



The Mainframe Software Partner  
For The Next 50 Years

---

# Hiperstation Scripting Reference

**Hiperstation for VTAM**  
**Hiperstation for Mainframe Servers**  
**Hiperstation for WebSphere MQ**

**Release 16.05**

Please direct questions about Hiperstation  
or comments on this document to:

**Compuware Customer Support**

**<https://go.compuware.com/>**

This document and the product referenced in it are subject to the following legends:

Copyright 1994-2016 Compuware Corporation. All rights reserved. Unpublished rights reserved under the Copyright Laws of the United States.

U.S. GOVERNMENT RIGHTS-Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in Compuware Corporation license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (OCT 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable. Compuware Corporation.

This product contains confidential information and trade secrets of Compuware Corporation. Use, disclosure, or reproduction is prohibited without the prior express written permission of Compuware Corporation. Access is limited to authorized users. Use of this product is subject to the terms and conditions of the user's License Agreement with Compuware Corporation.

Hiperstation, Hiperstation, Hiperstation/*Xchange*, Hiperstation/Code Coverage, Abend-AID for CICS, FrontLine, and Compuware are trademarks or registered trademarks of Compuware Corporation.

CICSplex, CICS Transaction Server, DB2, DFSMS, IBM, MQ Series, MVS, MVS/ESA, OS/390, VTAM, WebSphere MQ, and z/OS are trademarks or registered trademarks of International Business Machines Corporation.

Adobe® Reader® is a trademark of Adobe Systems Incorporated in the United States and/or other countries.

All other company and product names are trademarks or registered trademarks of their respective owners.

# Contents

<b>Figures</b> .....	<b>vii</b>
<b>Introduction</b> .....	<b>ix</b>
Accessing Hiperstation Documentation.....	ix
HTML Files.....	x
Using this Manual .....	x
Manual Contents.....	x
Notation Conventions .....	x
Command Syntax .....	xi
Reading Syntax Diagrams .....	xi
Accessibility .....	xii
Installing Windows Accessibility Features .....	xiii
Selecting Font and Font Size .....	xiii
Changing Color and Contrast.....	xiii
Setting Cursor Blink Rate .....	xiv
Using Keyboard Shortcuts.....	xiv
Accessibility Exceptions Work Arounds .....	xiv
Known Exceptions.....	xiv
Solutions .....	xv
Getting Help.....	xv
Compuware FrontLine Customer Support Website .....	xv
Contacting Customer Support .....	xv
Phone.....	xv
Web .....	xv
Mail .....	xv
Corporate Website.....	xvi
<b>Chapter 1. Hiperstation Script Language</b> .....	<b>1-1</b>
Statement of Compliance.....	1-1
Installation Requirements .....	1-1
Capabilities of the Language .....	1-1
Hiperstation Additions to REXX.....	1-2
Statements.....	1-2
Iterative Statements and Multiple Transactions .....	1-3
Conditional Statements.....	1-4
Language Constructs .....	1-4
Variables .....	1-4
Constants.....	1-4
Operators.....	1-5
Function Calls.....	1-5
Expressions .....	1-5
Assignments.....	1-6
Retrieving Data from External Sources.....	1-6
Retrieving Screen Data .....	1-6
Comments.....	1-6
Examples:.....	1-6
Using Language Statements .....	1-7
Terminal Input Substitution .....	1-7
Example .....	1-7
Modular Scripts .....	1-7
Examples .....	1-8

Script Reports . . . . .	1-8
Custom Return Codes . . . . .	1-8
Hiperstation REXX Variables . . . . .	1-9
3270 REXX Variables . . . . .	1-9
APPC REXX Variables. . . . .	1-13
APPC Error Determination REXX Variables. . . . .	1-13
APPC Environment Data REXX Variables . . . . .	1-14
APPC Application Data REXX Variables . . . . .	1-14
APPC General REXX Variables . . . . .	1-17
TCP/IP REXX Variables . . . . .	1-17
MQ REXX Variables . . . . .	1-20
Using REXX Variables . . . . .	1-24
Use REXX Variables as Input . . . . .	1-24
Perform File I/O from a Script . . . . .	1-24
Read Data from VSAM-GETVSAMR. . . . .	1-26
Analyze Application Screens . . . . .	1-27
Use Logic to Control Script Flow. . . . .	1-27
Use the REXX Log . . . . .	1-27
Building Hiperstation Script Statements Using <EXECUTE> . . . . .	1-33
Altering Hiperstation Script Statements Using <ALTER> Tags . . . . .	1-34
Using the <ADDCALL> and <DELCALL> Tags . . . . .	1-35
Hiperstation HSCMDS Command Environment . . . . .	1-35
HSCMDS Commands. . . . .	1-35
ALLOC . . . . .	1-36
Free . . . . .	1-36
GETVSAMR. . . . .	1-36
HSCMDS ALLOC Return Codes. . . . .	1-37
Other Examples. . . . .	1-38
Modify APPC Script Data - Scenario 1 . . . . .	1-38
Modify APPC Script Data - Scenario 2 . . . . .	1-38
Modify an MQ Script to Test the Requester. . . . .	1-40
Initialize a Count Variable . . . . .	1-43
Read Data from an External File . . . . .	1-44
Loop Until an Unsuccessful GET or Last Record is Encountered . . . . .	1-44
Process the MQ_PUT Message . . . . .	1-46
<b>Chapter 2. Common Programming Interface . . . . .</b>	<b>2-1</b>
Overview . . . . .	2-1
Call Interface . . . . .	2-1
Session IDs . . . . .	2-2
Parameters . . . . .	2-2
DD Requirements . . . . .	2-2
Verbs . . . . .	2-2
Logon . . . . .	2-3
Logoff . . . . .	2-4
Execute . . . . .	2-5
Send . . . . .	2-5
Receive . . . . .	2-6
Type . . . . .	2-6
Retrieve . . . . .	2-7
Copy3270 . . . . .	2-8
REPL3270 . . . . .	2-9
Sample REXX EXEC . . . . .	2-9
<b>Chapter 3. 3270 Repository Filtering and Copy Utility . . . . .</b>	<b>3-1</b>
Creating EHS3COPY Control Statements . . . . .	3-1
CONTROL Statement. . . . .	3-1

COPY Statement .....	3-2
Writing Exit Programs .....	3-3
Tips for Writing Exit Programs .....	3-4
Preparing the JCL .....	3-6
JCL Examples .....	3-6
<b>Chapter 4. Guidelines for Successful Stress Testing .....</b>	<b>4-1</b>
Planning the Stress Test .....	4-1
Create a Test Plan .....	4-1
Sample Stress Test Plan .....	4-1
Set Up the Test .....	4-3
Execute the Test .....	4-4
Using the Stress Test Results .....	4-5
Determine Performance Levels .....	4-5
Reveal Unexpected Results .....	4-5
Make Improvements .....	4-5
<b>Index .....</b>	<b>I-1</b>



## Figures

1-1.	Subroutine for Extracting Extended Attributes . . . . .	1-13
1-2.	Sample Script Using APPC Application Data REXX Variables . . . . .	1-15
1-3.	Sample Script Containing the APPC_ACTUAL Variable . . . . .	1-15
1-4.	Sample Script Containing the APPC_EXPECTED Variable . . . . .	1-16
1-5.	Sample Script Containing the APPC_ACTUAL_LENGTH Variable . . . . .	1-16
1-6.	Sample Script Containing the ACCP_EXPECTED_LENGTH Variable . . . . .	1-17
1-7.	External File Contents . . . . .	1-25
1-8.	Script Contents . . . . .	1-26
1-9.	Examples of Analysis, Using Logic, and Using the REXX Log (1 of 6) . . . . .	1-28
1-10.	Examples of Analysis, Using Logic, and Using the REXX Log (2 of 6) . . . . .	1-29
1-11.	Examples of Analysis, Using Logic, and Using the REXX Log (3 of 6) . . . . .	1-30
1-12.	Examples of Analysis, Using Logic, and Using the REXX Log (4 of 6) . . . . .	1-31
1-13.	Examples of Analysis, Using Logic, and Using the REXX Log (5 of 6) . . . . .	1-32
1-14.	Examples of Analysis, Using Logic, and Using the REXX Log (6 of 6) . . . . .	1-33
1-15.	REXX Example to Display Error Return Code . . . . .	1-37
1-16.	Script for Modifying Script Data Using the <REPLACE> Tag . . . . .	1-38
1-17.	Alternative Method for Modifying APPC Script Data . . . . .	1-39
1-18.	MQ Script Application Flow . . . . .	1-40
1-19.	Page 1 of Script Containing GET A and PUT B as Recorded . . . . .	1-42
1-20.	Page 2 of Script Containing GET A and PUT B as Recorded . . . . .	1-43
2-1.	Sample REXX EXEC . . . . .	2-10





# Introduction

Compuware is committed to providing user-friendly documentation in a variety of electronic formats. This section describes the available formats and how to access them, provides an overview of this manual, describes the conventions used within and the resources available to help you.

---

## Accessing Hiperstation Documentation

The Hiperstation documentation is available on the Compuware Go (FrontLine) customer support website at <http://go.compuware.com>. *Release Notes* are provided in HTML format and manuals in Portable Document Format (PDF):

- ***Release Notes*** — Provides recent information for the Hiperstation product. In this file, you can quickly access system requirements, technical notes, customer support contact information, and a list of the new features available in the release. The *Release Notes* may be updated throughout the life cycle of a release with the most current version located on FrontLine for easy access to the latest product information.
- ***Hiperstation Installation and Configuration Guide*** — Provides installation and configuration procedures.
- ***Hiperstation for VTAM User Guide*** — Explains how to use Hiperstation for VTAM to test 3270 and LU0 applications.
- ***Hiperstation for WebSphere MQ User Guide***— Explains how to use Hiperstation for WebSphere MQ to test WebSphere MQ applications.
- ***Hiperstation for Mainframe Servers User Guide*** — Explains how to use Hiperstation for Mainframe Servers to test APPC and TCP/IP applications.
- ***Hiperstation Auditor User Guide*** — Explains how to use the Hiperstation Archive Function.
- ***Hiperstation Automated Testing Vehicle (ATV) Manager User Guide*** — Explains how to use the Hiperstation ATV Manager to manage your testing environment and test cases.
- ***Hiperstation Messages and Codes*** — Explains the messages and codes that Hiperstation produces.
- ***Hiperstation Scripting Reference*** — Introduces advanced script editing concepts and provides reference information for technical users.
- ***Hiperstation Reference Summary*** — Summarizes the commands used in Hiperstation for VTAM's Domain Traveler and Session Demo features.
- ***Master Index*** — This file contains an indexed list of the contents of the entire manual set. To use this file, all of the book files and the master index file must be located in the same directory. Open the master index file and search for the desired term. Clicking on a search result will open the appropriate book at the desired page.

View and print PDF files with Adobe Reader. Download a free copy of the latest version of the reader from Adobe's web site: <http://www.adobe.com>.

**Note:** With a few minor exceptions, PDF files comply with the requirements of section 508 of the Rehabilitation Act of 1973. Refer to the Accessibility preface in any of the user guides for information.

For your convenience, Compuware also provides the Hiperstation manuals in the following formats:

- Hypertext Markup Language (HTML)

Access these formats on FrontLine, Compuware's Customer Support Web site at <http://frontline.compuware.com>.

1. Log-in.
2. Select the desired product.
3. Click the Documentation link on the left selection bar.
4. Select the desired release. FrontLine presents a documentation index containing links to each of the product's manuals in all of the available formats.

## HTML Files

View HTML files with any standard Web browser. Simply click the HTML link on the selected FrontLine documentation page.

**Note:** As you review the HTML content, you may encounter the known issue regarding screens and other graphic figures in which the image may be cropped along the left or bottom edge.

---

## Using this Manual

This section describes the:

- contents of this manual
- notation conventions used throughout the manual
- command syntax and syntax diagrams

## Manual Contents

This reference provides script manipulation information for advanced testing concepts, such as data-driven testing. It includes the following sections:

- **Chapter 1, "Hiperstation Script Language":** How to use REXX to modify your scripts for specific jobs.
- **Chapter 2, "Common Programming Interface":** How to use programs written in REXX to manage communications within your system.
- **Chapter 3, "3270 Repository Filtering and Copy Utility":** How to filter messages or sessions from a large repository for more manageable review or script creation.
- **Chapter 4, "Guidelines for Successful Stress Testing":** How to plan, set up, and execute application stress testing, and how to use the results of your tests.

## Notation Conventions

This document uses the following notations to describe Hiperstation screens and the information you enter on those screens:

- Technical revisions made to this document are indicated by revision bars in the left margin, as shown here.

- Sample screens generally show only the information appropriate to the accompanying text, for example:

```
ZOOM:PF23 ----- Hiperstation ----- LINE 1 OF 24
COMMAND ==> record                               SCROLL ==> HALF
Record OFF Play OFF Journal OFF Compare Log OFF autoDoc OFF
***USR2312 WELCOME TO CICS/MVS *** 10:11:42
```

- Blank lines or standard footings, as shown below, are usually omitted from screen illustrations.

```
Press ENTER to begin recording. Use END to cancel setup.
```

- Information you enter is printed in **boldface**.
- Words defined within paragraphs are *italicized*.
- The phrase “select an option” refers to typing a slash next to one of the presented options and pressing Enter.

## Command Syntax

Hiperstation uses statements and commands that you will learn about later. This section describes text conventions used in statement and command descriptions. It also explains how to read syntax diagrams.

- Keywords are shown in UPPERCASE letters. Enter keywords and special characters as shown. If a command or statement name can be abbreviated, the description shows only the abbreviation in uppercase letters.

```
Find string [NEXT|PREV|ALL]
```

The uppercase F in the command FIND means that you can enter the FIND command either as FIND or F.

- Variables and generic descriptions of parameter values are shown in lowercase letters.
- Vertical bars (|) are separators between mutually exclusive options.
- Brackets ([ ]) indicate optional parameters. All parameters not enclosed in brackets are required.
- Ellipses (...) indicate that the value can be repeated.

## Reading Syntax Diagrams

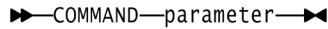
Syntax diagrams define primary command syntax.

A **parameter** is either a keyword or a variable.

All KEYWORDS are shown in uppercase characters and must be spelled exactly as shown. You cannot substitute another value. If any part of a KEYWORD is shown in lowercase characters, that part is optional.

Variables are user-specified values and are printed in lowercase italics. For example, *dataset-name* indicates you are to substitute a value.

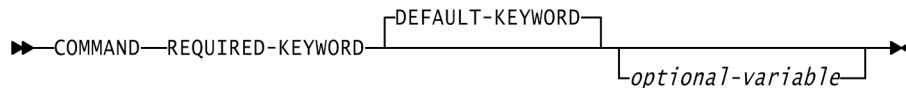
The syntax for commands is described in diagrams that help you visualize parameter use. The following example shows a command and a parameter:



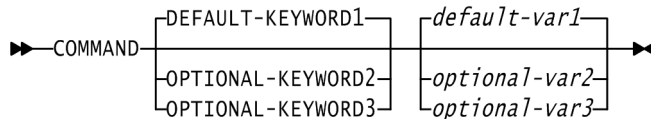
Read the diagrams from left to right and from top to bottom. These symbols help you follow the path of the syntax:

- ▶▶ indicates the beginning of a statement.
- indicates the statement is continued on the next line.
- ▶ indicates the statement is continued from the previous line.
- ▶▶ indicates the end of a statement.

Required parameters appear on the horizontal line (the main path). Optional parameters appear below the main path. Default parameters appear above the main path and are optional. The command executes the same regardless of whether the default parameter is included.



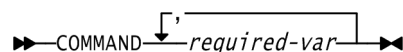
Vertically stacked parameters are mutually exclusive. If you must choose a parameter, one item of the stack appears on the main path. If the parameters are optional, the entire stack appears below the main path. If a parameter in a stack is the default, it appears above the main path.



If the same parameters are used with several commands, their syntax may be documented in a separate diagram. In the command syntax, these common parameters are indicated with separators before and after the parameter name.



An arrow returning to the left indicates a repeatable item. If the arrow contains a comma, separate the repeated items with a comma.




---

## Accessibility

In accordance with section 508 of the Rehabilitation Act of 1973, Compuware has committed to making its products and services easier to use for everyone including people with disabilities.

Hiperstation is a mainframe application that runs on IBM's OS/390 and z/OS operating systems. It has an ISPF interface that is accessed with IBM 327x-type terminals or with 3270 terminal emulator software. Since the mainframe environment offers few accessibility features, Compuware has focused its attention, with regard to accessibility, on 3270 terminal emulator software running on personal computers (PCs) with Microsoft Windows 2000 or more current. Hiperstation supports, with a few exceptions, Microsoft Windows accessibility features and Window-based Assistive Technology (AT) software and devices, such as Braille devices, screen readers, magnifiers, etc.

**Note:** Hiperstation is intended for use by mainframe software developers, programmers, and testers. Much of the input and output used or produced by Hiperstation, such as Job Control Language (JCL) and hexadecimal contents or dumps of memory, are not easily understood by the general public. Unfortunately, as in the case of hexadecimal dumps, data in these formats can be confusing to screen readers and therefore confusing to the people who use them. Effective use of this application requires the specialized knowledge of a mainframe systems software developer or programmer.

Hiperstation accessibility was evaluated using:

- Freedom Scientific's JAWS screen reader
- Attachmate Corporation's myExtra Presentation Services tn3270 emulator
- Microsoft's Windows accessibility features
- Adobe Reader using the "Read Out Loud" function

This evaluation not only identified accessibility exceptions, but revealed emulator and screen reader compatibility issues that in some cases can be remedied through appropriate configuration.

## Installing Windows Accessibility Features

Microsoft Windows operating systems offer several accessibility features to aid individuals who have difficulty typing or using a mouse, who are blind or have low vision, or who are deaf or are hard-of-hearing. Install these features during setup or later using the Windows installation disks. Refer to the "accessibility" topics in the Windows Help system for information on installing and using these features. Visit the Microsoft Web site, <http://www.microsoft.com/enable>, for additional information and tutorials.

