent être indiquées pour les valeurs souscrites de début et de rangée de fin, et la valeur de début doit être inférieur ou égal à la valeur de fin. Si la valeur de début est omise, elle se transfère sur 1.

Quand vous voulez rechercher une valeur d'une variable de rangée, vous pouvez employer des numériques propres , des variables de nombre entier ou des expressions arithmétiques complexes pour indiquer l'element(s). Par exemple:

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

Voyez également:

La Section de DÉFINITIONS

Valeurs Variables(Variable values)

Un ensemble de valeurs peut être associé à un variable, en utilisant une clause de valeur dans la définition variable. Elles sont employées par le *GENERATE* et les *NEXT commands*, qui permettent à la variable d'être assignée une valeur de la liste ou de la gamme, aléatoirement (en utilisant *GENERATE*) ou séquentiellement (en utilisant *NEXT*). Des valeurs peuvent être indiquées comme liste (des nombre entier et des caractère de variables) ou comme gamme (nombres entiers seulement). Note: Les listes peuvent contenir seulement différentes valeurs, et pas gammes.

Variables qui ont été déclarées car une rangée peut ne pas avoir une liste ou une gamme associée de valeur. Une liste de valeur a le format suivant:

```
(value1{, valeur 2, value3... })
```

Les valeurs doivent être du même type de données que la variable, c.-à-d. valeurs de nombre entier pour des variables de nombre entier et des valeurs de caractère pour des variables de caractère. Elles peuvent être des « « lettres » » ou des constantes qui ont été précédemment définies.

Note: Dans le cas des variables de caractère, la taille maximum d'une constante de caractère ou la chaîne littérale(literal string) est 65535 caractères.

Les gammes fournissent une méthode de sténographie pour définir une liste de valeurs adjacentes de nombre entier et ont le format suivant:

```
(start_value - end_value)
```

Si la valeur de début est moins que la valeur de fin, la variable est incrémentée par 1 sur chaque exécution de la PROCHAINE commande, jusqu'à ce que la valeur de fin soit atteinte. Si la valeur de début est plus grande que la valeur de fin, la variable est décrémentée par 1 sur chaque

exécution de la PROCHAINE commande, jusqu'à ce que la valeur de fin soit atteinte.

Si la variable est placée à la valeur de fin quand la PROCHAINE commande est exécutée, la variable sera remise à zéro à la valeur de début. Vous pouvez également remettre à zéro la variable explicitement, en employant *RESET command* .

Dans la liste suivante de définitions variables d'exemple comprenant des valeurs, les deux premières définitions sont équivalentes:

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

Voyez également:

La Section de DÉFINITIONS

Options Variables

Des attributs additionnels peuvent être assignés à une variable en utilisant des clauses d'option. Les options variables suivent les définitions de valeur (si présent), et sont introduites par une virgule. Il y a trois types de clause d'option disponibles: le premier définit la portée de la variable; la seconde est employée avec des variables avec des valeurs associées, pour définir à quel point des valeurs aléatoires doivent être produites, s'il y a lieu; le troisième est employé avec les variables qui sont définies comme paramètre pour le script.

Les sections suivantes décrivent les types de clause d'option de variable.

Voyez également:

Options Variables De Portée

Options De Variable Aléatoire

Option De Dossier

La Section de DÉFINITIONS

Options Variables De Portée(Variable Scope Options)

Les options variables de portée définissent comment largement accessible la variable est; elles sont mutuellement exclusif. Les options variables de portée sont:

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

Ces options sont décrites ci-dessous:

LOCAL

Les variables locales sont seulement accessibles au *thread running* le script dans ledquelles elles sont définies. Elles ne peuvent pas être consultées par aucuns autres *thread* ou script (scripts y compris référencés par le script principal). De même, un script ne peut accéder à aucune des variables locales définies dans aucun de ces scripts qu'il appelle.

L'espace pour des variables locales définies dans un script est assigné quand le script est activé et délocalisé quand l'exécution de script accomplit.

C'est le défaut si aucune option de portée n'est indiquée dans la définition variable.

SCRIPT

Les variables de script sont accessibles à n'importe quel thread courant le script dans lesquelles elles sont définies.

L'espace pour les variables de script définies dans un script est assigné quand le script est activé et il n'y a aucun fil courant actuellement le script. Si un ou plusieurs fils courent déjà le script, les données variables de script existant sont employées.

L'espace pour des variables de script est normalement délocalisé quand l'exécution d'un script se termine, et autre thread ne court pas le script. Dans certains cas, cependant, il peut être souhaitable de maintenir les teneurs des variables de script même s'il n'y a aucun fil accédant au script. Ceci peut être réalisé en employant la clause « ,keepalive » sur la commande *EXIT*. L'espace assigné aux variables de script est seulement supprimé quand un thread est le dernier thread accédant au

script et n'a pas indiqué clause ", keepalive ". Une utilisation particulière de cette clause est où le script s'appelle par un certain nombre de thread , mais il n'y a aucune garantie qu'il y aura au moins un thread accédant au script à tout moment.

THREAD

Les variables de thread sont accessibles de n'importe quel script exécuté par le thread qui déclare un exemple d'elles.

L'espace pour des variables de thread est délocalisé quand le thread est accomplit.

Les variables de thread ne peuvent pas avoir associé des listes de valeur ou des gammes.

GLOBAL

Les variables globales sont accessibles à n'importe quel thread courant n'importe quel script sous le même test manager .L'espace pour des variables globales est désaffecté quand le test manager en question est fermé.Les variables globales ne peuvent pas avoir associé des listes de valeur ou des gammes.

Voyez également:

Options Variables

Options De Variable Aléatoire(Random Variable Options)

Les options aléatoires sont seulement valides pour les variables qui ont un ensemble associé de valeurs; elles sont mutuellement exclusif. Les deux options aléatoires sont:

```
,RANDOM

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

Fonction de ces options comme suit:

RANDOM

Cette option indique qu'une valeur doit être choisie aléatoirement à partir d'une liste ou gamme, de le moment où la variable est employée en même temps que la commande de *GENERATE*. Les valeurs seront choisies dans un ordre différent chaque fois que elles sont produites; ceci est réalisé en produisant d'une valeur différente de multiple variable chaque fois que la variable est

initialisée. Des variables locales sont initialisées quand l'exécution de script commence. Des variables de script sont initialisées par le premier thread pour exécuter le script.

Cette option est particulièrement utile quand vous démarrez un test sur un système.

C'est le défaut si aucune option aléatoire n'est indiquée.

Voyez également:

Options Variables

REPEATABLE { RANDOM }

Cette option indique qu'une valeur doit être choisie aléatoirement à partir d'une liste ou d'une gamme, quand la variable est employée en même temps que la commande *GENERATE*, mais dans le même ordre chaque fois que le script est en cours . Ceci est réalisé en employant la même valeur de variable « multiple »chaque fois que la variable est initialisée.

Cette option est particulièrement utile dans le test de la régression quand l'entrée reproductible est exigée.

SEED = n

Cette option peut être employée en même temps que l'option REPEATABLE RADOM , pour indiquer la valeur multiple qui doit être employée en produisant de l'ordre aléatoire des nombres. Ceci permet pour employer un ordre différent des valeurs aléatoires pour chaque variable aléatoire qu'on peut répéter. "n" est un numérique propre dans la gamme -2147483648 à +2147483647.

Voyez également:

Options Variables

Option De fichier(Files options)

L'option variable de fichier associe un fichier des textes Ascii des valeurs - un par la ligne – avec variable:

```
, FILE = filename
```

où est écrit le "filename" est une chaîne de caractères citée qui définit le nom du fichier de textes Ascii, le nom de chemin et le fichier d’extension . Le fichier doit résider dans le dossier de données principal et avoir .FVR en extension de fichier .

Le dossier est employé par la *NEXT* commande, qui permet à la variable d'être assignée une valeur à partir du fichier séquentiellement.

Des valeurs sont tenues dans le fichier avec une valeur par la ligne. Les valeurs doivent être du même type de données que la variable, c.-à-d. valeurs de nombre entier pour des variables de nombre entier et des valeurs de caractère pour des variables de caractère. Par exemple, un fichier pour une variable de nombre entier a pu contenir les valeurs:

-100
0
100

Un fichier pour une variable de caractère a pu contenir les valeurs:

Pele
10
Cruyff
14

Note: Des commandes de caractère SCL ne sont pas identifiées dans les fichiers variables de fichier - le fichier devrait contenir les caractères crus Ascii seulement.

Des valeurs sont recherchées à partir du fichier lié à une variable en utilisant la *NEXT* commande. Cette commande recherche la prochaine valeur séquentielle à partir du fichier. Quand la *NEXT* commande est d'abord exécutée, elle recherchera la première valeur à partir du fichier. Si la variable est placée à la dernière valeur dans le fichier quand la *NEXT* commande est exécutée, la variable sera remise à zéro à la première valeur dans le fichier. Vous pouvez également remettre à zéro la variable explicitement, en employant la commande de *RESET*.

L'option de fichier n est pas la valable pour les variables qui:

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

Voyez également:

Options Variables

Définitions De Variable D'Exemple

Cette section montre à un certain nombre d'exemple des définitions variables:

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

Nombre entier	Isub	(100,200,300,400)
Nombre entier	ERRCOUNT	, global
Nombre entier	Jsub	(-400, -300, -200), Local, Aléatoires
Nombre entier	B	, Manuscrit, Qu'on peut répéter, Seed=30352
Nombre entier	Prdcod	, File="prd_codes "
Character:24	nom de famille	
Character*10	Alph	("A", "c"," e"), aléatoire qu'on peut répéter
Character*80	Prddsc	, File="prd_descriptions "
Constant	TAXrate =	17,5
Constant	confirmez =	"Confirmez [Y/n]:"

Voyez également:

La Section de DÉFINITIONS

La Section de CODE

La section obligatoire de code du fichier source de SCL contient toutes les commandes qui définissent le comportement du script.

Un fichier de script doit contenir la section (simple) de code comme dernière section dans le fichier. Il est présenté par la commande obligatoire de CODE.

Voyez également:

Codez La Structure De Section

Commandez Les Types

Traitement De Script

Variables

Étiquettes

Symboles

Marques de CORPS de la CHARGE RESPONSE_info

Codez Les Commandes De Section

Codez La Structure De Section(code Section Structure)

La section de code d'un fichier source de SCL se compose de:

Commands

Le SCL fournit un éventail de commandes qui commandent le comportement du script.

Une commande est normalement terminée vers la fin de la ligne de source, mais peut être continuée sur une ligne suivante en indiquant le caractère de suite comme dernier caractère sur une ligne - à part pour n'importe quel commentaire de ligne. Une esperluette ou un trait d'union ("et "ou" -") peut être employée comme caractère de suite; ceci est décrit dans des lignes suite .

Les espaces et des tabs sont traités comme séparateurs dans une commande, bien que les espaces soient significatifs quand ils apparaissent dans des arguments de chaîne de caractères.

Caractères ignorés par le compilateur(characters Ignored by the compiler)

Le compilateur de script permet à n'importe quel caractère avec une valeur Ascii dans la gamme HEX 00 à 20 ou le HEX 81 à 8F inclus d'apparaître au début d'une ligne ou de l'extrémité d'une ligne. Il ignore ces caractères, permettant à des tabs et à des chargements de page, par exemple, d'être employés pour faciliter la lisibilité.

Si n'importe quel caractère de commande Ascii apparaît ailleurs, le compilateur de script produira d'une erreur de compilation.

Voyez également:

La Section de CODE

Types De Commande(Command Types)

Le SCL offre un grand nombre de commandes pour soutenir la création des scripts puissants et flexibles. Ceux-ci entrent dans un certain nombre de catégories distinctes:

- Commandes de HTTP (http Commands)
- Commandes D'Entrée De Jet D'Entrée (Input Stream Entry Commands)
- Jet De Rendement Manipulant Des Commandes (Output Stream Handling Commands)
- Commandes De Commande D'Écoulement (Flow Control Commands)
- Dossier Manipulant Des Commandes (File Handling Commands)
- Commandes De Commande Formelles D'Test (Formal Test Control Commands)
- Commandes De Synchronisation (Synchronization Commands)
- Commandes Statistiques D'Enregistrement De Données (Statistical Data Logging Commands)
- Commandes Diagnostiques Diagnostic Commands
- Commandes Diverses Miscellaneous commands

Voyez également:

La Section de CODE

Traitement De Script(Script processing)

Quand un script est exécuté, la première commande dans le script est choisie et exécutée.

Des commandes sont traitées séquentiellement, à moins qu'une commande qui change l'écoulement de la commande soit exécutée, dans ce cas le traitement continue au point défini dans le script.

Un script se termine quand la fin du script est atteinte, quand une EXIT, ou DÉTACH la commande { THREAD } est exécutée, ou quand une erreur est détectée et le probleme d'erreur n'est pas permis pour le script.

Voyez également:

La Section de CODE

Variables

Toutes les variables ont accédé par un script doivent être prédéfinies dans la section de définitions du script. Si une variable non définie est consultée dans d'un fichier source de SCL, une erreur sera rapportée.

Toutes les variables de nombre entier sont au commencement placées à zéro, et les variables de caractère sont vides

Voyez également:

La Section de CODE

Étiquettes(labels)

Des labels peuvent être employées pour identifier des rapports de SCL. Un label se compose du nom du label suivi des deux points. Par exemple:

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

Un nom de label doit être un OpenSTA valide Dataname .

Aucun sous-programme défini peut ne pas mettre en référence des labels définies dans d'autres sections du code, puisque les labels sont locales au module dans lequel elles sont définies.

Voyez également:

La Section de CODE

Symboles(symbols)

Pendant la compilation, le compilateur maintient des tables de symbole de tous les symboles qu'il a rencontrés, de sorte qu'il puisse résoudre des références par rapport elles. Il y a les tables de symbole séparées pour des variables, des temporisateurs et des labels.

Tous les symboles dans une table de symbole doivent être uniques. Cependant, l'utilisation des tables de symbole séparées permet, par exemple, le même nom d'être employé pour un label quant à une variable.

En outre, parce que des labels ne sont pas propagées dans des sous-programmes ou vice versa, les

labels dans un sous-programme peuvent reproduire des labels dans d'autres sous-programmes, ou dans le corps principal du code.

Voyez également:

La Section de CODE

Marques de CORPS de la CHARGE RESPONSE_info

(LOAD RESPONSE_INFO BODY Identifiers)

La commande de *LOAD RESPONSE_INFO BODY* charge une variable de caractère avec l'ensemble ou une partie des données d'un corps de message de réponse HTTP pour un connexion indiqué en TCP. Pour de réponse *body* contenus dans un document HTML, la clause "WITH" peut être employé pour charger une variable de caractère avec un élément ou partie d'un élément du document.

la clause "WITH" a le format suivant:

,WITH identifier

Note: la marque est une chaîne variable et citée de caractère de caractères ou expression de caractère identifiant les données à rechercher du document de HTML dans le corps de message de réponse. Les sections suivantes décrivent le format de cette marque.

Adressage D'Élément de HTML

Un élément dans un document HTML est identifié par un élément d adresse string .

Format:

tag(tagnum){/tag(tagnum)}:element_type:{attribute}(element_num)

Paramètres:

Tag

The html tag name

tagnum

Un nombre identifiant le tag relatif à son tag de parent ou à la racine de document.

0 = d'abord étiquette d'enfant 0 = First child tag
1 = deuxième étiquette d'enfant 1 = Second child tag
n = nième étiquette d'enfant n = nth child tag

element_type

Le type d'élément de HTML. Ceci doit être l'un de ce qui suit:

ANONYMOUS ATTRIBUTE

ATTRIBUTE

COMMENT

SCRIPT

TEXT

attribut

Pour l'Attribut d'element_type, indique le nom de l'attribut de HTML.

element_num

Un nombre identifiant l'élément. Pour l'Attribut de type d'élément, le nombre identifie l'attribut relativement à son étiquette associée.

0 = premier attribut

1 = second attribut

n = nième attribut

Exemples:

```
HTML(0)/BODY(1)/TABLE(1)/TBODY(0)/TR(0)/TD(0):TEXT:(0)
```

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)
```

Notes:

- Il ne doit y avoir aucun whitespace entre aucun de ces composants d’un identifiant .
- Des identifiant ne sont pas validées au moment de la compilation .

Qualification d'une adresse d'élément de HTML(Qualifying an html Element address)

Une chaine complète d'élément HTML peut être recherchée d'un document de HTML en utilisant une marque contenant seulement une adresse d'élément HTML. Cependant, une sous-chaîne peut être choisie parmi elle employant une variété de qualificateurs. Ces qualificateurs suivent immédiatement l'adresse d'élément HTML et sont décrits ci-dessous.

Choix d'une sous-chaîne par Position et longueur(Selecting a Substring by Position and Length)

Une sous-chaîne d'élément de HTML peut être choisie en utilisant un identifiant spécifique indiquant l'offset la sous-chaîne et sa longueur.

Format:

element_addr[offset, length]

là où "[" et "]" sont les caractères et la partie littérale de la syntaxe exigée.

Paramètres:

element_addr

L'adresse d'élément de HTML dans le format décrit ci-dessus.

Offset(offset)

L’offset du premier caractère de la sous-chaîne dès le début de la chaîne d'élément.

Longueur(length)

Le nombre de caractères dans la sous-chaîne.

Notes:

- Si l'offset est invalide , une chaîne vide est retournée.
- Si la longueur est zéro, ou est invalid, tous les caractères dès le début l'offset à l'extrémité de la chaine d'élément sont retournés.

Exemple:

HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,5]

Choix d'une sous-chaîne en utilisant des délimiteurs(Selecting a Substring using Delimiters)

Une sous-chaîne d'élément HTML peut être choisie en indiquant un identifiant contenant deux délimiteurs de chaîne. La sous-chaîne retournée contient tous les caractères entre la première occurrence du premier délimiteur et la première occurrence de la seconde. La chaîne inclura également les deux chaînes de délimiteur.

Format:

element_addr[delimiter1,delimiter2]

là où "[" et "]" sont les caractères et la partie littérale de la syntaxe exigée.

Paramètres:

element_addr

L'adresse d'élément de HTML dans le format décrit ci-dessus.

delimiter1

Une chaîne - incluse dans des citations simples - identification des caractères au début de la sous-chaîne.

delimiter2

Une chaîne - incluse dans des citations simples - identification des caractères à la fin du

sous-chaîne.

Notes:

- Si delimiter1 ne peut pas être trouvé, une chaîne vide est retournée.

- Si delimiter2 ne peut pas être trouvé, tous les caractères de et inclure delimiter1 à l'extrémité de la chaîne d'élément sont retournés.

Exemple:

HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['document.cookie=',';']

Choix d'une sous-chaîne en utilisant la position, la longueur et la chaîne de délimiteur(Selecting a Substring Using Position, Length and Delimiter String)

Les deux méthodes ci-dessus de choix de sous-chaîne peuvent être combinées, permettant à une sous-chaîne d'élément HTML d'être identifiée par une chaîne de début et une longueur ou un offset et une chaîne d'arrêt.

Format:

element_addr[delimiter1,length]

ou

element_addr[offset,delimeter2]

là où "[" et "]" sont les caractères et la partie littérale de la syntaxe exigée.

Paramètres:

element_addr

L'adresse d'élément de HTML dans le format décrit ci-dessus.

delimiter1

Une chaîne - incluse dans des citations simples - identification des caractères au début de la sous-chaîne.

Longueur(length)

Le nombre de caractères dans la sous-chaîne.

Offset

L'offset du premier caractère de la sous-chaîne dès le début de la chaîne d'élément.

delimiter2

Une chaîne - incluse dans des citations simples - identification des caractères à la fin du

sous-chaîne.

Notes:

- Si delimiter1 ne peut pas être trouvé, une chaîne vide est retournée.
- Si l'offset est invalide, une chaîne vide est retournée.
- Si delimiter2 ne peut pas être trouvé, tous les caractères ensuite, et incluant, delimiter1 à l'extrémité de la chaîne d'élément sont retournés.
- Si la longueur est zéro, ou est invalide, tous les caractères de l'offset indiqué à l'extrémité de la chaîne d'élément sont retournés.

Exemples:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)['cookie=',3]

    HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';']
```

À l'exclusion des délimiteurs de choix(Excluding Delimiters from Selection)

La syntaxe étant décrit ci-dessus, toutes les chaînes de délimiteur indiquées sont incluses dans la sous-chaîne retournée. L'un ou l'autre ou les deux délimiteurs peuvent être exclus de la sous-chaîne retournée le plus presque en inversant le crochet au délimiteur, c.-à-d. à l'aide d'un crochet d'ouverture au lieu d'un crochet se fermant et vice versa.

Cette méthode peut également être employée avec des paramètres offset. Au lieu d'identifier l'offset du premier caractère de la sous-chaîne à choisir, en utilisant cette syntaxe alternative, l'offset devient l'offset du caractère juste avant que le premier caractère à choisir.

Les exemples suivants illustrent comment une sous-chaîne peut être choisie parmi la chaîne d'attribut CONTENT d'une MÉTA tag HTML.

Exemples:

```
        HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';']
```

Choisit la sous-chaîne qui commence à l'offset 3 à partir du commencement de la chaîne d'attribut et qui est terminée par - et inclut - le prochain point-virgule dans la chaîne.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)[2,';'[
```

Choisit la sous-chaîne qui commence à l'offset 2 à partir du commencement de la chaîne d'attribut et qui est terminée par - mais n'inclut pas - le prochain point-virgule dedans

la chaîne.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]2,';'[
```

Choisit la sous-chaîne qui commence à l'offset 3 à partir du commencement de la chaîne d'attribut et qui est terminée par – mais n'inclut pas – le prochain point-virgule dans la chaîne.

Ignorer les caractères au début d'un élément de HTML(Ignoring the Characters at the Beginning of an HTML Element)

Il y a des occasions quand il est utile d'employer les équipements ci-dessus à partir d'un certain point dans la chaîne d'élément, plutôt qu'au début de la chaîne. Ceci peut être réalisé en remettant à zéro la base de choix. Ceci peut être fait en indiquant la base de choix comme offset du commencement de la chaîne d'élément, ou en indiquant une sous-chaîne qui identifie les caractères au début de la sous-chaîne à examiner. L'offset ou la sous-chaîne est précédé par un de deux opérateurs ">" ou de ">=":

> offset

L'offset est celui du caractère juste avant que la sous-chaîne à examiner.

> substring

La sous-chaîne identifie les caractères à l'extrémité de la chaîne à ignorer. La sous-chaîne commence par le premier caractère après la sous-chaîne.

> = offset

L'offset est celui du premier caractère dans la sous-chaîne à examiner.

> = substring

La sous-chaîne identifie les caractères au début de la sous-chaîne à examiner.

Note:

Si l'offset ou la sous-chaîne ne peut pas être trouvé, une chaîne vide est retournée.

Les exemples suivants illustrent comment la base de choix est remise à zéro pour un choix à partir de la chaîne d un attribut CONTENT d'une MÉTA tag HTML.

Exemples:

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>'//
Cookie','document.cookie=',';']
```

L'offset de base de choix est placé à l'offset du premier caractère après la première occurrence de la chaîne "// Cookie "dans la chaîne d'élément. La sous-chaîne choisie commence par le caractère après "document.cookie =" et finie avec - et inclut - le prochain point-virgule.

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>='//
Cookie','document.cookie=',';']
```

Même chose qu'en haut, sauf que l'offset de base de choix est maintenant le premier caractère du "// cookie ".