## Selecting Font and Font Size

Microsoft Windows and emulator software packages offer font and font size settings to accommodate users with low vision. The emulator software's tool bars and dialog boxes typically use the font specified in the operating system, while the terminal presentation uses the font and font size specified in the emulator. To change the font or font size:

- Presented on the toolbars and dialog boxes, refer to the Windows Help system.
- Presented in the terminal window, refer to the emulator's documentation or Help.

Some screen readers recommend certain fonts and font sizes for compatibility. For example, Freedom Scientific recommends setting the font to a common or "plain" font such as Lucida, Courier, or Times New Roman, and setting the font size to 10 points or smaller. Refer to the screen reader's documentation or Help for these recommendations.

## Changing Color and Contrast

Color and contrast settings can assist users with low vision. ISPF and most emulator software packages offer color and contrast settings. If you are accessing Hiperstation with a terminal, use ISPF settings. Otherwise, adjust the color and contrast in the emulator software. Refer to ISPF Help or the emulator's documentation or Help.

## Setting Cursor Blink Rate

The blink rate of the cursor can affect users with photosensitive epilepsy. Additionally, some screen readers require a specific blink rate. Some readers automatically adjust the blink rate while others expect you to adjust the rate. Refer to:

- The Microsoft Windows Help to find out how to set the cursor blink rate.
- The screen reader's documentation or Help to find out the recommended blink rate.

## Using Keyboard Shortcuts

Keyboard access to application functions support users who cannot use a mouse.

Microsoft Windows provides keyboard access to all functions within the operating system, such as:

- Displaying or hiding the Windows Start Menu
- Showing the Desktop
- Minimizing all windows
- Searching for files
- Accessing the help system
- Controlling the behavior of the Windows accessibility features, for example, toggling the listening status to the microphone, or cycling focus backward and forward.

Most Windows-based applications also provide keyboard access to their functions. The combination of keys required to execute a given function is called a keyboard shortcut. Refer to the "Keyboard Shortcuts" topics in the Windows Help system for a complete list of Windows shortcuts. For a list of the shortcuts that are available in the emulator software or any third-party accessibility tool, such as the JAWS screen reader, refer to the software's documentation or Help.

## Accessibility Exceptions Work Arounds

During Hiperstation accessibility evaluation, some exceptions were encountered where some accessibility features or AT were not fully supported. The causes of and solutions for these exceptions are currently under investigation by Compuware Corporation.

### Known Exceptions

Accessibility exceptions include:

- Function Key (F Key) information at the bottom of the screen is not read by the screen reader on some screens. This is believed to be caused by an external interface. See "Solutions" for a viable work-around.
- Some system error and warning messages are not read by the screen reader when issued. Believed to be caused by an external interface. See "Solutions" for a viable work-around.
- Some pop-up dialog boxes or windows do not capture exclusive focus and are not read correctly by the screen reader. This is believed to be caused by an external interface. No known solution is currently available.
- System error and warning messages do not capture visual focus for the screen magnifier. This is believed to be caused by an external interface. No known solution is currently available.
- Some entry and display fields lack individual labels. When entry fields are accessed using the Tab key, the entire individual line is read.

- Current Web-based reports are not easily navigated using the keyboard and lack table element coordinate tags. Additionally, some of these reports contain color-coded elements — for example, the color of some elements conveys meaning.

## Solutions

When the screen reader fails to read the F Key information upon entry to a new screen, do one of the following:

- Use the arrow keys to move the cursor down to the lines with the F Key information. The screen reader reads each line as the cursor is placed on it.
- Press the Page Up key for the screen reader to reread the entire screen.

When the screen reader fails to read an error or warning message, an audio alert occurs if this feature is enabled on your system. Press the Up key to place the cursor on the line containing the error message, usually on the top or title line. The screen reader reads the line and its error message individually.

---

## Getting Help

Compuware provides a variety of support resources to make it easy for you to find the information you need.

### Compuware FrontLine Customer Support Website

You can access online information for Compuware products via our FrontLine customer support website at <https://go.compuware.com>.

Compuware FrontLine provides access to critical information about your Compuware products. You can review frequently asked questions, read or download documentation, access product fixes, or e-mail your questions or comments. The first time you access Compuware FrontLine, you are required to register and obtain a password. Registration is free.

### Contacting Customer Support

#### Phone

- USA and Canada: 1-800-538-7822 or 1-313-227-5444.
- All other countries: Contact your local Compuware office. Contact information is available at <https://go.compuware.com>.

#### Web

You can report issues via the Quick Link **Create & View Support Cases** on the Compuware FrontLine home page.

**Note:** Please report all high-priority issues by telephone.

#### Mail

Compuware Customer Support  
Compuware Corporation  
One Campus Martius  
Detroit, MI 48226-5099

## **Corporate Website**

To access Compuware's site on the Web, go to **<https://www.compuware.com>**.

The Compuware site provides a variety of product and support information.



# Chapter 1.

## Hiperstation Script Language

Hiperstation implements IBM's REXX as its script language. The script language utility significantly extends the capabilities of Hiperstation's playback function in both ISPF and unattended processing.

In addition to the REXX environment provided with MVS TSO/Extensions, Hiperstation offers a set of variables that allows information to be retrieved about the Hiperstation environment and data to be retrieved from the current and expected messages. Among the solutions that a script language offers are:

- Multiple ports that require unique input field values during batch execution.
- Scripts that require unique input field values each time they are played.
- Actions taken based on comparison of system output.
- The ability to call and share common scripts.

---

## Statement of Compliance

The implementation of the Hiperstation script language makes every reasonable attempt to function as described in the appropriate IBM TSO/E REXX Reference. Many, but not all, features and functions of the REXX language implemented in the MVS environment are available with Hiperstation.

The Hiperstation script language facilities are the same as REXX in the non-TSO/E environment. This means that all language statements provided by REXX are available with Hiperstation. The only unavailable facilities are functions that are part of the TSO/E function package such as LISTDSI, OUTTRAP, and SYSDSN.

Hiperstation's REXX implementation is limited to the functions in the MVS environment. The MVS, LINK, and ATTACH host command environments are available in the Hiperstation script language.

---

## Installation Requirements

Use of the Hiperstation script language requires the installation of IBM's TSO/E Version 2 or more current.

---

## Capabilities of the Language

In stress testing, where multiple terminal sessions are involved, systems often require unique data for each session. This data is usually a user ID and a password. Hiperstation's table processing and variable substitution allows a table of users to be *read in* and substituted based on criteria such as the terminal ID or the *port number* of the session.

In load testing, you may need to provide a unique input (for example, policy number or social security number) for each iteration or *cycle* of the script's execution. Hiperstation's external file access and variable CYCLE easily facilitate such a requirement.

You may need to perform a *fix-up* script based on the output of a particular transaction. For example, if a batch job tries to close a database, and users are still attached, you can

issue a message and reschedule the job. Hiperstation's conditional logic and CALL capability easily answer this requirement.

Some installations may need to create *special operator* scripts that perform operations that normally require terminal operators. For example, CICS installations might need to run the CEMT transaction to OPEN and CLOSE databases during nightly backups or reorganizations. Hiperstation addresses this requirement.

Features that the REXX language provides include:

- Variable substitution on simulated user-entered input parameters (<Inn>, <I(xx,xx)>, or <I CURSOR> level only)
- Variable assignment using simple and complex expressions
- Conditional logic and flow control: IF-THEN-ELSE, DO-WHILE, DO-UNTIL, controlled repetitive DO, and branching
- Inclusion of data from external datasets to be used as terminal input
- Random number generation
- Ability to call other Hiperstation scripts, REXX EXECs, or other programs from an executing Hiperstation script
- Ability to introduce user-generated messages in the Hiperstation log
- Internal and external function calls
- Integer, decimal, and floating point arithmetic
- Debugging and trace options

## Hiperstation Additions to REXX

Hiperstation adds predefined variables to the REXX language. This gives REXX access to the screen data and other environmental data such as the name of the current domain destination or the port number in unattended mode processing.

Hiperstation also adds additional commands not found in the non-TSO/E environment. Use these commands through the HSCMDS environment. For more information, see "Hiperstation HSCMDS Command Environment" on page 1-35.

## Statements

A 3270 script recorded by Hiperstation for VTAM is a series of input and output groups. The input group represents the actions taken at the terminal by the user. It records the data entered in each of the input fields and the PF key that was pressed to transmit the data. The output group records the output from the transaction. This data includes the output screen images and the response time of the transaction.

The recorded types are further described in this chapter. The combination appears as follows:

```
<INPUT>
...
</INPUT>
<OUTPUT>
...
</OUTPUT>
<INPUT>
```

**Note:** The ellipses (...) indicate that there are input or output records in the group. The record type format, which is delimited with angle brackets (< >) complies with the format used by the SAA Dialog Tag Language (DTL).

Hiperstation script language execution units consist of one or more REXX statements. The statements can be placed by themselves in a dataset member or in a dataset member that is combined with a recorded script. When placed in a script, the statements must follow an output group and precede an input group. The output group terminates with an </OUTPUT> record, and the input group begins with an <INPUT> record.

The following example shows the proper placement of script language statements:

```
</OUTPUT>
...
IF POS('LOGON IN PROGRESS',SCREEN) = 0
  SAY 'USERID' userid,index.1 'ALREADY IN USE, RETRYING'
  USERINDEX = USERINDEX + 1
  IF USERINDEX > MAXUSERIDS
    EXIT
  ELSE
    CALL RETRYLOG
<INPUT>
...
```

Script language statements can also be placed at the beginning of a script before the first input group. Typically, script language statements that initialize variables and tables are placed at the beginning of a script.

## Iterative Statements and Multiple Transactions

Iterative statements (such as DO WHILE or DO UNTIL) can span several transactions. For example, consider a set of transactions recorded in a Hiperstation script:

```
<INPUT>
...
</INPUT>
<OUTPUT>
...
</OUTPUT>
<INPUT>
...
</INPUT>
<OUTPUT>
...
</OUTPUT>
```

**Note:** The ellipses (...) represent one or more recorded statements.

The iterative statements can surround this entire group. For example, to repeat a series of transactions five times, you can add the following two statements — a DO and an END — at the beginning and end of the transactions:

```
DO I = 1 TO 5
  <INPUT>
  ...
  </INPUT>
  <OUTPUT>
  ...
  </OUTPUT>
  <INPUT>
  ...
  </INPUT>
  <OUTPUT>
  ...
  </OUTPUT>
END
```

## Conditional Statements

Conditional execution can be put in a script using any of the following REXX constructs: IF...THEN...ELSE. The DO statement provides iterative looping. The forms of the DO instruction are:

```
DO I=1 TO 5
...
END
DO WHILE I<10 /*While checked at the beginning of the loop*/
...
END
DO UNTIL I>10 /* Until checked at the end of the loop */
...
END
```

In addition, REXX includes statements that provide *elegant* exits from DO loops. The ITERATE statement transfers control to the top of the loop, and the LEAVE statement transfers control to the statement following the DO loop.

The REXX SELECT statement is used to conditionally run one of several alternative instructions. The SELECT statement is similar to the CASE statement of C.

---

## Language Constructs

This section introduces the REXX language and its relationship to Hiperstation scripts. It also provides several examples. An experienced REXX user will find portions of this material useful as a review of REXX capabilities and features. They are included here to initiate new REXX users and to assist those evaluating the Hiperstation program who do not have in-depth REXX knowledge.

Hiperstation script language statements are made up of a series of tokens. These tokens can be variable names, arithmetic operators, constants, and other language constructs. Following are examples of valid Hiperstation script language statements:

```
COLUMN = COLUMN + 1
IF COLUMN > 79
    COLUMN = 0
ROW = ROW + 1
IF ROW > 24
    SIGNAL END OF SCREEN
```

## Variables

A variable is an object whose value can change during the execution of a Hiperstation script. The process of changing the value of a variable is called assignment. The value of Hiperstation variables can be either characters or integers.

Variable names can be from 1 to 32 characters long and can be entered in upper or lower case. Hiperstation always treats variables as though they are entered in upper case. This means that the following expressions all refer to the same variable:

```
MyVar = 'Hello'
MYVAR = 123
myvar = -270
```

## Constants

The language allows the introduction of constants in two different forms. Only constants that consist of numbers and an optional minus sign can be entered as a character string of 1 to 16 digits. Numeric constants can be optionally prefixed with a minus sign to indicate a negative value. The following are valid numeric constants:

```
1
1234987
-222
```

Other constants, non-numeric and numeric, can be enclosed in either single or double quotes. These constants are called literal strings. The following are valid literal strings: 'Guacamole' and "The Great Gatsby".

## Operators

Following are examples of valid operators in the Hiperstation script language:

Table 1-1. Hiperstation Script Language Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Integer division

All operators supported by REXX are supported in the Hiperstation environment. For more information on the available operators, refer to the *TSO/E REXX Reference Guide*.

## Function Calls

Function calls can be included in an expression anywhere that a data term (either a variable name or a constant) is valid. The format of a function call is:

```
function_name (arg1, arg2, ...arg n)
```

The arguments can take the form of expressions. For example:

```
C_ROW = 10
C_COL = 32
MYVAR1=SUBSTR(SCREEN,C_ROW*80+C_COL,1+2*4)
```

The above call is equivalent to entering:

```
MYVAR1 = SUBSTR(SCREEN,832,9)
```

## Expressions

The purpose of an expression is to compute a value. Expressions are composed of operands such as constants and variables, operators, and possibly function calls. The process of evaluation is to substitute values for the operands and perform the operations.

REXX expression evaluation is from left to right. This is modified by parentheses and operator precedence:

- When parentheses are encountered (other than those that mark function calls or array subscripts) the entire sub-expression between the parentheses is evaluated immediately when the term is required.
- When the sequence:

```
term operator1 term operator2 term
```

is encountered, and operator2 has higher precedence than operator1, the expression (term operator2 term) is evaluated first, applying the same rule repeatedly as necessary.

## Assignments

A variable is an object whose value can be changed during the execution of a Hiperstation script. The process of changing the value of a variable is called *assignment*. The value of a variable is a single character string, from 1 to 256 characters long, that can contain any characters.

Statements of the form: **variable\_name=expression** are statements known as *assignments*. An *assignment* gives a variable a (new) value.

---

## Retrieving Data from External Sources

The REXX EXECIO function allows data to be read from external sources. In a single statement, REXX can read an entire dataset and place the records in a set of variables with a common prefix, or stem, plus one variable per record.

The data can be parsed using one or many of the REXX parsing functions such as PARSE. The following example reads in a set of user IDs and passwords from the DD name DDUSERS and places the values in USER.n and PASSWORD.n. The user IDs and passwords are placed in a dataset in *free format*, with one user and password per record, separated by one or more spaces.

```
"EXECIO * DISKR DDUSERS (STEM INPUT. FINIS"
DO I=1 TO INPUT.0
  PARSE VAR INPUT.I USER.I PASSWORD.I
END
```

## Retrieving Screen Data

Screen data can be retrieved from a screen using Hiperstation variables. The SCREEN variable contains a copy of the current screen image. Portions of the screen can be extracted using functions such as SUBSTR. The contents of the screen can be examined and searched using standard functions such as POS.

For example:

```
TITLE = SUBSTR(SCREEN,24,32)
```

retrieves 32 characters of screen data located in row 0, column 24, and places them in the TITLE variable.

## Comments

A comment can be introduced in two ways:

- Text enclosed in /\* ... \*/ is considered to be a comment. To begin a comment, use the characters /\*. End the comment with the characters \*/.
- A record that begins with the asterisk (\*) character in column 1 is considered to be a comment statement.

## Examples:

```
/*statement           Meaningful comment           */
IF A=B THEN           /* Set 'a' to the value in 'b' */
*                   This entire line is regarded as a comment.
```

## Using Language Statements

This section describes some of the common uses of REXX in Hiperstation scripts.

### Terminal Input Substitution

Terminal input substitution allows constants and variables to be substituted or assigned to the user's entered input of a Hiperstation script. For example, a script may have an input field where the user entered **TSOHP00** in input field *number 1*. Hiperstation for VTAM records this in the script as: `<I01>"TSOHP00"`

A variable can be substituted in the place of the constant data by replacing the text with a variable name: `<I01>USER`

**Note:** REXX variable substitution is sensitive to spaces. Any space between the closing angle bracket (`>`) and the REXX variable is included as part of the data. For example, if the variable **USER** contains **abcd**, then:

- `<I01>"abcd"` is the same as `<I01>USER`
- `<I01> USER` is the same as `<I01> "abcd"`.

Assuming that you previously set the **USER** variable to a value using an assignment statement, Hiperstation for VTAM places the value in the input field on the screen image, and is later transferred to the domain destination. The rules of substitution are described in the *TSO/E REXX Reference Manual*.

#### *String Concatenation Syntax*

```
<Inn> varname
```

### Example

Assume that the **USERS** variable is initialized from a dataset named by the DD statement **INDD** as follows:

```
"EXECIO * DISKR INDD (STEM USERS."
```

The following statements put a unique value in input field 1 for each port in unattended playback:

```
<I01>USERS.PORT
```

You can also assign values to variables and subsequently use them in input fields:

```
VARINPUT = 15*CYCLE /*Assign variable VARINPUT */
<I15>"THIS IS TYPED AS: "VARINPUT
```

Assuming the value currently assigned to **CYCLE** is 2, the following is input field number 15:

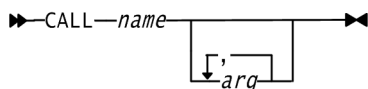
```
THIS IS TYPED AS: 30
```

## Modular Scripts

The **CALL** statement invokes:

- Other scripts of the same protocol, for example, call a 3270 script from a 3270 script
- Additional script logic implemented as REXX internal procedures
- Programs written in REXX, Assembler, C, or other high-level languages

Use this to create scripts that perform tasks and then return control to the original script.



#### Notes:

1. You cannot use a valid TSO command as a script name when using CALL statements.
2. A subroutine that is called from within a script member must be accessible for the Unattended Compare to execute properly. It either needs to be in the Script dataset or in a dataset concatenated to the DD SYSLIB in the Unattended Compare JCL.

## Examples

To call the CICSTERM script, code the following CALL statement:

```
CALL CICSTERM
```

To call the CUSTADD script, which expects two arguments (customer name and customer number), code the following CALL statement:

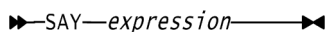
```
CUSTNUM='7234CB321'  
CUSTNAME='John Smith'  
CALL CUSTADD CUSTNAME,CUSTNUM
```

To accept the two arguments, the first non-comment line in the CUSTADD script is the REXX arg instruction. The ellipsis (...) represents one or more recorded statements.

```
arg name, number  
<VERSION>6  
<OUTPUT>. . .
```

## Script Reports

The REXX SAY statement provides a mechanism to create reports from an executing script.



The result of the expression evaluation is written to the Hiperstation log. In unattended processing, this is the SYSPRINT dataset. With Hiperstation for VTAM's Domain Traveler, this is the log defined by the RLOG command. For example, the following statement places the string **Compuware Corporation** in the log:

```
SAY 'Compuware Corporation'
```

The following statement puts a blank line in the log:

```
SAY ' '
```

The following statement puts the text **SCRIPT CICSTEST WAS EXECUTED ON 03 Apr 2006 at 11:22:58** in the log:

```
SAY 'SCRIPT CICSTEST WAS EXECUTED ON' date() 'at' time()
```

## Custom Return Codes

Create custom MVS job step return codes to provide more definitive playback results. For example, a playback job that ends successfully may result in a return code of 0, even if the application returned an unexpected value. Add REXX logic to the script to evaluate



message content and produce a specified return code value if the message content is incorrect.

To create a custom return code, add a REXX IF statement before the activity to be evaluated and include a RETURN or EXIT statement to specify the return code value. For example, the following logic returns a code of 108 and produces the message 'Unable to continue script execution' if 'IKJ54621I' appears anywhere on the input screen.

```
IF POS("IKJ54621I",SCREEN) > 0 THEN DO
  SAY "Unable to continue script execution"
  <INPUT>0000001
  <THINK>00.00.000 AT 00:00:00.000
  <KEY>PF3
  <CURSOR>01,01
  </INPUT>
  RETURN 108
END
```

On the RETURN or EXIT statement supply a numeric value between 0 and 4095. If you supply a non-numeric value or an out-of-range value, Hiperstation returns the decimal equivalent of the lower twelve bits. For example, if you specify 4100, Hiperstation returns 4. Additionally, be sure to select values that do not conflict with Hiperstation's internal return codes. Refer to the:

- *Hiperstation for VTAM User Guide* for a complete list of 3270 return codes.
- *Hiperstation for Mainframe Servers User Guide* for a complete list of APPC and TCP/IP return codes.
- *Hiperstation for WebSphere MQ User Guide* for a complete list of MQ return codes.

Each of these manuals also contains a detailed explanation of return code processing.

---

## Hiperstation REXX Variables

Hiperstation provides several predefined REXX variables that return playback information. Incorporate these variables into REXX logic that you add to your scripts for advanced testing. For example, use a variable that returns the replay iteration value to drive dynamic data replacement, or place a variable that returns the terminal ID into a SAY statement for troubleshooting purposes.

### Notes:

1. These variables are provided to the main script being played back, not to called scripts. This is due to the way TSO/E REXX separates and protects variables in external subroutines. Pass any required variables as arguments to called scripts.
2. Do not use variables set by Hiperstation on the left side of an assignment statement. Changing their values has no effect on Hiperstation, but can cause errors later in the script's execution if the variables are inspected.

This section defines the variables available for each type of testing you can perform with the Hiperstation suite of products.

## 3270 REXX Variables

The following predefined REXX variables return information from 3270 playback jobs. Incorporate these variables into REXX logic that you add to your scripts. For example, the CYCLE variable returns a value that you can use to drive dynamic data replacement.

**Note:** Understanding playback, especially the GROUP and SCRIPT statements, is integral to determining how to apply these variables. Refer to the *Hiperstation for VTAM User Guide* to learn more about these statements.

**COMPRESULT**

Indicates whether any mismatches occurred during the most recent screen comparison. It does not store a running count of mismatches. COMPRESULT returns a numeric value of 0 if there are no mismatches. If there are one or more mismatches, COMPRESULT returns a numeric value of 4. The following example shows how COMPRESULT might be used:

```
IF COMPRESULT > 0 THEN
  SAY 'Compare result is ' COMPRESULT
```

**COLUMNS**

Indicates how many columns are in each row of the terminal. For example, to set a loop counter for the number of columns in a screen:

```
C_COUNTER = COLUMNS
```

**CYCLE**

Returns the repeat value of the script. This value is set to one when a script begins execution.

The variable is incremented by one each time a REPEAT cycle occurs. You can enter REPEAT on the Play Setup screen in ISPF mode and by using the REPEAT keyword of the SCRIPT statement in unattended mode functions.

The CYCLE variable provides a unique value during each iteration of a script's execution.

The following example shows a list of policy numbers being read in using the EXECIO function. Then the variable POLICY is used in conjunction with CYCLE to provide a unique value during each iteration of the script.

```
"EXECIO * DISKR POLICIES (STEM POLICY."
. . .
<I01>POLICY.CYCLE
```

**GRPCYCLE**

Returns the repeat value of the group. This value is set to one when a group begins execution.

The variable is incremented by one each time a group finishes all of its scripts. You can use the REPEAT keyword on the GROUP statement in batch processing.

The GRPCYCLE variable provides a unique value during each iteration of a group's execution.

The following example shows a list of policy numbers being read in using the EXECIO function. Then the variable POLICY is used in conjunction with GRPCYCLE to provide a unique value during each iteration of the group.

```
"EXECIO * DISKR POLICIES (STEM POLICY."
. . .
<I01>POLICY.GRPCYCLE
```

**LASTAID**

Returns the 1-byte hex value of the AID key used in the previous input message. The AID key comes from the <KEY> script statement. The following example shows a message in the log if the CLEAR key was used:

```
IF LASTAID = '6D'X THEN
  SAY 'The CLEAR key was issued.'
```

**LASTKEY**

Returns a value listing the AID key used in the previous input message. The AID key comes from the <KEY> script statement. The following example shows the LASTKEY variable used to repeat the <KEY> that was last issued:

```
IF POS ('Processing...',screen) > 0 THEN
DO
  <INPUT>
  <EXECUTE>"<KEY>"||LASTKEY
</INPUT>
END
```

**LUNAME**

Returns the terminal ID for the session. The following example places the LUNAME variable in the log using the SAY statement:

```
SAY 'We are signed on to ' LUNAME
```

**OUTPUTNUM**

Returns the sequence number of the <OUTPUT> tag when the screen comparison is made. The following example places the OUTPUTNUM variable in the log using the SAY statement:

```
SAY 'Expected output sequence number ' OUTPUTNUM
```

**PORT**

Returns a value listing the port (terminal). In ISPF mode, the port number is always 1. In unattended mode, the PORT number is always a unique, sequentially assigned identifier.

The following example shows a list of TSO user IDs being read in using the EXECIO function. Then the USER variable is used in conjunction with PORT to provide a unique user ID for each port (terminal) signing on.

```
"EXECIO * DISKR POLICIES (STEM USER."
. . .
<I01>USER.PORT
```

**ROWS**

Indicates how many lines are in the terminal. The following example sets a variable for a loop counter to indicate the number of rows in the screen:

```
ROWCOUNT=ROWS
```

**SCREEN**

Contains a copy of the current screen buffer. The following example extracts 32 characters from row 12, column 20 of the current screen.

```
SCREENOUTPUT=SUBSTR(SCREEN,(12-1)*80+20,32)
```

For details about extended attributes in the SCREEN variable, see "SCREENXATR" on page 1-12.

**SCREENIN**

Examines a screen image after inputs are applied. This data is made available after the next output from the application. It contains all screen information including user-entered data, which can be extracted and referenced later in the script.

The following example extracts 32 characters from row 12, column 20 of the current screen.

```
USERINPUT=SUBSTR(SCREENIN,(12-1)*80+20,32)
```

### **SCREENEXPECT**

Contains a copy of the current screen expected buffer.

The following example extracts 32 characters from row 12, column 20 of the expected screen.

```
SCREENOUTPUT=SUBSTR(SCREENEXPECT,(12-1)*80+20,32)
```

### **SCREENEXPECT\_SIZE**

Contains the number of characters in the screen expected buffer. For a 24x80 screen, this variable is set to 1920.

The following statement assigns variable "BufferLength" to the size of the buffer for the expected screen.

```
BufferLength=SCREENEXPECT_SIZE
```

### **SCREENSIZE**

Contains the number of characters in the screen buffer. For a 24 x 80 screen, this variable is set to 1920.

The following statement assigns variable "BufferLength" to the size of the buffer for the current screen.

```
BufferLength=SCREENSIZE
```

### **SCREENXATR**

Contains the extended attributes referenced by the SCREEN variable.

The SCREEN variable contains the characters and attributes that represent the current screen image. To maintain a viewable SCREEN image, it is not possible to place the extended attributes in the SCREEN variable since the extended attribute may require up to nine bytes to represent a single byte on the screen. To accommodate this requirement, the SCREEN variable may contain a hexadecimal 22 to represent the existence of an extended attribute. The extended attribute can be found by taking the relative position of the extended attribute marker and multiplying it by the length of an extended attribute cell (nine bytes) and then adding one to reference the start of the cell. Following is the order of the nine attributes:

- byte 1: field attribute
- byte 2: outline mode
- byte 3: character set
- byte 4: shift in/shift out
- byte 5: foreground color
- byte 6: background color
- byte 7: highlighting
- byte 8: transparency
- byte 9: field validation

Figure 1-1 shows a subroutine that extracts the extended attributes for the screen from the SCREENXATR variable. A hex 22 in the screen buffer indicates the existence of an extended attribute.

To find the extended attribute, compute its offset in the SCREENXATR variable. To do this, take the relative position of the hex 22 and multiply it by the length of an extended attribute cell, which is nine bytes, then add one to reference the start of cell.

**Figure 1-1.** Subroutine for Extracting Extended Attributes

```
x = pos('22'x,screen,1)
do while (x>0)
  attrs = substr(screenxatr, (((x-1)*9)+1),9)
  say 'ext attrs for screen position 'x
  say 'field attribute = ' substr(attrs,1,1)
  say 'outline mode = ' substr(attrs,2,1)
  say 'character set = ' substr(attrs,3,1)
  say 'shift in/out = ' substr(attrs,4,1)
  say 'foreground color = ' substr(attrs,5,1)
  say 'background color = ' substr(attrs,6,1)
  say 'highlighting = ' substr(attrs,7,1)
  say 'transparency = ' substr(attrs,8,1)
  say 'field validation = ' substr(attrs,9,1)
  say
  x = pos('22'x,screen,x+1)
end
```

**TPFNAME**

Contains the name of the domain destination for the Hiperstation session. The following example checks that you are signed on to the domain destination called 'CICSTEST':

```
IF TPFNAME = 'CICSTEST' THEN
  SAY "Logon to CICSTEST is successful"
```

**APPC REXX Variables**

Hiperstation provides several predefined REXX variables that return information during playback. Incorporate the variables into REXX logic that you can optionally add to your scripts. This section provides definitions for the APPC REXX variable types:

**APPC Error Determination REXX Variables**

The variables in Table 1-2 provide information pertaining to an error encountered during script processing. During playback, Hiperstation issues VTAM macros to perform operations indicated in the script. Return codes from these macros are placed in the following REXX variables.

Many of these variables provide data from the Request Parameter List (RPL). These values are the character representations of binary values.

**Table 1-2.** APPC Error Determination REXX Variables

REXX Variables	Description
CPICRC	Contains the equivalent CPIC return code for the error. If there is no error, the CPICRC is zero.
ERRMSGNO	Contains the Hiperstation error condition message number.
RPLFBK2	Contains the feedback code from the previous APPC operation. It is the register 0 value.
RPLRTNCD	Contains the return code from the previous APPC operation. It is the register 15 value.
RPL6CCST	Contains the conversation state from the VTAM RPL of the previous APPC operation.
RPL6RCPR	Contains the primary reason code from the VTAM RPL of the previous APPC operation.
RPL6RCSC	Contains the secondary reason code from the VTAM RPL of the previous APPC operation.
RPL6SNSI	Contains the sense data from the VTAM RPL of the previous APPC operation.
RPL6WHAT	Contains the "what received" value from the VTAM RPL of the previous APPC operation.

## APPC Environment Data REXX Variables

The variables in Table 1-3 provide information pertaining to the APPC conversation. These variables cannot be used to set their corresponding conversation values. They are read only informational variables available for review while a script is running.

Table 1-3. APPC Environment Data REXX Variables

REXX Variables	Description
CONVCORR	Contains the conversation correlator.
LOGMODE	Contains the name of the logmode.
LUWINST	Contains the logical unit of work instance number for the conversation.
LUWNAME	Contains the logical unit of work name for the conversation.
LUWSEQ	Contains the logical unit of work sequence number for the conversation.
OPNRCDES	Contains the feedback code from the CNOS.
OPNRTNCD	Contains the return code from the CNOS.
RECEIVER	Contains the LU name receiving the ALLOCATE.
SECUSER	Contains the user ID used for security purposes.
SENDER	Contains the LU name sending the ALLOCATE.
TPNAME	Contains the transaction program name.

## APPC Application Data REXX Variables

The variables described in this section provide values for both the expected and actual data received from an application during playback. The variables are read only and are set after normal or expedited data is received from the partner application. The values are usable only when they immediately follow <CONTENT> tags associated with the <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verb.

The variables represent the data received from the partner application. They must be examined in the context of an <INBOUND> group of verbs in a Hiperstation-allocated conversation or in an <OUTBOUND> group of verbs in a Hiperstation-accepted conversation.

For example, an ALLOCATE statement in the conversation log runs the script shown in Figure 1-2. In this example, the first REXX SAY statement lists a null value (0 length string) because Hiperstation has not received data from the partner application. However, the four REXX SAY statements in the <INBOUND> group produces meaningful output.

**Figure 1-2.** Sample Script Using APPC Application Data REXX Variables

```

<VERSION>8
<ALLOCATE>
<OUTBOUND>
  <SEND_DATA>
  <CONTENT>...00000001This is outbound data
  say appc_actual          /* This is not meaningful, a null value is displayed. */
  <PREPARE_TO_RECEIVE>
</OUTBOUND>
<INBOUND>
  <SEND_DATA>
  <CONTENT>...00000002This is inbound data
  say appc_actual          /* The actual data received (up to 2048 bytes). */
  say appc_expected       /* The data we expect to receive (complete). */
  say appc_actual_length  /* The length of the actual data (may be more than 2048). */
  say appc_expected_length /* The length of data we expect to receive (complete). */
  <PREPARE_TO_RECEIVE>
</INBOUND>
<OUTBOUND>
  <DEALLOCATE>
</OUTBOUND>

```

### ***APPC\_ACTUAL***

The APPC\_ACTUAL variable contains the most recently received data from the partner application. This data can be either normal data from a <SEND\_DATA> verb or expedited data from a <SEND\_EXPEDITED\_DATA> verb. If the application sends more than 2048 bytes of data, only the first 2048 bytes are available in this variable.

The variable contains a relevant value only when it immediately follows <CONTENT> tags associated with <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verbs, and its value is reset when the next script tag is encountered. The variable generates useful data only when it is set following APPC verbs that represent data received by Hiperstation. As a result, using the variable in a section of the script where Hiperstation is sending data will not produce meaningful data.

Figure 1-3 shows the APPC\_ACTUAL variable following a <CONTENT> tag associated with a <SEND\_DATA> verb.

**Figure 1-3.** Sample Script Containing the APPC\_ACTUAL Variable

```

<INBOUND>
  <SEND_DATA>
  <CONTENT>...00000002This is inbound data
  say appc_actual          /* The actual data received (up to 2048 bytes). */
  <PREPARE_TO_RECEIVE>
  say appc_actual         /* A 0 length string is displayed. */
</INBOUND>

```

### ***APPC\_EXPECTED***

The APPC\_EXPECTED variable contains the data that Hiperstation expects to receive from the partner application as a result of the most recently issued <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verb. The value represents the data contained in the script or the associated detail file. The *complete* expected value is placed in the variable, even if it exceeds 2048 bytes (in contrast to the APPC\_ACTUAL variable).

The variable contains a relevant value only when it immediately follows <CONTENT> tags associated with <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verbs, and its value is reset when the next script tag is encountered. The variable generates useful data only when it is set following APPC verbs that represent data received by Hiperstation. As a result, using the variable in a section of the script where Hiperstation is sending data will not produce meaningful data.

Figure 1-4 shows the APPC\_EXPECTED variable following a <CONTENT> tag associated with the <SEND\_DATA> verb.

**Figure 1-4.** Sample Script Containing the APPC\_EXPECTED Variable

```

<INBOUND>
  <SEND_DATA>
  <CONTENT>...00000002This is inbound data
  say appc_expected /* Displays 'This is inbound data'. */
  <PREPARE_TO_RECEIVE>
  say appc_expected /* A 0 length string is displayed. */
</INBOUND>

```

**APPC\_ACTUAL\_LENGTH**

The APPC\_ACTUAL\_LENGTH variable contains the length of the data most recently received from the partner application. This value can be obtained from normal data from a <SEND\_DATA> verb or from expedited data from a <SEND\_EXPEDITED\_DATA> verb. This variable contains the *exact* length of the data received from the partner, even if it exceeds 2048 bytes. Consequently, this value may be larger than the result of the REXX built-in: LENGTH(APPC\_ACTUAL).

The APPC\_ACTUAL\_LENGTH variable contains a relevant value only when it immediately follows <CONTENT> tags associated with <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verbs, and its value is reset when the next script tag is encountered. The variable generates useful data only when it is set following APPC verbs that represent data received by Hiperstation. As a result, using the variable in a section of the script where Hiperstation is sending data will not produce meaningful data.

Figure 1-5 shows the APPC\_ACTUAL\_LENGTH variable following a <CONTENT> tag associated with the <SEND\_DATA> verb.