```
HTML(0)/HEAD(0)/META(1):ATTRIBUTE:CONTENT(1)]>=50,'document.cookie=','
;']
```

Même chose qu'en haut, sauf que l'offset de base de choix est maintenant 50 caractères dès le début de la chaîne d'élément.

Ignorer la caisse de caractères(Ignoring the Case of Characters)

Toutes les comparaisons de chaîne indiquées par des marques de LOAD RESPONSE_INFO BOSY sont par le cas de défaut sensible. La caisse de caractères peut être ignorée dans les comparaisons en mettant en tête la chaîne de recherche ou la chaîne de délimiteur par "I".

Exemple:

```
   /de HTML(0)/head(0)/meta(1):attribute:content(1)]>i '/Cookie', I'document.
cookie =',';']
```

La base de choix est remise à zéro en recherchant la chaîne d'élément "// cookie "; le bloc de caractères est ignorée dans la recherche.

Indication Des Citations Dans Des Marques(Specifying Quotes Within Identifiers)

Des chaînes de caractères citées dans le SCL sont délimitées, par des citations simples ou par de doubles citations. Puisque la syntaxe d'une marque de LOAD RESPONSE_INFO BODY inclut des citations simples, on lui recommande que des citations de double soient employées pour délimiter une chaîne de caractères citée contenant une telle marque.

Un caractère littéral de citation simple peut être inclus dans une chaîne de marque en la précédant avec un antislash. Par exemple:

```
"HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'\'']"
```

Ceci choisit une sous-chaîne terminée par une citation simple.

Un double caractère littéral de citation peut être indiqué dans une chaîne de marque, en utilisant la commande de caractère de SCL, ~<22 >. Par exemple,

```
"HTML(0)/HEAD(0)/META(1):ATTRIBUTE:XYZZY(1)[0,'~<22>']"
```

Ceci choisit une sous-chaîne terminée par une double citation.

Voyez également:

La Section de CODE

Code des Commandes De Section(Code Section Commands)

Cette section décrit les commandes qui peuvent être incluses dans la section de code d'un fichier script.

La section de code peut également contenir des labels et des commentaires. Davantage d'information sur ces articles est fournie dans la vue d'ensemble de la syntaxe d'ordres de gestion de script .

Référez-vous à la section de commandes de HTTP pour information concernant les commandes qui peuvent être employées avec le HTTP.

Voyez également:

Commandes de HTTP (http Commands)

Commandes D'Entrée De Jet D'Entrée (input Stream entry commands)

Jet De Rendement Manipulant Des Commandes (output Stream Handing Commands)

Commandes De Commande D'Écoulement (Flow control Commands)

Dossier Manipulant Des Commandes (file Handing commands)

Commandes De Commande Formelles D'Test (Formal Test Control commands)

Commandes De Synchronisation (Synchronisation Commands)

Commandes D'Entrée De Jet D'Entrée (input Stream Entry Commands)

Commandes Statistiques D'Enregistrement De Données (Statistical Data logging Commands)

Commandes Diagnostiques (diagnostic commands)

Commandes Diverses (Commands diverses)

La Section de CODE (la section code)

Commandes HTTP

Les commandes HTTP fournissent des équipements pour publier des requêtes HTTP des ressources, *examining/ interrogating* les messages de réponse et synchronisant des requêtes. Ces commandes sont seulement disponibles en scripts qui contiennent la déclaration HTTP MODE dans leur section d'environnement.

Les commandes de HTTP sont comme suit:

- RELIEZ La Commande (CONNECT Command)
- Commande de DÉCONNEXION (DISCONNECT command)

- OBTENEZ La Commande (GET Command)
- Commande PRINCIPALE (HEAD COMMAND)
- Commande de CORPS de la CHARGE RESPONSE_info (LOAD RESPONSE_INFO BODY Command)
- Commande d'En-tête de la CHARGE RESPONSE_info (LOAD RESPONSE_INFO HEADER Command)
- Commande de POTEAU (POST Command)
- SYNCHRONISEZ La Commande de REQUÊTES (SUNCHRONIZE REQUESTS Command)

Commandes D'Entrée De Jet D'Entrée(Input Stream Entry Commands)

L'entrée de jet d'entrée commande la contrôle de quelle manière le script alimente l'entrée au système en test.

Voyez également:

PRODUISEZ De la Commande

OBTENEZ La Commande

DIRIGEZ La Commande

PROCHAINE Commande

Commande de POTEAU

REMETTEZ À ZÉRO La Commande

PLACEZ La Commande

PRODUIRE De la Commande(GENERATE Command)

Description:

Cette commande charge une valeur aléatoire d'un ensemble de valeurs dans une variable.

La variable doit avoir une liste ou gamme des valeurs liées à elle dans la section de définitions. Si elle est définie comme " REPEATABLE RANDOM", des valeurs seront recherchées dans le même ordre aléatoire sur chaque démarrage . Si elle est définie comme " RANDOM", des valeurs seront recherchées dans différents ordres aléatoires sur chaque démarrage .

Format:

```
GENERATE variable
```

Paramètre:

variable

Le nom de la variable dans laquelle la valeur produite doit être chargée. La variable doit avoir un ensemble de valeurs liées à elle dans la section de définitions.

Exemple:

```
GENERATE Part-Number
```

Voyez également:

Commandes D'Entrée De Jet D'Entrée

OBTENIR La Commande(GET Command)

Description:

Cette commande publie un *HTTP GET* la requête d'une ressource indiquée. Elle est seulement valide dans un script qui a été défini comme HTTP MODE.

Le mot-clé *PRIMARY* facultatif dénote des requêtes primaires HTTP comme ceux visés par l'en-tête de "*referer*" dans des requêtes secondaires. Par exemple:

Une requête retirant un HTML PAGE d'un web server peut être suivie des requêtes retirant quelques images de GIF dont l'URL sont contenus dans la page indiquée.

Les champs d'en-tête de requête sont obtenus à partir de la clause *HEADER*. Ceux-ci peuvent être modifié en employeant les clauses *WITH* et *WITHOUT* .

La requête HTTP GET est asynchrone. Juste après que la requête est publiée, la prochaine

commande dans le script est traitée - elle n'attend pas un message de réponse à recevoir.

Un certificat du client peut être indiqué dans une requête le dossier ou de nom en employant les clauses " CERTIFICATE FILE " et " CERTIFICATE NAME ".

Il y a une clause facultative " RESPONSE TIMER ", qui peut être employée pour indiquer qu'une paire d'enregistrements du TIMER réponse doivent être écrites dans les logs de statistiques. Le premier enregistrements est écrit quand le message de requête est envoyé, et la seconde est écrite à la réception du message de requête de réponse du serveur.

Le code de réponse dans le message de réponse peut être recherché en employant la clause facultative " RETURNING CODE response_code " pour indiquer la variable de nombre entier pour tenir le code de réponse. La variable est chargée quand le message de réponse est reçu du serveur. En outre, la clause facultative " RETURNING STATUS response_status " peut être employée pour indiquer la variable de nombre entier pour tenir une de deux valeurs indiquant si la requête a réussi ou a échoué. Il y a un SCL incluent le fichier " response_codes.inc " fourni avec OpenSTA, qui définit des constantes de nombre entier SCL pour les *response code* et les *response status values*

La connexion TCP utilisé pour la requête dépend au moment si la connexion a été déjà établi pour l'identification indiquée de la connexion en utilisant la commande *CONNECT*. S'il a, les requêtes utilisent la connexion. S'il n'a pas, la connexion TCP sera établi au centre serveur identifié par l'uri-httpversion, sur le port 80.

Par défaut, si une erreur se produit tout en établissant le la connexion TCP ou publiant la requête, un message d'erreur sera écrit au log d'audit et la connexion sera avorté. Cependant, si la découverte d'une erreur est permise , le contrôle sera transférée au code *error-handling*.

Format:

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

Paramètres:

uri-httpversion

une variable de caractère, citant une chaîne de caractère ou une expression de caractère , contenant l'Uri (Uniform Resource Identifier) de la ressource sur laquelle pour appliquer la requête , et la version HTTP, séparée par un caractère d'espace simple. La version HTTP indique le format du message et de la capacité de l'expéditeur pour comprendre davantage la communication HTTP.

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant l'identification de la connexion ID du TCP sur lequel on publie la requête .

http_header

Chaîne variable ,citée une chaîne de caractères, expression de caractère ou liste de valeur de caractère contenant *the request header fields.*

header_value

Chaîne variable et citée d'une chaîne de caractères, liste d'expression de caractère ou de valeur de caractère contenant zéro ou plus *request header fields*. Ces champs de *request-header* sont ajoutés à ceux indiqués dans "http_header". si un champ de requête d'en tête(*request-header)* apparaît "http_header" et "header_value", le champ indiqué ici dépasse celui indiqué dans le "http_header".

header_field

Chaîne variable et citée d'une chaîne de caractères, expression de caractère ou liste de valeur de caractère contenant les noms de champ de requête d'en tête(*request-header)* des champs à exclure de la requête .

cert_filename

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant le nom d'un fichier . Le fichier contient un certificat de client(client certificate).

cert_name

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant un nom de certificat d'un client.

timer_name

Le nom d'un temporisateur(*timer*) a déclaré dans la section de définitions du script.

response_status

Une variable de nombre entier dans laquelle le statut de réponse du message de la réponse HTTP est chargé quand le message de réponse HTTP est reçu.

response_code

Une variable de nombre entier dans laquelle le code de réponse du message de la réponse HTTP est chargé quand le message de réponse HTTP est reçu.

Exemples:

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

Voyez également:

Commandes D'Entrée De Jet D'Entrée

Commande PRINCIPALE(HEAD command)

Description:

Cette commande publie une requête http HEAD d'une ressource indiquée. Elle est seulement valide dans un script qui a été défini comme MODE HTTP.

Le mot-clé PRIMARY facultatif dénote des requêtes primaires HTTP comme ceux visés par l'en-tête "referer" *(« referer » header)* dans des requêtes secondaires. Par exemple:

Une requête retirant un HTML PAGE d'un web server peut être suivie des requêtes retirant quelques images de GIF dont l'URL sont contenus dans la page indiquée.

Les champs d'en-tête de requête sont obtenus à partir de la clause d'cEn-tête. Ceux-ci peuvent être employer modifié AVEC et SANS des clauses.

La requête *http head* est asynchrone. Juste après que la requête soit publiée, la prochaine commande dans le script est traitée - elle n'attend pas un message de réponse à recevoir.

Un certificat de client peut être indiqué dans une requête soit par fichier, soit par nom en employant les clauses " CERTIFICATE FILE " et " CERTIFICATE NAME ".

Il y a une clause facultative " RESPONSE TIMER ", qui peut être employée pour indiquer qu'une paire d'enregistrement de temporisateur(timer) de réponse doivent être écrites dans statistiques log . Le premier enregistrement est écrit quand le message de requête est envoyé, et le second est écrit à la réception du message de requête de réponse du serveur.

Le code de réponse dans le message de réponse peut être recherché en employant la clause facultative " RETURNING CODE response_code " pour indiquer la variable de nombre entier pour tenir le code de réponse. La variable est chargée quand le message de réponse est reçu du serveur. En outre, la clause facultative " RETURNING STATUS response_status " peut être employée pour indiquer la variable de nombre entier pour tenir une des deux valeurs indiquant si la requête a réussi ou a échoué. Il y a un SCL incluent le fichier "response_codes.inc" fourni avec OpenSTA, qui définit des constantes de nombre entier SCL pour les réponses de valeurs de statut de code et les codes de réponse.

la connexion TCP utilisé pour la requête dépend au moment si la connexion a été déjà établi pour l'identification indiquée de la connexion en utilisant la commande de CONNECT. S'il a, les requêtes utilisent la connexion. S'il n'a pas, la connexion TCP sera établi au centre serveur identifié par l'uri-httpversion, sur le port 80.

Par défaut, si une erreur se produit tout en établissant la connexion TCP ou publiant la requête, un message d'erreur sera écrite dan le log d'audit et la connexion sera avorté. Cependant, si la détection d'erreur est permis, la control sera transférée au code error-handling.

Format:

```
{PRIMARY} HEAD [ URI | URL ] uri-httpversion{&}
        ON conid{&}
                HEADER http_header {&}
                {,WITH header_value} {&}
                {,WITHOUT header_field}{&}
```

```
{,CERTIFICATE FILE cert_filename}{&}
{,CERTIFICATE NAME cert_name}{&}
{,RESPONSE TIMER timer_name} {&}
{,RETURNING STATUS response_status} {&}
{,RETURNING CODE response_code}
```

Paramètres:

uri-httpversion

Chaîne variable et chaîne de caractères citée ou expression de caractère, contenant l'Uri (Uniform Resource Identifier) de la ressource sur laquelle pour appliquer la requête, et la version HTTP, séparée par un caractère d'espace simple. La version HTTP indique le format du message et de la capacité de l'expéditeur pour comprendre davantage la communication HTTP.

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant la connexion ID de la connexion TCP sur lequel on publie la requête.

http_header

Chaîne variable et chaîne de caractères citée, expression de caractère ou liste de valeur de caractère contenant les champs requêter-en-tête(the request-header fields).

header_value

Un caractère variable, chaîne de caractères citée, liste d'expression de caractère ou de valeur de caractère contenant zéro champs ou plus de requête-en-tête. Ces champs d'en-tête de requête sont ajoutés à ceux indiqués dans "http_header". Si un champ d'en-tête de requête apparaît dans "http_header" et "http_value", le champ indiqué ici dépasse celui indiqué dans "http_header".

header_field

Chaîne variable et chaîne de caractères citée, expression de caractère ou liste de valeur de caractère contenant les noms de champ d'en-tête de requête des champs à exclure de la requête.

cert_filename

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant le nom d'un fichier. Le fichier contient un certificat de client.

cert_name

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant un nom de certificat de client.

timer_name

Le nom d'un temporisateur*(timer)*a déclaré dans la section de définitions du script.

response_status

Une variable de nombre entier dans laquelle le statut de réponse du message de HTTP est chargé quand le message de réponse HTTP est reçu.

response_code

Une variable de nombre entier dans laquelle le code de réponse du message de réponse HTTP est chargé quand le message de réponse HTTP est reçu.

Exemples:

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

Voyez également:

Commandes D'Entrée De Jet D'Entrée

PROCHAINE Commande(NEXT Command)

Description:

Cette commande charge une variable avec la prochaine valeur séquentielle d'un ensemble de valeurs. Ceci pourrait être une liste ou une gamme liée à cette variable, ou à partir d'un fichier qui est associé à la variable.

Quand la commande *NEXT* est d'abord exécutée, elle recherchera la première valeur. L'ensemble est traité en tant que répétition: quand la dernière valeur a été recherchée, la prochaine valeur recherchée sera la première dans l'ensemble.

Cette commande peut être employée pour remettre à zéro l'indicateur de valeur lié à une variable de sorte que la première commande *NEXT* est exécuté après que RESET retrouve la première valeur dans l'ensemble.

La variable doit avoir un ensemble de valeurs ou d'un fichier lié à elle dans la section de définitions.

Format:

```
NEXT variable
```

Paramètre:

variable

Le nom d'une variable dans laquelle la prochaine valeur de l'ensemble est chargée. La variable doit avoir un ensemble de valeurs ou d'un fichier lié à elle dans la section de définitions.

Exemple:

```
NEXT Emp-Name
```

Voyez également:

Commandes D'Entrée De Jet D'Entrée

Commande de « POTEAU »(POST Command)

Description:

Cette commande publie une requête HTTP POST d'une ressource indiquée. Elle est seulement

valide dans un script qui a été défini comme HTTP MODE.

Le mot-clé PRIMARY facultatif dénote des requêtes primaires HTTP comme ceux visés par l'en-tête de "referer" dans des requêtes secondaires. Par exemple:

Une requête retirant un HTML PAGE d'un web server peut être suivie des requêtes retirant quelques images de GIF dont l'URL sont contenus dans la page indiquée.

Les champ de requête en-têtes à employer dans la requête sont obtenus à partir de la clause d'En-tête, convenablement modifiée par des clauses WITH et WITHOUT, si specifié.

La requête HTTP POST est asynchrone. Juste après que la requête soit publiée, la prochaine commande dans le script est traitée - elle n'attend pas un message de réponse à recevoir.

Un certificat de client peut être indiqué dans une requête le dossier ou de nom en employant " CERTIFICATE FILE " et les clauses " CERTIFICATE NAME ".

Il y a une clause facultative " RESPONSE TIMER ", qui peut être employée pour indiquer qu'une paire d'enregistrement de temporisateur(timer) de réponse doivent être écrites à la statistiques log . Le premier enregistrement sera écrit quand le message de requête est envoyé, et la second sera écrit à la réception du message de requête de réponse du serveur.

Le code de statut dans le message de réponse peut être recherché en employant la clause facultative " RETURNING CODE response_code " pour indiquer la variable de nombre entier pour tenir le code de réponse. La variable est chargée quand le message de réponse est reçu du serveur. En outre, la clause facultative " RETURNING STATUS response_status " peut être employée pour renvoyer un de deux valeurs indiquant si la requête a réussi ou a échoué. Il y a un SCL incluent le fichier "response_codes.inc" fourni avec OpenSTA, cela définit des constantes de nombre entier de SCL pour de valeurs de statut de code et les codes de réponse.

la connexion TCP utilisé pour la requête dépend au moment si la connexion a été déjà établi pour l'identification indiquée de la connexion en utilisant la commande de CONNECT. S'il a, les requêtes utilisent la connexion. S'il n'a pas, la connexion TCP sera établi au centre serveur identifié par l'uri-httpversion, sur le port 80.

Par défaut, si une erreur se produit tout en établissant la connexion TCP ou publiant la requête, un message d'erreur sera écrite dan le log d'audit et la connexion sera avorté. Cependant, si la détection d'erreur est permis, la control sera transférée au code error-handling.

Format:

```
{PRIMARY}        POST [ URI | URL ] uri-httpversion {&}
           ON conid {&}
                   HEADER http_header {&}
           {,{BINARY} BODY http_body} {&}
           {,WITH header_value} {&}
           {,WITHOUT header_field} {&}
                          {,CERTIFICATE FILE cert_filename} {&}
           {,CERTIFICATE NAME cert_name}{&}
           {,RESPONSE TIMER timer_name} {&}
           {,RETURNING STATUS response_status} {&}
           {,RETURNING CODE response_code}
```

Paramètres:

uri-httpversion

Chaîne variable et chaîne de caractères citée ou expression de caractère, contenant l'Uri (Uniform Resource Identifier) de la ressource sur laquelle pour appliquer la requête, et la version HTTP, séparée par un caractère d'espace simple. La version HTTP indique le format du message et de la capacité de l'expéditeur pour comprendre davantage la communication http.

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant la connexion ID de la connexion TCP sur lequel est publie la requête.

http_header

Chaîne variable , chaîne de caractères citée, expression de caractère ou liste de valeur de caractère contenant les champs d'en-tête de requête.

http_body

Chaîne variable , chaîne de caractères citée ou expression de caractère contenant le corps de la requête.

header_value

Chaîne variable , chaîne de caractères citée, liste d'expression de caractère ou de valeur de caractère contenant zéro ou plus champs d'en-tête de requête. Ces champs d'en-tête de requête sont ajoutés à ceux indiqués dans "http_header". Si un champ d'en-tête de requête apparaît dans "http_header" et "http_value", le champ indiqué ici dépasse cela indiqué dans "http_header".

header_field

Chaîne variable , chaîne de caractères citée, expression de caractère ou liste de valeur de caractère contenant les noms de champ d'en-tête de requête des champs à exclure de la requête.

cert_filename

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant le nom d'un fichier . Le fichier contient un certificat de client.

cert_name

Un caractère variable, chaîne de caractères citée, expression de caractère, contenant un nom de certificat de client.

timer_name

Le nom d'un temporisateur(timer) a déclaré dans la section de définitions du script.

response_status

Une variable de nombre entier dans laquelle le statut de réponse du message de réponse de HTTP est chargé quand le message de réponse de HTTP est reçu.

response_code

Une variable de nombre entier dans laquelle le code de réponse du message de réponse HTTP est chargé quand le message de réponse HTTP est reçu.

Exemples:

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

POST URL "http://dogbert.abebooks.com/abe/IList HTTP/1.0" on
SEARCH_PAGE &
        HEADER post_header &
        ,WITH ("Host: dogbert.abebooks.com", &
                "Referer: http://dogbert.abebooks.com/abe/IList") &
        ,BODY "bu=New+Search"
 POST URI "http://abc.com/~pascal/don.gif HTTP/1.0" ON 2 &
```

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

Voyez également:

Commandes D'Entrée De Jet D'Entrée

REMETTREÀ ZÉRO La Commande(RESET Command)

Description:

Cette commande remet à zéro l'indicateur de valeur pour une variable à la première valeur dans l'ensemble de valeur associé. Ceci pourrait être une liste ou une gamme liée à cette variable, ou à partir d'un fichier qui est associé à la variable. Dans le cas d'une variable aléatoire qu'on peut répéter, la variable multiple peut être remise à zéro à une valeur indiquée ou transférée.

La commande RESET ne change pas le contenu de la variable. La valeur à laquelle la variable a été remise à zéro est seulement recherchée sur l'exécution de la première commande NEXT après la commande de RESET.

Format:

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

Paramètres:

variable

Le nom de la variable dont l'indicateur de valeur doit être remis à zéro. La variable doit avoir un ensemble ou un fichier lié à elle dans la section de définitions.

valeur

Un nombre numérique entier littéral dans la gamme -2147483648 à +2147483647. Si la clause "SEED" est omise de la commande de RESET, la variable de Seed sera remise à zéro à la valeur indiquée quand la variable a été définie, ou à la valeur a indiqué par une commande précédente de RESET.

Exemples:

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

Voyez également:

Commandes D'Entrée De Jet D'Entrée

PLACE La Commande(Set Command)

Description:

Cette commande permet à une valeur d'être assignée à un nombre entier ou à une variable de caractère. Les valeurs peuvent être toutes les valeurs de nombre entier ou de caractère ou une référence de fonction, mais leurs types de données doivent assortir cela de la variable. Les valeurs peuvent être dérivées en raison des opérations arithmétiques.

Si la variable est une variable de nombre entier, l'expression de tâche peut être une autre variable de nombre entier ou une littéralement numérique, ou une expression arithmétique complexe se composant de deux valeurs ou plus de nombre entier ou variables, chacune séparée par un opérateur. Les opérateurs suivants sont soutenus:

+	pour l'addition
-	pour la soustraction
*	pour la multiplication
/	pour la division
%	pour le modulo
et	au niveau du bit ET
|	au niveau du bit inclus OU
^	au niveau du bit exclusivité OU

+	for addition
-	for subtraction
*	for multiplication
/	for division
%	for modulo
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR

La valeur résultant d'une opération de division sera un nombre entier, c.-à-d. le reste sera ignoré. Le calcul de modulo est l'inverse de cette opération, c.-à-d. la variable sera placée à la valeur du reste. Par exemple:

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

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

Si B = 13 et C = 2, alors A sera placé à 6 et à D à 1.

Des parenthèses peuvent être indiquées pour déterminer l'ordre de la priorité. Si des parenthèses ne sont pas indiquées, alors l'expression est évaluée de gauche à droite sans l'autre ordre de la priorité appliqué.

Le soin devrait être pris en utilisant des expressions arithmétiques, puisqu'il n'y a aucun contrôle pour le débordement de nombre entier au temps d'exécution. Si un débordement de nombre entier se produit une erreur de script sera rapportée.

Si la variable est une variable de caractère, l'expression de tâche peut se composer d'une ou plusieurs variables ou coquilles de caractère. Des opérandes sont séparés par l'opérateur d'addition si les opérandes doivent être ajoutés ensemble; si le deuxième opérande doit être soustrait dès le début, ils sont séparés par l'opérateur de soustraction.

La fonction ~EXTRACT de caractère peut être mise en référence dans une commande SET d'extraire une sous-chaîne à partir d'une chaîne variable ou citée de caractère de caractères dans une variable de caractère.

La fonction ~LOCATE de nombre entier peut être mise en référence dans une commande SET de charger l'offset d'une sous-chaîne dans une chaîne variable ou citée de caractère de caractères dans une variable de nombre entier.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label à laquelle la commande devrait être transférée en cas d'une erreur. Une erreur pourrait se produire si, par exemple, une fonction de ~EXTRACT est indiquée avec un offset inadmissible, ou une tentative est faite de se diviser par zéro.

Format:

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

Paramètres:

variable

Le nom d'un nombre entier ou d'une variable de caractère dans lesquels le résultat de l'opération doit être placé.

operand1

La valeur dont le résultat initial d'opération sera pris. Pour une commande de SET de caractères, l'opérande(operand) peut être une chaîne variable , chaîne de caractères citée ou référence de fonction de caractère. Pour des commandes SET de nombre entier, l'opérande peut être une référence de nombre entier littéral ou une variable.

opérator

L'opération qui doit être exécutée sur le précédent et suivant des opérandes. Pour des commandes de SET de caractères, elle peut être "+" ajouter le premier opérande à la seconde, ou "-" pour soustraire le deuxième opérande dès le début. Pour des commandes SET de nombre entier, tous les opérateurs sont valides.

opérand

La variable ou la valeur qui sont employées pour modifier la valeur courante pour la "variable". Pour une commande SET de caractères, l'opérande peut être une chaîne variable, chaine de caractères citée ou référence de fonction de caractère. Pour des commandes SET de nombre entier, l'opérande peut être un nombre entier littéral ou une variable.

err_label

un label définie dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemples:

```
SET STRING1 = STRING2 - "ERROR"
SET STRING1 = STRING2 + STRING3 + STRING4
SET STRING1 = STRING2 - '"END MARKER"' &
            ON ERROR GOTO Error_report
```

Voyez également:

Commandes D'Entrée De Jet D'Entrée

Jet De Rendement Manipulant Des Commandes(output Stream Handling Commands)

output Stream Handling Commands de contrôle comment les scripts examinent et maninpulent le rendement du système, dans le script lui-même ou en sauvegardant les données pour la comparaison postérieure.

Voyez également:

CONVERTISSEZ La Commande (CONVERT Command)

Commande de ~extract (~EXTRACT Command)

Commande de FORMAT (FORMAT Command)

Commande de CORPS de la CHARGE RESPONSE_info (LOAD RESPONSE_INFO BODY Command)

Commande d'cEn-tête de la CHARGE RESPONSE_info (LOAD RESPONSE INFO HEADER Command)

Commande de ~locate (~LOCATE Command)

commande de CONVERTION(CONVERT Command)

Description:

Cette commande permet à la valeur dans une variable de nombre entier d'être convertie à une chaîne Ascii, ou vice versa. La radix de défaut pour la conversion est 10, mais ceci peut être dépassé en incluant une clause de "RADIX" dans la commande.

Pour des conversions de nombre-à-caractère, des options de format peuvent être indiquées. Ces options peuvent causer la chaîne Ascii d'être laissées ou droit-justifiées dans l'amortisseur de rendement(output buffer), ou avoir de principaux zéros ou espaces, ou rendez la conversion signée ou non signée.

Dans la description de format ci-après, le caractère "|" indiquent des options de mutuellement exclusif.

Les options par défaut sont SIGNED et LEFT JUSTIFY. Si RIGHT JUSTIFY est en fonction, le defaut remplissant is LEADING ZEROS.

Si l'amortisseur de rendement(output buffer) est trop petit pour tenir la chaîne de rendement, il sera rempli de caractères de d'astérisque ("*").

Pour des conversions de caractère-à-nombre entier, menant et traînant des espaces sont eNewlineevée de la chaîne Ascii avant la conversion. Des spécifications d'une chaîne non-numérique Ascii, ou d'une chaîne Ascii qui est convertie en numérique en dehors de la gamme d'un *interger*4*, causeront un message pour être noté au fichier de contrôle indiquant une chaîne de caractères inadmissible pour la convertion. La connexion (thread) sera avorté.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label au quelle la commande devrait être transférée en cas d'une erreur.

Format:

```
CONVERT variable1 TO variable2 {&}
    {,[SIGNED][UNSIGNED} {,LEADING [ZEROS]|[SPACES]} {&}
    {,[LEFT]|[RIGHT] JUSTIFY} {,RADIX=radix} {&}
    {,ON ERROR GOTO err_label}
```

Paramètres:

variable1

Une variable contenant la variable à convertir.

variable2

Une variable dans laquelle la variable convertie doit être placée.

radix

Une variable de nombre entier ou littéral dans la gamme 2 à 36.

err_label

un label défini dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemples:

```
 CONVERT Number To String
 CONVERT Number To Employee-Code, RIGHT JUSTIFY
 CONVERT Ascii-code To Numeric_code
CONVERT Ascii-code To Hex_code, RADIX=16, &
        ON ERROR GOTO Conv_error
```

Voyez également:

Produisez Le Jet Manipulant Des Commandes

Commande de ~extract(~Extract command)

Description:

Cette commande est une fonction et peut seulement être mise en référence dans une commande SET. Elle renvoie la partie de la chaîne de source identifiée par l'offset et la longueur indiqués.

Si la chaîne identifiée par l'offset et la longueur recouvre l'extrémité de la chaîne de source, seulement les caractères jusqu'à l'extrémité de la chaîne de source seront retournés.

Si l'offset ne se trouve pas en dessous des limites de la chaîne de source quand le script est exécuté, un message sera écrit au log d'audit, indiquant qu'une mauvaise valeur de paramètre a été indiquée. L'exécution de script sera alors avortée, ou l'action indiquée sera prise si la détection d'erreur est permise via *ON ERROR command*.

Format:

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

Return value :

La sous-chaîne de caractère extraite à partir de la chaîne de source.

Paramètres:

offset

Une variable ou une valeur de nombre entier définissant l'offset dans la chaîne de source du premier caractère qui doit être extrait. Le premier caractère de la chaîne de source est à l'offset zéro.

Length

Une variable ou une valeur de nombre entier définissant le nombre de caractères pour extraire pour former la sous-chaîne.

String

La valeur de caractère ou la variable de caractère à partir duquel la sous-chaîne doit être extraite.

Exemple:

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

Voyez également:

Jet De Rendement Manipulant Des Commandes

Commande de FORMAT(FORMAT command)

Description:

Cette commande traduit des caractères d'un format en des autres. Ceci le facilite pour manipuler les chaînes de caractères qui ont été produites du système en test , ou qui doivent être entrées dans ce système.

Dans toutes les traductions, la commande exige trois éléments:

La variable de cible qui recevra la valeur traduite. Ceci peut être un caractère variable ou une variable de nombre entier.

Une chaîne de format définissant le type de traduction requis. Pour une variable de cible de nombre entier, la chaîne de format doit seulement contenir une un simple de format d'identification ; pour une variable de caractère, la chaîne de format peut contenir des

identifications multiples et/ou les caractères ordinaires qui doivent être copiés sans changement à la variable de cible.

Une ou plusieurs valeurs à traduire; celles-ci peuvent être indiquées comme variables ou en tant que texte littéral. Une valeur simple doit être indiquée pour chacune des format d' identifications dans la chaîne de format; le type de données de chacun doit être conforme à la format d'identification associée et au type de données de la variable de cible, comme discuté ci-dessous. Notez que toutes les anomalies à cet égard sont détectées au temps d'exécution et ne sont pas sélectionnées par le compilateur.

Les types suivants de traduction sont supporté :

%U - traduisez chaque lettre dans la chaîne d'entrée en son équivalent majuscule. Les variables de source et de cible doivent être des variables de caractère. La chaîne de source au besoin est tronquée pour adapter la variable de cible.