**Figure 1-5.** Sample Script Containing the APPC\_ACTUAL\_LENGTH Variable

```

<INBOUND>8
  <SEND_DATA>
  <CONTENT>...00000002This is inbound data
  say appc_actual_length /* The length of the actual data (may be more than 2048). */
  <PREPARE_TO_RECEIVE>
  say appc_actual_length /* A 0 value is displayed.*/
</INBOUND>

```

**APPC\_EXPECTED\_LENGTH**

The APPC\_EXPECTED\_LENGTH variable contains the length of the data that Hiperstation expects to receive from the partner application. This can be normal data from a <SEND\_DATA> verb or expedited data from a <SEND\_EXPEDITED\_DATA> verb. The value is the length of the data contained in the script or associated detail file. The length does not include any of the Hiperstation header information that appears on <CONTENT> tags (the first 12 bytes).

This variable contains a relevant value only when it immediately follows <CONTENT> tags associated with <SEND\_DATA> or <SEND\_EXPEDITED\_DATA> verbs, and its value is reset when the next script tag is encountered. The variable generates useful data only when it is set after APPC verbs that represent data received by Hiperstation. As a result, using the variable in a section of the script where Hiperstation is sending data will not produce meaningful data.

Figure 1-6 shows the APPC\_EXPECTED\_LENGTH variable after a <CONTENT> tag associated with a <SEND\_DATA> verb.



**Figure 1-6.** Sample Script Containing the ACCP\_EXPECTED\_LENGTH Variable

```

<INBOUND>8
<SEND_DATA>
<CONTENT>...00000002This is inbound data
say appc_expected_length /* The length of data expected to receive (20 bytes). */
<PREPARE_TO_RECEIVE>
say appc_expected_length /* A 0 value is displayed. */
</INBOUND>

```

## APPC General REXX Variables

The following variables return general playback information.

### **CYCLE**

Returns the script repeat value. This value is set to one when a script begins execution.

### **GRPCYCLE**

Returns the repeat value of the ALLOCATE or APPCGRP statement. This value is set to one when an ALLOCATE or APPCGRP statement begins execution.

The variable is increased by one each time an ALLOCATE or APPCGRP statement finishes all of its scripts. You can use the REPEAT keyword on the ALLOCATE or APPCGRP statement in batch processing.

The GRPCYCLE variable provides a unique value during each iteration of an ALLOCATE or APPCGRP statement's execution.

### **PORT**

Returns a value listing the port assigned to each ALLOCATE or APPCGRP statement. The variable is increased by one for each ALLOCATE or APPCGRP statement. You can use the COUNT keyword on an ALLOCATE or APPCGRP statement in batch processing.

## TCP/IP REXX Variables

The following predefined REXX variables return information from TCP/IP playback jobs. Incorporate these variables into REXX logic that you add to your scripts to control the way they play back. For example, the REPLAY\_ID variable returns a value that you can use to drive dynamic data replacement.

**Note:** Understanding playback, especially the SOCKETS and SCRIPT statements, is integral to determining how to apply these variables. Additionally, do not forget to set the SET\_REXX\_STREAM\_VARIABLES playback control parameter to YES to enable the use of these variables during playback. Refer to the *Hiperstation for Mainframe Servers User Guide* for more information.

### **TCPIP\_ACTUAL**

Returns the content of the most recent message processed by Hiperstation. This variable's value is similar to the RequestBody or ResponseBody reporting fields, depending on whether the data came from the client or server.

### **TCPIP\_EXPECTED**

Returns the content of an inbound message that Hiperstation expects to receive. This is the server content from your script. This variable is similar to the ExpResponseBody reporting field.

**TCPIP\_ACTUAL\_LENGTH**

Returns the length of the most recent message processed by Hiperstation.

**TCPIP\_EXPECTED\_LENGTH**

Returns the length of an inbound message that Hiperstation expects to receive.

**Note:** To use the TCPIP\_ACTUAL, TCPIP\_EXPECTED, TCPIP\_ACTUAL\_LENGTH, AND TCPIP\_EXPECTED\_LENGTH variables, place them after the last server script tag that you want to reference. If you place these variables before the last server script tag, only information prior to the variables is returned from the playback.

**CONNECTION\_NBR**

Returns the current ID of the connection that is playing back. This value may be the same across multiple sockets since it is pulled from the script.

**REPLAY\_ID**

Returns a sequential value for each instance of a SOCKETS replay. For example, a SOCKETS statement with COUNT(2) REPEAT(3), replays two instances of a script simultaneously, repeated three times, for a total of six unique replay IDs. Use REPLAY\_ID in volume testing to provide a unique message for each connection to your application.

For example, to test 1000 simultaneous customer requests accessing your application, each with a different customer number and name:

1. Create an external file containing customer numbers and names with one customer per line.
2. Add data replacement logic to your script, for example:

```
/* Access external file and parse customer info */
"EXECIO * DISKR CUSTFILE (STEM cust.FINIS"
parse var cust.REPLAY_ID 1 cust_nbr 11 cust_name
/* Replace content in script with info from file */
<REPLACE FIELD =(0,cust_nbr) FIELD =(10,cust_name)>
/* Send the message */
<CLIENT...
```

3. Set the count value in the SOCKETS statement to 1000, that is, **SOCKETS COUNT(1000)**. This generates 1000 unique customer requests by starting 1000 simultaneous sessions, all with unique REPLAY\_IDs that correspond to the customer information from your external file.

**SOCKETS\_COUNT\_NBR**

Returns the COUNT value of the current SOCKETS replay. For example, if the SOCKETS statement specifies COUNT(3), the first instance returns COUNT = 1; the second, COUNT = 2; the third, COUNT = 3. This value is set when the SOCKETS replay begins.

Use this variable and SOCKETS\_REPEAT\_NBR to extend testing capability.

**SOCKETS\_REPEAT\_NBR**

Returns the REPEAT value of the given SOCKETS. This value is set when a socket begins execution. Use SOCKETS\_COUNT\_NBR and SOCKETS\_REPEAT\_NBR to replace script data

on specific COUNTS and REPEATS. For example, to test three products (A, B, C) with two accounts (1, 2):

Iteration	All COUNTS execute simultaneously		
	COUNT 1	COUNT 2	COUNT 3
Repeat 1	Product A, Account 1	Product B, Account 1	Product C, Account 1
Repeat 2 (begins after Repeat 1 completes)	Product A, Account 2	Product B, Account 2	Product C, Account 2

1. Create an external file for each parameter you want to replace. Each file should contain one parameter value per line.
2. Add data replacement logic to your script, for example:

```
/* Access external files */
"EXECIO * DISKR PRODFILE (STEM product.FINIS"
"EXECIO * DISKR ACCTFILE (STEM account.FINIS"
product_nbr = product.MQGROUP_COUNT_NBR
account_nbr = account.MQGROUP_REPEAT_NBR
/* Replace content in script with info from file */
<REPLACE FIELD =(5,account_nbr) FIELD =(32,product_nbr)>
/* Send the message */
<CLIENT...
```

3. Set the count and repeat values in the SOCKETS statement as follows: **SOCKETS COUNT(3),REPEAT(2)**. This generates six unique customer requests, each with a unique combination of product and account parameter values.

#### **SCRIPT\_REPEAT\_NBR**

Returns the REPEAT value of the current script playback. This value is set to one when the script begins execution and increments with every repeat.

#### **ACTUAL\_DB2\_USERID**

Returns the ID of the user associated with the DB2 transaction being played back. This value comes from the script and is sent to the server by playback.

#### **ACTUAL\_DB2\_RDBNAM**

Returns the name of the relational database associated with the DB2 transaction being played back. This value comes from the script and is sent to the server by playback.

#### **ACTUAL\_DB2\_SQLSTMT**

Returns the SQL Statement associated with the DB2 transaction being played back. This value comes from the script and is sent to the server by playback.

#### **ACTUAL\_DB2\_SQLSTATE**

Returns the value of the last SQL State returned by the application during playback. This is a live value that is received by playback from the server being tested.

#### **ACTUAL\_DB2\_SQLCODE**

Returns the value of the last SQL code returned by the application during playback. This is a live value that is received by playback from the server being tested.

#### **ACTUAL\_DB2\_ANSWER\_SET**

Returns the server's response to an SQL Query/Statement being played back. This is a live value that is received by playback from the server being tested.

***ACTUAL\_IMS\_CLIENT***

Returns the unique client identifier associated with the IMS transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_IMS\_TRANS***

Returns the IMS transaction code (target application) associated with the IMS transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_IMS\_DSTORE***

Returns the IMS Datastore associated with the IMS transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_IMS\_USERID***

Returns the ID of the user associated with the IMS transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_CTG\_APPLID***

Returns the APPLID of the VTAM application being accessed by the CTG transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_CTG\_PROGID***

Returns the name of the program being accessed by the CTG transaction being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_ECI\_USERID***

Returns the ID of the user associated with the ECI call being played back. This value comes from the script and is sent to the server by playback.

***ACTUAL\_ECI\_PROGID***

Returns the name of the program invoked in CICS by the ECI call being played back. This value comes from the script and is sent to the server by playback.

## MQ REXX Variables

The following predefined REXX variables return information from MQ playback jobs. Incorporate these variables into REXX logic that you add to your scripts to control the way they play back. For example, the `REPLAY_ID` variable returns a value that you can use to drive dynamic data replacement logic.

**Note:** Understanding playback, especially the `MQGROUP` and `SCRIPT` statements, is integral to determining how to apply these variables. Additionally, do not forget to set the `REXONN` and `SET_REXX_STREAM_VARIABLES` playback control parameters to enable the use of these variables during playback. Refer to the *Hiperstation for WebSphere MQ User Guide* for more information.

***ACTUAL\_COMPCODE***

Returns the completion code (`COMPCODE`) of an `MQ_PUT`, `MQ_PUT1`, or `MQ_GET` executed during playback. Place this variable after each script tag that executes a `GET`, `PUT`, or `PUT1` of interest. For example, to report the completion code for a particular `GET` and `PUT`, incorporate this variable into REXX `SAY` statements placed after each of the applicable script tags. For example:

```

<MQ_GET,
CONNECTION_ID = 2,
OBJECT_ID = 1,
COMPCODE = 0,
REASON = 0,
MQMD_VERSION = 1,
...>

say 'Comp code of GET = 'ACTUAL_COMPCODE'

<MQ_PUT
...>

say 'Comp code of PUT = 'ACTUAL_COMPCODE'

```

**EXPECTED\_COMPCODE**

Returns the completion code (COMPCODE) that was recorded in the script for an MQ\_PUT, MQ\_PUT1, or MQ\_GET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest. For example, to see the actual and expected completion codes for a particular GET, insert the following REXX SAY statements after the MQ\_GET tag that executes the call:

```

<MQ_GET,
...>

say 'Comp code of GET = 'ACTUAL_COMPCODE'
say 'Expected comp code of GET = 'EXPECTED_COMPCODE'

```

**ACTUAL\_CORRELID**

Returns the correlation ID (MQMD\_CORRELID), in character format, from an MQ\_PUT, MQ\_PUT1, or MQ\_GET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest. If the transaction does not complete successfully, this variable returns 'NULL'. This variable is used in the example provided in "Modify an MQ Script to Test the Requester" on page 1-40.

**EXPECTED\_CORRELID**

Returns the correlation ID (MQMD\_CORRELID) from the script, in character format, for an MQ\_PUT, MQ\_PUT1, or MQ\_GET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest.

**ACTUAL\_MSG**

Returns the content of the most recent message processed by Hiperstation. This variable's value is similar to the MQMessageOutbound or MQMessageInbound reporting fields depending on whether an MQ\_GET or MQ\_PUT was processed. Refer to the *Hiperstation for WebSphere MQ User Guide* for detailed testing scenarios that illustrate the use of this variable.

**EXPECTED\_MSG**

Returns the content of an inbound MQ message that Hiperstation is expecting to receive. Normally, this is the content of an MQ\_GET from your script. However, if you reverse the direction of playback, it is the content from an MQ\_PUT.

**ACTUAL\_MSGID**

Returns the message ID (MQMD\_MSGID) from the script, in character format, for an MQPUT, MQPUT1, or MQGET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest. If the transaction does not complete successfully, this variable returns 'NULL'.

**EXPECTED\_MSGID**

Returns the message ID (MQMD\_MSGID) from the script, in character format, for an MQPUT, MQPUT1, or MQGET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest.

**ACTUAL\_LENGTH**

Returns the length of the most recent message processed by Hiperstation.

**EXPECTED\_LENGTH**

Returns the length of an inbound MQ message that Hiperstation expects to receive.

**ACTUAL\_REASON**

Returns the reason code (REASON) from an MQPUT, MQPUT1, or MQGET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest. This variable is used in the example provided in “Modify an MQ Script to Test the Requester” on page 1-40.

**EXPECTED\_REASON**

Returns the reason code (REASON) from the script for an MQPUT, MQPUT1, or MQGET executed during playback. Place this variable after each script tag that executes a GET, PUT, or PUT1 of interest.

**OBJECT\_ID**

Returns the current ID of the object (queue) associated with the activity that is playing back. This variable's value may be the same across multiple MQGROUPs since it is pulled from the script.

**REPLAY\_ID**

Returns a sequential value for each instance of an MQGROUP replay. For example, an MQGROUP statement with COUNT(2) REPEAT(3), replays two instances of a script simultaneously, repeated three times, for a total of six unique replay IDs. Use REPLAY\_ID in volume testing to provide a unique message for each connection to your application.

For example, to test 1000 simultaneous customer requests accessing your application, each with a different customer number and name:

1. Create an external file containing customer numbers and names with one customer per line.
2. Add data replacement logic to your script, for example:

```
/* Access external file and parse customer info */
"EXECIO * DISKR CUSTFILE (STEM cust.FINIS"
parse var cust.REPLAY_ID 1 cust_nbr 11 cust_name

/* Replace content in script with info from file */
<REPLACE FIELD =(0,cust_nbr) FIELD =(10,cust_name)>

/* Send the message */
<MQ_PUT...
```

3. Set the count value in the MQGROUP statement to 1000, that is, **MQGROUP COUNT(1000)**. This generates 1000 unique customer requests by starting 1000 simultaneous sessions, all with unique REPLAY\_IDs that correspond to the customer information from your external file.

**PUT\_QUEUE\_NAME**

Returns the name of the queue most recently used for an MQ\_PUT.

**GET\_QUEUE\_NAME**

Returns the name of the queue most recently used for an MQ\_GET.

**MQGROUP\_COUNT\_NBR**

Returns the COUNT value of the current MQGROUP replay. For example, if the MQGROUP statement specifies COUNT(3), the first instance returns COUNT = 1; the second, COUNT = 2; the third, COUNT = 3. This value is set when the MQGROUP begins execution.

Use this variable, and MQGROUP\_REPEAT\_NBR, to extend testing capability. Refer to the *Hiperstation for WebSphere MQ User Guide* for detailed testing scenarios that illustrate the use of this variable.

**MQGROUP\_REPEAT\_NBR**

Returns the REPEAT value of the MQGROUP replay. This value is set when a group begins execution. Use MQGROUP\_COUNT\_NBR and MQGROUP\_REPEAT\_NBR to replace script data on specific COUNTS and REPEATS. For example, to test three products (A, B, C) with two accounts (1, 2):

Value	All COUNTS execute simultaneously		
	COUNT 1	COUNT 2	COUNT 3
Repeat 1	Product A, Account 1	Product B, Account 1	Product C, Account 1
Repeat 2 (begins after Repeat 1 completes)	Product A, Account 2	Product B, Account 2	Product C, Account 2

1. Create an external file for each parameter you want to replace. Each file should contain one parameter value per line.
2. Add data replacement logic to your script, for example:

```

/* Access external files */
"EXECIO * DISKR PRODFILE (STEM product.FINIS"
"EXECIO * DISKR ACCTFILE (STEM account.FINIS"
product_nbr = product.MQGROUP_COUNT_NBR
account_nbr = account.MQGROUP_REPEAT_NBR
/* Replace content in script with info from file */
<REPLACE FIELD =(5,account_nbr) FIELD =(32,product_nbr)>
/* Send the message */
<MQ_PUT...
    
```

3. Set the count and repeat values in the MQGROUP statement, for example, **MQGROUP COUNT(3),REPEAT(2)**. This generates six unique customer requests, each with a unique combination of product and account parameter values.

Refer to the *Hiperstation for WebSphere MQ User Guide* for detailed testing scenarios that illustrate the use of this variable.

**SCRIPT\_REPEAT\_NBR**

Returns the REPEAT value of the current script playback. This value is set to one when the script begins execution, and increments with every repeat.

## Using REXX Variables

### Use REXX Variables as Input

One of the basic features of using REXX with Hiperstation is to replace recorded inputs with variable names to dynamically replace input data with unique values during playback.

**Note:** When using REXX variables to substitute 3270 or APPC input data, the length of the REXX variable's value cannot exceed 256 bytes.

The following example replaces the employee number field with a variable named `employee_number`. You can also add a loop to increment `employee_number` by 1 on each iteration of the loop.

#### Original

```
<INPUT>0000006
<THINK>00.07.907 AT 00:00:21.014
<KEY>ENTER
<CURSOR>01,07
<I01>"add"
<I02>"11111"
</INPUT>
```

#### Modified

```
<INPUT>0000006
<THINK>00.07.907 AT 00:00:21.014
<KEY>ENTER
<CURSOR>01,07
<I01>"add"
<I02>employee_number
</INPUT>
```

It is possible with unattended playback to run one script across multiple terminals. Using variable substitution, you can enter unique input data across the different terminals. For example, you can play back a single script that adds an employee number to the database, while simulating multiple terminals with each terminal entering a unique employee number to the application. To do this, insert a compound variable using `PORT` as the compound variable extension. In this example, the unattended playback controls are `GROUP H01AC054 TERM(4) SUMMARY (*) SCRIPT (TEST1)`.

```
employee_number.1 = '11111'
employee_number.2 = '22222'
employee_number.3 = '33333'
employee_number.4 = '44444'
<INPUT>
<KEY>ENTER
<I01>employee_number.port
</INPUT>
```

### Perform File I/O from a Script

Reading a file from a Hiperstation script is done the same way as reading a file in a stand-alone REXX program (see "Retrieving Data from External Sources" on page 1-6). Before you read or write to a file, the file must be allocated. `ALLOCATE` is a TSO command, so you cannot invoke it from REXX in a Hiperstation script. Hiperstation does provide its own `ALLOCATE` command. You can use it by invoking `ADDRESS HSCMDS`.