%L - traduisez chaque lettre dans la chaîne d'entrée en son équivalent minuscule. Les variables de source et de cible doivent être des variables de caractère. La chaîne de source au besoin est tronquée pour adapter la variable de cible.

%D - convertissez une valeur de chaîne de date de caractères en format numérique (représentant le nombre de jours depuis la date de référence Smithsonienne de 17-Nov-1858). La variable de cible doit être une variable de nombre entier, et la variable de source une chaîne de caractères contenant une date valide; ceci peut être l'un ou l'autre dans le modèle de défaut pour la plateforme sur laquelle le script fonctionne ou dans le format fixe " DD-MMM-CCYY " (où "CC" est facultatif).

Ce format d' identification peut également être employée pour convertir une valeur numérique de date (représentant le nombre de jours depuis la date de référence Smithsonienne de 17-Nov-1858) en chaîne de caractères dans le format fixe " DD-MMM-CCYY ". La variable de source doit être une variable de nombre entier et la variable de cible une chaîne de caractères, qui sera tronquée au besoin.

%T - convertissez une valeur de temps de chaîne de caractères en format numérique (la représentation du nombre de 10 millisecondes faitde « tic tac »depuis le minuit). La variable de cible doit être une variable de nombre entier, et la variable de source une chaîne de caractères contenant un temps valide; ceci peut être l'un ou l'autre dans le modèle de défaut pour la plateforme sur laquelle le script fonctionne ou sous la forme " HH:MM:SS.MMM " (où "MMM" est facultatif).

Cette marque de format peut également être employée pour convertir une valeur numérique de

temps (représentant le nombre de 10 millisecondes de ticks depuis le minuit) en chaîne de caractères dans le format fixe " HH:MM:SS.MMM ". La variable de source doit être une variable de nombre entier et la variable de cible une chaîne de caractères, qui sera tronquée comme exigée.

Format:

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

Paramètres:

Target-variable

Le nom d'un nombre entier ou d'une variable de caractère dans lesquels le résultat de l'opération est placé.

format-string

Une chaîne de caractères citée contenant la chaîne pour être composé et contenant un certain nombre de format d'identification. Les format d'identifications doivent être compatibles avec les types de données des variables qui suivent.

variable

Une ou plusieurs nombre entier ou variables de caractère ou littéral à traduire. Le nombre de variables doit correspondre au nombre de marques de format dans la chaîne de format. Le type de données de chaque variable doit assortir l' identification correspondante au format et la variable de cible.

err_label

un label définie dans la portée courante du script, auquel le contrôle s'embranche si une erreur se produit.

Exemples:

```
FORMAT (date_string, &
       "The date is %D today, and the time is %T", &
       int-date, int-time), ON ERROR GOTO end
FORMAT (date_value, "%D", char-date), ON ERROR GOTO frm_err
FORMAT (uc_string, "Name in uppercase is %U", lc_string)
```

Voyez également:

Produisez Le Jet Manipulant Des Commandes

Commande de CORPS de la CHARGE RESPONSE_info (LOAD RESPONSE_INFO BODY Command)

Description:

Cette commande charge une variable de caractère avec l'ensemble ou une partie des données d'un corps de message de réponse HTTP pour un la connexion indiqué TCP. Elle est employée après qu'une commande GET, HEAD ou *POST command* .

OpenSTA attendra automatiquement jusqu'à ce que n'importe quelle requête sur l'identification indiquée de la connexion soit complète avant d'exécuter cette commande. Il n'est pas néctestre que le script fasse ceci explicitement.

Si la chaîne de données est trop longue pour s'adapter dans la variable de cible, elle sera tronquée. Pour une réponse d un message de corps contenant un document HTML, la clause "WITH" peut être employé pour charger une variable de caractère avec un élément ou une partie d'un élément du document.

Format:

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

Paramètres:

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant connexion ID de la connexion TCP sur lequel le message de réponse HTTP sera reçu.

variable

Le nom d'une variable de caractère dans laquelle le corps du message de réponse HTTP, ou la partie choisie de lui, sont chargés.

identifier

Chaîne variable ,chaîne de caractères citée ou expression de caractère identifiant les données à rechercher du corps de message de réponse. Pour une définition du format de marque voir les marques de CORPS de la CHARGE RESPONSE_info .

Exemple:

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

Voyez également:

Produisez Le Jet Manipulant Des Commandes

Commande d'En-tête de la CHARGE RESPONSE_info (LOAD RESPONSE_INFO HEADER Command)

Description:

Cette commande charge une variable de caractère avec tous ou certains champs d'en-tête de message de réponse HTTP pour un la connexion indiqué TCP.

OpenSTA attendra automatiquement jusqu'à ce que n'importe quelle requête sur l'identification indiquée de la connexion soit complète avant d'exécuter cette commande. Il n'est pas néctestre que le script fasse ceci explicitement.

Si la chaîne de données est trop longue pour s'adapter dans la variable de cible, elle sera tronquée.

la clause "WITH" peut être employé pour indiquer les noms d'un champ d'en-tête dont la valeur doit être recherchée du message de réponse HTTP. Si cette clause est omise, tous les champs d'en-tête de message de réponse sont recherchés.

Format

```
LOAD RESPONSE_INFO HEADER ON conid INTO variable{&}
```

```
{,WITH identifier}
```

Paramètres:

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant la connexion ID de la connexion TCP sur lequel le message de réponse de HTTP sera reçu.

variable

Le nom d'une variable de caractère dans laquelle les en-têtes de message de réponse HTTP, ou les en-têtes choisis, sont chargés.

Identifier

Chaîne variable , chaîne de caractères citée ou expression de caractère contenant le nom du champ d'en-tête de message de réponse à rechercher.

Exemple:

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

Voyez également:

Produisez Le Jet Manipulant Des Commandes

Commande de ~locate(~LOCATE Command)

Description:

Cette commande est une fonction et peut seulement être mise en référence dans une commande SET . Elle renvoie une valeur de nombre entier, correspondant à l'offset de la sous-chaîne indiquée dans la chaîne de source. L'offset du premier caractère dans la chaîne de source est zéro. Si la sous-chaîne n'est pas trouvée, la fonction renvoie une valeur de -1.

Par défaut, l'assortiment est cas sensible. Les chaînes "Londres" et "LONDRES", par exemple, ne produiraient pas un résultat , parce que la caisse des caractères n'est pas identique. Ceci peut être dépassé en indiquant ", CASE_BLIND".

La chaîne de source est balayée de gauche à droite. Si la sous-chaîne apparaît plus d'une fois dans la chaîne de source, la fonction renverra toujours l'offset de la première occurrence.

Format:

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

Return Value :

L'offset de la sous-chaîne dans la chaîne de source. Si la sous-chaîne n'était pas trouvée, alors une valeur de -1 est retournée.

Paramètres:

Substring

La valeur de caractère définissant la sous-chaîne à situer dans la chaîne de source. Ceci peut être une chaîne variable ou chaîne de caractères citée.

String

La valeur de caractère pour rechercher la sous-chaîne indiquée. Ceci peut être une chaîne variable ou de chaîne de caractères citée.

Exemple:

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

Voyez également:

Produisez Le Jet Manipulant Des Commandes

Contrôle De Commande D'Écoulement(Flow Control Commands)

Les contrôle de commande d'écoulement (Flow Control Commands)déterminent quelles sections

d'un script sont traitées, et dans quelles ordre.

Voyez également:

APPELEZ La Commande (CALL Command)

APPELEZ La Commande de SCRIPT (CALL Script Command)

DÉCOMMANDEZ SUR La Commande (CANCEL ON Command)

DÉTACHEZ La Commande (DETACH Command)

Commandez (DO Command)

FINISSEZ La Commande de SOUS-programme (END SUBROUTINE Command)

Commande d'Entrée (ENTRY Command)

SORTEZ La Commande (EXIT Command)

Commande GOTO (GOTO Command)

SI Commande (IF Command)

SUR La Commande d'cErreur (ON ERROR Command)

Commande DE RETOUR (RETURN Command)

Commande de SOUS-programme (SUBROUTINE Command)

Commande d'Appel(CALL Command)

Description:

Cette commande appelle un sous-programme dans un script. Les sous-programmes doivent suivre la section principale de code et ne doivent pas être enfoncés dans elle. Ils partagent les définitions variables du module principal.

Il n'est pas possible de s'embrancher dans ou hors d'un sous-programme, parce qu'un label ne peut pas être mise en référence en dehors du module ou du sous-programme principal en lesquels elle se produit. Ceci signifie, cependant, que chaque sous-programme permet à un script de définir jusqu'à 255 labels en plus de ceux utilisées dans le code principal.

Un maximum de huit paramètres peut être passé du code appelé (calling code) au sous-programme appelé. Les paramètres passés peuvent être des variables de caractère ou de nombre entier, littérales ou des chaînes de caractères citées. Le code d'appeler doit passer exactement le même nombre de paramètres au sous-programme appelé car le sous-programme appelé a défini dans sa déclaration de SOUS-programme. Les noms des variables dans l'appel n'ont pas besoin d'être identiques à la liste de paramètre de sous-programme, mais les types de données de chacun des paramètres doivent s'assortir. Le manque de se conformer dans ces conditions aura comme conséquence une erreur de script étant produite.

Les valeurs des variables définies comme paramètres dans la définition de sous-programme ne sont pas copiées de nouveau aux variables dans l'appel, sur le retour du sous-programme. Cependant, si les mêmes noms variables sont employés dans l'appel et la liste de paramètre de sous-programme, la valeur de la variable dans l'appel sera changée par un changement du sous-programme; c'est parce que le code d'appeler et la part appelée de sous-programme les mêmes définitions de données. Réciproquement, si différents noms variables sont employés, aucun changement fait aux variables dans le sous-programme n'affectera les variables dans l'appel.

Format:

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

Paramètres:

Subroutine

Le nom du sous-programme appelé. Le nom doit être un OpenSTA valide Dataname .

paramètre

Une variable de caractère, variable de nombre entier, valeur de nombre entier ou une chaîne de caractères citée. Jusqu'à 8 paramètres peuvent être déclarés dans la commande d'Appel. Il doit y avoir le même nombre de paramètres dans cette liste comme sont dans la définition du sous-programme, et les types de données des paramètres doivent s'assortir.

Exemples:

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

Voyez également:

Commandes De Commande D'Écoulement

Commande de SCRIPT d'Appel(CALL SCRIPT Command)

Description:

Cette commande appelle un script d'un autre script. Quand la commande est exécutée, la commande est transférée au script appelé; quand le script appelé sort, la commande est retournée au script d'appeler, renvoyant sur option un statut du script appelé. Il n'y a aucune limite sur le nombre de scripts qui peuvent être mis en référence par n'importe quel script.

En général, un script appelé est considéré comme prolongation au script d'appeler, et tous les changements faits du script appelé sont propagés de nouveau au script d'appeler sur la sortie. Cependant, certains changements (par exemple plus loin ON ERROR handlers) demeurent seulement en vigueur pour la durée du script appelé (ou des scripts appelés par elle); l'état original est rétabli quand la commande est retournée au script d'appeler.

Pour des scripts, un maximum de huit paramètres peut être passé du script d'appeler au script appelé. Un paramètre omis est indiqué par deux virgules consécutives ",,". que le script d'appeler doit passer exactement le même nombre de paramètres au script appelé car le script appelé a défini dans sa déclaration ENTRY(comptabilité pour tous paramètres omis). En outre, les types de données de chacun des paramètres doivent s'assortir. Le manque de se conformer dans ces conditions aura comme conséquence une erreur de script étant produite.

Les valeurs des paramètres sont passées du visiteur dans les variables définies dans le rapport d'Entrée du script appelé. Toutes les modifications aux valeurs des variables sont copiées de nouveau au visiteur sur le retour du script appelé.

Une valeur facultative de statut peut être retournée du script appelé en employant la clause "RETURNING" pour indiquer la variable de nombre entier qui doit tenir la valeur de retour de statut.

Par défaut, si une erreur se produit dans un script appelé, un message d'erreur est écrit à la notation d'audit et aux arrêts de fil; la commande n'est pas retournée au script d'appeler. Cependant, si le

piégeage d'erreur est permis dans le script d'appeler et l'erreur était une erreur de script, puis commande sera retourné au code de gestion d'erreur du script d'appeler.

La clause "ON ERROR GOTO err_label" peut être indiquée pour définir un label auquel la commande devrait être transférée en cas d'une erreur tout en essayant d'appeler le script.

Format:

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

Paramètres:

name

Un caractère variable ou chaîne de caractères citée définissant le nom du script à s'appeler. Le nom doit être un OpenSTA valide Dataname .

paramètre

Une variable de caractère, variable de nombre entier, a cité la chaîne de caractères, la valeur de nombre entier ou l'identification de dossier à passer au script appelé. Un maximum de 8 paramètres peut être passé entre les scripts.

statuts

Une variable de nombre entier pour recevoir le statut retourné du script appelé. Si aucun statut n'est retourné du script appelé, alors cette variable contiendra le dernier statut retourné de n'importe quel script appelé.

err_label

un label défini dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemples:

```
CALL SCRIPT Script-Name
CALL SCRIPT "TEST"
CALL SCRIPT "CALC_TAX" [COST, RATE, TAX]
CALL SCRIPT "GET_RESPONS" returning Response &
```

```
ON ERROR GOTO Problem
```

Voyez également:

Commandes De Commande D'Écoulement

ANNULATION SUR La Commande(CANCEL command)

Description:

Cette commande termine la detection automatique des erreurs de script, qui est permis avec la commande *ON ERROR* . Toutes les erreurs de script produites causeront le fil d'être avorté.

Cette commande affectera seulement la détection automatique des erreurs de script dans le script ou les scripts courants appelés par elle. Sur la sortie de ce script, ON ERROR handler établi par un script d'appeler en seront rétablis.

Format:

```
CANCEL ON {ERROR}
```

Paramètres:

Aucun

Exemples:

```
CANCEL ON
CANCEL ON ERROR
```

Voyez également:

Commandes De Commande D'Écoulement

DÉTACHEZ La Commande(DETACH Command)

Description:

Cette commande cause le fil courant à la sortie. Le programme sort de tous les scripts ou sous-programmes qui se sont appelés (appels nichés y compris) jusqu'à ce que la commande revienne au script primaire. Le fil est alors détaché du test Executer.

Format:

```
    DETACH {THREAD}
```

Paramètres:

Aucun

Exemples:

```
DETACH
DETACH THREAD
```

Voyez également:

Commandes De Commande D'Écoulement

Commandez(DO Command)

Description:

La commande DO et ENDDO permettent un ensemble de commandes d'être répété un nombre de fois fixe. La section d'un script à répéter est terminée par une commande ENNDO.

Format:

```
DO variable = value1, value2 {, step}
       command{s}
ENDDO
```

Paramètres:

variable

Le nom de la variable de commande ou d'index qui est ajustée chaque fois la boucle s'exécute. L'ajustement est déterminé par la valeur de la variable d'étape. Ceci doit être une variable de nombre entier.

Value 1

La valeur commençante de la variable de commande. Ceci doit être une variable de nombre entier ou une valeur de nombre entier.

Value 2

La valeur de terminaison de la variable de commande. Ceci doit être une variable ou une valeur de nombre entier, et peut être plus haute ou inférieur la valeur 1 . Quand la variable de commande contient une valeur qui est plus grande que cette valeur (ou inférieur si l'étape est négative), la boucle sera terminée.

Step

Une variable ou une valeur de nombre entier déterminant la valeur par laquelle la variable de commande ou la variable d'index est incrémentée chaque fois la boucle s'exécute. Si valeur 2 est moins que la valeur 1 puis l'étape la valeur doit être négative. Si une variable d'étape n'est pas indiquée, puis l'étape la valeur se transférera sur 1.

Exemples:

```
DO Empno = 1, 1000
       NEXT Name
       LOG 'Employee number: ', Empno, '; Name: ', Name
ENDDO
DO Empno = START, END, 10
       NEXT Name
       LOG 'Employee number: ', Empno, '; Name: ', Name
ENDDO
```

Voyez également:

Commandes De Commande D'Écoulement

Commande de SOUS-programme de FIN(END SUBROUTINE Command)

Description:

Cette commande termine un sous-programme. Elle doit suivre toutes autres commandes exécutables dans le sous-programme. Les seuls rapports qui peuvent suivre une commande de END SUBROUTINE sont un commentaire, une nouvelle commande de SOUS-programme ou une commande INCLUDE; le script inclus doit contenir plus de définitions de sous-programme.

Un sous-programme est lancé par la commande de SOUS-programme.

Format:

```
    END SUBROUTINE
```

Paramètres:

Aucun

Exemple:

```
END SUBROUTINE
```

Voyez également:

Commandes De Commande D'Écoulement

Commande d'Entrée(ENTRY Command)

Description:

Cette commande, si indiquée, doit être le premier article dans la section de code du script, à l'exclusion des caractères de format et des commentaires. Elle identifie quelles variables doivent recevoir des valeurs passées comme paramètres d'un script d'appeler

Il est recommandé que les variables avouées dans la commande d'cEntrée n'aient pas une liste de valeur ou une gamme ou un dossier associée. Les valeurs ont passé de cette façon seront recouvertes quand l'initialisation de script a lieu suivant la commande d'cEntrée.

Format:

```
ENTRY [parameter{, parameter ...}]
```

Paramètre:

paramèter

Une variable de caractère (de jusqu'à 50 caractères de longueur), variable de nombre entier ou identification de dossier avouée dans la section de définitions du script. Jusqu'à 8 paramètres peuvent être déclarés dans la commande ENTRY . Il doit y avoir le même nombre de paramètres dans cette liste comme sont passés au script (paramètres omis y compris), et les types de données de paramètres correspondants doivent s'assortir.

Exemple:

```
ENTRY [DATE_PARAM, TIME_PARAM, CODE_PARAM]
```

Voyez également:

Commandes De Commande D'Écoulement

SORTEZ La Commande(EXIT Command)

Description:

Cette commande fait terminer l'exécution du script courant immédiatement. Aucune entrée supplémentaire ne sera fournie du dossier de script et d'aucunes commandes exécutés.

Une valeur facultative de statut peut être retournée quand le script en question s'est appelé d'un autre script. Ceci est réalisé en employant le statut variable pour placer une valeur dans la variable de retour de statut indiquée sur l'appel à ce script. Si aucun statut n'est indiqué, mais le visiteur attend un, alors le statut retourné sera retourné par le dernier script qui a sorti avec un statut. Ceci permet à un statut d'être recherché d'un script profondément niché où aucun renvoi explicite de statut n'a été utilisé.

Au temps d'exécution, un script est automatiquement terminé quand la fin du script est atteinte. Il n'est pas néctestre d'inclure une commande de EXIT comme dernière commande dans un script, de terminer l'exécution de script.

Si le script s'est appelé, en utilisant la commande CALL SCRIPT, l'exécution du script d'appeler reprendra à la commande juste après la commande CALL SCRIPT.

Quand une commande EXIT est traitée et il n'y a aucun autre fil exécutant le script, les données de script sont jetées. Cependant, si l' option ", KEEPLIVE" est indiqué sur la commande de EXIT, puis les données de script qui ne seront pas supprimées même s'il n'y a aucun autre fil l'exécutant. Ceci permet aux fils suivants d'exécuter le script et d'accéder à n'importe quelle installation de données de script par un fil précédent.

Format:

```
EXIT {status} {,KEEPALIVE}
```

Paramètre:

statuts

Une variable de nombre entier ou une valeur de nombre entier à retourner comme statut de ce script au visiteur. Le statut sera retourné dans la variable de nombre entier indiquée sur la commande CALL .

Exemples:

```
EXIT
EXIT RETURN-STATUS
```

Voyez également:

Commandes De Commande D'Écoulement

Commande GOTO(GOTO Command)

Description:

Cette commande transfère le contrôle à un label de script spécifié . Le transfert du contrôle est immédiat et sans conditions.

Des branches conditionnelles peuvent être faites employer en utilisant la commande IF.

Format:

```
    GOTO label
```

Paramètre:

Label

Un label défini dans la portée courante du script.

Exemples:

```
GOTO Start
GOTO End-Of-Script
```

Voyez également:

Commandes De Commande D'Écoulement

SI Commande(IF Command)

Description:

Cette commande réalise des tests sur les valeurs des variables contre d'autres variables ou littéral , et transfère le contrôle à un label indiqué dépendant des résultats des tests .

Alternativement, structuré les commandes IF peuvent être employées pour exécuter une ou plusieurs commandes dépendant du succès ou de l'échec des tests.

Par défaut, l'assortiment est cas sensible. Les chaînes "Londres" et "LONDRES", par exemple, ne produiraient pas un resultat , parce que la caisse des caractères n'est pas identique. Ceci peut être dépassé en indiquant la clause ", CASE_BLIND ".

Format:

```
1.    IF condition GOTO label
2.    IF condition THEN
```

```
     commands{s}
{ ELSEIF condition THEN
    command{s} }
         :
         :
{ ELSEIF condition THEN
    command{s} }
{ ELSE
    command{s} }
  ENDIF
```

Paramètres:

condition

Une condition du format suivant:

{NOT}(operand1 operator operand2 {, CASE_BLIND}) &
{AND/OR condition ...}

Les deux opérandes peuvent chacun être une variable, une chaîne de caractères citée ou une valeur de nombre entier.

L'option "CASE_blind" peut être indiquée pour "operand2", pour requêter une comparaison cas-peu sensible des opérandes.

"NOT" inverse le résultat de l'état encadré qu'il précède.

Les opérateurs binaires sont:

=	operand1 égale operand2
<>	operand1 n'égale pas operand2
<	operand1 est moins qu'operand2
<=	operand1 est inférieur ou égal à operand2
>	operand1 est operand2 plus grand que
>=	operand1 est supérieur ou égal à operand2
^	operand1 contient operand2
CONTIENT	operand1 contient operand2
<^>	operand1 ne contient pas operand2
PAS CONTIENT	operand1 ne contient pas operand2
NOT_contains	operand1 ne contient pas operand2

=	operand1 equals operand2
<>	operand1 does not equal operand2
<	operand1 is less than operand2
<=	operand1 is less than or equal to operand2
>	operand1 is greater than operand2
>=	operand1 is greater than or equal to operand2
^	operand1 contains operand2
CONTAINS	operand1 contains operand2
<^>	operand1 does not contain operand2
NOT CONTAINS	operand1 does not contain operand2
NOT_CONTAINS	operand1 does not contain operand2

Toutes les conditions sont évaluées de gauche à droite.

Label

Un label défini dans la portée courante du script.

command

Tout nombre de script commande - incluant plus loin les commandes IF ou DO, à condition que le niveau d'emboîtement maximum de 100 ne soit pas excédé.

Exemple:

```
IF ( NOT(isub=10) AND (NOT(isub=99)) ) THEN
      LOG "...continued"
ELSE
      LOG " Completed loop"
ENDIF
```

Voyez également:

Commandes De Commande D'Écoulement

La Commande ON Error(ON ERROR Command)

Description:

Cette commande permet aux erreurs de script - qui causeraient normalement le fil étant exécuté pour avorter - d'être capturée, et à l'exécution de script à reprendre à un label prédéfini. Le ON ERROR handler est en ligne global à toutes les sections du script; il est propagé dans tous les sous-programmes et scripts appelés.

la commande ON ERROR capture toutes les erreurs qui se produisent l'un ou l'autre dans le script dans lequel on lui a déclaré ou dans n'importe quels scripts plus bas appelés par elle. Toutes les erreurs de script, telles qu'une mauvaise erreur de paramètre sur la commande de ~EXTRACT, ou une tentative d'appel un script inexistant, peuvent être arrêtées et traitées par cette commande.

Si une erreur de script est produite, alors un message sera écrit le log d’audit , identifiant et localisant où l'erreur s'est produite. Si l'erreur s'est produite dans un script à un niveau plus bas que cela dans lequel la commande ON ERROR a été déclarée, alors tous les scripts seront avortés

jusqu'à ce que le script exigé soit trouvé.

ON ERROR handler peut être dépassé par " ON ERROR GOTO " ou la clause " ON TIMEOUT GOTO " pour la durée d'une commande simple. Elle peut également être dépassée par la commande ON ERROR dans un script ou un sous-programme appelé; une telle modification affectera seulement ces scripts et sous-programmes à ce niveau d'emboîtement ou s'abaissera. Sur la sortie du script ou du sous-programme, précédemment défini ON ERROR handler sera rétabli.

Quand la vérification ON ERROR est établi, elle peut être neutralisée en employant la commande CANCEL, comme suit:

```
CANCEL ON ERROR
```

Format:

```
ON ERROR GOTO label
```

Paramètre:

Label

Le nom du label dans la portée courante du script, auquel la commande s'embranche si une erreur de script est produite.

Exemple:

```
ON ERROR GOTO SCRIPT-ERROR
```

Voyez également:

Commandes De Commande D'Écoulement

Commande DE RETOUR(RETURN Command)

Description:

Cette commande renvoie le control d'un sous-programme appelé à l'instruction après l'appel à ce sous-programme.

Format:

```
RETURN
```

Paramètres:

Aucun

Exemple:

```
RETURN
```

Voyez également:

Commandes De Commande D'Écoulement

Commande de SOUS-programme(SUBROUTINE Command)

Description

Cette commande définit le début d'une section discrète du code qui est lié par les commandes de SOUS-programme (SUBROUTINE)et de SOUS-programme de FIN(END SUBROUTINE).

Des sous-programmes sont appelles du code principal avec une commande du format " CALL name ". Ils renvoient la commande au code principal au moyen de la commande RETURN . Un maximum de 255 sous-programmes peut être défini dans un script.

Les sous-programmes partagent les mêmes définitions variables que le code principal mais ont leurs propres labels . Un label ne peut être mise en référence en dehors du module principal ou de l'extérieur du sous-programme dans lequel il se produit. Ceci a l'effet de la neutralisation s'embranchant dans et hors des sous-programmes, et également des moyens que chaque sous-programme peut employer des 255 labels plus encore en plus de ceux utilisées dans le code principal.

Format:

```
    SUBROUTINE name {[parameter{, parameter..}]}
```

Paramètres:

Name

Le nom du sous-programme. Ceci doit être un OpenSTA valide Dataname et doit être unique dans le script.

paramètre

Une variable de caractère ou la variable de nombre entier a déclaré dans la section de définitions du script. Jusqu'à 8 paramètres peuvent être déclarés dans la commande de SOUS-programme. Il doit y avoir le même nombre de paramètres dans cette liste pendant qu'il y a dans l'appel de sous-programme, et les types de données des paramètres doivent s'assortir.

Exemples:

```
SUBROUTINE GET_NEXT_VALUE
SUBROUTINE CREATE_FULL_NAME [subchr_1, subchr_2, subchr_3]
        SET full_name = subchr_3 + subchr_1 + subchr_2
       RETURN
END SUBROUTINE
```

Voyez également:

Commandes De Commande D'Écoulement

Fichier Manipulant Des Commandes(File Handling Commands)

Le fichier manipulant des commandes aident des scripts et des fichiers de données externes pour échanger des données.

Voyez également:

CLÔTUREZ La Commande (CLOSE Command)

OUVREZ La Commande (OPEN Command)

Commande LUE (READ Command)

Commande de REBOBINAGE (REWIND Command)

Commande FERMER(CLOSE Command)

Description:

Cette commande ferme un fichier de données externe. Le fichier doit avoir été déjà ouvert par la commande OPEN.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label à laquelle le control devrait être transférée en cas d'une erreur.