**Note:** You must specify the fully qualified dataset name to be allocated, without using quotation marks, which is a slight difference from how you use TSO's ALLOCATE command.

**Figure 1-7.** External File Contents

```

File Edit Confirm Menu Utilities Compilers Test Help
-----
EDIT          USERID1.TRAINING.SCRIPT.INPUT          Columns 00001 00072
Command ==>                                     Scroll ==> CSR
***** ***** Top of Data *****

000100 18751 Covell Ryan 050.00 040.00
000200 23452 Woods Tiger 045.00 900.00
000300 56781 Couples Fred 020.00 840.00
000400 16531 Norman Greg 023.00 560.00
000500 86791 Watson Tom 042.00 480.00
***** ***** Bottom of Data *****

```

Figure 1-8. Script Contents

```

<VERSION>7
<INPUT>0000002
<KEY>ENTER
</INPUT>
<INPUT>0000003
<KEY>ENTER
<I01>"userab0"
<N03>"n -Vb1"
</INPUT>
<INPUT>0000004
<KEY>CLEAR
</INPUT>

ADDRESS HSCMDS"ALLOC F(MYINDD) DA(USERAB0.TRAINING.SCRIPT.INPUT) SHR"
"EXECIO * DISKR MYINDD (STEM input_rec. FINIS "
ADDRESS HSCMDS"FREE F(MYINDD)"

DO ptr = 1 to input_rec.0
  PARSE VAR input_rec.ptr ,
           employee_number.ptr ,
           last_name.ptr ,
           first_name.ptr ,
           hours_worked.ptr ,
           hourly_rate.ptr
  employee_name.ptr = first_name.ptr||' '||last_name.ptr
  SAY employee_number.ptr
  SAY employee_name.ptr
  SAY hours_worked.ptr
  SAY hourly_rate.ptr
END

<INPUT>0000005
<KEY>ENTER
<I01>"plaf"
</INPUT>
DO x = 1 TO input_rec.0
  <INPUT>0000006
  <KEY>ENTER
  <I01>"add"
  <I02>employee_number.x
  </INPUT>
  <INPUT>0000007
  <KEY>ENTER
  <I01>employee_name.x
  <I02>hours_worked.x
  <I03>hourly_rate.x
  </INPUT>
  <INPUT>0000014
  <KEY>ENTER
  <I01>"del"
  <I02>employee_number.x
  </INPUT>
  <INPUT>0000015
  <KEY>ENTER
  </INPUT>
END

<INPUT>0000016
<KEY>CLEAR
</INPUT>
<INPUT>0000017
<KEY>ENTER
<I01>"cesf logoff"
</INPUT>

```

## Read Data from VSAM-GETVSAMR

The command,

```
ADDRESS HSCMDS "GETVSAMR"
```

retrieves a VSAM record. Four REXX variables are used for this command:

- HS\_VSAM\_DD
- HS\_VSAM\_KEY
- HS\_VSAM\_RC
- HS\_VSAM\_REC

Before issuing this command, set HS\_VSAM\_DD with the allocated DD name of the VSAM dataset and HS\_VSAM\_KEY with the key. GETVSAMR sets HS\_VSAM\_RC as follows:

```

/*****/
/*HS_VSAM_RC :Notes */
/*0 :Successful */
/*4 :No record found */
/*8 :HS_VSAM_DD problem */
/*12 :HS_VSAM_KEY problem */
/*14 :HS_VSAM_REC problem */
/*16 :HS_VSAM_DD OPEN error */
/*24 :Not licensed */
/*****/

```

When HS\_VSAM\_RC is zero, HS\_VSAM\_REC will contain the VSAM record that matches the specified HS\_VSAM\_key.

## Analyze Application Screens

You can code REXX in the script to interrogate screen contents. This information can then be used to make IF/THEN/ELSE decisions, to handle synchronization issues, and as input later in the script (as in data replacement).

Use commands such as POS and SUBSTR against the SCREEN variable to analyze data displayed on the screen. Figure 1-9 on page 1-28 shows an example of this analysis.

## Use Logic to Control Script Flow

You can use REXX to control playback. Use conditional logic such as IF/THEN/ELSE structures, DO loops, and SELECT structures to do this.

You can wrap DO/END statements around portions of the script. Figure 1-8 shows a DO loop wrapped around the section of the script that adds and deletes the employee information to the application. See Figure 1-9 on page 1-28 for additional examples.

## Use the REXX Log

The REXX log allows you to create custom reports to document what happens during script execution.

The REXX log creates a file that contains the results of any SAY statements executed from the script.

To create a REXX log during playback via Hiperstation for VTAM's Domain Traveler, enter the RLOG command on the command line. To create a REXX log during an unattended playback, specify RLOG(??) on the GROUP statement.

Figure 1-9 through Figure 1-14 shows a complete 3270 script containing REXX logic that:

- Sends the Enter command (up to 100 times) until the system returns a job complete return code
- and
- Reports the return code to the REXX log

**Figure 1-9.** Examples of Analysis, Using Logic, and Using the REXX Log (1 of 6)

```

<VERSION>7

/* Let's check the tpf before starting with the script execution */
IF tpfname <> 'TSO' THEN
  DO
    SAY TIME() 'Oops. wrong tpf. Playback aborted'
    EXIT
  END

SAY TIME() 'The script execution has started'
<IMSUNLOK>N
<OUTPUT>0000001
<RESPONSE>01.27.709
<S01> IKJ56700A ENTER USERID -
<S02>
<S03>
<S04>
<S05>
<S06>
<S07>
<S08>
<S09>
<S10>
<S11>
<S12>
<S13>
<S14>
<S15>
<S16>
<S17>
<S18>
<S19>
<S20>
<S21>
<S22>
<S23>
<S24>
</OUTPUT>
<BASETIME>05/14/07 21:17:24.394
<INPUT>0000002
<THINK>00.04.985 AT 00:00:00.000
<KEY>ENTER
<CURSOR>01,07
<I02>"userab1"
</INPUT>
<OUTPUT>0000002
<RESPONSE>00.00.053
<S01> ----- TSO/E LOGON -----
<S02>
<S03>
<S04> Enter LOGON parameters below: RACF LOGON para
<S05>
<S06> Userid ==> USERAB1
<MSK> NNNNNNNNNNNNNNNN
<S07>
<S08> Password ==>
<MSK> NNNNNNNN New Password ==
<S09>
<S10> Procedure ==> AISP41 Group Ident ==
<S11>
<S12> Acct Nmbr ==> OVPBAS5.1.0MNT
<S13>
<S14> Size ==> 4096
<S15>
<S16> Perform ==>
<S17>
<S18> Command ==>
<S19>
<S20> Enter an 'S' before each option desired below:
<S21> -Nomail -Nonotice S -Reconnect -OID
<S22>
<S23> PF1/PF13 ==> Help PF3/PF15 ==> Logoff PA1 ==> Attention P
<S24> You may request specific help information by entering a '?' in any
</OUTPUT>

```

Figure 1-10. Examples of Analysis, Using Logic, and Using the REXX Log (2 of 6)

```

<INPUT>0000003
<THINK>00.03.102 AT 00:00:03.412
<KEY>ENTER
<CURSOR>07,25
<NO1>"n -Vb1"
</INPUT>
<OUTPUT>0000010
<RESPONSE>00.07.775
<S01> ICH70001I USERAB1 LAST ACCESS AT 21:15:28 ON MONDAY, MAY 14, 2007
<S02> USERAB1 LOGON IN PROGRESS AT 21:17:28 ON MAY 14, 2007
<S03> NO BROADCAST MESSAGES
<S04> *****
<S05> *
<S06> * Compuware Mainframe System News (restricted to authorized user
<S07> *****
<S08> * This system is now running OS/390 V1R2M0
<S09> * Due to the nature of Server Pack operating system installation,
<S10> * many SYSRES/DLIB dataset names have changed. Please reference
<S11> * SYS2.SW.DOCLIB(NEWDSNS) for detailed information on OLD vs NEW D
<S12> * name changes. Please direct any questions to the help desk.
<S13> *****
<S14> * Attention Boole CICS Manager Users: Change all references of
<S15> * SYS2A.BOOLE.** to SYS2.BOOLE.** in your in your CICS region JCL
<S16> *****
<S17> * Attention VSE Users: VSE migration to the P/390 is scheduled fo
<S18> * May 24-25. Reference SYS2.SW.DOCLIB(VSEMIGR) for details.
<S19> *****
<S20> ***
<S21>
<S22>
<S23>
<S24>
</OUTPUT>
<INPUT>0000011
<THINK>00.01.328 AT 00:00:12.782
<KEY>ENTER
<CURSOR>19,05
</INPUT>
<OUTPUT>0000011
<RESPONSE>00.00.384
<S01> Menu Utilities Compilers Options Status Help
<S02> -----
<S03> ISPF Primary Option Menu
<S04> Option ==>
<S05>
<S06> 0 Settings Terminal and user parameters User ID .
<S07> 1 View Display source data or listings Time. . .
<S08> 2 Edit Create or change source data Terminal.
<S09> 3 Utilities Perform utility functions Screen. .
<S10> 4 Foreground Interactive language processing Language.
<S11> 5 Batch Submit job for language processing Appl ID .
<S12> 6 Command Enter TSO or Workstation commands TSO logon
<S13> 7 Dialog Test Perform dialog testing TSO prefi
<S14> 8 LM Facility Library administrator functions System ID
<S15> 9 IBM Products IBM program development products MVS acct.
<S16> 10 SCLM SW Configuration Library Manager Release .
<S17> D DB2 & QMF DB2I/QMF access to DB2 subsystems
<S18> M MIS MIS Dialogs
<S19> P Products COMPUWARE Product Menu
<S20> S SDSF System Display and Search Facility
<S21> U User User Dialogs
<S22>
<S23>
<S24> Enter X to Terminate using log/list defaults
</OUTPUT>

```

**Figure 1-11.** Examples of Analysis, Using Logic, and Using the REXX Log (3 of 6)

```

<INPUT>0000012
<THINK>00.05.324 AT 00:00:18.760
<KEY>ENTER
<CURSOR>03,16
<I01>"3.4"
</INPUT>
<OUTPUT>0000012
<RESPONSE>00.00.430
<S01>  Menu  RefList  RefMode  Utilities  Help
<S02> -----
<S03>                                     Data Set List Utility
<S04> Command ==>
<S05>
<S06>      blank Display data set list          P Print data set list
<S07>      V Display VTOC information          PV Print VTOC information
<S08>
<S09> Enter one or both of the parameters below:
<S10>  Dsname Level . . . USERAB1.LOGON.CLIST
<S11>  Volume serial . .
<S12>
<S13> Data set list options:
<S14>  Initial View . . . 1 1. Volume          Enter "/" to select o
<S15>                                     2. Space          / Confirm Delete
<S16>                                     3. Attrib
<S17>                                     4. Total
<S18>
<S19> The following actions will be available when the list is displayed
<S20>  Enter a "/" on the data set list command field for command prom
<S21>  Enter TSO commands, CLIST, REXX execs, or "=" to execute previo
<S22>
<S23>
<S24>
</OUTPUT>
<INPUT>0000013
<THINK>00.05.105 AT 00:00:24.573
<KEY>ENTER
<CURSOR>09,44
<I02>"userab0.hiper.scripts"
</INPUT>
<OUTPUT>0000013
<RESPONSE>00.00.263
<S01>  Menu  Functions  Sort  Confirm  View  Utilities  Compilers  Help
<S02> -----
<S03> DSLIST - Data Sets Matching USERAB0.HIPER.SCRIPTS
<S04> Command ==>                               Scro
<S05>
<S06> Command - Enter "/" to select action          Message
<S07> -----
<S08>          USERAB0.HIPER.SCRIPTS
<S09> ***** End of Data Set list *****
<S10>
<S11>
<S12>
<S13>
<S14>
<S15>
<S16>
<S17>
<S18>
<S19>
<S20>
<S21>
<S22>
<S23>
<S24>
</OUTPUT>

```

Figure 1-12. Examples of Analysis, Using Logic, and Using the REXX Log (4 of 6)

```

<INPUT>0000014
<THINK>00.02.431 AT 00:00:27.527
<KEY>ENTER
<CURSOR>07,02
<I03>"e          USERABO.HIPER.SCRIPTS          "
</INPUT>
<OUTPUT>0000014
<RESPONSE>00.00.736
<S01>  Menu Functions Utilities Help
<S02> -----
<S03> EDIT          USERABO.HIPER.SCRIPTS          Row 00
<S04> Command ==>          Scro
<S05>  Name          VV MM Created      Changed      Size  Init
<S06> . ABC00000      00.00 07/04/22 07/04/22 15:58   319   10
<S07> . ASIA1         01.01 06/03/13 06/03/13 17:42   168  168
<S08> . ASIA2         00.00 06/03/13 06/03/13 17:16   307   10
<S09> . ASIA2         00.00 06/03/13 06/03/13 18:05   472   15
<S10> . A1            01.01 06/08/26 06/08/26 16:50  1089 1089
<S11> . BATCH         01.97 05/03/10 07/05/12 21:16    16   16
<S12> . BATCHPB       01.00 07/05/12 07/05/12 21:17    16   16
<S13> . BATCH1        01.02 06/04/02 06/04/02 14:16    33   33
<S14> . BATCH2        01.81 04/07/14 06/03/26 10:23    21   16
<S15> . BATCH3        01.01 06/03/26 06/06/10 16:38    15   21
<S16> . BATCH4        01.00 06/03/26 06/03/26 10:21    21   21
<S17> . BBB           00.10 06/08/08 06/08/15 10:13   160    9
<S18> . BBBIN         01.01 06/08/08 06/08/14 17:03     4    4
<S19> . CALLER        01.10 06/07/25 06/07/25 10:50     4    4
<S20> . CHECK2        01.05 06/10/01 06/10/02 09:13    10   14
<S21> . CICS LIST     00.00 06/10/29 06/10/29 17:07   647   17
<S22> . CICS LIST     00.00 06/10/29 06/10/29 17:10   538   14
<S23> . CICS OFF      00.00 06/08/08 06/08/08 14:19   103    3
<S24> . CICS ON       00.05 06/08/08 06/09/05 14:41   146    5
</OUTPUT>
<INPUT>0000015
<THINK>00.04.650 AT 00:00:33.172
<KEY>ENTER
<CURSOR>03,23
<I01>"s batchpb"
</INPUT>
<OUTPUT>0000015
<RESPONSE>00.00.370
<S01>  File Edit Confirm Menu Utilities Compilers Test Help
<S02> -----
<S03> EDIT          USERABO.HIPER.SCRIPTS(BATCHPB) - 01.00          Columns
<S04> Command ==>          Scro
<S05> ***** Top of Data *****
<S06> ==MSG> -CAUTION- Profile changed to CAPS ON (from CAPS OFF) becaus
<S07> ==MSG>          data does not contain any lower case characters.
<S08> 000001 //USERABOC JOB ('OVPBAS5.0.ODSP',P87,9,9),'RYAN COVELL',REG
<S09> 000002 //          NOTIFY=USERABO,CLASS=2,MSGCLASS=R
<S10> 000003 //*****
<S11> 000004 //*
<S12> 000005 //*          SAMPLE JCL TO EXECUTE Hiperstation
<S13> 000006 //*          IN UNATTENDED MODE
<S14> 000007 //*
<S15> 000008 //*****
<S16> 000009 //HIPER2 EXEC PGM=EHSBATCH,REGION=4096K
<S17> 000010 //STEPLIB DD DISP=SHR,DSN=SYS2.HPER.V531.LOAD
<S18> 000011 //SYSLIB DD DSN=USERABO.REVENUE.SCRIPTS,DISP=SHR
<S19> 000012 //SYSPRINT DD SYSOUT=*
<S20> 000013 //SYSUDUMP DD SYSOUT=*
<S21> 000014 //SYSIN DD *
<S22> 000015 GROUP H01AC054 TERM(5) SUMMARY(*) XLOG(*)
<S23> 000016 SCRIPT(TEST1)
<S24> ***** Bottom of Data *****
</OUTPUT>
<INPUT>0000016
<THINK>00.16.845 AT 00:00:50.644
<KEY>ENTER
<CURSOR>03,17
<I01>"sub"
</INPUT>

```

**Figure 1-13.** Examples of Analysis, Using Logic, and Using the REXX Log (5 of 6)

```

<OUTPUT>0000019
<RESPONSE>00.01.182
<S01> File Edit Confirm Menu Utilities Compilers Test Help
<S02> -----
<S03> EDIT USERAB0.HIPER.SCRIPTS(BATCHPB) - 01.00 Column
<S04> Command ==> sub Scr
<S05> ***** ***** Top of Data *****
<S06> ==MSG> -CAUTION- Profile changed to CAPS ON (from CAPS OFF) becau
<S07> ==MSG> data does not contain any lower case characters.
<S08> 000001 //USERAB1C JOB ('OVPBAS5.0.ODSP',P87,9,9),'RYAN COVELL',RE
<S09> 000002 // NOTIFY=USERAB1,CLASS=2,MSGCLASS=R
<S10> 000003 //*****
<S11> 000004 //*
<S12> 000005 //* SAMPLE JCL TO EXECUTE Hiperstation
<S13> 000006 //* IN UNATTENDED MODE
<S14> 000007 //*
<S15> 000008 //*****
<S16> 000009 //HIPER2 EXEC PGM=EHSBATCH,REGION=4096K
<S17> 000010 //STEPLIB DD DISP=SHR,DSN=SYS2.HPER.V531.LOAD
<S18> 000011 //SYSLIB DD DSN=USERAB0.REVENUE.SCRIPTS,DISP=SHR
<S19> 000012 //SYSPPRINT DD SYSOUT=*
<S20> 000013 //SYSUDUMP DD SYSOUT=*
<S21> JOB USERAB1C(JOB23067) SUBMITTED
<S22> ***
<S23>
<S24>
</OUTPUT>

/* Press enter until the 'JOB SUBMITTED' message is displayed */
DO loop_ctr = 1 TO 100
  x = POS('SUBMITTED',screen)
  IF x > 0 THEN
    LEAVE
  <INPUT>0000020
<KEY>ENTER
</INPUT>
END
IF loop_ctr >= 100 THEN
  DO
    SAY TIME() 'Job did not successfully submit. Playback aborted.'
  EXIT
END

/* Grab the JOBID off from the screen */
job_id = SUBSTR(screen,x - 7,5)

/* Display a message */
SAY TIME() 'Job successfully submitted. The JOBID is 'job_id

/* Now loop until the job completes */
DO loop_ctr = 1 TO 100
  IF POS('USERAB1C ENDED',screen) > 0 THEN
    LEAVE
  <WAIT>00.10
  <INPUT>0000020
  <KEY>ENTER
</INPUT>
END
IF loop_ctr = 100 THEN
  DO
    SAY TIME() 'Job did not seem to end. Playback aborted.'
  EXIT
END

/* Display the return code status */
return_code_position = POS('MAX COND',screen)
return_code_status = SUBSTR(screen,return_code_position,19)
SAY TIME() 'The batch job ended, status is 'return_code_status

```



**Figure 1-14.** Examples of Analysis, Using Logic, and Using the REXX Log (6 of 6)

```

<OUTPUT>0000028
<RESPONSE>00.00.024
<S01> 21.18.29 JOB23067 $HASP165 USERAB1C ENDED AT CW01 - MAX COND CODE
<S02>NAL)
<S03>
<S04>
<S05>
<S06>
<S07>
<S08>
<S09>
<S10>
<S11>
<S12>
<S13>
<S14>
<S15>
<S16>
<S17>
<S18>
<S19>
<S20>
<S21>
<S22>
<S23>
<S24>
</OUTPUT>

```

The above script actually continues to go the JES log to check the output messages.