Format:

```
CLOSE fileid {{,}ON ERROR GOTO err_label}
```

Paramètres:

fileid

Le nom lié au fichier quand il a été ouvert.

err_label

Un label défini dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemple:

```
CLOSE datafile ON ERROR GOTO Close_error
```

Voyez également:

Classez Manipuler Des Commandes

OUVREZ La Commande(OPEN Command)

Description:

Cette commande ouvre un fichier de données externe pour l'accès d'entrée et associe un OpenSTA

Dataname avec elle, pour la future référence.

Quand les enregistrements sont lus à partir d'un fichier , des données seront lues jusqu à qu ils ne comprennent pas à un caractère NEWLINE. Le caractère NEWLINE sera sauté pour placer le fichier au début du prochain enregistrement à lire.

L' enregistrement lu sera tronqué comme exigé pour remplir variable indiquée.

Les lectures sont indépendants pour chaque fil.

Un maximum de 10 fichiers de données externes peut être ouvert pour chaque fil n'importe quand. Essayer d'ouvrir plus que ce nombre aura comme conséquence une erreur de script étant rapportée.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel la commande devrait être transférée en cas d'une erreur. Ceci doit être la dernière clause dans la commande.

Format:

```
OPEN filename AS fileid {ON ERROR GOTO err_label}
```

Paramètres:

Filename

Un caractère variable ou chaîne de caractères citée contenant le nom de fichier (à l'exclusion du nom de chemin) du fichier à ouvrir. Le fichier doit résider dans l'annuaire de données du dossier .

fileid

Un OpenSTA Dataname lié au fichier quand il est ouvert; il est employé pour identifier le fichier dans de futures références. Le "fileid" doit être déclaré dans un rapport de FILE dans la section de définitions du script.

err_label

Un label défini dans la portée courante du script, auquel la control s'embranche si une erreur se produit.

Exemples:

```
OPEN "Usernames" AS datafile ON ERROR GOTO file-error
OPEN myfile AS datafile ON ERROR GOTO file-error
```

Voyez également:

Classez Manipuler Des Commandes

Commande LUE(READ Command)

Description:

Cette commande lit un enregistrement simple à partir d'un fichier externe qui est actuellement ouvert dans une variable. Si le fichier enregistré est plus long que la variable, les données enregistrées sont tronquées.

L'enregistrement lu sera délimité par un caractère NEWLINE dans le fichier. Ce caractère NEWLINE est employé purement comme délimiteur d'enregistrement et ne fait pas partie de l'enregistrement.

Par défaut, le fichier sera rebobiné quand un statut " End-of-File " est retourné par la commande READ . Cette action peut être modifiée au moyen de la clause " AT END GOTO label ".

Le fichier est lu séquentiellement.

Format:

```
READ variable FROM fileid
{AT END GOTO label}  {ON ERROR GOTO err_label}
```

Paramètres:

variable

Une variable de caractère dans laquelle le prochain enregistrement à partir du fichier est lu.

fileid

Le nom s'est associé au fichier quand il a été ouvert.

Label

Un label dans la portée courante du script, auquel l'exécution de script s'embranchera si le statut " End-of-File " est produit.

err_label

Un label dans la portée courante du script, auquel l'exécution de script s'embranchera si une erreur se produit.

Exemples:

```
READ data_record FROM datafile
READ data FROM datafile AT END GOTO EXIT_LABEL &
      ON ERROR GOTO read_error
```

Voyez également:

Classez Manipuler Des Commandes

Commande de REBOBINAGE(REWIND Command)

Description:

Cette commande cause un fichier de données externe d'être rebobiné. Le fichier doit avoir été déjà ouvert par la commande OPEN.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel la contrôle devrait être transférée en cas d'une erreur.

Format:

```
    REWIND fileid {ON ERROR GOTO err_label}
```

Paramètres:

fileid

Le nom s'est associé au fichier quand il a été ouvert.

err_label

Un label défini dans la portée courante du script, auquel le contrôle s'embranche si une erreur se produit.

Exemple:

```
    REWIND datafile
```

Voyez également:

Classez Manipuler Des Commandes

Commandes De Commande Formelles D'Test(Formal Test Control Commands)

Les commandes de contrôle formelles de test fournissent l'appui formel pour dépister les résultats de chaque test, de sorte qu'il soit possible de voir facilement à quel point le test va.

Voyez également:

FINISSEZ La Commande De Cas d'espèce (END TEST-CASE Command)

ÉCHOUEZ La Commande De Cas d'espèce (FAIL TEST-CASE Command)

Commande d'cHistoire (HISTORY Command)

PASSEZ La Commande De Cas d'espèce (PASS TEST-CASE Command)

RAPPORTEZ La Commande (REPORT Command)

COMMENCEZ La Commande De Cas d'espèce (START TEST-CASE Command)

Commande De Cas d'espèce de FIN(END TEST-CASE Command)

Description:

La commande END TEST-CASE termine une section du script qui commence par une commande START TEST-CASE, pour créer un cas de test individuel.

Si la commande END TEST-CASE est atteinte pendant l'exécution du script, le test est considéré comme avoir réussi, et le message indiqué dans la définition de test est envoyé à la rapport log.

Examinez les cas ne peut pas être niché. Cependant, il n'y a aucune restriction à appeler un autre script qui contient un test, en dedans d'une section de test.

Format:

```
    END TEST-CASE
```

Paramètres:

Aucun

Exemple:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate < minimum) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

Voyez également:

Commandes De Commande Formelles D'Test

Commande De Cas d'espèce d'Échec(FAIL TEST-CASE Command)

Description:

Cette commande indique que le test en cours a échoué. Le message d'échec du test est envoyé au log de rapport, et le compte d'anomalie du test est incrémenté.

L'exécution de script est reprise à la première instruction suivant l'extrémité de la section de cas d'test (c.-à-d. la commande END TEST-CASE). Si une clause "GOTO" est indiquée, l'exécution de script est reprise au point identifié par la clause label . Si une commande valide suit immédiatement la commande FAIL TEST-CASE qui ne serait pas exécutée en raison du saut dans l'exécution de script, le compilateur de script produit un message d'avertissement quand le script est compilé, mais produit toujours un fichier d'exécution (supposant il n'y a aucune erreur).

Cette commande est seulement valide dans une section du test d'un script. Il peut répéter aussi souvent comme exigé dans un test individuel.

Format:

```
    FAIL TEST-CASE {GOTO label}
```

Paramètre:

Label

Un label défini dans la portée courante du script, auquel la commande s'embranche.

Exemple:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate < minimum) THEN
        FAIL TEST-CASE
    ELSEIF (dist_rate > maximum) THEN
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

Voyez également:

Commandes De Commande Formelles D'Test

Commande d'Histoire(HISTORY Command)

Description:

Le log d'histoire contient une histoire des exécutions d'un test. Par conséquent, le programme essaye toujours d'ouvrir log historique existant chaque fois qu un test est exécuté.

La commande HISTORTY vous permet d'indiquer un message à entrer dans ce fichier. Chaque message aura une date, chronomètre et filète(thread name) associé nommé à lui dans le log d'histoire.

Un message d'histoire peut se composer de tout nombre de différentes valeurs séparées par des virgules. Tous les caractères non-imprimable Ascii en valeurs de caractère sont remplacés par des valeurs de nombre entier des périodes (".") sont écrits en tant que valeurs signées, en utilisant seulement autant de caractères selon les besoins.

Format:

```
    HISTORY value {, value...}
```

Paramètres:

Value

La valeur ou la variable à écrire au log d'histoire. Ceci peut être une variable ou une chaîne de caractères citée.

Exemples:

```
HISTORY "Test Run Completed." &
        ' Actions = ', action_count
HISTORY "This message contains a character command " &
       "to represent the tilde character ~~"
HISTORY "This message contains a 'single quoted section'" &
       'and "a double one here".'
```

Voyez également:

Commandes De Commande Formelles D'Test

PASSEZ La Commande De Cas d'espèce(PASS TEST-CASE Command)

Description:

Cette commande indique que le test en cours a réussi. Le message de succès du test est envoyé au log de rapport.

Si aucune clause GOTO n'est indiquée, l'exécution de script est reprise à la première commande suivant l'extrémité de la section du test (c.-à-d. la commande END TEST-CASE). Si une clause GOTO est indiquée, l'exécution de script est reprise au point identifié par la clause label . Si une commande valide suit immédiatement la commande PASS TEST-CASE qui ne serait pas exécutée en raison du saut dans l'exécution de script, le compilateur produit un message d'avertissement quand le script est compilé, mais produit toujours un fichier d'exécution (supposant il n'y a aucune erreur).

Cette commande est seulement valide dans une section du test d'un script. Il peut répéter aussi souvent comme exigé dans un test .

Si la commande END TEST-CASE est atteinte pendant l'exécution du script, le cas test est automatiquement considéré comme avoir réussi, et le message de succès est envoyé au log de rapport.

Format:

```
PASS TEST-CASE {GOTO label}
```

Paramètre:

Label

Un label défini dans la portée courante du script, auquel la commande s'embranche.

Exemple:

```
START TEST-CASE "Checking distribution rate"
    IF (dist_rate >= minimum) THEN
        PASS TEST-CASE
    ELSE
        FAIL TEST-CASE
    ENDIF
END TEST-CASE
```

Voyez également:

Commandes De Commande Formelles D'Test

RAPPORT La Commande(REPORT Command)

Description:

Les logs de rapport contiennent l'information passagère concernant l'exécution d'un test.

La commande de REPORT permet à l'utilisateur d'indiquer un message à entrer dans ce fichier. Chaque message aura une date, chronomètre et filète(thread name) associé nommé à lui dans le log de rapport.

Un message de rapport peut se composer de tout nombre de différentes valeurs séparées par des virgules.

Tous les caractères non-imprimable Ascii en valeurs de caractère sont remplacés par des périodes où des valeurs de nombre entier de ("."). sont écrites en tant que valeurs signées, et emploient seulement autant de caractères que nécessaire.

Format:

```
    REPORT value{, value...}
```

Paramètres:

Value

La valeur ou la variable à écrire au log de rapport. Ceci peut être une variable ou une chaîne de caractères citée.

Exemples:

```
REPORT "Section 1 Completed after ", loops, &
       ' Iterations'
REPORT "This is a long message ", &
        "that is continued on this line ", "and this line"
REPORT "This message contains a character command " &
        "to represent the tilde character ~~"
REPORT "This message contains a 'single quoted section'" &
        'and "a double one here".'
```

Voyez également:

Commandes De Commande Formelles D'Test

COMMENCEZ La Commande De Cas d'espèce(START TEST-CASE Command)

Description:

La commande START TEST-CASE présente une section du code qui est groupé ensemble dans un test. La section est terminée par une commande END TEST-CASE.

La commande START TEST-CASE doit inclure une description d'un test. Le statut du test et la description et du test sont écrits au log de rapport quand le test est exécuté.

Des cas test ne peuvent pas être nichés, ainsi un test doit être terminé avec une commande END TEST-CASE avant qu'une nouvelle section de cas test puisse être commencée. Cependant, il n'y a aucune restriction à appeler un autre script qui contient des tests, en dedans d'une section de test.

Format:

```
START TEST-CASE description
```

Paramètre:

description

Une chaîne de variable ou d'expression entre guillemets de caractère contenant le texte qui décrit le test.

Exemples:

```
START TEST-CASE "Checking for appearance of UNITS field"
 IF (no_units = 0) THEN
        FAIL TEST-CASE
 ENDIF
 END TEST-CASE
```

```
SET tc_desc_str = "Checking for appearance of UNITS field"
START TEST-CASE tc_desc_str
IF (no_units = 0) THEN
        FAIL TEST-CASE
ENDIF
END TEST-CASE
```

Voyez également:

Commandes De Commande Formelles D'Test

Commandes De Synchronisation(Synchronization Commands)

Ces commandes adressent des événements que les scripts peuvent devoir attendre avant continuer leur exécution.

Voyez également:

ACQUÉREZ La Commande de MUTEX (ACQUIRE MUTEX Command)

DÉGAGEZ La Commande de SÉMAPHORE (CLEAR SEMAPHORE Command)

LIBÉREZ La Commande de MUTEX (RELEASE MUTEX Command)

PLACEZ La Commande de SÉMAPHORE (SET SEMAPHORE Command)

SYNCHRONISEZ La Commande de REQUÊTES (SYNCHRONIZE REQUESTS Command)

ACQUÉREZ La Commande de MUTEX(ACQUIRE MUTEX Command)

Description:

Cette commande acquiert l'accès exclusif à une ressource partagée, connue sous le nom de mutex .

Le mutex est identifié par son nom et portée (qui doivent être "LOCAL" ou "TEST-WIDE"). Un mutex test-wide est un qui est partagé par tous les scripts fonctionnant en tant qu'élément d'un test distribué; un mutex local est seulement partagé entre les scripts fonctionnant sur le noeud local (local node).

Par défaut, si une tentative est faite d'acquérir un mutex qui a été déjà acquis par un autre script (dans la même portée), puis le fil sera suspendu jusqu'à ce que le mutex soit libéré. Cependant, si une période de délai est indiquée, ceci représente le nombre maximum des secondes qu'OpenSTA attendra le mutex à libérer avant la synchronisation hors de la requête. Une période de zéro indique que la requête devrait être chronométrée dehors immédiatement si le mutex a été acquis par un autre script.

La clause " ON TIMEOUT GOTO tmo_label " peut être indiquée pour définir un label auquel la commande devrait être transférée chronomètre la fin de la requête . En outre, la clause " ON TIMEOUT GOTO tmo_label " peut être indiquée pour définir un label auquel la commande devrait être transférée en cas d'une erreur, ou si les temps de requête sont dépasses et là n'étaient aucune clause " ON TIMEOUT GOTO tmo_label".

Format:

```
ACQUIRE {scope} MUTEX mutex_name {&}
        {,WITH TIMEOUT period {,ON TIMEOUT GOTO tmo_label}} {&}
       {,ON ERROR GOTO err_label}
```

Paramètres:

Scope

La portée du mutex à acquérir. Ceci doit être "LOCAL" ou "TEST-WIDE", et par défauts "LOCAL".

mutex-name

Une variable de caractère, ou chaîne de caractères citée, contenant le nom du mutex qui doit être acquis. le "mutex-name" doit être un OpenSTA valide Dataname .

Period

Une variable ou une valeur de nombre entier, définissant le nombre de secondes pour attendre avant une requête mécontente est dépassée le temps compté. La gamme valide est 0-2147483647.

tmo_label

Un label défini dans la portée courante du script, auquel la commande s'embranche si un délai se produit.

err_label

un label définie dans la portée courante du script, auquel la commande s'embranche si une erreur se produit, ou la commande depasse le temps et le "tmo_label" n'est pas indiqué.

Exemple:

```
ACQUIRE LOCAL MUTEX "MUMPS-SERVER", ON ERROR GOTO mumps-error
```

Voyez également:

Commandes De Synchronisation

Commande CLAIRE de SÉMAPHORE(CLEAR SEMAPHORE Command)

Description:

Cette commande remet à zéro une sémaphore nommée à son état "Clear". La sémaphore est identifiée par son nom et portée (qui doivent être "LOCAL" ou "TEST-WIDE"). Une sémaphore test-wide, est une qui est partagée par tous les scripts fonctionnant en tant qu'élément d'un test distribué; une sémaphore locale est seulement partagée entre les scripts fonctionnant sur le noeud local.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel la commande devrait être transférée en cas d'une erreur.

Format:

```
CLEAR {scope} SEMAPHORE semaphore-name {&}
       {,ON ERROR GOTO err_label}
```

Paramètres:

Scope

La portée de la sémaphore à l'espace libre. Ceci doit être "LOCAL" ou "TEST-WIDE", et défauts "LOCAL".

sémaphore-name

Une variable de caractère, ou chaîne de caractères citée, contenant le nom de la sémaphore à l'espace libre.

err_label

Un label défini dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemple:

```
    CLEAR LOCAL SEMAPHORE "SERVER-RUNNING"
```

Voyez également:

Commandes De Synchronisation

LIBÉREZ La Commande de MUTEX(RELEASE MUTEX Command)

Description:

Cette commande libère un mutex nommé . Le mutex à libérer est identifié par son nom et la portée, qui doit correspondre aux valeurs indiquées sur la correspondance à la commande ACQUIRE MUTEX.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel la commande devrait être transférée en cas d'une erreur. Notez qu'une erreur se produit toujours si le script qui publie la requête du RELEASE MUTEX ne l'a pas précédemment acquise.

Format:

```
RELEASE {scope} MUTEX mutex_name {,ON ERROR GOTO err_label}
```

Paramètres:

Scope

La portée du mutex à libérer. Ceci doit être "LOCAL" ou "TEST-WIDE", et des défauts "LOCAL".

mutex-name

Une variable de caractère, ou chaîne de caractères citée, contenant le nom du mutex pour libérer.

err_label

un label définie dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemple:

```
    RELEASE LOCAL MUTEX "MUMPS-SERVER"
```

Voyez également:

Commandes De Synchronisation

PLACEZ La Commande de SÉMAPHORE(SET SEMAPHORE Command)

Description:

Les jeux de ces commandes une sémaphore nommé au son "Set" l'état. La sémaphore est identifiée de nom et la portée (qui doit être "LOCALE" ou "TEST-WIDE"). Une sémaphore test-wide est une qui est partagée par tous les scripts fonctionnant en tant qu'élément d'un test distribué; une sémaphore locale est seulement partagée entre les scripts fonctionnant sur le noeud local.

La clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel la contrôle devrait être transférée en cas d'une erreur.

Format:

```
SET {scope} SEMAPHORE semaphore-name {&}
```

```
{,ON ERROR GOTO err_label}
```

Paramètres:

Scope

La portée de la sémaphore à placer. Ceci doit être "LOCAL" ou "Test-large", et des défauts "LOCAUX".

sémaphore-name

Une variable de caractère, ou chaîne de caractères citée, contenant le nom de la sémaphore à placer.

err_label

un label défini dans la portée courante du script, auquel la commande s'embranche si une erreur se produit.

Exemple:

```
SET LOCAL SEMAPHORE "SERVER-RUNNING"
```

Voyez également:

Commandes De Synchronisation

SYNCHRONISEZ La Commande de REQUÊTES (SYNCHRONIZE REQUESTS Command)

Description:

Des requêtes HTTP sont publiées asynchronously. Juste après qu'une requête HTTP a été publiée, la prochaine commande dans le script est traitée. OpenSTA n'attend pas une réponse à recevoir pour une requête HTTP.

Cette commande cause le fil s'exécutant actuellement pour être suspendu immédiatement, jusqu'à ce que des réponses aient été reçues pour toutes les requêtes qui ont été publiées par le fil. Elle est seulement valide dans un script qui a été défini comme HTTP MODE.

Le ` ON TIMEOUT GOTO tmo_label' peut être indiqué pour définir le label auquel la commande sera transférée si la requête chronomètre dépasse .

Format:

```
[SYNCHRONIZE | SYNCHRONISE] REQUESTS {&}
{, WITH TIMEOUT period {, ON TIMEOUT GOTO tmo_label}}
```

Paramètres

période

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier définissant le nombre de secondes pour attendre avant la commande est chronométrée dehors. La gamme valide est 0 - 32767.

tmo_label

un label défini dans la portée courante du script, auquel la contrôle s'embranche si un délai se produit.

Exemples:

```
SYNCHRONIZE REQUESTS
SYNCHRONISE REQUESTS &
, WITH TIMEOUT 60, ON TIMEOUT GOTO timed_out
```

Voyez également:

Commandes De Synchronisation

Commandes D'Entrée De Jet D'Entrée(Input Stream Entry Commands)

la commande Input stream entry contrôle comment le script alimente l'entrée au système sous test.

Voyez également:

ATTENDEZ La Commande (WAIT Command)

ATTENDEZ La Commande de SÉMAPHORE (WAIT FOR SEMAPHORE Command)

Commande d'Attente(WAIT Command)

Description:

Cette commande suspend l'exécution de script pour le nombre indiqué de secondes. L'unité est ou des secondes ou les millisecondes dépendant de la valeur de déclaration d'environnement WAIT UNIT.

Format:

```
WAIT period
```

Paramètre:

période

Une variable ou une valeur de nombre entier définissant le nombre de secondes où l'exécution de script doit être suspendue. La gamme valide est 0-2147483647.

Exemples:

```
WAIT 5
WAIT Wait-Period
```

Voyez également:

Commandes D'Entrée De Jet D'Entrée

ATTENTE La Commande de SÉMAPHORE(WAIT FOR SEMAPHORE Command)

Description:

Cette commande stoppe le script jusqu'à ce que la sémaphore indiquée soit dans son état "Set" .La sémaphore est identifiée par son nom et portée (qui doivent être "LOCAL" ou "TEST-WIDE").

Une sémaphore test-wide est une qui est partagée par tous les scripts fonctionnant en tant qu'élément d'un test distribué; une sémaphore locale est seulement partagée entre les scripts fonctionnant sur le noeud local.

Par défaut, si la sémaphore est dans son état "CLEAR" quand la commande WAIT FOR SEMAPHORE est publiée, le fil sera suspendu jusqu'à ce qu'il soit placé dans son "SET" état. Cependant, si une période de délai est indiquée, ceci représente le nombre maximum des secondes qu'OpenSTA attendra la sémaphore à placer avant la synchronisation hors de la requête. Une période de zéro indique que la requête devrait être chronométrée dehors immédiatement si la sémaphore n'est pas placée.

La clause " ON TIMEOUT GOTO tmo_label " peut être indiquée pour définir un label auquel le contrôle devrait être transférée si la requête chronomètre dehors. En outre, la clause " ON ERROR GOTO err_label " peut être indiquée pour définir un label auquel le contrôle devrait être transférée en cas d'une erreur, ou si les temps de requête dehors et il n y aurait aucune clause " ON TIMEOUT GOTO tmo_label ".

Format:

```
WAIT {period} FOR {scope} SEMAPHORE semaphore-name {&}
        {,ON TIMEOUT GOTO tmo_label} {&}
       {,ON ERROR GOTO err_label}
```

Paramètres:

période

Une variable ou une valeur de nombre entier définissant le nombre de secondes pour attendre. La gamme valide est 0-2147483647.

Scope

La portée de la sémaphore à attendre. Ceci doit être "LOCAL" ou "TEST-WIDE", et a défauts "LOCAUX".

sémaphore-name

Une variable de caractère, ou chaîne de caractères citée, contenant le nom de la sémaphore pour attendre.

tmo_label

un label définie dans la portée courante du script, auquel le contrôle s'embranche si un délai se produit.

err_label

Un label défini dans la portée courante du script, auquel le contrôle s'embranche si une erreur se produit, ou la commande chronomètre dehors et le "tmo_label" n'est pas indiqué.

Exemple:

```
WAIT 10 FOR SEMAPHORE "SERVER-RUNNING"
```

Voyez également:

Commandes D'Entrée De Jet D'Entrée

Commandes Statistiques D'Enregistrement De Données (Statistical Data Logging Commands)

Les commandes diagnostiques vous aident à analyser des scripts afin de diagnostiquer une anomalie.

Voyez également:

FINISSEZ La Commande de TEMPORISATEUR (END TIMER Command)

COMMENCEZ La Commande de TEMPORISATEUR (START TIMER Command)

Commande de TEMPORISATEUR de fin (END TIMER Command)

Description:

Cette commande coupe *Stop-watch* nommé et écrit un enregistrement ‘end timer’ dans le logs statistique, même si le timer est déjà arrêté.

Un temporisateur de chronomètre est alimenté par la commande START TIMER.

Format:

```
END TIMER name
```

Paramètre:

Name

Le nom de temporisateur. Le temporisateur doit être déclaré dans une declaration de TEMPORISATEUR dans la section de définitions du script.

Exemple:

```
END TIMER Transaction
```

Voyez également:

Commandes Statistiques D'Enregistrement De Données

COMMENCEZ La Commande de TEMPORISATEUR (START TIMER Command)

Description:

Cette commande démarre *Stop-watch* nommé et écrit un enregistrement ‘end timer’ dans le logs statistique.

Il n'y a aucune limite au nombre de *stop-watch* timer qui peuvent être alimentés en même temps. Cependant, si un temporisateur est alimenté deux fois sans être arrêté dans l'intérim, le premier temporisateur est efficacement décommandé et est jeté quand il est remis en marche.

Un *Stop-watch* est coupé avec la commande END TIMER.

Format:

```
START TIMER name
```

Paramètre:

Name

Le nom de temporisateur. Le temporisateur doit être déclaré dans une déclaration de TEMPORISATEUR dans la section de définitions du script.

Exemple:

```
START TIMER Transaction
```

Voyez également:

Commandes Statistiques D'Enregistrement De Données

Commandes Diagnostiques(Diagnostic Commands)

Pendant le développement du test, il y a de temps en temps un besoin de découvrir plus au sujet de quel script fait afin de diagnostiquer une anomalie. Les commandes diagnostiques aident à ce processus.

Voyez également:

NOTEZ La Commande (LOG Command)

NOTEZ La Commande (NOTE Command)

Commande de NOTATION(LOG Command)

Description:

OpenSTA maintient une vérification rétrospective de son activité et événements relatifs. La commande LOG permet à l'utilisateur d'indiquer un message à écrire au log d'audit. Chaque message dans ce fichier aura une date, chronomètre et filète associé a son nom..

Un message de log peut se composer de tout nombre de différentes valeurs séparées par des virgules.

Tous les caractères non imprimable Ascii en valeurs de caractère sont remplacés par des périodes où des valeurs de nombre entier de ("."). sont écrites en tant que valeurs signées, en utilisant seulement autant de caractères que nécessaires.

Format:

```
    LOG value{, value...}
```

Paramètres:

value

La valeur ou la variable à noter. Ceci peut être une variable ou une chaîne de caractères citée.

Exemples:

```
LOG "Customer Name = ", Cust-Name, &
     ' Customer Code = ', Cust-Code
LOG "This is a long message " &
     "that is continued on this line " &
     "and this line"
LOG "This message contains a 'single quoted section'" &
     'and "a double one here".'
```

Voyez également:

Commandes Diagnostiques

NOTEZ La Commande(NOTE Command)

Description:

Cette commande associe une liste variables d'ou chaînes de caractères citées au fil courant. La

valeur courante peut être regardée dans tab de surveillance ou fenetre actif de test dans le commander.

Format:

```
NOTE value{,char_value,...}
```

Paramètres:

value

La valeur ou la variable à noter. Ceci peut être une variable ou une chaîne de caractères citée.

Exemples:

```
NOTE Emp-Name
NOTE "Searching for 'End Of File' failures"
```

Voyez également:

Commandes Diagnostiques

Commande de TRACE(TRACE Command)

Description:

Cette commande écrit les messages utilisateur-définissables au log traçante de script.

Format:

```
TRACE value{,value...}
```

Paramètres:

value

La valeur ou la variable à écrire au log de trace. Ceci peut être une variable ou une chaîne de caractères citée.

Exemples:

```
TRACE 'Trace point following "overflow" condition'
TRACE "Trace point ", trcpos
```

Voyez également:

Commandes Diagnostiques

Commandes Diverses (Miscellaneous Commands)

Les commandes diverses fournissent d'autre fonctionnalité qui s'est avérée utile en créant des scripts.

Voyez également:

RELIEZ La Commande (CONNECT Command)

Commande de DÉCONNEXION (DISCONNECT Command)

Commande de la CHARGE ACTIVE_users (LOAD ACTIVE USERS Command)

Commande de DATE de CHARGE (LOAD DATE Command)

Commande de la CHARGE NODENAME (LOAD NODENAME Command)

Commande de SCRIPT de CHARGE (LOAD SCRIPT Command)

Commande d'cTest de CHARGE (LOAD TEST Command)

Commande de FIL de CHARGE (LOAD THREAD Command)

Commande du MOMENT de CHARGEMENT (LOAD TIME Command)

Commande de TEMPORISATEUR de CHARGE (LOAD TIMER Command)

CONECTER La Commande(CONNECT Command)

Description:

Cette commande peut être employée pour établir la connexion de TCP à un hôte nommé. Elle est seulement valide dans un script qui a été défini comme HTTP MODE.

Cette commande indique un ID pour le la connexion TCP. Ceci peut être employé dans suivant des commandes GET, HEAD, POST et LOAD RESPONSE_INFO pour employer la connexion TCP. la connexion TCP peut être fermé en utilisant la commande de DISCONNECT. Il sera également terminé quand le fil(thread) sort le script.

la connexion ID indiquée ne doit pas correspondre à une connexion TCP déjà établi précédemment en utilisant la commande CONNECT. Autrement une erreur de script sera rapportée.

Format:

```
CONNECT TO host ON conid
```

Paramètres:

Host

Chaîne variable , chaîne de caractères citée ou expression de caractère, contenant le nom d'hôte ou le IP ADDRESS de la ressource pour s'y relier et , par option, du numéro de port sur lesquels la connexion doit être faite. Si un port est indiqué, il doit être séparé *host field* par des deux points (":"). si le champ du numéro de port est vide ou non indiqué, le port se transfère sur TCP 80.

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier définissant de la connexion ID . Ceci est employé dans toutes les opérations suivantes sur ce la connexion.

Exemples:

```
CONNECT TO "proxy.dev.mynet:3128" ON 1
CONNECT TO myhost ON 2
CONNECT TO 'abc.com' ON conid
```

Voyez également:

Commandes Diverses

Commande de DÉCONNEXION(DISCONNECT Command)

Description:

Cette commande ferme une ou tous les la connexions TCP établis utilisant la commande CONNECT. Elle est seulement valide dans un script qui a été défini comme HTTP MODE.

Si la clause " FROM conid " est indiquée, la connexion TCP identifié par cette connexion ID sera fermé. Si le " ALL " mot-clé est employé, tous les la connexions TCP établis par le fil courant seront fermés.

Par défaut, la commande DISCONNECT attendra jusqu'à ce que toutes les requêtes sur le connexion(s) soient fermés et complètes avant de les fermer. Si la clause WITH CLAUSE est indiqué, le connexion(s) sera fermé immédiatement.

la connexion ID indiquée doit correspondre à la connexion TCP établi en utilisant la commande CONNECT, autrement une erreur de script sera rapportée.

Format:

```
    DISCONNECT [FROM conid | ALL ] {,WITH CANCEL}
```

Paramètres:

conid

Une variable de nombre entier, une valeur de nombre entier ou une expression de nombre entier identifiant la connexion ID de la connexion TCP à fermer.

Exemples:

```
DISCONNECT FROM 1
DISCONNECT FROM conid
DISCONNECT FROM 1, WITH CANCEL
DISCONNECT ALL
DISCONNECT ALL, WITH CANCEL
```

Voyez également:

Commandes Diverses

Commande de la CHARGE ACTIVE_users(LOAD ACTIVE_USERS Command)

Description:

Cette commande permet le nombre de fils(thread) qui sont actuellement en activité sur le directeur test courant à charger dans une variable de nombre entier pour l'usage postérieur.

Le compte de fils actifs inclut tous les fils qui exécutent leur script primaire ou un script secondaire. Il n'inclut pas les fils qui traitent une mise en train retardent ou qui sont actuellement suspendus.

Format:

```
LOAD ACTIVE_THREADS INTO variable
```

Paramètre:

variable

Une variable de nombre entier dans laquelle le compte de fils actifs est chargé.

Exemple:

```
LOAD ACTIVE_THREADS INTO active-count
```

Voyez également:

Commandes Diverses

Commande de DATE de CHARGE(LOAD DATE Command)

Description:

Cette commande charge une variable de nombre entier avec le nombre de jours depuis la date de base de système, ou une variable de caractère avec la date de système.

Pour des variables de caractère, la date de système sera chargée dans le format de défaut de système (par exemple, " DD-MMM-CCYY "); la date sera tronquée comme exigé pour s'adapter dans la variable de cible.

Format:

```
LOAD DATE INTO variable
```

Paramètre:

variable

Le nom d'une variable de caractère ou de nombre entier dans lequel la date est chargé.

Exemples:

```
LOAD DATE INTO INT-DATE
LOAD DATE INTO CHAR-DATE
```

Voyez également:

Commandes Diverses

Commande de la CHARGE NODENAME(LOAD NODENAME Command)

Description:

Cette commande charge le nom courant de noeud dans une variable.

Format:

```
LOAD NODENAME INTO variable
```

Paramètre:

variable

Une variable de caractère dans laquelle le nom de noeud est chargé. Le nom de noeud sera tronqué comme exigé, pour s'adapter dans la variable de cible.

Exemple:

```
LOAD NODENAME INTO Node-name
```

Voyez également:

Commandes Diverses

Commande de SCRIPT de CHARGE(LOAD SCRIPT Command)

Description:

Cette commande charge le nom du script étant exécuté, dans une variable de caractère.

Format:

```
LOAD SCRIPT INTO Scriptname
```

Paramètre:

variable

Une variable de caractère dans laquelle le nom de script est chargé. Le nom de script sera tronqué comme exigé, pour remplir variable de cible.

Exemple:

```
LOAD SCRIPT INTO Scriptname
```

Voyez également:

Commandes Diverses

Commande Test de CHARGE(LOAD TEST Command)

Description:

Cette commande charge le nom du test duquel le script est une partie, dans une variable. Le nom du test sera tronqué comme exigé pour s'adapter dans la variable de cible. La taille maximum de la chaîne est retournée par cette commande est 64 caractères.

Format:

```
LOAD TEST INTO variable
```

Paramètre:

variable

Une variable de caractère dans laquelle le nom du test est chargé.

Exemple:

```
LOAD TEST INTO variable
```

Voyez également:

Commandes Diverses

Commande de FIL de CHARGE(LOAD THREAD Command)

Description:

Cette commande charge le nom du fil(thread) sur lequel le script s'exécute actuellement, dans une variable de caractère.

Déclarez la variable de caractère à 32 bytes longs, en utilisant la commande **`CHARACTER*32`**. 32 bytes devraient être assez longs pour manipuler la plupart des noms de fil.

Le nom de fil sera tronqué comme exigé pour remplir variable de cible si vous ne déclarez pas une

valeur assez grande pour faire face aux noms de fil.

Format:

```
LOAD THREAD INTO variable
```

Paramètre:

variable

Une variable de caractère dans laquelle le nom de fil est chargé.

Exemple:

```
    LOAD THREAD INTO Thread-Name
```

Voyez également:

Commandes Diverses

Commande du MOMENT de CHARGEMENT(LOAD TIME Command)

Description:

Cette commande charge une variable avec l'un ou l'autre le nombre de 10 ms de ` ticks ' depuis minuit (si la variable est une variable de nombre entier), ou le temps de système (si la variable est une variable de caractère).

Pour des variables de caractère, le temps de système sera chargé dans le format de défaut de système, tronqué si la variable n'est pas assez longue pour le tenir.

Format:

```
    LOAD TIME INTO variable
```

Paramètre:

variable

Le nom d'une variable de caractère ou de nombre entier dans laquelle le temps est chargé.

Exemples:

```
LOAD TIME INTO Int-time
LOAD TIME INTO Char-time
```

Voyez également:

Commandes Diverses

Commande de TEMPORISATEUR de CHARGE(LOAD TIMER Command)

Description:

Cette commande charge une variable de nombre entier avec la valeur courante - pendant qu'un certain nombre de 10ms fait tic tac - du temporisateur indiqué. La valeur courante d'un temporisateur est calculée en prenant le temps pour le dernier **Stop timer** et en soustrayant d'elle le moment pour le temporisateur **start timer** . Si aucune commande de temporisateur de début/ temporisateur d'arrêt n'a été exécutée pour le temporisateur indiqué par le fil courant une erreur se produira. Ceci ou avortera l'exécution de script, ou prenez l'action indiquée si le piégeage d'erreur est permis par l'intermédiaire la commande ON ERROR .

Format:

```
    LOAD TIMER name INTO variable
```

Paramètres:

name

Le nom de temporisateur. Le temporisateur doit être déclaré dans une declaration de TEMPORISATEUR dans la section de définitions du script.

variable

Le nom d'une variable de nombre entier dans laquelle la valeur de temporisateur - dans les coutils

10ms - est chargée.

Exemple:

```
LOAD TIMER Transaction INTO Timval
```

Voyez également:

Commandes Diverses

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

Closed

Unfortunately due to occasional over-use of this service we can no longer afford to provide a live version of the demonstration Web site.

If someone (or more than one person) would like to volunteer another live version then we are more than willing to provide DNS redirects and links here.

Send mail to: **demosite@opensta.org** to volunteer.

Or simply post to the **User List** to announce your servers availability

Web Applications

Why provide demonstrations?

Early on while working with OpenSTA we realised that to be able to provide any sort of examples or tutorials we needed a common Web application to provide the examples against. Knowing that users would get much more from running tests against local machines we decided to write a very trivial example application ourself. Hopefully you'll find these easy to install and they'll help you to learn and experiment with OpenSTA.

FindPres

Which US President?

This Web application provides a very simple search engine which has a login page and uses cookies to perform session tracking and timout (albeit in a very contrived fashion). This application is used throughout the OpenSTA **Getting Started Guide** and the contrived way it uses cookies was initially to illustrate a point in this document. The GSG is a bit out of date now and the last editor seems to have left some parts of the their updates incomplete... the next version will probably use an updated version of this application.

The application consists of 2 files, a perl CGI style script and a plain text database. To install simply put the perl file somewhere your Web server will treat it as a CGI executable. By default the perl script looks for its database in the same directory as itself but if you want to put this elsewhere just edit the script. These 2 files in a zip archive can be downloaded:

- **by HTTP here**
- or FTP in **ftp.opensta.org/demosite/**

As we've been asked this so many times - the easiest way to get this script running in IIS is using **ActiveState's ActivePerl** installation. This is in fact why the perl script has the filename extension it has... The next version of this application will also be available as ASP and PHP to make

installation even easier in a variety of environments. We already have working versions of these they, just need checking and packaging.

The application is very simple and its use should be self explanatory. The only start up information you should need is that the login password is checked to be the reverse of the entered username - this was done this way to allow an unlimited amount of users to be simulated without having a real user database.

---

*Proud to be **Open**, prouder to be **Free***

Questions, Comments, Suggestions?

Last Updated: 2003-DEC-26

HTTP/S Testing, Getting Started Guide

Introduction

OpenSTA is a distributed testing architecture that enables you to create and run performance Tests to evaluate Web Application Environments (WAEs) and production systems. It can be employed at all stages of WAE development as well as being used to continuously monitor system performance once a WAE goes live.

Use it to develop load Tests that include an HTTP/S load element, known as Scripts, to help assess the performance of WAEs during development, and to create Collector-only Tests that monitor and record performance data from live WAEs within a production environment.

OpenSTA enables you to run Tests against the same target system within both load testing and production monitoring scenarios. This means that you can directly compare the performance of the target system within these two environments.

This guide is intended to give new users a practical introduction to OpenSTA by explaining how to create and run a simple HTTP/S load Test targeting the demonstration Web site, *Which US President?*. It is structured according to the procedural sequence for developing an HTTP/S load Test, from Script and Collector creation through to running a Test and results display.

The key features and procedures you need to use are included in the **Contents** list below. Begin with the OpenSTA Overview section and work your way through the guide.

Before you start refer to the checklist below to ensure that your computer is correctly configured.

Checklist

- Download and run the demonstration Web site, *Which US President?*

  Launch http://opensta.org/demosite/, for download and installation instructions for this Web site.

- Download and install the latest stable version of OpenSTA. This guide was prepared with version 1.3 of OpenSTA but you should should use the latest stable release to avoid known bugs.

  Launch http://opensta.org/download.html for download and installation instructions.

Contents

Notes

- Make use of the Glossary at the end of this guide if you come across unfamiliar terminology.
- If you have already used OpenSTA and want to know how to perform a specific task, please refer to the appropriate section in the *HTTP/S Load User's Guide*, which you can view or download from OpenSTA.org.
- Use the *SCL Reference Guide* for information on SCL commands used in Script modeling, which you can view or download from OpenSTA.org.

Next Section: **OpenSTA Overview**

OpenSTA Overview

OpenSTA supplies versatile Test development software that enables you to create and run Tests tailor-made for the environment you are assessing. The contents and structure of a Test will depend on the type of test you are conducting, the nature of the system you are targeting and the aims of your performance test.

OpenSTA supports the creation of HTTP/S load Tests that include Scripts which supply the load element, and may also include Collectors which are used to record additional performance data. You can also use OpenSTA to develop Collector-only used Tests for monitoring production environments.
It is possible to use the same Test in both these scenarios.

Running a Test with both Script and Collector Task Groups enabled allows you to test and record the performance of a WAE during development. After the WAE goes live, Script-based Task Groups can be disabled and the Test re-run within a production environment. This enables you to generate useful comparative performance results and to continuously monitor the target WAE.

The example in this tutorial works through the creation of an HTTP/S load Test which includes Script-based and Collector-based Task Groups.

HTTP/S Load Test

OpenSTA is designed to create and run HTTP/S load Tests to help assess the performance of WAEs. Tests can be developed to simulate realistic Web scenarios by creating and adding Scripts to a Test to reproduce the activity of hundreds to thousands of users. Resource utilization information and response times from WAEs under test can be monitored and collected during a Test-run and then displayed. This enables you to identify system limitations or faults before launch, or after Web services have been modified, in order to help you create reliable Web sites that meet your load requirements.
Load Tests can also incorporate Collectors which monitor and record the performance of target components that comprise the system under test.

The Scripts used in a Test can be disabled when the WAE goes live allowing you to use the same Test and the Collectors it includes, to monitor and record performance data during a production-based Test-run. Test results can then be directly compared to assess the performance of the target system within a load Test and production environment.

Production Monitoring Test

OpenSTA supports the creation of Collector-only Tests. The ability to develop Tests without an HTTP/S load element enables you to create and run Tests which monitor and collect performance data from target systems in a production scenario. In this environment Tests are used to monitor and collect performance data within a production system where the load is generated externally by the normal use of the system.

OpenSTA Architecture

OpenSTA supplies a distributed software testing architecture based on CORBA which enables you to create then run Tests across a network. The OpenSTA Name Server configuration utility is the component that allows you to control your Test environment.

After installing OpenSTA you will notice that the OpenSTA Name Server is running indicated by  , in the Windows Task Bar. This component must be running before you can run a Test.

If no icon appears click **Start > Programs > OpenSTA > OpenSTA NameServer**.

If the OpenSTA Name Server stops the Name Server Configuration utility icon appears  , in the Task Bar. You can start it by right-clicking  , and selecting **Start Name Server** from the menu.

Commander

Commander is the Graphical User Interface that runs within the OpenSTA Architecture and functions as the front end for all Test development activity. It is the program you need to run in order to use OpenSTA.

Launch Commander

- Click **Start > Programs > OpenSTA > OpenSTA Commander**.

The Commander Interface

Commander combines an intuitive user interface with comprehensive functionality to give you control over the Test development process, enabling you to successfully create and run HTTP/S performance Tests.

Use the menu options or work from the Repository Window to initiate the creation of Collectors and Tests. Right-click on the predefined folders in the Repository Window and choose from the functions available.

Work within the Main Window of Commander to create Collectors and Tests. The Main Window houses the Repository Window and supplies the workspace for Test creation using the Test Pane, and Collector creation using the Collector Pane. Use Script Modeler to create the Scripts you need.

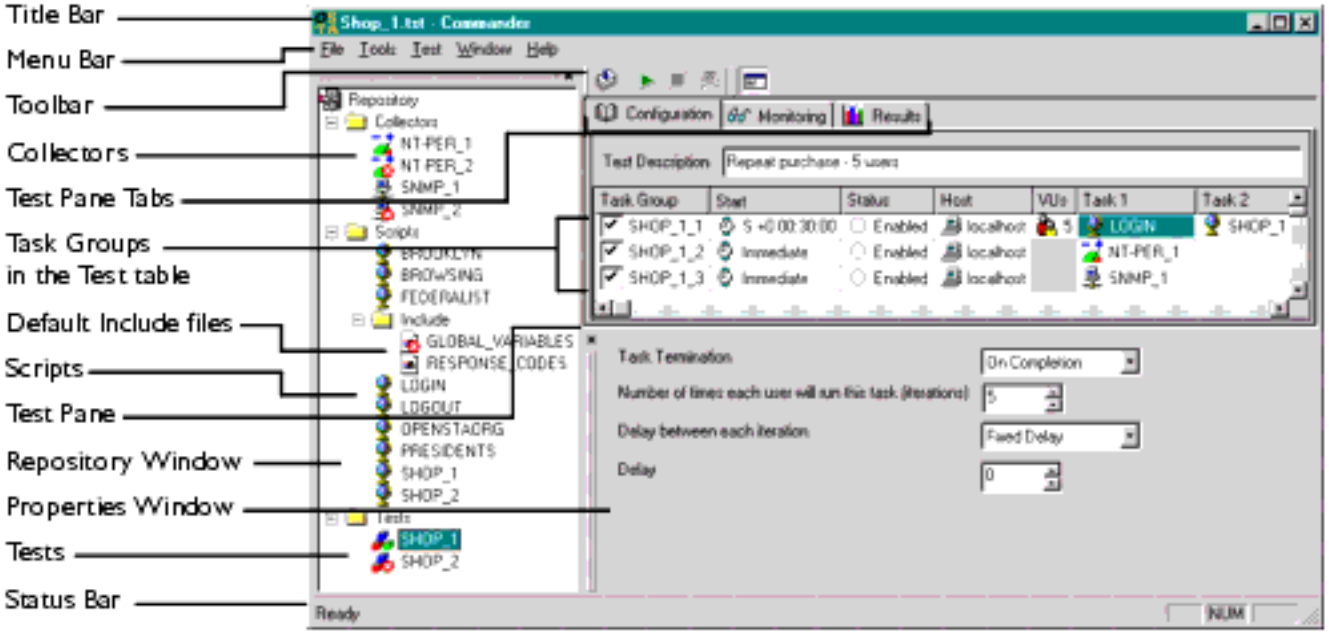
After you have created or edited a Test or Collector in the Main Window it is automatically saved when you switch to another procedure or exit from Commander.

Commander Interface Features

The Commander interface is divided up into three primary areas:

- Commander Toolbars and Function Bars.
- The Repository Window.
- The Commander Main Window.

The main features of the Commander interface are detailed below:



Next...

Now you have an overview of OpenSTA and Commander, you are ready to create a Script to include in a new Test. Move on to the next section for details on the Script creation process.

Next Section: **Recording a Script**

Back to **Contents**

A - B - C - D - E - F - G - H - I - J - K - L - M - N - O - P - Q - R - S - T - U - V - W - X - Y - Z

Index

A

B

C

D

E

F

G

H

L

M

N

O

P

R

S

T

V

W

Glossary

Collector

An OpenSTA Collector is a set of user-defined queries which determine the performance data that is monitored and recorded from target Hosts when a Test is run. They are used to monitor and collect performance data from the components of Web Application Environments (WAEs) and production systems during Test-runs to help you evaluate their performance.

Collectors are stored in the Repository and are included in Tests by reference, so any changes you make to a Collector will have immediate affect on all the Tests that use them.

The HTTP/S Load release of OpenSTA (Open Source release) supplies the NT Performance Module and the SNMP Module for Collector creation.

NT Performance Collectors are used for collecting performance data from Hosts running Windows NT or Windows 2000.

SNMP Collectors are used for collecting SNMP data from Hosts and other devices running an SNMP agent or proxy SNMP agent.

Collector Pane

The Collector Pane is the workspace used to create and edit Collectors. It is displayed in the Commander Main Window when you open a Collector from the Repository Window.

Commander

OpenSTA Commander is the Graphical User Interface used to develop and run HTTP/S Tests and to display the results of Test-runs for analysis.

Each OpenSTA Module, provides its own Plug-ins and supplies Module-specific

Test Configuration, data collection, Test-run monitoring and Results display facilities. All Plug-in functionality is invoked from Commander.

Cookie

A packet of information sent by a Web server to a Web browser that is returned each time the browser accesses the Web server. Cookies can contain any information the server chooses and are used to maintain state between otherwise stateless HTTP transactions.

Typically cookies are used to store user details and to authenticate or identify a registered user of a Web site without requiring them to sign in again every time they access that Web site.

CORBA

Common Object Request Broker Architecture.

A binary standard, which specifies how the implementation of a particular software module can be located remotely from the routine that is using the module. An Object Management Group specification which provides the standard interface definition between OMG-compliant objects. Object Management Group is a consortium aimed at setting standards in object-oriented programming. An OMG-compliant object is a cross-compatible distributed object standard, a common binary object with methods and data that work using all types of development environments on all types of platforms.

Document Object Model or DOM

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents (Web pages). It defines the logical structure of documents and the way a document is accessed and manipulated.

With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model, with a few exceptions - in particular, the DOM interfaces for the XML internal and external subsets have not yet been specified.

For more information:

- What is the Document Object Model?

  www.w3.org/TR/1998/REC-DOM-Level-1-19981001/introduction.html

- The Document Object Model (DOM) Level 1 Specification

  www.w3.org/TR/REC-DOM-Level-1/

Gateway

The OpenSTA Gateway interfaces directly with the Script Modeler Module and enables you to create Scripts. The Gateway functions as a proxy server which intercepts and records the HTTP/S traffic that passes between browser and Web site during a Web session, using SCL scripting language.

Host

An OpenSTA Host is a networked computer or device used to execute a Task Group during a Test-run. Use the Test Pane in Commander to select the Host you want to use a to run Task Group.

Host also refers to a computer or device that houses one or more components of a Web Application Environment under Test, such as a database. Use Collectors to define a Host and the type of performance data you want to monitor and collect during a Test-run

HTML

Hypertext Markup Language. A hypertext document format used on the World-Wide Web. HTML is built on top of SGML. Tags are embedded in the text. A tag consists of a **<**, a case insensitive directive, zero or more parameters and a **>**. Matched pairs of directives, like **<TITLE>** and **</TITLE>** are used to delimit text which is to appear in a special place or style.

.HTP file

See Script.

HTTP

HyperText Transfer Protocol. The client-server TCP/IP protocol used on the World-Wide Web for the exchange of HTML documents. HTTP is the protocol which enables the distribution of information over the Web.

HTTPS

HyperText Transmission Protocol, Secure. A variant of HTTP used by Netscape for handling secure transactions. A unique protocol that is simply SSL underneath HTTP. See SSL.

HTTP/S

Reference to HTTP and HTTPS.

HTTP/S Load

This release of OpenSTA includes the following Modules/components:

The release includes:

- Script Modeler Module - used to capture, model and replay HTTP/S-based data.
- OpenSTA Architecture.
- SNMP Module - Collector creation for performance data monitoring and recording.
- NT Performance Module - Collector creation for performance data monitoring and recording.

HTTP/S Load User's Guide

Hard copy and on-line versions of this guide are available.

In Commander click **Help > Commander Help > Contents**.

You can view or download a copy from **http://opensta.org/**

Load Test

Using a Web Application Environment in a way that would be considered operationally normal with a normal to heavy number of concurrent Virtual Users.

Modules

See OpenSTA Modules.

Monitoring Window

The Monitoring Window lists all the display options available during a Test-run or a single stepping session in a directory structure which you can browse through to locate the monitoring option you need. Each Task Group included in the Test is represented by a folder which you can double-click on to open and view the display options contained.

Use the Monitoring Window to select and deselect display options in order to monitor the Task Groups included in your Test and to view additional data categories, including summary data and an error log. The monitoring display options available vary according on the type of Task Groups included in a Test.

The Monitoring Window is located on the right-hand side of the Monitoring Tab by default, but can be moved or closed if required.

Name Server

See OpenSTA Name Server.

O.M.G.

Object Management Group. A consortium aimed at setting standards in object-oriented programming. In 1989, this consortium, which included IBM Corporation, Apple Computer Inc. and Sun Microsystems Inc., mobilized to create a cross-compatible distributed object standard. The goal was a common binary object with methods and data that work using all types of development environments on all types of platforms. Using a committee of organizations, OMG set out to create the first Common Object Request Broker Architecture (CORBA) standard which appeared in 1991. The latest standard is CORBA 2.2.

Open Source

A method and philosophy for software licensing and distribution designed to encourage use and improvement of software written by volunteers by ensuring that anyone can copy the source code and modify it freely.

The term Open Source, is now more widely used than the earlier term, free software, but has broadly the same meaning: free of distribution restrictions, not necessarily free of charge.

OpenSTA Dataname

An OpenSTA Dataname comprises between 1 and 16 alphanumeric, underscore or hyphen characters. The first character must be alphabetic.

The following are not allowed:

- Two adjacent underscores or hyphens.
- Adjacent hyphen and underscore, and vice versa.
- Spaces.
- Underscores or hyphens at the end of a dataname.

Note: Where possible avoid using hyphens in the names you give to Tests, Scripts and Collectors. The hyphen character functions as an operator in SCL and conflicts can occur during Test-runs.

OpenSTA Modules

OpenSTA is a modular software system that enables users to add additional functionality to the system by installing new OpenSTA Modules. When a new Module is installed existing functionality is enhanced, enabling users to develop their configuration of OpenSTA in line with their performance Testing requirements. Each Module comes complete with its own user interface and run-time components.

OpenSTA Modules are separate installables that bolt on to the core architecture to add specific functionality, including performance monitoring and data

collection for all three layers of Web Application Environment activity:

- Low-level - Hardware/Operating System performance data
- Medium-level - Application Performance Data
- High-level - Transaction Performance Data

OpenSTA Name Server

The OpenSTA Name Server allows the interaction of multiple computers across a variety of platforms in order to run Tests. The Name Server's functionality is built on the Object Management Group's CORBA standard.

Performance Test

One or more Tests designed to investigate the efficiency of Web Application Environments (WAE). Used to identify any weaknesses or limitations of target WAEs using a series of stress Tests or load Tests.

Proxy Server

A proxy server acts as a security barrier between your internal network (intranet) and the Internet, keeping unauthorized external users from gaining access to confidential information on your internal network. This is a function that is often combined with a firewall.

A proxy server is used to access Web pages by the other computers. When another computer requests a Web page, it is retrieved by the proxy server and then sent to the requesting computer. The net effect of this action is that the remote computer hosting the Web page never comes into direct contact with anything on your home network, other than the proxy server.

Proxy servers can also make your Internet access work more efficiently. If you access a page on a Web site, it is cached on the proxy server. This means that the next time you go back to that page, it normally does not have to load again from the Web site. Instead it loads instantaneously from the proxy server.

Repository

The OpenSTA Repository is where Scripts, Collectors, Tests and results are stored. The default location is within the OpenSTA program files directory structure. A new Repository is automatically created in this location when you run Commander for the first time.

You can create new Repositories and change the Repository path if required. In Commander click **Tools > Repository Path**.

Manage the Repository using the Repository Window within Commander.

Repository Host

The Host, represented by the name or IP address of the computer, holding the OpenSTA Repository used by the local Host. A Test-run must be started from the Repository Host and the computer must be running the OpenSTA Name Server.