The contents of the REXX log created by the above script is displayed below.

```

Menu Utilities Compilers Help
-----
BROWSE      USERAB0.TRAINING.REXX.LOG          Line 00000000 Col 001 080
Command ==>                               Scroll ==> CSR
***** Top of Data *****
13:33:08 The script execution has started
13:33:31 Job successfully submitted. The JOBID is 27313
13:33:53 The batch job ended, status is MAX COND CODE 0000
13:34:15 The script execution has completed. Bye.
***** Bottom of Data *****

```

---

## Building Hiperstation Script Statements Using <EXECUTE>

Hiperstation allows you to build script statements (<Inn>, <KEY>, <Snn>, <HEX>, etc.) for immediate execution using the <EXECUTE> statement. The data after the <EXECUTE> statement can contain a mixture of constants and variables as outlined under variable substitution.

**Note:** Hiperstation allows you to use the <EXECUTE> tag to build script statements but not REXX statements. The REXX programming language includes the INTERPRET statement to provide the same capability for REXX commands, such as CALL, PARSE, DO, and ADDRESS.

The most common use of the <EXECUTE> statement is to vary the number of inputs entered on a screen. For example, an order entry application has a screen where as many as 25 items can be entered per order. The order numbers and item numbers are in an external file. Each statement has an order number and item number. On the screen, order number is input field 1, item numbers are input fields 2 through 26.

```

"EXECIO * DISKR ORDERS (STEM INPUT. FINIS"
DO K=1 TO INPUT.0
  PARSE VAR INPUT.K ORDER.K ITEM.K
END
DO J=1 TO INPUT.0
<INPUT>
<KEY>ENTER
<IO1>ORDER.J
DO K=2 to INPUT.0
<EXECUTE>"<IO"&K&"||K||">"ITEM.K /*Builds <IOn>ITEM.K*/
END
</INPUT>
END

```

You can also use the <EXECUTE> tag to replace other Hiperstation script pieces with variables. For example, to use REXX to alter which key should be pressed on an input message, you can use the <EXECUTE> tag.

```

IF error = 'YES' THEN
  input_key = 'PF3'
ELSE
  input_key = 'ENTER'
<INPUT>
<EXECUTE>"<KEY>"||input_key
</INPUT>

```

You can also use <EXECUTE> to decide which input field number to use.

```

IF command_line_location = 'AT THE TOP' THEN
  field_number = '01'
ELSE
  field_number = '15'
<INPUT>
<KEY>ENTER
<EXECUTE>"<I"&field_number&"||'">"submit"
</INPUT>

```

---

## Altering Hiperstation Script Statements Using <ALTER> Tags

Hiperstation allows you to modify script statements during playback using the <ALTER> statement. Data after the <ALTER> statement can contain a mixture of constants and variables as discussed in "Terminal Input Substitution" on page 1-7. During playback, the resultant string (after variable substitution) completely replaces the script record that follows the <ALTER> statement.

The most common use of the <ALTER> statement is to alter a screen row. For example, an order entry application has a screen with a date field. During playback, a user wants to replace the date in the date field with a new date. Following is sample REXX code to do this:

```

IF y2k = 'Y' THEN
  new_row = "<S02> Hire Date: 10/12/2000" ELSE
  new_row = "<S02> Hire Date: 10/12/1997"

<OUTPUT>
<S01>
<ALTER>new_row
<S02> Hire Date: 10/12/1997
<S03>
</OUTPUT>

```

---

## Using the <ADDCALL> and <DELCALL> Tags

The <ADDCALL> tag adds a call to a REXX EXEC before each subsequent <INPUT> section. For example, to call a routine named CHECKER that takes an argument — SCREEN — use:

```
</OUTPUT>
<ADDCALL>CHECKER SCREEN.
<INPUT>
```

<ADDCALL> is powerful. You can use it to automatically invoke a routine that checks every screen for items such as error messages or if a compare mismatch occurred. The routine invoked can then take an action such as stopping playback.

**Note:** There can be only one active EXEC at a time. If another <ADDCALL> is done before a <DELCALL>, the routine it specifies replaces the first <ADDCALL>.

The <DELCALL> tag removes a REXX EXEC added with <ADDCALL>. Basically, it turns off the <ADDCALL>.

---

## Hiperstation HSCMDS Command Environment

Hiperstation has its own command environment for processing commands not found in the non-TSO/E REXX environment. These additional commands add to the functionality of the script language.

### HSCMDS Commands

The commands available in the HSCMDS environment are:

- **ALLOC** — Allocates a dataset for use by the EXECIO statement.
- **FREE** — Frees an allocated dataset.
- **GETVSAMR** — Retrieves a VSAM record.

These commands are run using the REXX ADDRESS statement. The ADDRESS statement lists the environment to which the command is submitted and the command itself.

```
►—ADDRESS—HSCMDS—"command"—►
      |environment|
```

#### environment

Indicates the name of the environment to which the command is submitted. For Hiperstation, this is always HSCMDS.

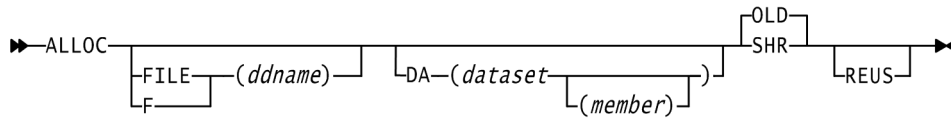
#### command

The command string. Enclose in double quotes except for those portions that contain REXX variables.

For example, run the Hiperstation FREE command to free a previously allocated file. The DD name of the file to free is in the REXX variable ddname.

```
ADDRESS HSCMDS "FREE F(ddname)"
```

## ALLOC



### ddname

The DD name to which the dataset is allocated.

### dataset

The fully qualified dataset name. You can enter a member name at the end of the dataset name by enclosing the member name in parentheses. Do not enclose the dataset name in quotes.

### SHR|OLD

The disposition of the dataset. OLD is the default.

### REUS

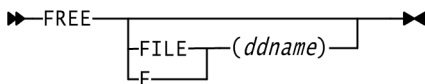
An existing allocation to the DD name is freed.

If the execution of the ALLOC command issues a -1 REXX return code (RC), the true return code can be found in variable HSCMDRC. The value in HSCMDRC is the return code from dynamic allocation. See "HSCMDS ALLOC Return Codes" on page 1-37 for information.

For example, member EMPNO in dataset USER1.RECORDS is to be allocated to DD name FILETEST. Allocate the dataset as shared and free any existing allocation.

```
ADDRESS HSCMDS "ALLOC F(FILETEST) DA(USER1.RECORDS(EMPNO)) SHR REUS"
```

## Free



### ddname

The DD name of the allocated dataset to be freed. For example:

```
ADDRESS HSCMDS "FREE F(FILETEST)"
```

## GETVSAMR



```
ADDRESS HSCMDS "GETVSAMR"
```

This command retrieves a VSAM record. Four REXX variables are used for this command:

- HS\_VSAM\_DD
- HS\_VSAM\_KEY
- HS\_VSAM\_RC
- HS\_VSAM\_REC

Before issuing this command, HS\_VSAM\_DD must be set with the allocated DD name of the VSAM dataset and HS\_VSAM\_KEY must be set with the key. GETVSAMR sets HS\_VSAM\_RC as follows.

```

/*****
/* HS_VSAM_RC : Notes
/* 0 : Successful
/* 4 : No record found
/* 8 : HS_VSAM_DD problem
/* 12 : HS_VSAM_KEY problem
/* 14 : HS_VSAM_REC problem
/* 16 : HS_VSAM_DD OPEN error
/* 24 : Not licensed
*****/

```

When HS\_VSAM\_RC is zero, HS\_VSAM\_REC will contain the VSAM record that matches the one specified in HS\_VSAM\_KEY.

## HSCMDS ALLOC Return Codes

REXX sets a special variable (RC) that contains the return code from the environment command processor. For the HSCMDS environment, the values for RC and their descriptions are as follows:

**Table 1-4.** HSCMDS ALLOC Return Codes

RC	Description
-3	The command was not found.
-2	The command contained an invalid parameter.
-1	Execution of the command was not successful.
0	Execution of the command was successful.

If RC is negative, then an ERROR condition can be trapped in the script or EXEC using the SIGNAL ON ERROR or CALL ON ERROR statements. For more information on these statements, see the *TSO/E REXX Reference Manual*.

Hiperstation sets two variables to aid in problem determination for HSCMDS commands. The variable, HSCMD, is set to the name of the last command entered; and the variable, HSCMDRC, is set to the value of the return code resulting from the command. When the REXX RC value is set to -1, check the value of HSCMDRC to determine the return code from the actual command.

You can add REXX code after the ADDRESS HSCMDS ALLOC statement to show the REXX error return code. Figure 1-15 shows an example of this code.

**Figure 1-15.** REXX Example to Display Error Return Code

```

ADDRESS HSCMDS "ALLOC F(LOGONFIL)DA(HIPER.DATA.CNTL(LOGON)) SHR REUS"
IF RC /= 0 THEN
DO
  IF RC = -1 THEN
    SAY 'ERROR: HSCMDS ALLOC SVC 99 RC IS: 'žžHSCMDRC
  ELSE
    SAY 'ERROR: HSCMDS ALLOC RETURN CODE IS: 'žžRC
END

```

**Note:** This example only shows how to view the return code. It takes no other action when an error occurs. If the return code was placed in HSCMDRC, it is an SVC 99 action code and is not explained in the Hiperstation manuals. Refer to IBM's *MVS AUTH Assembler Services Guide* for information.

## Other Examples

This section provides script editing examples.

### Modify APPC Script Data - Scenario 1

The following script uses the recommended method for modifying APPC script data — using the <REPLACE> tag inside the <OUTBOUND> section to replace <CMSEND> <CONTENT> data. <REPLACE> uses an offset method to indicate where to start replacing.

**Note:** Both the offset count and the string of replacement data can be REXX variables (as they are in this example).

**Figure 1-16.** Script for Modifying Script Data Using the <REPLACE> Tag

```

<VERSION>8
<ADDCALL>COMPAPPC appc_expected, appc_actual, appc_expected_length, appc_actual_length
target_domain = "H01AC079"
source_domain = "H01APPCP"
"EXECIO * DISKR MESSAGES (STEM msg_in. FINIS"
say "Message is:" msg_in.port "length:" length(msg_in.port)
<DETAIL COMBINED='*'>
<TIME>2006/11/02_11:22:12.380705
<PIU>
<CONTENT> 100000001          BÇ-#... 1,Ã Ò ^^&úgiig¹          ³          BH01AC079          #INTERÌ.Š@iW%IeUSCWXN01.H01AC079 BH01AC074-"Áí
  øÿhDUSCWXN01.CWXCDRM3USCWXN01.H01AC079ÉÍB
<TIME>2006/11/02_11:22:12.385948
<PIU>
<CONTENT> i00000002          -BÇ#... ŽōÃ Ò ^^&ú ÁiiÃ ¹          ³          #INTERÌO.Š@iW%I USCWXN01.H01AC074
  -"ÁíøÿhDUSCWXN01.CWXCDRM
<TIME>2006/11/02_11:22:12.411075
<PIU>
<CONTENT> |00000003          BÇ- ...næü J H3S ÓUSCWXN01.CW010001.Š@dì< è- AA1 üRIGHT SAID FRED
<TIME>2006/11/02_11:22:12.411075
<CMALLC TPNAME=H3S SENDER=USCWXN01.³³source_domain,
RECEIVER=USCWXN01.³³target_domain,
LOGMODE=#INTER TYPE=MAPPED SYNCLVL=NONE SECURITY=NONE
STIME='2006/11/02_11:22:12.411075'
FTIME='2006/11/02_11:22:12.725288
CONV CORR='-AA1' LUWNAME=USCWXN01.CW010001 LUWINST=,
'4B9B218446B8'X LUWSEQ='0001'X>
say "Receiver is:" receiver "sender is:" sender
send_data = "RIGHT SID CHRIS"
say "Port number is:" port
dat_len = X2C(length(msg_in.port))
dat_pos = 3
<OUTBOUND>
  <TIME>2006/11/02_11:22:12.411075
  <REPLACE FIELD=(dat_pos,msg_in.port) TYPE=ONE>
  <CMSEND DATATYPE=APPLDATA>
  <CONTENT> Í 00000004RIGHT SID CHRIS
  <TIME>2006/11/02_11:22:12.411075
  <CMPTR TYPE=FLUSH>
</OUTBOUND>
<INBOUND>
  <TIME>2006/11/02_11:22:12.725288
  <PIU>
  <CONTENT> Í00000005          -BÇ j üDERF DIAS THGIR
  <TIME>2006/11/02_11:22:12.725288
  <CMSEND DATATYPE=APPLDATA>
  <CONTENT> Í 00000006DERF DIAS THGIR
  <TIME>2006/11/02_11:22:12.725288
  <CMDEAL TYPE=FLUSH LOGDATA=NO>
</INBOUND>

```

### Modify APPC Script Data - Scenario 2

This is an alternative method of modifying APPC script data.

Figure 1-17. Alternative Method for Modifying APPC Script Data

```

<VERSION>8
<ADDCALL>COMPAPPC appc_expected, appc_actual, appc_expected_length, appc_actual_length
parse source op_sys how_called scr_name rest
address HSCMDS "ALLOC F(DETAILS) DA(USR2503.TSTEAM.DETAIL("scr_name")),
" SHR REUS"
"EXECIO * DISKR DETAILS (STEM data_in. FINIS"
address HSCMDS "FREE F(DETAILS)"
parse var data_in.2 len_type "00000002" rest
"EXECIO * DISKR MESSAGES (STEM msg_in. FINIS"
say "Message is:" msg_in.port "length:" length(msg_in.port)
out_data = msg_in.port
data_in.2 = '00'X||D2C(length(out_data)+12)||substr(len_type,3,2)||,
"00000002"||out_data
say "length:" C2D(substr(data_in.2,1,2))
new_mem = "H3C1NW"||RIGHT(port,2,'0')
say "New Member:" new_mem
address HSCMDS "ALLOC F("new_mem") DA(USR2503.TSTEAM.DETAIL("new_mem")) OLD
REUS"
"EXECIO" data_in.0 "DISKW" new_mem "(STEM data_in. FINIS"
address HSCMDS "FREE F("new_mem)"
detail_mem = "H3C10000"
<DETAIL COMBINED='DETAIL('||new_mem||')'>
<TIME>2006/11/02_11:25:54.424879
<PIU LABEL=00000001>
<TIME>2006/11/02_11:25:54.424879
<CMALLC TPNAME=H3S SENDER=USCWXN01.H01APPCP RECEIVER=USCWXN01.H01AC079
LOGMODE=#INTER TYPE=MAPPED SYNCLVL=NONE SECURITY=NONE
STIME='2006/11/02_11:25:54.424879' FTIME='2006/11/02_11:25:54.435640'
CONVCORR='-AA1' LUWNAME=USCWXN01.CW010001 LUWINST=,
'4B9BBC105B62'X LUWSEQ='0001'X>
SAY "Receiver is:" RECEIVER "Sender is:" SENDER "Port:" PORT
<OUTBOUND>
<TIME>2006/11/02_11:25:54.424879
<CMSEND DATATYPE=APPLDATA LABEL=00000002>
<TIME>2006/11/02_11:25:54.424879
<CMPTR TYPE=FLUSH>
</OUTBOUND>
<INBOUND>
<TIME>2006/11/02_11:25:54.435640
<PIU LABEL=00000003>
<TIME>2006/11/02_11:25:54.435640
<CMSEND DATATYPE=APPLDATA LABEL=00000004>
<TIME>2006/11/02_11:25:54.435640
<CMDEAL TYPE=FLUSH LOGDATA=NO>
</INBOUND>

```

To explain the code, we will look at individual pieces.