Repository Window

The Repository Window displays the contents of the Repository which stores all the files that define a Test. Use the Repository Window to manage the contents of the Repository by creating, displaying, editing and deleting Collectors, Scripts and Tests.

The Repository Window is located on the left-hand side of the Main Window by default and displays the contents of the Repository in three predefined folders  **Collectors**,  **Scripts**, and  **Tests**. These folders organize the contents of the Repository into a directory structure which you can browse through to locate the files you need.

Double-click on the predefined folders to open them and display the files they contain.

Right-click on the folders to access the function menus which contain options for creating new Collectors, Scripts and Tests.

Results Window

The Results Window lists all the results display options available after a Test-run or a single stepping session is complete. The display options are listed in a directory structure which you can browse through to locate the results option you need. Each Collector-based Task Group included in the Test is represented by a folder which you can double-click on to open and view the display options contained.

Use the Results Window to select and deselect display options in order to view and analyze the results data you need. The results display options available vary according on the type of Task Groups included in a Test.

The Results Window is located on the right-hand side of the Results Tab by default, but can be moved or closed if required.

SCL

See Script Control Language.

SCL Reference Guide

Hard copy and on-line versions of this guide are available.

In Script Modeler click **Help > SCL Reference**.

Script

Scripts form the basis of HTTP/S load Tests using OpenSTA. Scripts supply the HTTP/S load element used to simulate load against target Web Application Environments (WAE) during their development.

A Script represents the recorded HTTP/S requests issued by a browser to WAE during a Web session. They are created by passing HTTP/S traffic through a proxy server known as the Gateway, and encoding the recorded data using Script Control Language (SCL). SCL enables you to model the content of Scripts to more accurately generate the Web scenario you need reproduce during a Test.

Scripts encapsulate the Web activity you need to test and enable you to create the required Test conditions. Use Commander to select Scripts and include them in a Test then run the Test against target WAEs in order to accurately simulate the way real end users work and help evaluate their performance.

Scripts are saved as an .HTP file and stored in the Repository.

Script Control Language

SCL, Script Control Language, is a scripting language created by CYRANO used to write Scripts which define the content of your Tests. Use SCL to model Scripts and develop the Test scenarios you need.

Refer to the *SCL Reference Guide* for more information.

Script Modeler

Script Modeler is an OpenSTA Module used to create and model Scripts produced from Web browser session recordings, which are in turn incorporated into performance Tests by reference.

Script Modeler is launched from Commander when you open a Script from the Repository Window.

Single Stepping

Single stepping is a debugging feature used to study the replay of Script-based Task Groups included in an HTTP/S load Test. Run a single stepping session to follow the execution of the Scripts included in a Task Group to see what actually happens at each function call, or when a process crashes.

SNMP

Simple Network Management Protocol. The Internet standard protocol developed to manage nodes on an IP network. SNMP is not limited to TCP/IP. It can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs, toasters and jukeboxes.

For more information visit the NET_SNMP Web site:

- What is it? (SNMP)

  http://net-snmp.sourceforge.net/

SSL

Secure Sockets Layer. A protocol designed by Netscape Communications Corporation to provide encrypted communications on the Internet. SSL is layered beneath application protocols such as HTTP, SMTP, Telnet, FTP, Gopher, and NNTP and is layered above the connection protocol TCP/IP. It is used by the HTTPS access method.

Stress Test

Using a Web Application Environment in a way that would be considered operationally abnormal. Examples of this could be running a load test with a significantly larger number of Virtual Users than would normally be expected, or running with some infrastructure or systems software facilities restricted.

Task

An OpenSTA Test is comprised of one or more Task Groups which in turn are composed of Tasks. The Scripts and Collectors included in Task Groups are known as Tasks. Script-based Task Groups can contain one or multiple Tasks. Tasks within a Script-based Task Group can be managed by adjusting the Task Settings which control the number of Script iterations and the delay between iterations when a Test is run.

Collector-based Task Groups contain a single Collector Task.

Task Group

An OpenSTA Test is comprised of one or more Task Groups. Task Groups can be of two types, Script-based or Collector-based. Script-based Task Groups represent one or a sequence of HTTP/S Scripts. Collector-based Task Groups represent a single data collection Collector. Task Groups can contain either Scripts, or a Collector, but not both. The Scripts and Collectors included in Task Groups are known as Tasks.

A Test can include as many Task Groups as necessary.

Task Group Definition

An OpenSTA Task Group definition constitutes the Tasks included in the Task Group and the Task Group settings that you apply.

Task Group Settings

Task Group settings include Schedule settings, Host settings, Virtual User settings and Task settings and are adjusted using the Properties Window of the Test Pane. Use them to control how the Tasks and Task Group that comprise a Test behave when a Test is run.

Schedule settings determine when Task Groups start and stop.

Host settings specify which Host computer is used to run a Task Group.

Virtual User settings control the load generated against target Web Application Environments during specifying the number of Virtual Users running a Task Group. Set Logging levels to determine the amount of performance statistics collected from Virtual Users running the Tasks. You can also select to Generate Timers for each Web page returned during a Test-run.

Task settings control the number of times a Script-based Tasks are run including the delay you want to apply between each iteration of a Script during a Test-run.

Test

An OpenSTA Test is a set of user controlled definitions that specify which Scripts and Collectors are included and the settings that apply when the Test is run. Scripts define the test conditions that will be simulated when the Test is run. Scripts and Collectors are the building blocks of a Test which can be incorporated by reference into many different Tests.

Scripts supply the content of a Test and Collectors control the type of results data that is collected during a Test-run. Test parameters specify the properties that apply when you run the Test, including the number of Virtual Users, the iterations between each Script, the delay between Scripts and which Host computers a Test is run.

Commander provides you with a flexible Test development framework in which you can build Test content and structure by selecting the Scripts and Collectors you need. A Test is represented in table format where each row within it represents the HTTP/S replay and data collection Tasks that will be carried out when the Test is run. Test Tasks are known as Task Groups of which there are two types, either Script-based and Collector-based.

Test Pane

The Test Pane is the workspace used to create and edit Tests, then run a Test and monitor its progress. After a Test-run is complete results can be viewed and compared here. The Test Pane is displayed in the Commander Main Window when you open a Test from the Repository Window.

Transaction

A unit of interaction with an RDBMS or similar system.

URI

Uniform Resource Identifier. The generic set of all names and addresses which are short strings which refer to objects (typically on the Internet). The most common kinds of URI are URLs and relative URLs.

URL

Uniform Resource Locator. A standard way of specifying the location of an object, typically a Web page, on the Internet. Other types of object are described below. URLs are the form of address used on the World-Wide Web. They are used in HTML documents to specify the target of a hyperlink which is often another HTML document (possibly stored on another computer).

Variable

Variables allow you to vary the fixed values recorded in Scripts. A variable is defined within a Script. Refer to the Modeling Scripts section for more information.

Virtual User

A Virtual User is the simulation of a real life user that performs the activity you specify during a Test-run. You control the activity of your Virtual Users by recording and modeling the Scripts that represent the activity you want. When the Test that includes the Script is run, the Script is replayed exactly as the browser made the original requests.

Web Application Environment, WAE

The applications and/or services that comprise a Web application. This includes database servers, Web servers, load balancers, routers, applications servers, authentication/encryption servers and firewalls.

Web Applications Management, WAM

Consists of the entirety of components needed to manage a Web-based IT environment or application. This includes monitoring, performance testing, results display, results analysis and reporting.

Web Site

Any computer on the Internet running a World-Wide Web server process. A particular Web site is identified by the host name part of a URL or URI. See also Web Application Environment, WAE.

Running a Test Over the Web

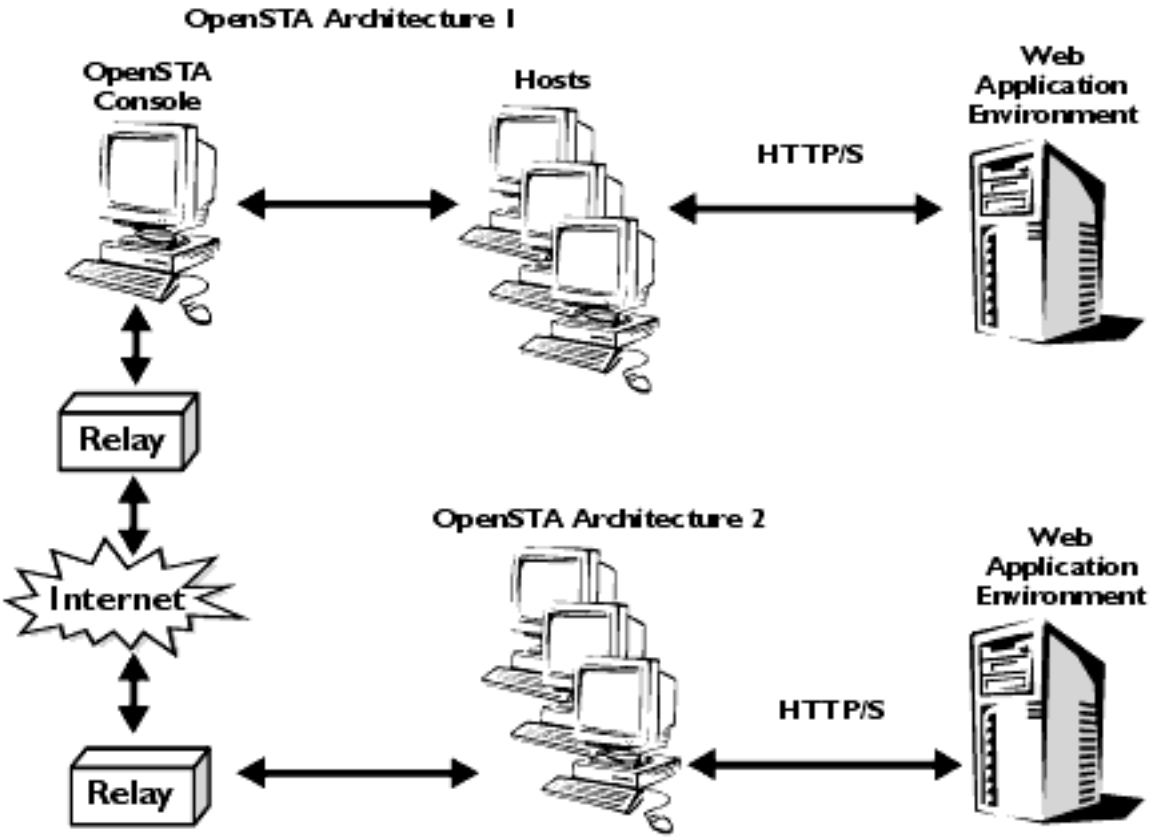
OpenSTA's distributed software architecture enables Test execution on remote Web-based Hosts. This is achieved using a Web Relay Daemon facility which allows the CORBA-based communications within the OpenSTA architecture to be transmitted between machines that are located over the Web.

The Web Relay Daemon facilitates configuration of the components that comprise the Web Relay environment. These consist of the Web Relay Daemon, a Web server and the OpenSTA architecture. Normal Test control communications use XML over HTTP. OpenSTA Web-based replay allows two modes of file transfer: HTTP or FTP. The system also allows SSL-based data transfer.

Use the Web Relay Daemon to map all the machines that need to connect to one another in an OpenSTA architecture which includes Web-based machines. These facilities offer the full range of OpenSTA communications between single or groups of Web-based machines running OpenSTA.

After configuring the Web Relay Daemon remote Hosts can be selected to run a Task Group as normal. For more information see Select the Host used to Run a Task Group.

Web Relay Daemon Architecture



Note: OpenSTA Console refers to a Host computer that has an installation of OpenSTA. This includes the OpenSTA Architecture and Commander, and may also include the Repository, where all Test related files and results are stored.

Configuring the Web Relay Daemon

The configuration of the Web Relay Daemon involves:

- Configuring the Web Server
- Configuring the Relay Map
- Setting the Trace Level

Configuring the Web Server

1. Activate the OpenSTA Web Relay facility:
   Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon appears in the task bar.
   Note: It is assumed that you already have a Web server installed that supports ISAPI.
2. Right-click on and select **Edit Server Settings** from the pop-up menu to open the Server Settings window.

Note: If the Web Relay Daemon is inactive the icon is visible.

3. Enter the port number of the local Web server in the **Port** field.
4. Check the **Use SSL** option if SSL security is required.
5. Type the path and root directory of the Web server in the **Root Directory** field.
   A DLL is automatically entered in the **ISAPI Extension** field and a cache file in the **File Cache** field.
6. If you want to use FTP file transfer for data transmission, check the Enable FTP File Transfer option and enter your settings in the complete the optional Local FTP Server Settings fields.
7. Click on **Save** to apply your settings.

Configuring the Relay Map

1. Activate the OpenSTA Web Relay facility:
   Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon appears in the task bar.
   Note: It is assumed that you already have a Web server installed that supports ISAPI.
2. Right-click on and select **Edit Relay Map** from the pop-up menu to open the Edit Relay Map Settings window.
   Note: If the Web Relay Daemon is inactive the icon is visible.
3. Click on in the toolbar to add the Relay Map settings of the remote Host you want to connect to.
4. In the Edit Relay Map Settings window enter the name of the remote host in the **Alias** field.
5. In the **IP Address** field, enter the IP address of the remote host.
6. Type the ISAPI extension in the **Extension Path** field.
   Note: This entry is identical to the one in the ISAPI Extension field in the Web server configuration settings.
7. Enter the port number of the Web server in the **Port** field.
8. In the **User Name** field, enter the user name.
9. Type the password in the **Password** field.
10. Check the **Use SSL** option if SSL security is required.
11. Click **OK** to confirm the Relay Map settings.
    Note: Repeat this process on the remote Host to complete the mapping of the two machines.

Setting the Trace Level

1. Activate the OpenSTA Web Relay facility:
   Click **Start > Programs > OpenSTA > OpenSTA Web Relay**. The Web Relay Daemon icon  appears in the task bar.
   Note: It is assumed that you already have a Web server installed that supports ISAPI.
2. Right-click on  and select **Set Trace Level** from the pop-up menu to open the Set Trace Level dialog box.
   Note: If the Web Relay Daemon is inactive the  icon is visible.
3. Click  to the right of the **Trace Level** field and select a trace level setting from the drop down list.
   Tip: The trace level you select effects the amount of information you receive about the Test executer processes if problems are encountered during a Test-run. The default setting is `None'.
4. Click on **OK** to confirm the setting.

Next Section: **Glossary**

Back to **Contents**

Increase the Load Generated During a Test-run

In order to increase the amount of data generated during a Test-run and to produce some varied results for analysis, you can raise the load generated against the target Web site by modifying some of the Script-based Task Group settings.

To increase the load generated during a Test-run you can:

- Edit the Number of Script Iterations and the Delay Between Iterations

Increasing the number of Script iterations from one to five for example, will result in each Virtual User running a Script from beginning to end five times.

You can also increase the load by increasing the number of Virtual Users:

- Specify the Number of Virtual Users to run a Script-based Task Group

Try experimenting with your Task Group settings and running the Test again. Allocate five Virtual Users and start them at five second intervals. This technique allows you to ramp-up the load generated against the target Web site.

Use the Batch Start Options when defining your Virtual User settings to control when and how many Virtual Users are active during a Test-run.

Use the results display functions to view and compare the performance data you have collected.

Conclusion

The Test you have created and run whilst working through this guide will

hopefully have given you an understanding of the basic techniques involved in successfully developing HTTP/S performance Tests using OpenSTA. As well as indicating the potential of the software for analyzing your own Web sites and improving their performance.

Note: OpenSTA's distributed architecture enables you to utilize remote Hosts to execute Task Groups during a Test-run across the Web. For more information on this functionality move on to the next section.

Next Section: **Running a Test Over the Web**

Back to **Contents**

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Documentation feedback

Displaying Test Results

After a Test-run is complete, results are stored in the Repository from where they are available for immediate display and analysis. The data collected can be displayed alongside results from previous Test-runs associated with the Test, to provide comparative information about the performance of target systems.

Use Commander to control which results are displayed and how they are presented, in order to help you analyze the performance of target WAEs and the network used to run the Test.

Open the Test you want from the Repository Window and click on the  **Results** Tab in the Test Pane, then choose the results you want to display using the Results Window. Depending on the category of results you select, data is displayed in graph or table format. You can choose from a wide range of tables and customizable graphs to display your results which can be filtered and exported for further analysis and print.

Use the Results Window to view multiple graphs and tables simultaneously to compare results from different Test-runs.

When a Test is run a wide range of results data is collected automatically. Virtual User response times and resource utilization information is recorded from all Web sites under test, along with performance data from WAE components and the Hosts used to run the Test.

Results categories include the Test Summary option which presents a brief description of the Test and the Task Groups settings that applied during a Test-run. The Test Audit log records significant events that occur during a Test-run and the HTTP Data List records the HTTP/S requests issued, including the response times and codes for every request. The Timer List option records the length of time taken to load each Web page defined in the Scripts referenced by a Test.

Creating and referencing Collectors in a Test helps to improve the quality and

extend the range of the results data produced during a Test-run. NT Performance and SNMP Collectors give you the ability to target the Host computers and devices used to run a Test and the components of WAEs under test, with user-defined data collection queries.

Results collected by all the SNMP Collectors included in a Test are saved in the Custom SNMP file. Results collected by all the NT Performance Collectors you include are saved in the Custom NT Performance file. Results are displayed by opening a Test, then using the Results Window displayed in the Results Tab of the Test Pane to open the display options listed. Results data can be can be exported to spreadsheet and database programs for further analysis, or printed directly from Commander.

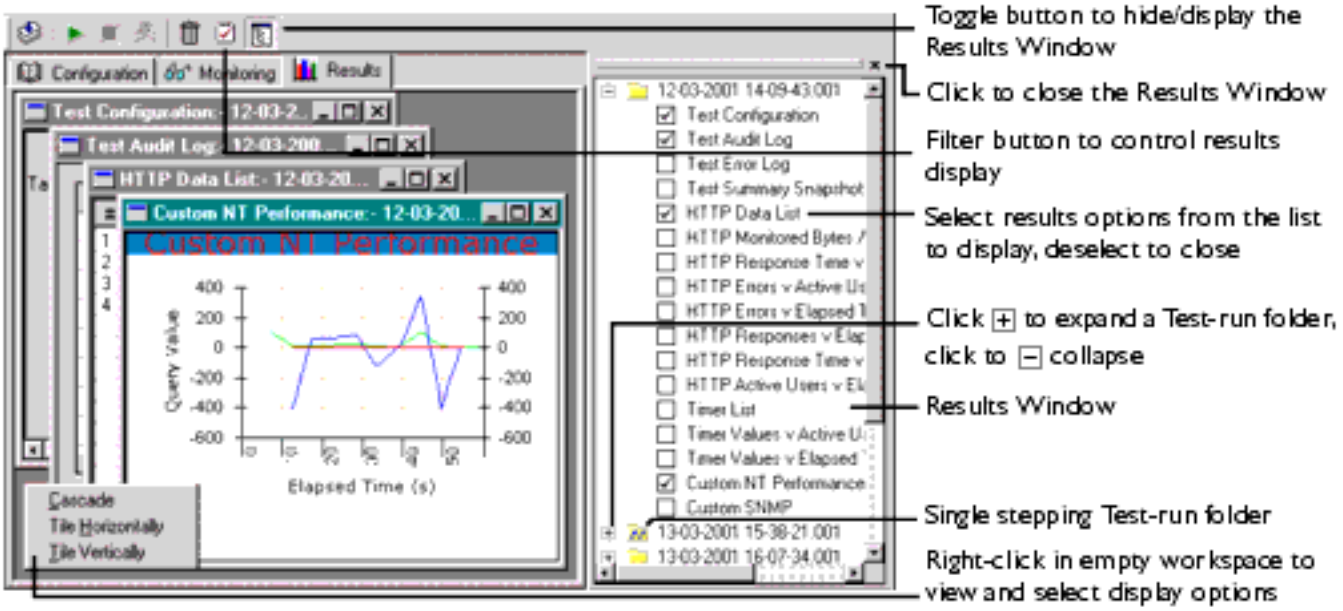
The range of results produced during a Test-run can depend on the content of the Scripts that are referenced in a Test. For example Report and History logs are only produced if the Scripts included have been modeled to incorporate the SCL commands used to generate the data content for these logs.

Results Tab

Results are stored in the Repository after a Test-run is complete. You can view them by working from the Repository Window to open the Test you want, then click on the **Results** Tab in the Test Pane.

Use the Results Window to select the results you want to view in the workspace of the Test Pane. You can reposition the Results Window by floating it over the Main Window to give yourself more room for results display, or close it once you have selected the results options you want to view.

The Results Tab of the Test Pane



The Results Window

When you click on the **Results** Tab, the Results Window opens automatically. Its default location is on the right-hand side of the Test Pane where it is docked. Use it to select and display results from any of the Test-runs associated with the current Test.

Test-runs are stored in date and time stamped folders which you can double-click on to open, or click ⊞ . When you open a Test-run folder, the available results are listed below. Display the results you want by clicking on the options and ticking the check boxes to the left of the results options. The results you choose are displayed in the Test Pane.

Multiple graphs and tables from different Test-runs associated with the current Test can be displayed concurrently. Use the Results Window to select additional Test-runs and equivalent results options to compare Test results and help evaluate performance.

Display Test Results

1. In the Repository Window, double-click **Tests** to expand the directory structure.
2. Double-click **PRESIDENT_SEARCH** , to open the Test.
3. In the Test Pane click the **Results** Tab.

   The Results Window opens automatically listing all Test-runs associated with the current Test. Results are stored in date and time stamped folders.
4. In the Results Window, double-click on a Test-run folder or click ⊞ , to open it and display the available results.
5. Click on a results option to display your selection in the Test Pane.

   A ticked check box to the left of a results option indicates that it is open in the Test Pane.

   Note: Click , in the title bar of a graph or table to close it or deselect the results option in the Results Window by clicking on the option.

   Tip: All available results have display and output options associated with them, These may include filtering, customizing and exporting. Right-click within a graph or table to display and select from the choices available.

   Use the **Windows** option in the Menu Bar to control the display of graphs and tables. Alternatively, right-click within the empty workspace of the Test Pane to access these functions.

Results Tab Display Options

Graphs can be customized to improve the presentation of data by right-clicking within a graph then selecting **Customize**. This function includes options that enable you to modify the graph style from the default line plot to a vertical bar, as well as controlling the color of elements within the graph display.

You can control the information displayed in some graphs and tables by filtering the data they represent. Right-click within a graph or table, then select **Filter** or **Filter URLs**, or click the Filter button  in the toolbar and make your selection. You can also opt to export results data for further analysis and printing. Right-click and select **Export to Excel** or **Export** from the menu.

You can also zoom in on a graph by clicking and dragging over the area of the graph you want to study. Use the **Windows** option to control the presentation of results options in the Test Pane, or right-click within the empty workspace of the Test Pane to access these functions as illustrated in the diagram above.

Next...

Now that you have created and run your first Test, try experimenting with the Task Group settings to increase the load produced and generate a variety of results for comparison.
Move on to the next section for details.

Next Section: **Increase the Load Generated During a Test-run**

Back to **Contents**

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Documentation feedback

Single Stepping HTTP/S Load Tests

Make use of the single stepping functionality provided during Test development to check your HTTP/S load Tests and to help resolve errors that may occur during a Test-run.

When you run a Script-based Task Group within a single stepping session HTTP is returned from target WAEs in response to the Scripts that are executed during replay. You can single step through the browser requests contained in Scripts and monitor HTTP responses to check that the Task Group is behaving as required.

The main difference between single stepping a Script-based Task Group and a normal Test-run, is that replay can be paused by inserting breakpoints against the HTTP requests included in Scripts. When a breakpoint is reached Task Group replay is halted allowing you to study the HTTP returned from the target WAE for all HTTP requests issued before the breakpoint. Breakpoints can be inserted before and during a Test-run, or you can single step through the Test-run using the Single Step button  on the toolbar.

The number of Virtual Users running a Task Group can be set within the session, either to the number previously configured, or to one, for the duration of the replay. Cutting down the number of Virtual Users reduces the amount of HTTP data returned, which is useful if you are running a large volume load Test that involves hundreds or thousands of Virtual Users.

You can also select the Script items you want to view during replay using the toolbar display buttons to help make monitoring easier. Comments, Transaction Timers and Call Scripts need to be manually inserted by modeling Scripts before running a single stepping session. Comments are useful to help explain the HTTP requests and other Script content during replay. Transaction Timers are used to measure the duration of user-defined HTTP transactions when a Test is run. The Call Script command enables you to execute a Script that is not included in the Task Group. For more information on modeling Scripts in

preparation for a single stepping session refer to the *HTTP/S Load User's Guide*.

When you replay a Task Group during a single stepping session it runs in the same way as a normal Test-run. It is executed according to existing Task Group settings and any changes you may have made to the Virtual User settings from within the single stepping session.

Results collected during a single stepping session are unrealistic in comparison to data from a a normal Test-run. In this mode the continuous replay characteristic of a regular Test-run, is disrupted by breakpoints and some Task Group settings are overridden. Single Stepping results can assist you in Test development but are unreliable for WAE performance analysis.

Begin a single step session by opening a Test, right-clicking on a Script-based Task Group in the Test table, then selecting **Single Step Task Group** from the menu. Use the Configuration Tab to setup your Scripts before running a Task Group. Select the Script you want to configure from the Tasks selection box at the bottom of the display. Then choose the Script items you want to view by right-clicking inside the Script Item list and picking the display options required. Insert the breakpoints you need by right-clicking on an HTTP request, then selecting **Insert/Remove Breakpoint**.

Run the Task Group by clicking  in the Monitoring Tab toolbar. When a breakpoint is reached Task Group replay is halted allowing you to view the WAE responses displayed in the HTTP section at the bottom of the Monitoring Tab. Make sure that the HTTP check box to the right of an HTTP request is ticked before you run the Task Group if you want to view the HTTP returned.

If you are using the single step method there is no need to add breakpoints in the Script Item list before running a Task Group. Simply click on the Monitoring Tab, then click  to run the Task Group one HTTP request at a time. After an HTTP request is issued and the HTTP response is complete, the replay is automatically paused. Move through the Task Group from one HTTP request to the next by clicking  until replay is complete. You can click  at any stage to revert to continuous replay and use the Break button  in the toolbar to insert a break and pause the run.

To stop the Task Group replay click  in the Monitoring Tab toolbar. To end a single step session click  in the Configuration Tab toolbar.

Configure a Single Stepping Session

Note: Before beginning a single stepping session you should compile the Test by clicking  , in the toolbar, to check that the Scripts it contains are valid.

1. Open a Test with the **Monitoring** tab of the Single Stepping Test Pane displayed.
2. Right-click on a Script-based Task Group and select **Single Step Task Group** from the menu. The first Script Task in a sequence is displayed by default in the Monitoring tab.
3. The Script or sequence of Scripts included in the Task Group are listed in the Tasks selection box at the bottom of the Single Step window.

   Click on a Script Task to display it in the workspace above.

4. The top section of the window displays the List tab view by default which includes a list of Script items included in the selected Script.
   Use the toolbar to select the Script items you want to display. Primary HTTP requests are always displayed. Choose from:

   Secondary URLs **Timers**
   Comments **Transaction Timers**
   Waits **Call Scripts**

   Note: The display options you select apply to all the Scripts in the Task Group.

   Tip: Use the Script tab to view the SCL commands that constitute the Script.
   You can double-click on any Script item to display the SCL commands associated with your selection in the Script tab.

5. Insert a breakpoint on an HTTP request by right-clicking on the request then selecting **Insert/Remove Breakpoint**.

   Breakpoints can be inserted on Primary HTTP requests and Secondary Gets. They are indicated by to the left of the HTTP request.

   Note: Breakpoints inserted using this method are saved after you end the single stepping session. Breakpoints inserted using the Break button are temporary and not saved.

   Use the HTTP check boxes to the right of an HTTP request to control whether HTTP responses are displayed when the Task Group is run. By default the check boxes are checked and HTTP is returned for all requests.

6. Click on a check box to check or uncheck it.

   Tip: You can quickly configure the HTTP data displayed during replay for all HTTP requests by clicking on the column title **HTTP** to select the entire column, then uncheck or check a single check box to uncheck or check all boxes.

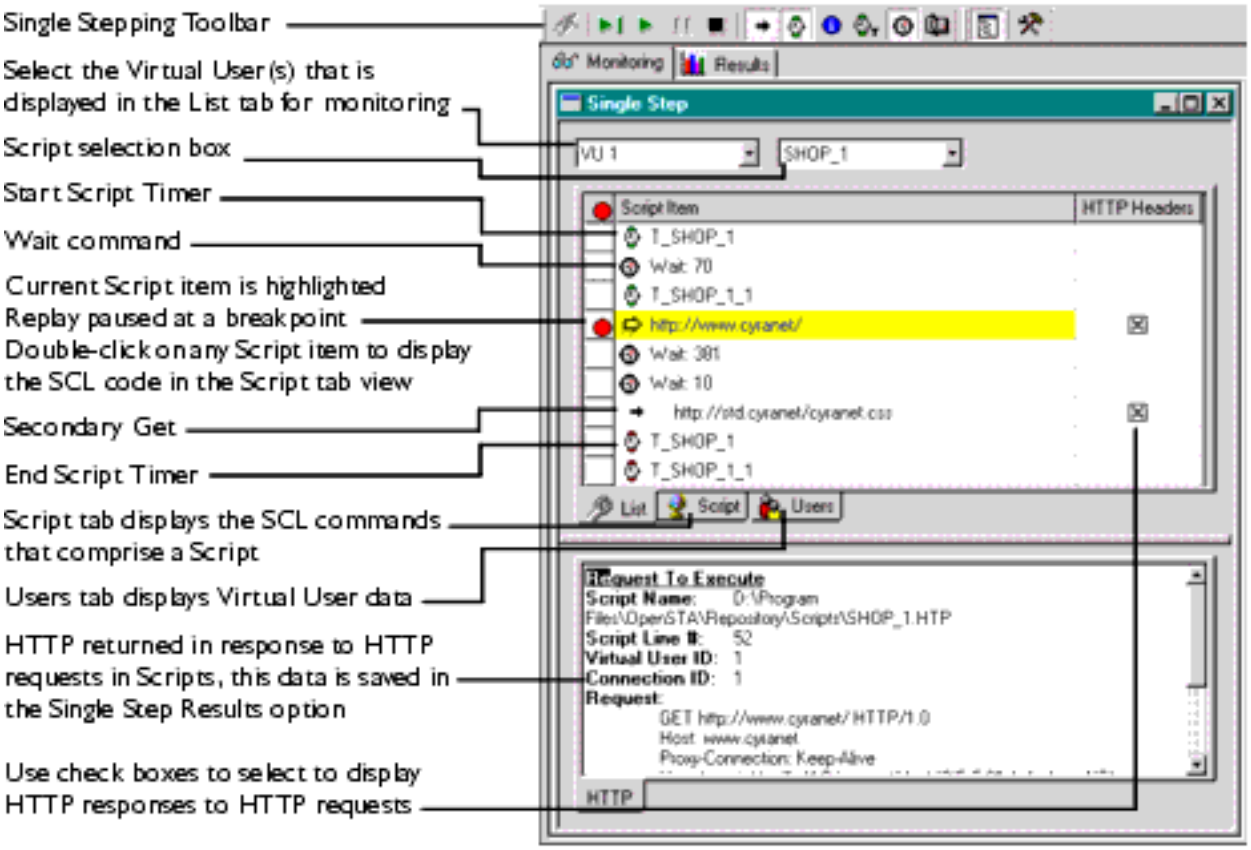
Run a Single Stepping Session

1. Open a Test with the **Monitoring** tab of the Single Stepping Test Pane displayed.
2. Right-click on a Script-based Task Group and select **Single Step Task Group** from the menu.
3. Make sure that you have configured your Scripts.
4. Run the Task Group from the Monitoring tab by clicking in the toolbar to replay up to the first breakpoint.

   Or click in the toolbar to replay the Task Group one HTTP request at a time.
   Replay is automatically halted after the response is complete. Keep clicking to single step through the Task Group.

   Tip: Use the break button , to pause the replay.

   You can monitor the replay of the Task Group from the List tab, Script tab or Users tab in the Scripts Item list. Use the Monitoring Window options to view data categories collected during replay.



Note: While the replay is paused you can reconfigure your Task Group

replay options from the Monitoring tab if required. You can insert new breakpoints, edit Task Group settings control the HTTP returned and change the Script items you display.

5. Click the Run button  or the Step button  to restart the Task Group replay.

   Use the break button  , to pause the replay. Click  or  to restart.

6. Click  to stop the Task Group replay.

7. End a single stepping session from the Monitoring tab by clicking  in the toolbar.

   On completion of the Test-run click the  **Results** tab and use the Results Window to access the **Single Step Results** option.

   The Test-run folders that store single stepping session results are identified  , to distinguish them from normal test-run folders  .

Next...

After you have run your Test, use the results display functions to view the data collected during the Test-run. Move on to the next section for details on how to do this.

Next Section: **Displaying Test Results**

Back to **Contents**

Running a Test

Running a Test enables you to simulate real end user Web activity and to generate the results data you need in order to analyze and assess the performance of target WAEs.

Running a Test is a straightforward procedure, because the Task Group settings have already been specified during Test creation. Open the Test you want to run and click the Start Test button  , in the toolbar.

Dynamic Tests

In OpenSTA Tests are dynamic, which means that the Test contents and settings can be modified while it is running, giving you control over a Test-run and the results that are generated.

New Task Groups can be added and the contents and settings of the existing Task Groups that comprise a Test can be individually edited by temporarily stopping the Task Group, making the changes required, then restarting them. These facilities give you control over the load generated and enable you to modify the type of performance data you monitor and record without stopping the Test-run.

Note: It is not possible to remove a Task Group from a Test during a Test-run.

While a Test is running you can:

- Add a new Task Group.

  Scripts and Collectors can be added to a Test and the Task Groups that contain them started.

- View the settings and status of Task Groups using the Properties Window

and the Status column of the Configuration Tab.

- Modify Task Group settings when the selected Task Group has stopped.

  These settings are:

  Schedule settings

  Host settings

  Virtual User settings (Script-based Task Groups only)

  Task settings (Script-based Task Groups only)

- Stop/Start a Task Group.

  Task Groups can be stopped and started during a Test-run using the **Stop** and **Start** buttons in the new Control column of the Configuration Tab. The **Stop** button is displayed if the Task Group is Active and a **Start** button is displayed if the Test is running and the Task Group is stopped, otherwise no button is displayed.

Run a Test

1. In the Repository Window, double-click **Tests** to open the folder and display the Tests contained.
2. Double-click the Test, **PRESIDENT_SEARCH** , you want to run, which launches the Test Pane in the Commander Main Window.
3. Check the Test contains the Scripts and Collectors you want and that the Task Group settings are correct, then click in the toolbar to run the Test.

   Note: When you click , the Test is automatically compiled. If there is an error during compilation the Compile Errors dialog box appears with a description of the fault(s) to assist you in resolving any problems.

   After your Test has been compiled successfully, the Starting Test dialog box appears which displays a brief status report on the Test-run.

   Tip: Click on the **Monitoring** Tab within the Test Pane during a Test-run and select a Collector or Task Group, to monitor the performance of target Web sites and Hosts used to run the Test, in graph and table format.

Monitoring a Test-run

Task Groups and the Scripts and Collectors they contain can be monitored using

the Monitoring Tab of the Test Pane during a Test-run. When you run a Test that includes Collectors you can monitor:

- A summary of current Test-run activity.
- Script-based Task Groups: All the Scripts in a Task Group and the Virtual Users currently running each Script.
- Collector-based Task Groups: All the data collection queries defined in a Collector.

Monitor Scripts and Virtual Users

1. Make sure the **PRESIDENT_SEARCH** Test is open and running with the  **Monitoring** Tab of the Test Pane displayed.

   Note: Ensure that the entry in the Status column of the Configuration Tab reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click ⊞ , next to a Script-based Task Group folder to open it. The Script-based Task Group folder lists the Script Tasks it contains.
3. Select a Script from the Monitoring Window to track Virtual User activity.

   Data for all the Virtual Users running the selected Script-Task are displayed in the Test Pane. The data categories are Virtual User ID, Duration, Current Script-Task iteration and Note Text connected with each Virtual User. Note text is included for the last NOTE command executed by a Virtual User.

   Note: When a Test-run is complete, the entry in the Test Status box at the top of the Monitoring Window reads **INACTIVE** and the display options in the Monitoring Window are cleared.

Monitoring Collectors

1. Make sure the **PRESIDENT_SEARCH** Test is open and running with the  **Monitoring** Tab of the Test Pane displayed.

   Note: Ensure that the entry in the Status column of the Configuration Tab reads **ACTIVE**, indicating that the Test is running.

2. In the Monitoring Window click ⊞ , to open a Task Group folder that contains an NT Performance or an SNMP Collector.

   The data collection queries defined in the selected folder are listed below. They represent the display options available.

3. Select one or more of the data collection queries you want to monitor from the Monitoring Window.

Note: When a Test-run is complete, the entry in the Test Status box at the top of the Monitoring Window reads **INACTIVE** and the display options in the Monitoring Window are cleared.

Next...

After you have run your Test, use the results display functions to view the data collected during the Test-run. Or move on to the next section for details on how to run a Test in single stepping mode.

Next Section: **Single Stepping HTTP/S Load Tests**

Back to **Contents**

Creating a Test

After you have planned your Test use Commander to coordinate the Test development process by selecting the Scripts and Collectors you need and combining them into a new Test with the required Task Group settings applied.

Use Task Group settings to control the load generated during a Test-run, which Host is used to run a Task Group and when it starts and stops during a Test-run. Select a Task Group cell in the Test table then use the Properties Window below to apply your settings.

Tests can be developed and then run using remote Hosts across a network to execute the Task Groups that comprise a Test. Distributing Task Groups across a network enables you to run Tests that generate realistic heavy loads simulating the activity of many users. In order to do this, OpenSTA must be installed on each Host and the OpenSTA Name Server must be running on each and configured to specify the Repository Host for the Test.

Tasks and Task Groups

Work from the Repository Window, located by default on the left of the Commander Main Window, to create new Tests and to open existing ones.

The Repository Window displays the contents of the Repository and functions as a picking list from where you can select the Scripts and Collectors you want to include in a Test. Use it in conjunction with the Configuration Tab of the Test Pane to develop the contents of a Test. Select a Script or Collector from the Repository Window then drag and drop it on to a Task column of a Test to create a new Task within a new Task Group.

The Scripts and Collectors you add to a Test are referred to as Tasks. One or a sequence of Script Tasks are represented by a Script-based Task Group. A Collector Task is represented by a Collector-based Task Group. When you add a Script or Collector to a Test, you can apply the Task Group settings you require or you can accept the default settings and return later to edit them.

Some of the Task Group cells in the Test table are dynamically linked to the Properties Window below, select them one at a time to display and edit the associated Task Group settings in the Properties Window.

Select the **Start** or **Host** cells in a Task Group row to control the Schedule and Host settings. Script-based Task Groups and the Script Tasks they contain have additional settings associated with them. Select the **VUs** and **Task** cells to control the load levels generated when a Test is run.

Use the Disable/Enable Task Group function to control which Task Groups are executed when a Test is run by clicking the check box in the Task Group column cell. This is a useful feature if you want to disable Script-based Task Groups to turn off the HTTP/S load element. The Test can then be used to monitor a target system within a production scenario.

Task Group Settings include:

- **Schedule Settings**: Control when a Task Group starts and stops to determine the period of data collection during a Test-run.
- **Host Settings**: Specify the Host computer used to execute a Task Group during a Test-run. The Host computers you use can be local Hosts on your Test network or remote, Web-based Hosts. For more information see Running a Test Over the Web.
- **Virtual User Settings**: Control the load generated against target WAEs during a Test-run.
- **Task Settings**: Control the number of times a Script is run and the delay you may want to apply between each iteration of a Script during a Test-run. You can also specify the type of delay, between each Script iteration, which can be either Fixed or Random.

The Test Pane

Use the Test Pane to create and edit a Test, then apply the Task Group settings you require to control how they behave during a Test-run. Run and monitor the Test-run then display your results for analysis.

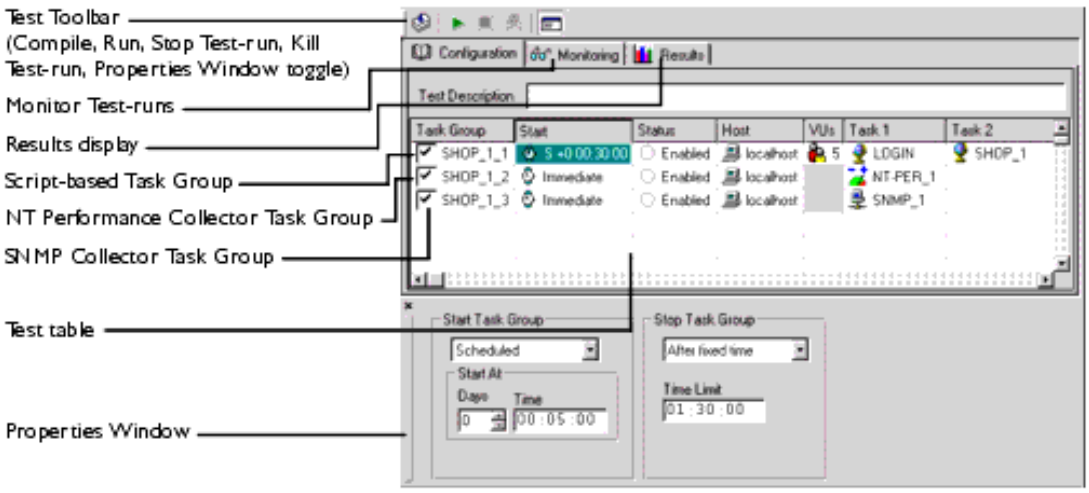
The Test Pane is displayed in the Main Window when you open a Test by double-clicking a new Test , or an existing Test , in the Repository Window.

The Test Pane comprises three sections represented by the following tabs:

- **Configuration**: This is the default view when you open a Test and the workspace used to develop a Test. Use it in combination with the Repository Window to select and add Scripts and Collectors. It displays the Test table which has categorized column headings that indicate where Script and Collector Tasks can be placed and the Task Group settings that apply to the contents of the Test.
  Select a Task Group cell to view and edit the associated settings using the Properties Window displayed below the Test table.
- **Monitoring**: Use this tab to monitor the progress of a Test-run. Select the display options you want from the Monitoring Window, including a Summary and data for individual Task Groups.
- **Results**: Use this tab to view the results collected during Test-runs in graph and table format. Use the Results Window to select the display options available which are dependent on the type of Test you are running.

Test Pane Features

The Configuration Tab view of the Test Pane is displayed below:



The Test Development Process

The Test development process typically includes the following procedures:

- Create a Test
- Add a Script to a Test
- Add Collectors to a Test
- Define Task Group settings, these include:
- Edit the Task Group Schedule Settings
- Select the Host used to Run a Task Group
- Specify the Number of Virtual Users to run a Script-based Task Group
- Edit the Number of Script Iterations and the Delay Between Iterations
- Save and Close a Test

Create a Test

1. In Commander select **File > New Test > Tests**.

   Or: In the Repository Window, right-click **Tests**, and select **New Test > Tests**.

   The Test appears in the Repository Window with a small crossed red circle over the Test icon , indicating that the file has no content. As soon as you open the Test and add a Script or a Collector, the icon changes to reflect this and appears .
2. In the Repository Window give the Test a name, in this example **PRESIDENT_SEARCH**, then press **Return**.

   Note: The new Test is saved automatically in the Repository when you switch to a different function or exit from Commander.

Add a Script to a Test

1. In the Repository Window, locate your new Test and double-click **PRESIDENT_SEARCH**, to open it with the **Configuration** Tab of the Test Pane displayed.

   The Configuration Tab displays the Test table where you can add Tasks, and the Properties Window which is used to apply Task Group settings.
2. Double-click **Scripts**, in the Repository Window to open the folder.
3. In the Repository Window, click on the **FINDBYNAME** Script then drag it across to the Test table and drop it in a new row under the **Task 1** column.

   The selected Script, **FINDBYNAME**, appears in the first empty row under the first Task column in a new Task Group. Additional Scripts can be added in sequence within the same row.

- The Task Group name is taken from the Test name and includes a number suffix which is automatically incremented for each new Task Group added to the Test.

  Use the Task Group cell to disable and enable a Task Group.

  Note: Uniquely naming Task Groups enables you to identify and select them for monitoring during a Test-run.
- The Start column indicates the Task Group Schedule settings. For more information on Task

Group scheduling, see Edit the Task Group Schedule Settings.

- The Status column displays Task Group activity and status information.
- The Host column defaults to **localhost**, which refers to the computer you are currently working on.
  The Host you select here determines which computer or device will run the Task Group during a Test-run. For more information on selecting a Host, see Select the Host used to Run a Task Group.
- The VUs column displays the number of Virtual Users assigned to run a Task Group. The default is a single Virtual User **1**.
  The number of Virtual Users running the Task Group can be changed by selecting the VUs cell and using the Properties Window to enter a new value. For more information, see Specify the Number of Virtual Users to run a Script-based Task Group.

  With the Script Task you have just added selected, use the Properties Window at the bottom of the Configuration Tab to specify the Task settings. For more information, see Edit the Number of Script Iterations and the Delay Between Iterations.

  Note: If your Task Group incorporates more than one Script, select the next Script from the Repository Window, then drag and drop it into the same Task Group row under the next **Task** column cell. Repeat this process until your Script sequence is complete.

  You can add additional Scripts to a Test in a new Task Group by dragging and dropping them into the next empty row.

  Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

Add Collectors to a Test

1. In the Repository Window, locate your new Test and double-click **PRESIDENT_SEARCH**, to open it with the **Configuration** Tab of the Test Pane displayed.

   The Configuration Tab displays the Test table where you can add Test Tasks and the Properties Window used to apply Task Group settings.

2. Double-click **Collectors**, in the Repository Window to open the folder and display the contents.
3. In the Repository Window, click on the **NT_PERFORMANCE** Collector then drag it across to the Test Pane and drop it in a new row under the **Task 1** column.

   The selected Collector **NT_PERFORMANCE**, appears in the next empty row under the first Task column in a new Task Group.

   Note: Collector-based Task Groups can only contain a single Task.

- The Task Group name is taken from the Test name and includes a number suffix which is automatically incremented as new Task Groups are added to the Test.

  Use the Task Group cell to disable and enable a Task Group.

  Note: Uniquely naming Task Groups enables you to select and monitor them during a Test-run from the Monitoring Tab.
- The Start column indicates the Task Group Schedule settings. For more information on Task Group scheduling, see Edit the Number of Script Iterations and the Delay Between Iterations.
- The Status column displays Task Group activity and status information.
- The Host column defaults to **localhost**, which refers to the computer you are currently

working on.

The Host you select here determines which computer or device will run the Task Group during a Test-run. For more information on selecting a Host, see Select the Host used to Run a Task Group.

4. Repeat step **3**, but this time select the **SNMP** Collector that you created earlier and add it to the Test in the next empty row.

   Note: Your changes are saved automatically in the Repository when you switch to a different function or exit from Commander.

Edit the Task Group Schedule Settings

1. Open a Test with the **Configuration** Tab of the Test Pane displayed.
2. Click on the **Start** cell in a Task Group.

   The current Schedule settings are displayed in the Properties Window at the bottom of the Configuration Tab. The default setting is for an **Immediate** start when the Test is run.

3. In the Start Task Group section of the Properties Window, click to the right of the selection box and choose a Start option:

- **Scheduled**: The Task Group starts after the number of days and at the time you set.
  Enter a time period using the Days and Time text boxes.
- **Immediately**: The Task Group starts when the Test is started.
- **Delayed**: The Task Group starts after the time period you set, (days: hours: minutes: seconds), relative to when the Test was started.
  Enter a time period using the Days and Time text boxes.

  Note: Your settings are displayed in the Test table.

4. In the Stop Task Group section of the Properties Window, click to the right of the selection box and choose a Stop option:

- **Manually**: The Task Group will run continuously until you click the **Stop** button in the Status column of the Task Group that activates during a Test run.
- **After fixed time**: The Task Group is stopped after a fixed period of time.
  Enter a time period using the Time Limit text box.
- **On Completion**: The Script-based Task Group is stopped after completing a number of iterations.
  Enter the number of Task Group iterations in the Iterations text box.

  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

  Note: During a Test-run Schedule settings cannot be edited, but they can be overridden manually using the **Start** and **Stop** buttons in the Status column of each Task Group.

Select the Host used to Run a Task Group

Note: Collector-based Task Groups include a Collector which defines a set of data to be recorded from one or more target Hosts during a Test-run. The Host you select in the Test table determines which computer or device will run the Task Group during a Test-run, not the Host from which data is collected.

1. Make sure the **PRESIDENT_SEARCH** Test is open with the **Configuration** Tab of the Test

Pane displayed.

2. Click on the Host cell in a Task Group.

   The current Host settings are displayed in the Properties Window at the bottom of the Configuration Tab. The default setting is localhost, which refers to the computer you are currently using.

3. In the Host Name text box of the Properties Window, enter the name of the Host to run the Task Group. Your settings are then displayed in the Test table.

   Note: The Host you select must have the OpenSTA Name Server installed and running with the Repository Host setting pointing to the local Host.

   Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

Specify the Number of Virtual Users to run a Script-based Task Group

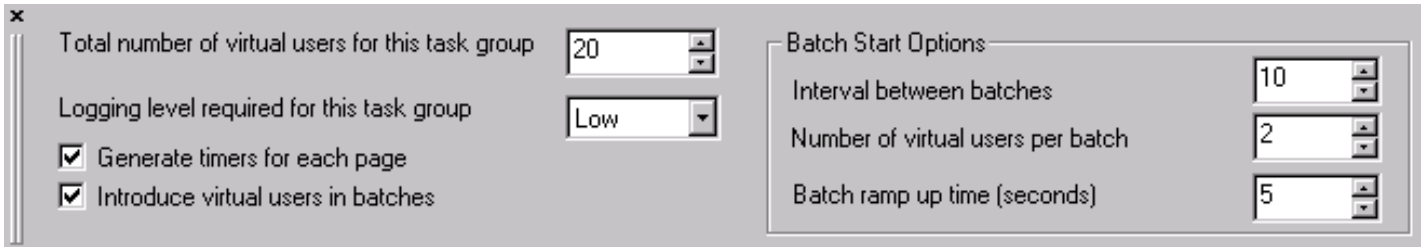
You can accept the default settings for your first Test-run then experiment with the settings to increase the load and compare Test-run results.

1. Make sure the **PRESIDENT_SEARCH** Test is open with the **Configuration** Tab of the Test Pane displayed.
2. Click on the VUs cell of the Task Group whose Virtual User settings you want to edit. The current Virtual User settings are displayed in the Properties Window at the bottom of the Configuration Tab. Use it to help control the load generated during a Test-run by specifying the number of Virtual Users and when they start.
3. In the Properties Window enter a value in the first text box to specify the total number of Virtual Users for the Task Group, or use to set a value.
4. Select the Logging level required for the Task Group to control the level of performance statistics and Timers gathered from Virtual Users. Click , and select either:
   Low: Information collected from the first 10 Virtual Users in the Task Group.
   High: Information collected from all the Virtual Users in the Task Group.
   None: No performance statistics or Timers are gathered.
5. Click the **Generate Timers For Each Page** check box, to record results data for the time taken to load each Web page specified in the Scripts, for every Virtual User running the Scripts. Timer information is recorded for the duration of the complete Script if the box is checked or unchecked.
6. Click on the **Introduce Virtual Users in batches** check box if you want to ramp up the load you generate by controlling when the Virtual Users you have assigned run. This is achieved by starting groups of Virtual Users in user defined batches.
7. Use the Batch Start Options section to control your Virtual user batch settings.

- **Interval between batches**, specifies the period of time in seconds between each ramp up period. No new Virtual Users start during this time.
- **Number of Virtual Users per batch**, specifies how many Virtual Users start during the batch ramp up time.
- **Batch ramp up time (seconds)**, specifies the period during which the Virtual Users you have assigned to a batch start the Task Group. The start point for each Virtual User is evenly staggered across this period.

  The example below depicts the Properties Window, where 20 Virtual Users are assigned to a Script-based Task Group.

  When the Task Group is run 2 Virtual Users (the number of Virtual Users per batch) will start over a period of 5 seconds (batch ramp up time) with a 10 second delay between each batch running.



Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

Edit the Number of Script Iterations and the Delay Between Iterations

You can accept the default settings for your first Test-run then experiment with the settings to increase the load and compare Test-run results.

1. Make sure the **PRESIDENT_SEARCH** Test is open with the **Configuration** Tab of the Test Pane displayed.
2. Click on the **FINDBYNAME** Script Task in the Test table, to display the current Task settings in the Properties Window at the bottom of the Configuration Tab.
3. With a Script Task selected, use the Properties Window to specify how long the Task runs. Click on the Task Termination box and select an option, either:

- **On Completion**: set a value to control the number of times (iterations) a Virtual User will run the Script during a Test-run.
- **After Fixed Time**, specify a time period to control when the task completes.

  Enter a value in the text box below or use .

4. You can specify a Fixed or Variable delay between each iteration of a Script Task.
   In the Properties Window, click on the Delay Between Each Iteration box and select an option, either:

- **Fixed Delay**: set a time value in seconds using the Delay text box.

  Or, you can choose to introduce a variable delay between Scripts:

- **Variable Delay**: set a value range in seconds using the Minimum and Maximum text boxes to control the upper and lower limits of variable iteration delay.

  Note: Your changes are saved automatically in the Repository when you switch to a different function in or exit from Commander.

Save and Close a Test

- The Test related work you perform is automatically saved in the Repository and the Test is closed when you switch to a different function or exit Commander.

Next...

After you have created a Test, by adding a Scripts and Collector, and applied the Task Group settings required, you are ready to run it against the demonstration Web site.
Move on to the next section for details on how to do this.

Next Section: **Running a Test**

Back to **Contents**

Creating Collectors

A Collector is a set of user-defined data collection queries which determine the type of performance data that is monitored and recorded from one or more Host computers or devices during a Test-run. Include them in your Tests to target specific components of the WAE under test and the Hosts used to run a Test, with precise data collection queries to record the performance data you need. Create Collectors and incorporate them into your Tests, then run the Test to generate the results data required.

OpenSTA also supports the creation of Collector-only production monitoring Tests. These Tests are used to monitor and collect performance data within a production system where the load is generated externally by the normal use of the system.

Collectors give you the flexibility to collect and monitor a wide range of performance data at user defined intervals during a Test-run. A Collector can contain a single data collection query and be used to target a single Host. Or alternatively, they can contain multiple queries and target multiple Hosts. The specific data collection queries defined within a Collector can be selected and monitored from the Monitoring Tab view of the Test Pane during a Test-run.