1. Retrieve the user data from the associated DETAIL member:

```

parse source op_sys how_called scr_name rest
address HSCMDS "ALLOC F(DETAILS) DA(USR2503.TSTEAM.DETAIL("scr_name")),
"SHR REUS"
"EXECIO * DISKR DETAILS (STEM data_in.FINIS"
address HSCMDS "FREE F(DETAILS)"

```

2. Determine which user <CONTENT> data record to modify using the attached LABEL and extract record type information:

```

parse var data_in.2 len_type "00000002" rest

```

3. Retrieve new data from the MESSAGE file and use the replay PORT number (provided by Hiperstation) to select the replacement record for this user:

```

"EXECIO * DISKR MESSAGES (STEM msg_in.FINIS"
out_data = msg_in.port

```

4. Build the replacement user <CONTENT> data record:

```

data_in.2 = '00'X||D2C(length(out_data)+12)||substr(len_type,3,2)||,
"00000002"||out_data

```

5. Build member name for replacement DETAIL member that contains the replacement user <CONTENT> record, again using the PORT number for uniqueness:

```

new_mem="H3C1NW"||RIGHT(port,2,'0')

```

6. Create new DETAIL member:

```
address HSCMDS "ALLOC(F("new_mem") DA(USR2503.TSTEAM.DETAIL("new_mem")) OLD
REUS"
"EXECIO" data_in.0. "DISKW" new_mem "(STEM data_in. FINIS"
address HSCMDS "FREE F("new_mem")"
```

7. Modify <DETAIL COMBINED=...> tag to point to the replacement DETAIL member:

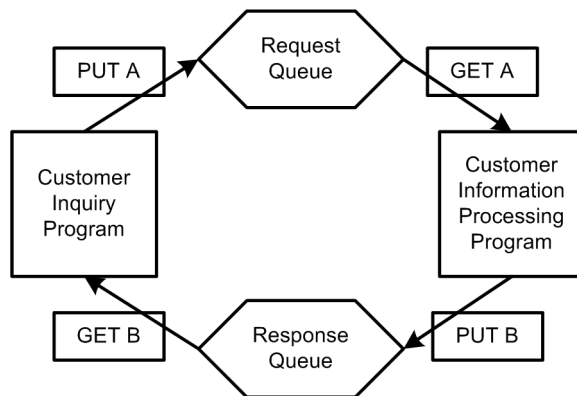
```
<DETAIL COMBINED='DETAIL('||new_mem||')>
```

## Modify an MQ Script to Test the Requester

This section provides an example that shows how to modify an MQ script to test the requesting application. In this example, the application flow is as follows:

1. The Customer Inquiry Program PUTs an inquiry to the request queue (PUT A).
2. The Customer Information Processing Program GETs the request (GET A) and processes the inquiry.
3. The processing program PUTs the response on the response queue (PUT B).
4. The inquiry programs GETs the response from the response queue (GET B) and presents it to the user.

**Figure 1-18.** MQ Script Application Flow



The inquiry program assigns a unique correlation ID to the request message (PUT A). The correlation ID remains intact all of the way through processing. When the inquiry program retrieves the response (GET B), it uses the correlation ID to match the response with the request.

In this example, the user interface for the inquiry program must be tested to ensure that it is displaying information correctly, yet the processing program is unavailable.

To test the inquiry program, the player must act as the processing program. That is, it must execute GET A and PUT B. To do this, the request queue must contain a request that the player can GET (GET A). Input inquiries after playback starts to populate the queue. However, the correlation ID from the new requests will not match the correlation ID that was initially recorded. To make this scenario work, the player must replace the recorded correlation ID with the new correlation ID.

Further, thorough testing requires displaying several customer records. Achieve this by replacing the customer information in the recorded response (PUT B) with information from an external file. Each time you submit an inquiry, playback GETs the request and returns a response generated from the records in the external file. Add logic that terminates playback if the GET is unsuccessful or after all of the records have been used.



Start with a script that contains the processing program's activity, GET A and PUT B (Figure 1-19 and Figure 1-20). Then add REXX logic to:

- "Initialize a Count Variable", page 1-43
- "Read Data from an External File", page 1-44
- "Loop Until an Unsuccessful GET or Last Record is Encountered", page 1-44
- "Process the MQ\_PUT Message", page 1-46

Each of these sections explains the REXX logic required to achieve the example and indicates where in the script to add the logic. Each section presents the relevant portion of the script along with the additional logic in **bold** text. If you have trouble with context, compare the section with Figure 1-19 and Figure 1-20. They show the script as it was recorded. Additionally, the samples library, typically SQQFSAMP, contains the edited script sample (MQTSTREQ) complete with comments.

Figure 1-19. Page 1 of Script Containing GET A and PUT B as Recorded

```

*
* SCRIPT STARTED ON 2007/02/13 AT 11:12:01
* DESC:
*
<VERSION>07.01.00
<DETAIL COMBINED='*'>
*
* MQ_CONNECT *
*
<TIME>2007/02/13_10:56:01.327075
<MQ_CONNECT,
  CONNECTION_ID = 2,
  COMPCODE = 0,
  REASON = 0,
  QMGR_NAME = 'QMGR',
  SYSTEM_NAME = 'SYS1',
  JOBNAME = 'JOB',
  STIME = '2007/02/13_10:56:01.327075'>1
*
* MQ_OPEN *
*
<TIME>2007/02/13_10:56:01.325201
<MQ_OPEN,
  CONNECTION_ID = 2,
  OBJECT_ID = 3,
  COMPCODE = 0,
  REASON = 0,
  MQOO = (FAIL_IF QUIESCING,SAVE_ALL_CONTEXT,INPUT_SHARED),
  MQOD_VERSION = 3,
  MQOD_OBJECTNAME_OUT = 'CLIENT.REQUEST.QUEUE',
  MQOD_OBJECTNAME_IN = 'CLIENT.REQUEST.QUEUE',
  MQOD_OBJECTQMGRNAME = '',
  MQOD_DYNAMICQNAME = 'CSQ.*'>2
*
* MQ_OPEN *
*
<TIME>2007/02/13_10:56:01.323695
<MQ_OPEN,
  CONNECTION_ID = 2,
  OBJECT_ID = 2,
  COMPCODE = 0,
  REASON = 0,
  MQOO = (FAIL_IF QUIESCING,PASS_ALL_CONTEXT,OUTPUT),
  MQOD_VERSION = 3,
  MQOD_OBJECTNAME_OUT = 'CUSTOMER.RESPONSE.QUEUE',
  MQOD_OBJECTNAME_IN = 'CUSTOMER.RESPONSE.QUEUE',
  MQOD_OBJECTQMGRNAME = '',
  MQOD_DYNAMICQNAME = 'CSQ.*'>3
*
* MQ_GET *
*
<TIME>2007/02/13_10:56:01.327075
<MQ_GET,
  CONNECTION_ID = 2,
  OBJECT_ID = 3,
  COMPCODE = 0,
  REASON = 0,
  MQMD_EXPIRY = '00001387'X,
  MQMD_ENCODING = '00000222'X,
  MQMD_CODEDCHARSETID = 437,
  MQMD_FORMAT = 'MQSTR',
  MQMD_PRIORITY = '',
  MQMD_PERSISTENCE = NOT_PERSISTENT,
  MQMD_MSGID = '0000000000000000000000000000000000000000000000000000000000000000'X,
  MQMD_BACKOUTCOUNT = '',
  MQMD_REPLYTOQMGR = 'QMGR',
  MQMD_USERIDENTIFIER = 'ASPNET',
  MQMD_ACCOUNTINGTOKEN = '0000000000000000000000000000000000000000000000000000000000000000'X,
  MQMD_PUTAPPLTYPE = WINDOWS_NT,
  MQMD_PUTAPPLNAME = 'APPL',
  MQMD_PUTDATE = '20070213',
  MQMD_PUTTIME = '15560215',
  MQGMO_VERSION = 3,

```



## Read Data from an External File

Reading data from the external file is the next required step. After variable initialization, before the connection is established, insert an EXECIO statement to read the records into a REXX stem variable, 'reply\_message'. This results in a series of reply\_message.*n* variables, where *n* indicates the record number. Include a FINIS parameter to close the file after all of the records have been read.

```
*
"EXECIO*DISKRINDATA(STEMreply_message.FINIS"

*
* MQ_CONNECT      *
*
<TIME>2007/02/13_10:56:01.327075
<MQ_CONNECT,
  CONNECTION_ID = 2,
  COMPCODE = 0,
  REASON = 0,
  QMGR_NAME = 'QMGR',
  SYSTEM_NAME = 'SYS1',
  JOBNAME = 'JOB',
  STIME = '2007/02/13_10:56:01.327075'>1
*
* MQ_OPEN *
*
<TIME>2007/02/13_10:56:01.325201
<MQ_OPEN,
  CONNECTION_ID = 2,
  OBJECT_ID = 3,
  COMPCODE = 0,
  REASON = 0,
  MQOO = (FAIL_IF_QUIESCING,SAVE_ALL_CONTEXT,INPUT_SHARED),
```

## Loop Until an Unsuccessful GET or Last Record is Encountered

Because the script contains only a single transaction, looping logic must be added to the script to execute the GET and process the response message the correct number of times. If the GET is unsuccessful or there are no more records to process, the loop must terminate. To do this:

1. Before the MQ\_GET tag that retrieves the request, insert a DO statement to execute the MQ\_GET until all records have been processed (DO i=1 TO reply\_message.0). 'reply\_message.0' represents the total number of records.
2. Add a WAIT parameter to the MQGMO\_OPTIONS to avoid executing multiple MQ\_GETS on an empty queue.
3. After the MQ\_GET tag, insert an IF statement that checks the reason code to ensure that the GET is successful. Use the Hiperstation for WebSphere MQ REXX variable ACTUAL\_REASON to return the reason code. If the reason code equals zero, increment the count and set the length value (new\_len) used for processing the response message to the length of the current record (reply\_message.i).
4. After the MQ\_PUT that delivers the response, insert an END statement that ends the current loop and returns to the beginning of the DO loop, providing the IF statement condition is met. That is, the MQ\_GET is successful.
5. After the END statement, insert an ELSE statement that terminates the loop if the MQ\_GET is unsuccessful. To terminate the loop, set the value of i to 'reply\_message.0'.
6. After the ELSE statement, insert another END statement to terminate the DO loop when all records from the external file have been processed.





```
      MQCO = (NONE)>8
*
* MQ_CLOSE *
*
<TIME>2007/02/13_10:56:03.801748
<MQ_CLOSE,
      CONNECTION_ID = 2,
      OBJECT_ID = 3,
      COMPCODE = 0,
      REASON = 0,
      MQCO = (NONE)>7
*
* MQ_DISCONNECT *
*
<TIME>2007/02/13_10:56:03.823516
<MQ_DISCONNECT,
      CONNECTION_ID = 2,
      COMPCODE = 0,
      REASON = 0>9
```





## Chapter 2.

# Common Programming Interface

The Hiperstation high-level language, common programming interface (HLL CPI) allows programs written in REXX to accomplish the following:

- Establish sessions with transaction processing facilities (such as CICS and IMS/DC) that use VTAM and 3270 communication protocol
- Simulate a terminal operator's typed input
- Send data to a transaction processing facility in either 3270 or field level formats
- Interrogate and examine the output terminal data
- Run Hiperstation scripts
- Terminate the session.

The terms *input* and *output* are used throughout this chapter. These terms are consistent with the CPU's point of view. Input is data flowing from the 3270 devices to the domain destination. Output is data coming from the domain destination and going to the virtual 3270.

---

## Overview

The Common Programming Interface (CPI) is not related to embedding REXX code in a Hiperstation script and, therefore, does not deal with script language capability. Use the CPI to write a normal REXX program that can talk to an online application.

Perhaps, you want to write a REXX program that logs on to a CICS region, performs a sign-on, executes some online transactions, and logs off from the application. Rather than accomplishing this by playing back a script, use the CPI to take Hiperstation's automation capabilities to another level. If you wrote a REXX program that executed continuously all day as a started task, this program could periodically read from a transaction file, take that data, log onto IMS, and automatically enter that data into the online application. Or, it could periodically scan other areas (files and screens, for example) and automatically log on to the e-mail system and send a message to a system programmer if it discovers a problem. If you are running a web server system on your mainframe, you can create web pages that communicate to legacy applications. You do all this by writing a REXX CGI program that uses Hiperstation's REXX CPI to handle communication to the legacy online application.

By interfacing with Hiperstation, the REXX program "talks" to the online application through a Hiperstation virtual terminal. You do this by invoking a function called HSCPI, which is not a standard REXX function and is not documented in any REXX manual.

Before you can invoke the HSCPI function from a REXX program, you must define the HSCPI function to REXX (see "Define a Function Package Directory for REXX CPI" in the *Hiperstation Installation and Configuration Guide*).

---

## Call Interface

The CPI operates as a REXX function. The function name is HSCPI. A typical call might look like:

```
HSCPI('LOGON', 'SESID', TSOVAR)
```

The first argument in the CALL parameter list identifies the action to be taken by the CPI. This argument is referred to as the verb.

The second argument always contains the name of a REXX variable to be used as a session identifier or session-ID. The session-ID is returned by the LOGON verb and is used in subsequent calls to identify the session on which the verb is to operate.

The HSCPI function call also returns a return code. The return code is placed in a REXX variable via an assignment statement:

```
RC = HSCPI('LOGON', 'SESID', TSOVAR)
```

## Session IDs

Hiperstation uses a session ID to identify the session. The session is created during the successful execution of a LOGON verb, and the session ID is returned to the calling program as one of the parameters. Subsequent calls to the CPI pass the session-ID as a means of identifying the session to Hiperstation as in:

```
HSCPI('LOGON', 'SESID', tsovar)
```

## Parameters

The CPI uses two types of parameters: (1) character strings of a variable length, and (2) integers.

All integers are of the full word type. In some calls, a variable amount of data may flow back and forth between the caller and called program.

Some of the parameters are mandatory. Others are optional. Optional parameters are shown in square brackets [ ]. In the following example, the verb and session-ID are required, and the key-ID is optional:

```
HSCPI('SEND', 'SESID', [key-ID])
```

## DD Requirements

The CPI uses three DD statements:

- HPERIN DD
- HPERLIB DD
- HPEROUT DD.

The HPERIN DD contains the CONTROL statement that defines Hiperstation's execution environment.

The HPERLIB DD defines the datasets that contain any prerecorded scripts used by the EXECUTE verb. If the EXECUTE verb is not used, the HPERLIB DD is not required.

The HPEROUT DD defines the dataset containing Hiperstation status messages as well as any output from REXX that is generated via an executed script.

---

## Verbs

The Hiperstation HLL CPI provides the following verbs for communication management:

- **LOGON:** Establishes a session with a domain destination.
- **LOGOFF:** Terminates a session created by LOGON.

- **EXECUTE:** Executes a prerecorded Hiperstation for VTAM script.
- **SEND:** Transmits data from the input buffer to the domain destination. Hiperstation waits for a response from the domain destination.
- **RECEIVE:** This verb is obsolete. Its function has been included in the SEND verb.
- **TYPE:** Places data in an input field.
- **RETRIEVE:** Copies data from the screen to variables in the caller's program.
- **COPY3270:** Copies the current screen in 3270 format to a variable in the user's program.
- **REPL3270:** Replaces the 3270 input data stream.

This section describes the purpose of each verb, the operands associated with it, and its return codes.

## Logon

The LOGON verb establishes a session with a domain destination. The domain destination name is set as one of the operands.

The session, or port-ID, is returned in the listed REXX variable following successful session establishment.

### *Logon Verb Operands*

```
HSCPI(
  'LOGON',
  'session-ID',
  domain-destination-data)
```

#### **session-ID**

The name of the REXX variable containing the returned session-ID. This identifier is required for further communication with the CPI.

#### **domain-destination-data.**

The name of the domain destination.

**Return Codes****Table 2-1.** Hiperstation Return Codes for the Logon CPI Verb

Code	Description
0	The session has been successfully connected.
8	Initialization error: <ul style="list-style-type: none"> <li>All virtual terminals busy.</li> <li>Domain destination not active.</li> <li>HPERIN not allocated.</li> </ul> Check the system log for messages EHSnnnn or ETnnnnnn for more detail.
16	Error: Invalid script statement found
20	Serious Error: <ul style="list-style-type: none"> <li>Port could not be started because LU2 application is not valid or not active.</li> <li>Port started, but could not be initialized.</li> <li>REXX EXEC/script not found.</li> </ul>
24	Serious Error: The first port to start could not get started — the unattended mode process is terminated. If other ports started, unattended mode continues.
28	SEVERE ERROR: Port subtask could not be started. <ul style="list-style-type: none"> <li>SYSPRINT could not be opened or error processing SYSPRINT.</li> <li>SYSIN could not be opened or error processing SYSIN.</li> <li>SYSCNTL could not be opened or error processing SYSCNTL.</li> <li>User failed security authorization.</li> <li>Either LU2 or LU6.2 pool names could not be built from the provided prefix and suffix values.</li> <li>Product is not licensed for LU6.2 (APPC) unattended mode processing.</li> <li>LU6.2 is not supported for LU6.2 scripts.</li> <li>Trace could not be initialized.</li> <li>No terminals found to be played back.</li> <li>VTAM initialization error.</li> <li>A syntax error exists on the JOURNAL, LOG, XLOG, RLOG, AUTODOC, SUMMARY, or TRACE keywords on the CONTROL, GROUP, ALLOCATE, or COMPARE statements.</li> </ul>
32	SERIOUS ERROR: A print file is full or could not be processed. The print files are the JOURNAL, LOG, XLOG, RLOG, AUTODOC, SUMMARY, and TRACE files.
40	SEVERE ERROR: Hiperstation is about to expire or has expired. Contact Hiperstation Customer Support for a date extension and new release availability information.

**Logoff**

LOGOFF terminates the session with the domain destination and frees all resources acquired during the session.

**Logoff Verb Operands**

```
HSCPI(
  'LOGOFF',
  'session-ID')
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**Return Codes**

Table 2-2. Return Codes for Logoff CPI Verb

Code	Description
0	The session has been terminated.
64	Session-ID was invalid.

**Execute**

The EXECUTE verb executes a Hiperstation script.

**Execute Verb Operands**

```
HSCPI(
  'EXECUTE',
  'session-ID',
  scriptname)
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**scriptname**

The eight-byte name of a Hiperstation for VTAM script. The script must be in the HPERLIB DD statement.