OpenSTA supplies two Modules which facilitate the creation of Collectors:

- NT Performance Module
- SNMP Module

In this example the following procedures take you through the creation of one NT Performance Collector and one SNMP Collector.

NT Performance Collectors

NT Performance Collectors are used to monitor and collect performance data from your computer or other networked Hosts running Windows NT or Windows 2000 during a Test-run. Creating and running NT Performance Collectors as part of a Test enables you to record comprehensive data to help you assess the performance of systems under test.

Use NT Performance Collectors to collect performance data during a Test-run from performance objects such as Processor, Memory, Cache, Thread and Process on the Hosts you specify in the data collection queries. Each performance object has an associated set of performance counters that provide information about device usage, queue lengths, delays, and information used to measure throughput and internal congestion.

NT Performance Collectors can be used to monitor Host performance according to the data collection queries defined in the Collector during a Test-run. Performance counters can be displayed graphically by selecting the Task Group that contains the Collector from the Monitoring Window in the Monitoring Tab of the Test Pane.

The results recorded using a Collector can be monitored then viewed after the Test-run is complete. Select a Test and open up the Custom NT Performance graph from the Results Tab of the Test Pane to display your results.

Note: If you are using an NT Performance Collector to target a Web server that is running Microsoft IIS (Internet Information Server), you can monitor and collect performance from it by selecting the Web Service object from the Performance Object text box when you set up a new query.

In this example the procedure below takes you through adding two data collection queries targeting the same Host.

Create an NT Performance Collector

1. In Commander, select **File > New Collector > NT Performance**.

   Or: In the Repository Window, right-click **Collectors**, and select **New Collector > NT Performance**.

   The Collector appears in the Repository Window with a small crossed red circle over the Collector icon , indicating that the Collector has no content.

   Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon changes to reflect this .

2. Give the new Collector a name within the Repository Window, in this example **NT_PERFORMANCE**, then press **Return**.
3. In the Repository Window, double-click the new Collector

NT_PERFORMANCE, to open the Collector Pane in the Commander Main Window, where you can setup your data collection queries.

The Edit Query dialog box opens automatically when you open a new Collector  , or double-click on a row of an open Collector. Use this dialog box to add NT Performance data collection queries.

4. In the Name text box enter a unique title for the data collection query, in this case **Processor**.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring Tab of the Test Pane. You can select and monitor queries during a Test-run. Query names also appear in the Custom SNMP graph with the associated results data. Use the Results Window in the Results Tab of the Test Pane to display them.

5. Click the **Browse Queries** button to open the Browse Performance Counters dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit Query dialog box.

6. In the Browse Performance Counters dialog box, select the Host you want to collect data from. You can select to either:

- **Use local computer counters**: Collects data from the computer you are currently using.
- Or, **Select counters from computer**: Enables you to specify a networked computer. Type **** then the name of the computer, or click  and select a computer from the list.

7. In the Performance object selection box select a performance object, in this example **Processor**. Click  , to the right of the selection box and choose an entry from the drop down list.
8. In the Performance counters selection box choose a performance counter, in this example **% Processor Time**.

   Note: Click **Explain** to view a description of the currently selected Performance counter.

9. In the Instances selection box pick an instance of the selected performance counter.
10. Click **OK** to confirm your choices and return to the Edit Query dialog box.
11. In the Interval text box enter a time period in seconds, for example **5**, to control the frequency of data collection, or use  , to set a value.
12. Leave the Delta Value column check box unchecked to record the raw

data value, or check the box to record the Delta value.

Note: Delta value records the difference between the data collected at each interval.

13. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

14. Use  , in the toolbar to add an additional query then repeat steps 4-13. This time select the **Memory** Performance object and **Page Faults/sec** Performance counter.

    Tip: Double-click on a query to edit it. Select a query then click  , in the toolbar to delete it.

    Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

SNMP Collectors

SNMP Collectors (Simple Network Management Protocol) are used to collect SNMP data from Host computers or other devices running an SNMP agent or proxy SNMP agent during a Test-run. Creating then running SNMP Collectors as part of a Test enables you to collect results data to help you assess the performance of production systems under test.

SNMP is the Internet standard protocol developed to manage nodes on an IP network. It can be used to manage and monitor all sorts of equipment including computers, routers, wiring hubs and printers. That is, any device capable of running an SNMP management process, known as an SNMP agent. All computers and many peripheral devices meet this requirement, which means you can create and include SNMP Collectors in a Test to collect data from most components used in target WAEs.

SNMP data collection queries defined in a Collector can be displayed graphically during a Test-run to monitor the performance of the target Host. Select a Task Group that contains an SNMP Collector from the Monitoring Window in the Monitoring Tab of the Test Pane then choose the performance counters you want to display.

In this example the procedure below takes you through adding two data collection queries targeting the same Host.

Create an SNMP Collector

1. In Commander, select **File > New Collector > SNMP**.

Or: In the Repository Window, right-click **Collectors**, and select **New Collector > SNMP**.

The Collector appears in the Repository Window with a small crossed red circle over the icon , indicating that the Collector has no content.

Note: After you have opened a Collector and defined a data collection query using the Edit Query dialog box in the Collector Pane, the icon changes to reflect this .

2. Give the new Collector a name within the Repository Window, in this example **SNMP**, then press **Return**.
3. In the Repository Window, double-click the new Collector **SNMP**, to open the Collector Pane in the Commander Main Window, where you can setup your data collection queries.

   The Edit Query dialog box opens automatically when you open the new Collector , or double-click on a row of an open Collector. Use this dialog box to add SNMP data collection queries.
4. In the Name text box enter a unique title for the data collection query, in this example **IP In**.

   Note: When you run a Test the query name you enter is listed in the Available Views text box which is displayed in the Monitoring Tab of the Test Pane. You can select query names to monitor the progress of the Test-run.

   Query names also appear in the Custom SNMP graph with the associated results data. Use the Results Window in the Results Tab of the Test Pane to display them.
5. In the SNMP Server text box enter the Host name or the IP address you want to collect data from.

   Tip: You can run the SNMP Server Scan by clicking in the toolbar, to identify all networked SNMP Servers currently running an SNMP agent, then click , to the right of the SNMP Server text box to display the list and select an SNMP server.
6. In the Port text box enter the port number used by the target SNMP Server.

   Note: Port 161 is the default port number that an SNMP agent runs from.
7. Click the **Browse Queries** button to open the Select Query dialog box and define the query.

   Tip: You can enter a query directly into the Query text box in the Edit

Query dialog box.

8. In the Select Query dialog box, click  to the right of the Category selection box and choose a category from the drop down list, in this example **ip**.
9. In the Query selection box below, pick a query associated with the category you have chosen, in this example **ipInReceives.0**.

   Note: The Current Value of the query must contain a numeric counter in order to generate data to populate the results graphs.

10. Click **Select** to confirm your choices and return to the Edit Query dialog box.

    The selected query, **public ip.ipInReceives.0**, records the total number of input datagrams received from interfaces, including those received in error.

11. In the Edit Query dialog box use the Interval text box to enter a time period in seconds, for example **5**, to control the frequency of data collection, or use  , to set a value.
12. Leave the Delta Value column check box unchecked to record the raw data value, or check the box to record the Delta value.

    Note: Delta value records the difference between the data collected at each interval.

13. Click **OK** to display the data collection query you have defined in the Collector Pane.

    Each row within the Collector Pane defines a single data collection query.

14. Use  , in the toolbar to add an additional query then repeat steps 4-13. This time select the **ip** category and the **ipOutRequests.0** query.

    This query appears as **public ipOutRequests.0**. It records the total number of IP (Internet Protocol) datagrams, which local IP user and protocols (including ICMP) supplied to IP in requests for transmission. This counter does not include any datagrams counted in ipForwDatagrams.

    Tip: Double-click on a query to edit it. Select a query then click  , in the toolbar to delete it.
    Note: The Collector is saved automatically in the Repository when you switch to a different function or exit from Commander.

It is also possible to create new SNMP data collection categories which can then be selected during the Collector creation process. Follow the procedure below for details.

Create New SNMP Data Collection Categories

Use this option to create new SNMP data collection categories which you can select when you define a new query in the Select Query dialog box.

1. In the Repository Window double-click on an SNMP Collector to open the Collector Pane in the Main Window.
2. Click  , in the toolbar.
3. In the Category Definition dialog box, click in the Name text box and enter the title of the new data collection category.

   Note: The new category can be chosen from the Category text box of the Select Query dialog box when you are defining a query.
4. In the Walk Point text box enter the query definition.

   Note: The Walk Point you define can be selected in the Query text box of the Edit Query dialog box and the Category text box of the Select Query dialog box when you are choosing a query.
5. Click **Apply** to make the new category available for selection. Click **Close** to cancel.

   Note: Edit the Walk Point of a category by clicking  , to the right of the Name text box to display and select a category, then enter the new query definition.

Next...

After you have created your Collectors the next step is to add them to a Test. Move on to the next section for details on how to create a new Test.

Next Section: **Creating a Test**

Back to **Contents**

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Documentation feedback

Modeling a Script

After creating a Script you can model it by using variables to more accurately simulate the behavior of real users when the Test that references the Script is run.

The following example documents the procedures involved in modeling a user name and password to enable the simulation of multiple Virtual Users when a Test is run.

Modeling a Script involves:

- Open a Script from Commander
- Create and Apply Variables

Note: Use the *SCL Reference Guide* if you need help using Script Control Language. There is an on-line copy shipped with the Script Modeler Module, or you can download or view it from OpenSTA.org.

Open a Script from Commander

1. In the Repository Window within Commander, double-click **Scripts**, to expand the directory structure.
2. Double-click on the Script **FINDBYNAME**.

   Script Modeler is launched, opening the Script you have previously recorded.

Create and Apply Variables

1. With your FINDBYNAME Script open, click **Variable > Create**.

Shortcut: Click  in the Variable Toolbar.

2. In the Variable Creation dialog box, enter a name for your new variable. In this example the name is **USERNAME**.

   Note: The name you give must be an OpenSTA Dataname.

3. Select the Scope of your variable.
   In this example the selection is **Script**.

   Note: The scope of a variable relates to which Virtual Users and Scripts can make use of the variables you create.

4. Select the Value Source of your variable.
   In this example the selection is **Value List**.
5. Select the order in which the variable values are selected when a Test is run.
   In this example the selection is **Sequential**.
6. Select the data types of the variable.
   In this example the selection is **Character**.
7. Click **Next** when you have made your selections.
8. In the Value List dialog box you need to enter the variable values, or names that will represent the Virtual Users you need when the Test is run. In this example there are five values or user names entered manually within the Value List dialog box, in this example they are:

   phillip, **allan**, **david**, **robert** and **donna**.

   Click **Add Value** and enter the first name.

   Repeat this process until you have entered all the values.

9. Click **Finish** when the setup process is complete.
10. Repeat this process to create the **PASSWORD** variable, which your five Virtual Users will need in order to access the *Which US President?* Web site.
    Note: This Web site requires a password to be the reverse spelling of the login name.

    The variables you have created are represented as text strings within the Definitions section of the Script, as illustrated below:

```
CHARACTER*512 USERNAME ( "phillip", "allan", "david" &
, "robert", "donna" ), SCRIPT
CHARACTER*512    PASSWORD ( "pillihp", "nalla", "divad" &
, "trebor", "annod" ), SCRIPT
```

The variables are now ready to be substituted for the original login identity

recorded in the Script.
Obviously something happened here ... there are no instructions on how to then make use of these vars in your script :-(
Please Check this documentation for help on using the variables

11. Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar, to compile your Script.

    Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-model to resolve the problem.

12. It is a good idea to replay the Script to check the activity you have recorded before you incorporate it into a Test.

    Select **Capture > Replay** or click  , in the Capture/Replay Toolbar. The replay activity is displayed in the Output Pane.

13. Click  , to save your Script, or click **File > Save**.

Next...

The Script is now successfully modeled and ready to be incorporated into a new Test. Go on to the next section for information on creating Collectors, which are used to monitor and collect performance data during a Test-run.

Next Section: **Creating Collectors**

Back to **Contents**

Recording a Script

Scripts form the content of an HTTP/S performance Test using OpenSTA. After you have planned a Test the next step is to develop its content by creating the Scripts you need.

Launch Script Modeler from Commander to create and model Scripts. Then incorporate the Scripts into your performance Tests.

Script creation involves:

- Create a New Script Using Commander
- Familiarizing yourself with the Script Modeler
- Planning your Script
- Configuring Script Modeler for Script Creation
- Recording a Web Session

Create a New Script Using Commander

1. In Commander select **File > New Script > HTTP**.

   Or: In the Repository Window, right-click **Scripts**, and select **New Script > HTTP**.

   The Script appears in the Repository Window with a small crossed red circle over the Script icon , indicating that the file has no content. As soon as you open the Script and record a Web session, the icon changes to reflect this and appears .

2. Name the new Script **FINDBYNAME**, from the Repository Window, then press **Return**.

Note: Script names must be OpenSTA Datanames.

3. After you have created an empty Script it appears in the Repository Window, Scripts folder.

   Double-click the new Script **FINDBYNAME** , to launch Script Modeler.

Script Modeler

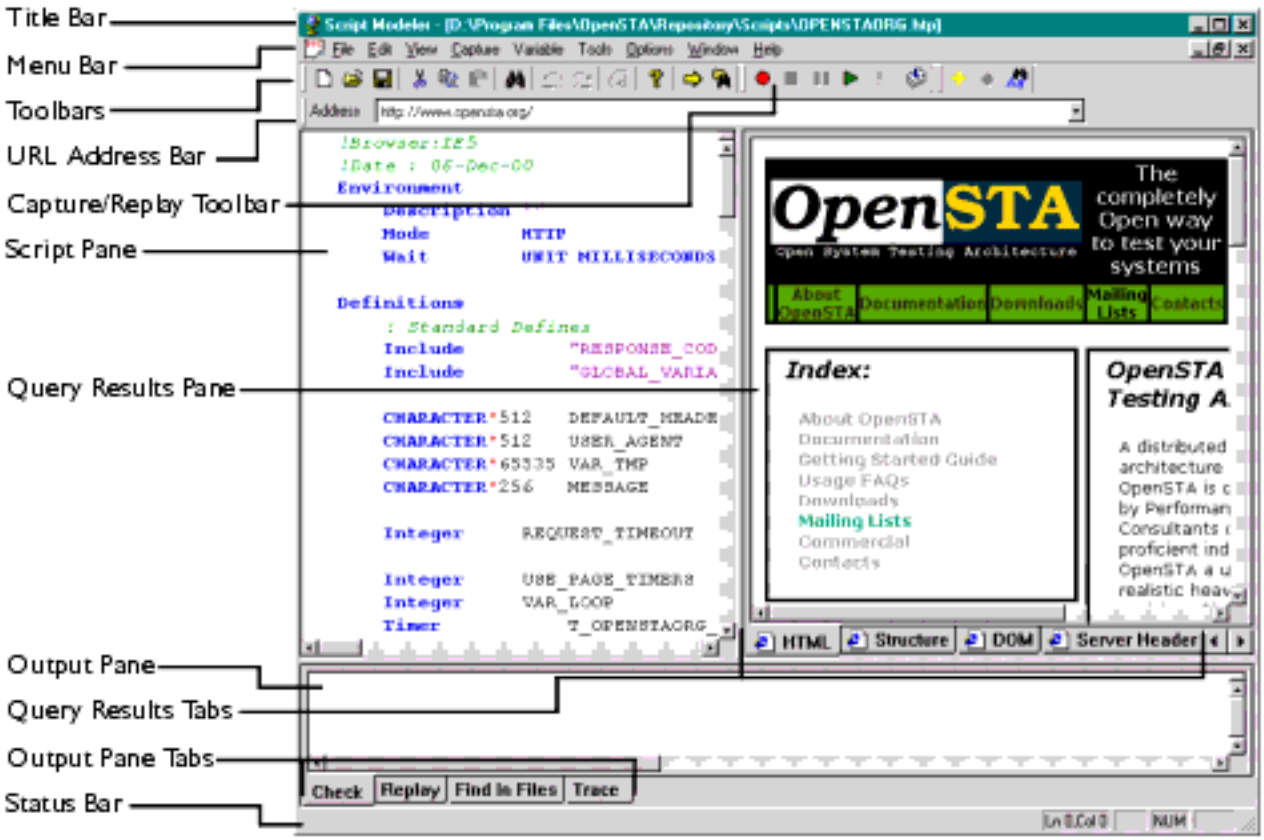
Script Modeler supplies versatile Script creation and modeling functionality. Use the menu bar and right-click menu options to create and model Scripts.

After you create a Script or when you open one, it is displayed in the Script Pane on the left-hand side of the main window. It is represented using SCL code which enables you to model it using the menu options or directly by keying in the SCL commands you need.

The Query Results Pane is used to display Web site responses. HTML information is recorded during the same Web session as the corresponding Script and is directly related to it, which enables additional modeling options.

The Output Pane. Displays the results of Script compilation. Scripts need to be compiled after modeling to check the validity.

Script Modeler Interface Features



Planning your Script

Before recording a Script make sure you are familiar with the Web site you are going to test. Ensure that it is functioning correctly and that you understand its basic behavior.

Below is a summary of the demonstration Web site, *Which US President?*.

The Demo Web Site

Which US President? is a simple WAE that provides a variety of information concerning past presidents of the United States. The presidents are selected through a filtering search.

The application has five sections of interest:

- Login Page: POST Form with text and password fields and a Submit button.

  The Login page requires a user ID and password. To keep the CGI simple yet have a realistic login, there is no pre-defined user database: passwords are authenticated by being the reverse of the user ID. For example, the password for the user ID "guest" is "tseug". The login session is maintained by the use of a site cookie.

- Search Form: POST Form with a text field, a set of radio buttons, a select menu and a Submit button.
- Predefined search links: Cause GET requests to access the search form, but via an alternative URL encoded mechanism.
- Search output: titles and a table containing the formatted results of the query.
- Log out link: Cause GET request with URL encoded values that logs the application out.

  The application implements a simple session time-out mechanism. This is done by encoding the last session access time into the Web site cookie, which is updated on each access.

Configuring Script Modeler for Script Creation

The next step is to set up the correct Script recording environment. This includes selecting the type of browser you want to use to record Scripts and ensuring that your browsers proxy server settings are correctly configured.

Script Modeler's recording mechanism makes use of a proxy server, known as the Gateway, which is used to intercept and record the HTTP/S traffic generated during a Web session. This recording mechanism involves temporarily adjusting

the proxy server settings of the browser you use for the duration of the recording.

There is currently a limitation in this recording mechanism. If your browser gets its proxy configurations through any of the available automated mechanisms, Script Modeler cannot adjust your proxy server settings and correctly configure the Gateway. For the recording mechanism to work, the browser must either have no proxy server configured or have a manually supplied proxy server and port configured. If you are unsure if this is the case, or do not know how to configure your browser in this manner, talk to your system administrator.

Before you can create a Script you need to select the browser you want to use for the recording.

Select Browser Type for Script Recording

1. Create a new Script using Commander.
2. Open the Script from the Repository Window.
3. In Script Modeler, select **Options > Browser**.
4. In the Select Browser dialog box, click  , and select the browser you want to use from the list, either **Internet Explorer 4**, **Internet Explorer 5** or **Netscape**.

   Note: The **Netscape** option refers to Netscape Navigator version 4.7
5. Click **OK** to save your settings.

   When you start recording using Script Modeler, the selected browser is launched automatically.

Recording a Web Session

After you have selected a browser and configured the Gateway, you are ready to record a Web session and create a Script.

When you begin recording a Web session use the browser as normal and move through the Web site performing the steps you want to record. In this example, the first step is to go to the login page of the demonstration Web site *Which US President?*, by typing in the URL of your locally installed copy.

Recording a Script

1. After you have created an empty Script it appears in the Repository Window, Scripts folder.

   Double-click the new Script **FINDBYNAME**  , to launch Script Modeler.

2. Click the Record button  , in the Capture/Replay Toolbar, or select **Capture > Record**, to begin the HTTP/S capture process.

   This action launches the Gateway and the Web browser you have selected.

   Your browser's Home page Internet option is overridden by Script Modeler when you start recording. The setting is replaced with **about: blank**, which specifies that your home page will be a blank HTML page. This ensures that your normal Home page is not launched and recorded in the Script.

3. Type in a URL of the demonstration Web site *Which US President?*, and hit **Return**.

   The browser displays the Log-in page of the Web site which requires you to enter a log-in name and password. The Password is the reverse of the Log-in so you can use the same single character in both text boxes.

4. Type **A** in the Log-in and Password text boxes.
5. Click the **login** button.

   The Web site displays a page which enables you to search through a simple database comprising a list of past US presidents and some of their personal details including political party and religion.

6. In the Presidents Name text box, type in part of a president's name, for example **Truman**.
7. Click the **Submit Query** button.

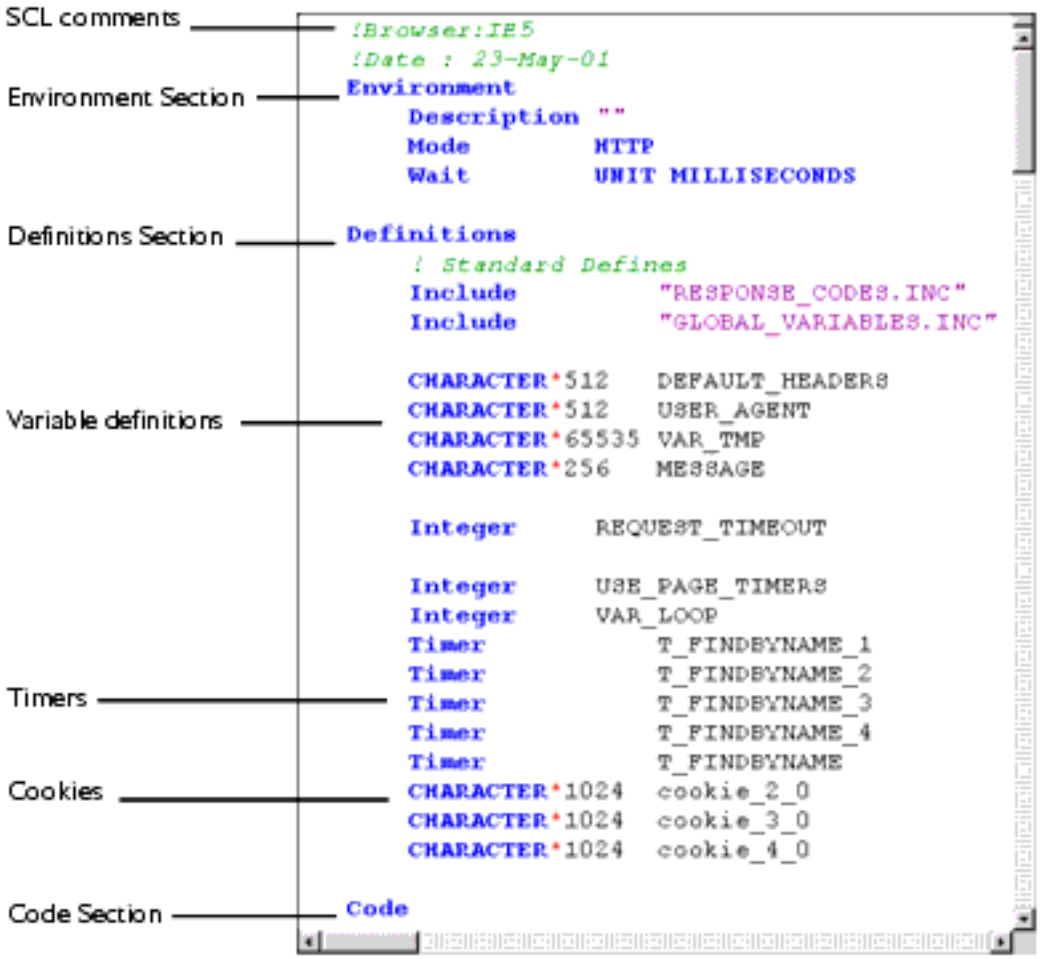
   The Web site displays a combined search and results page which displays a single entry for Harry S Truman.

8. After you have completed your search you need to Log-out.

   Click the **LOG OUT** link in the top right-hand side of the page.

9. After you have completed the browser session, either close the browser window to end the recording, or switch back to Script Modeler and click the Stop button  , in the Capture/Replay Toolbar.

   When you have finished recording the Script the SCL formatted data is displayed in the Script Pane as illustrated below:



The Script represents the data it contains using syntax coloring to help identify the different elements. For example, SCL keywords and commands are represented in blue. A Script is divided into three sections represented by the following SCL keywords; **Environment**, **Definitions** and **Code**.

The Environment Section

The Environment section is always the first part of a Script. It is introduced by the mandatory Environment keyword. It is preceded by comments written by the Gateway which note the browser used and the creation date. This section is used to define the global attributes of the Script including a Description, if you choose to add one, the Mode and Wait units.

The Definitions Section

The Definitions section follows the Environment section and is introduced by the mandatory **Definitions** keyword. It contains all the definitions used in the Script, including definitions of variables and constants, as well as declarations of timers and file definitions.

It also contains the **global_variables.INC** file which is used to hold variable definitions of global and Script scope which are shared by Virtual Users during a Test-run, and the **Response_Codes.INC**, an include file which contains the definitions of constants which correspond to HTTP/S

response codes.

The Code Section
The Code section follows the Definitions section and is introduced by the mandatory **Code** keyword. It contains commands that represent the Web-activity you have recorded and define the Script's behavior. The Code section is composed of SCL commands that control the behavior of the Script.

10. Before you save your new Script you need to compile it using the Syntax Check option to ensure the validity of the recording.

    Select **Capture > Syntax Check** or click  , in the Capture/Replay Toolbar. Compilation results are reported in the Output Pane. If compilation is unsuccessful, you may need to re-record the Script or model the contents to resolve the problem.

    Note: You can record over the top of an existing Script until you achieve the content you need.

11. After compilation replay the Script to check the activity you have recorded.

    Select **Capture > Replay** or click  , in the Capture/Replay Toolbar

12. When you have finished recording, click  , in the Standard Toolbar to save your Script in the Repository, or click **File > Save**.
13. Select **File > Close** to close the current Script or **File > Exit** to exit Script Modeler.

    Note: If you have unsaved Scripts open in Script Modeler, you are automatically prompted to save them before the program closes. Closing down Script Modeler also closes the browser which restores your original browser settings.

Query Results Pane

After you have finished recording you can also view the HTML information recorded at the same time as the Script. You can access this data by opening a Script in Script Modeler then selecting a URL you want to view from the URL Address Bar and clicking the URL Details button  , in the Standard Toolbar. HTML information is organized into five categories:

- **HTML**: Shows a browser view of the HTML document that has been retrieved.
- **Structure**: Shows the basic elements of the page in a collapsing tree view.
- **DOM**: Shows the page structure in the Document Object Model, as a collapsing tree view.

- **Server Header**: Shows the HTTP response headers that the Web server returned to the browser.
- **Client Header**: Shows the HTTP request headers provided by the browser for the Web server.

Before you exit from Script Modeler make sure you have saved your Script then select **File > Exit** to exit Script Modeler.

Next...

Now you have created a Script you are ready to model it.

Modeling a Script is not an essential procedure but it can be useful to modify Scripts using variables for example, to better simulate the activity of real Web users when a Test is run.

Next Section: **Modeling a Script**

Back to **Contents**

TOC PREV NEXT INDEX

OpenSTA.org
Mailing Lists
Documentation feedback