**Return Codes**

Table 2-3. Return Codes for Execute CPI Verb

Code	Description
0	The script ran successfully.
4	The script was not found in the HPERLIB DD definition.
8	The HPERLIB DD statement was not defined.
64	Session-ID was invalid.

**Send**

Following a SEND operation, Hiperstation waits until either a response is received from the domain destination or until a timeout occurs. Available input data is sent to the domain destination. The SEND verb allows you to choose the PF or PA key to be transmitted.

**Send Verb Operands**

```
HSCPI(
  'SEND',
  'session-ID',
  [key-ID])
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**key-ID**

The name of the attention identifier key used to transmit the data (for example, PF1, PA2, etc.). If this operand is omitted, transmit the data using the Enter key.

**Return Codes****Table 2-4.** Return Codes for Send CPI Verb

Code	Description
0	The data was sent.
4	Invalid AID key.
64	Session-ID was invalid.

**Receive**

This verb is obsolete. Its function has been included in the SEND verb. It appears here for downward compatibility.

**Receive Verb Operands**

```
HSCPI(
  'RECEIVE',
  'session-ID')
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**Return Codes****Table 2-5.** Return Codes for Receive CPI Verb

Code	Description
0	The data has been received.
64	Session-ID was invalid.

**Type**

The TYPE verb places data in an input field in the 3270 screen image. You can enter the data at screen locations by row and column coordinates or by 3270 field number.

**Type Verb Operands**

```
HSCPI(
  'TYPE',
  'session-ID',
  [row],
  [column],
  [field-number],
  data)
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**row**

Designates the row number of the input field. This number is relative to one.

**column**

Designates the column number of the input field. This number is relative to one.

**field-number**

Indicates the input field number to place the data in. This number is relative to one.

**data**

Data to be placed in the input field.

**Return Codes**

**Table 2-6.** Return Codes for the Type CPI Verb

Code	Description
0	The data has been placed in the input field.
4	The field number does not exist.
8	No input field at the entered row and column.
12	The data was truncated.
64	Session-ID was invalid.

**Retrieve**

Retrieve obtains data from the screen buffer and places it in the caller's variables.

The data can be retrieved by row and column coordinates or by a 3270 field number. You can obtain the entire screen image by omitting both the row/column and field numbers.

When the entire screen image is returned, the attributes are translated into pseudo-attributes to distinguish them from the textual data.

**Retrieve Verb Operands**

```
HSCPI(
  'RETRIEVE',
  'session-ID',
  [row],
  [column],
  [field-number],
  'data',
  max-data-length,
  ['returned-data-length'])
```

**session-ID**

The name of the REXX variable that contains the session-ID returned by the LOGON verb.

**row**

Designates the row number of a field. This number is relative to one.

**column**

Designates the column number of a field. This number is relative to one.

**field-number**

Indicates the field number from which to retrieve data. This number is relative to one.

**data**

The name of the REXX variable containing the returned data.

**max-data-length**

Maximum length of the data that can be returned.

**returned-data-length**

The name of the REXX variable containing the actual length of the returned data.

***Return Codes***

**Table 2-7.** Return Codes for Retrieve CPI Verb

Code	Description
0	The data has been retrieved.
4	The field number does not exist.
8	The row and column exceed the screen dimensions.
12	The requested data was truncated because it exceeded the max-data-length.
64	Session-ID was invalid.

## Copy3270

This verb copies the 3270 output screen buffer to the applications buffer. The screen is built as a 3270 data stream. The returned buffer contains both the write command and the WCC.

***Copy3270 Verb Operands***

```
HSCPI(
'COPY3270',
'session-ID',
'buffer',
max-buffer-length,
['returned-buffer-length'])
```

**session-ID**

The session-ID returned by the LOGON verb.

**buffer**

The name of a REXX variable containing the returned buffer.



**max-buffer-length**

The largest size data stream that the application can accept. The buffer should be approximately 4096 bytes long.

**returned-buffer-length**

The name of a REXX variable containing the actual length of the returned data.

**Return Codes****Table 2-8.** Return Codes for Copy3270 CPI Verb

Code	Description
0	Buffer successfully built.
12	The requested buffer was truncated because it exceeded the max-buffer-length.
64	Session-ID was invalid.

**REPL3270**

REPL3270 Replaces the 3270 input buffer. The data replaced is in 3270 format. The data must be in 3270 short read format, and it must be preceded by a 3270 AID definition (the row and column of the cursor followed by the screen data).

**REPL3270 Verb Operands**

```
HSCPI(
  'REPL3270',
  'session-ID',
  buffer)
```

**session-ID**

The session-ID returned by the LOGON verb.

**buffer**

A buffer in 3270 input format.

**Return Codes****Table 2-9.** Return Codes for REPL3270 CPI Verb

Code	Description
0	The data was successfully stored in the buffer.
64	Session-ID was invalid.

**Sample REXX EXEC**

Figure 2-1 on page 2-10 is a sample REXX EXEC to log a user onto TSO, issue =x at the ISPF Primary Option Menu, and issue logoff at the TSO READY prompt to terminate the session. Calls to the “display” subroutine are made after each type and send request to show the current screen contents. This EXEC can be found in member CPIREXX of SQQFSAMP.

Figure 2-1. Sample REXX EXEC

```

PURPOSE: DEMONSTRATE THE USE OF HSCPI  Before you can use this REXX example, you
need to do the following:
1. Follow the Install Instructions for the HSCPI (refer to "Define a Function
Package Directory for REXX CPI" in the Hiperstation Installation Guide.
2. Allocate the HPERIN and HPEROUT DDs to your TSO session. See "DD Requirements"
on page 3-2 for more information.
3. Allocate the HPERLIB DD if you want to be able to run Hiperstation scripts
within REXX.

"ALLOC F(HPERIN) DSNAME('USERABO.REXX.EXEC(CONTROL)') SHR REUSE"
"ALLOC F(HPEROUT) DSNAME('USERABO.REXX.EXEC(HIPROUT2)') SHR REUSE"
/* REXX */
y = 'TSO' /* data can be in a variable */
x = hscpi('logon','SESID',y)
say 'session ID is:' SESID /* what is our session ID? */
say 'return from logon:' x /* a little trace */

x = hscpi('type','SESID',2,2,,'TSOHP22') /* type in userID */
say 'return from type:' x
call display SESID

x = hscpi('send','SESID','enter') /* send data with enter key */
say 'return from send:' x
call display SESID

x = hscpi('type','SESID',8,21,,'BLUE') /* type in password */
say 'return from type:' x
call display SESID

z = 'enter'
x = hscpi('send','SESID',z) /* send data with enter key */
call display SESID

x = hscpi('send','SESID','enter') /* get rID of *** */
call display SESID

x = hscpi('type','SESID',2,15,,'=X') /* put =x on ispf main menu */
say 'return from type:' x
call display SESID

x = hscpi('send','SESID','enter')
call display SESID

x = hscpi('type','SESID',2,2,,'logoff') /* logoff at ready prompt */
say 'return from type:' x
call display SESID

x = hscpi('send','SESID','enter')
call display SESID

x = hscpi('logoff','SESID')
say 'return from logoff:' x
exit

display: procedure /* display current screen image */
arg SESID
x = hscpi('retrieve','SESID',0,0,0,'DATA',1920)
DATA = translate(DATA, ' ', '01 02 03 04 05 06 07 08 09'x)
do x = 24 to 1 by -1 /* remove attribute at start of each line */
z = ((x-1)*80)+1
DATA = delstr(DATA,z,1)
end
say 'session ID is:' SESID
say DATA
return
"FREE F(HPERIN)"
"FREE F(HPEROUT)"

```

## Chapter 3.

# 3270 Repository Filtering and Copy Utility

Use the 3270 Repository Filtering and Copy Utility (EHS3COPY) to create subset repositories containing specific activity. Filter by any of the following criteria:

- Terminal LU
- VTAM APPLID
- User ID
- Session date-timestamps
- Message date-timestamps

Write the filtered activity to a repository dataset or pass it to an exit program for further processing. For example, you need to audit the activity that transpired on a specific day relevant to a specific vehicle, but the repository contains several days worth of activity. To create a repository containing only the activity of interest:

1. Write an exit-program that scans the repository for information specific to the vehicle, such as Vehicle Identification Number (VIN).
2. Run the EHS3COPY utility with session date-time criteria and specify the exit-program to receive the filtered activity.

EHS3COPY is a batch utility that requires control statements. Plug the control statements into the JCL used to run the utility. Hiperstation provides a series of samples to help you write exit-programs and build JCL.

---

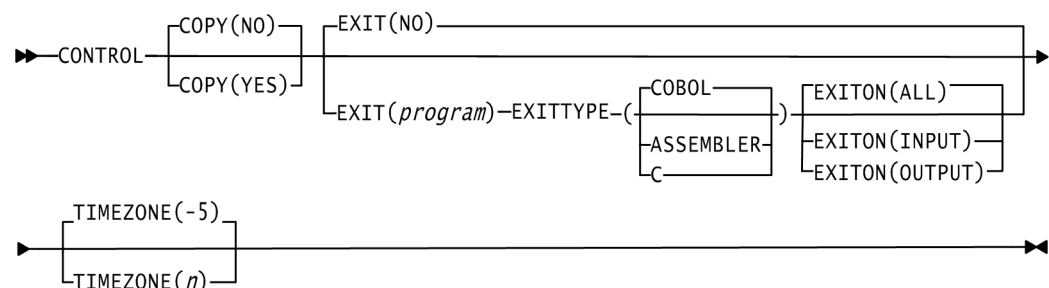
## Creating EHS3COPY Control Statements

EHS3COPY is a batch utility that requires the following control statements:

- “CONTROL Statement” on page 3-1
- “COPY Statement” on page 3-2

Place these statements in the JCL used to run the utility. Save time by starting with one of the JCL samples that Hiperstation provides. See “Preparing the JCL” on page 3-6 for more information.

### CONTROL Statement



**COPY(NO|YES)**

Specifies whether to create an output repository. Default is NO (no output). YES creates an output repository containing all 3270 sessions that satisfy established filters.

**Note:** If you use the sample JCL to run EHS3COPY, and you pass the data to an exit program modeled on one of the provided samples, do not supply the COPY parameter. The sample exit programs generate CONTROL and COPY statements into the CONTROL DD. CONTROL DD is used as the SYSIN for the second invocation of EHS3COPY, the copy step.

**EXIT(NO|program)**

Specifies whether an exit program is called with each terminal input and output. Only 3270 data is passed. Default is NO (no exit program is called). Entering a program name specifies an exit program to call. Program must be in COBOL II, Assembler, or C. If you specify an exit program, you must specify the EXITON() parameter.

Any non-zero return code from EXIT terminates the copy function.

**EXITTYPE(COBOL|ASSEMBLER|C)**

The language of the exit program specified on the EXIT parameter, if any.

**EXITON(ALL|INPUT|OUTPUT)**

Specifies when an exit program is to be called based on type of message. If you use this parameter, you must specify an exit program using EXIT(*program*).

**ALL** calls a user-exit program for all messages. **INPUT** calls a user-exit program for input messages only. **OUTPUT** calls a user-exit program for output messages only.

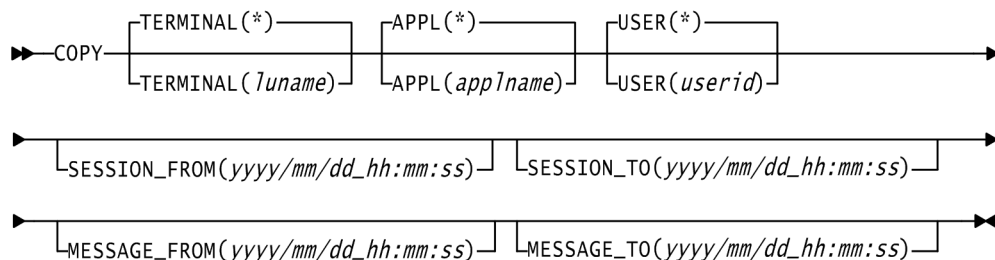
**TIMEZONE(-5|n)**

Time difference in your area from Greenwich Mean Time (GMT). Default is -5 (Eastern Standard time).

## COPY Statement

The COPY statement provides filtering criteria. Place the COPY statement after the CONTROL statement in the JCL.

The COPY statement is optional. However, if you omit a COPY statement, but specify COPY(YES) on the CONTROL statement, all 3270 repository records are copied to the output repository. If COPY statements are added, there is a limit of 1,000 per execution.

**TERMINAL(\*|luname)**

The LU name of the terminal associated with the sessions you need to copy. If filtering by terminal ID is unnecessary, omit this parameter. Otherwise, supply:

- A specific LU name.
- A range of LU names — An LU name prefix followed by an asterisk (\*) wildcard character.

**APPL(\*|applname)**

The APPLID associated with the sessions you need to copy. If filtering by application name is unnecessary, omit this parameter. Otherwise, supply:

- A specific APPLID.
- A range of APPLIDs — An APPLID prefix followed by an asterisk (\*) wildcard character.

**USER(\*|userid)**

The user ID associated with the sessions you need to copy. If filtering by user ID is unnecessary, omit this parameter. Otherwise, supply:

- A specific user ID.
- A range of user IDs — A user ID prefix followed by an asterisk (\*) wildcard character.

**SESSION\_FROM(yyyy/mm/dd\_hh:mm:ss)**

If specified, a session time greater than or equal to the specified time is copied.

**SESSION\_TO(yyyy/mm/dd\_hh:mm:ss)**

If specified, a session time lesser than or equal to the specified time is copied.

**MESSAGE\_FROM(yyyy/mm/dd\_hh:mm:ss)**

If specified, a message time greater than or equal to the specified time is copied.

**MESSAGE\_TO(yyyy/mm/dd\_hh:mm:ss)**

If specified, a message time lesser than or equal to the specified time is copied.

---

## Writing Exit Programs

User-written exit programs can be written in Assembler, COBOL, and C. The Hiperstation sample library (SQQFSAMP) contains sample programs including source code, exit data area, compile JCL, and run JCL.

Data passed to the user-exit program includes (from COBOL example in EHS3XDCB):

```
*01  EHS3COPY-EXIT-DATA.
      10  EXIT-ID          PIC X(8).
          88  EXIT-DATA-ID  VALUE 'EXITDATA'.
      10  EXIT-DATA-LENGTH PIC S9(8)  COMP SYNC.
      10  FUNC            PIC S9(8)  COMP SYNC.
          88  EHS3COPY-EXIT-INIT  VALUE 1.
          88  EHS3COPY-EXIT-VSBE  VALUE 2.
          88  EHS3COPY-EXIT-TERM  VALUE 3.
          88  EHS3COPY-EXIT-ABEND  VALUE 4.
      10  RC              PIC S9(8)  COMP SYNC.
      10  TERM           PIC X(8).
      10  APPL           PIC X(8).
      10  USER           PIC X(8).
      10  VSSMSTR        PIC X(8).
      10  SESSION-START-TIME.
*      YYYYY/MM/DD_HH:MM:SS.MMMMMM  FORMAT
      20  EHS3YEAR        PIC X(4).
      20  S1              PIC X.
      20  EHS3MNTH        PIC X(2).
      20  S2              PIC X.
      20  EHS3DAY         PIC X(2).
      20  U               PIC X.
      20  EHS3HOUR        PIC X(2).
      20  C1              PIC X.
      20  EHS3MIN         PIC X(2).
```

```

20 C2          PIC X.
20 EHS3SEC     PIC X(2).
20 D           PIC X.
20 EHS3MCRS    PIC X(6).
10 MESSAGE-TIME.
*   YYYYY/MM/DD_HH:MM:SS.MMMMMM FORMAT
20 EHS3YEAR    PIC X(4).
20 S1          PIC X.
20 EHS3MNTH    PIC X(2).
20 S2          PIC X.
20 EHS3DAY     PIC X(2).
20 U           PIC X.
20 EHS3HOUR    PIC X(2).
20 C1          PIC X.
20 EHS3MIN     PIC X(2).
20 C2          PIC X.
20 EHS3SEC     PIC X(2).
20 D           PIC X.
20 EHS3MCRS    PIC X(6).
10 VSBE.
20 VSBEID      PIC X(4).
   88 VSBE-DATA-ID VALUE 'VSBE'.
20 VSBELEN     PIC S9(8) COMP SYNC.
20 VSBEDIR     PIC X.
   88 VSBEIN     VALUE X'02'.
   88 VSBEOUT    VALUE X'01'.
20 VSBEFLAG    PIC X.
20 VSBETIME    PIC X(8).
20 VSBETH.
   30 FID        PIC X.
   30 FILLER     PIC X(23).
   30 FID4DCF    PIC S9(4) COMP SYNC.
*   THE SIZE OF TIOA = FID4DCF - 3.
20 VSBEDATA.
   30 VSBERH     PIC X(3).
   30 VSBE-DATA  PIC X(16297).
   30 TIOA       REDEFINES VSBE-DATA.
   40 FILLER     PIC X(16297).

```

## Tips for Writing Exit Programs

1. The user-exit program is loaded once and called at each message. Initialize at INIT call (for example, reading search strings) once for performance, and write the result at the TERM call. The call type is passed in the FUNC area with the following level 88 values:

```

10 FUNC          PIC S9(8) COMP SYNC.
   88 EHS3COPY-EXIT-INIT VALUE 1.
   88 EHS3COPY-EXIT-VSBE VALUE 2.
   88 EHS3COPY-EXIT-TERM VALUE 3.
   88 EHS3COPY-EXIT-ABEND VALUE 4.

```

2. The VSBEDIR field contains the following values:

```

20 VSBEDIR      PIC X.
   88 VSBEIN     VALUE X'02'.
   88 VSBEOUT    VALUE X'01'.

```

VSBEIN is 3270 input. VSBEOUT is 3270 output.

3. In the exit program, TIOA size is calculated by this formula:

```
SIZE OF TIOA = FID4DCF-3
```

Write the exit program so that it does not reference storage beyond the address of TIOA + TIOA length. Otherwise, the exit program may get storage protection errors (MVS abend 0C4).

You need to know some 3270 datastream basics to write the exit program.

- a. For output, the first byte in TIOA is 3270 command code.

```
Command
=====
(X'F1')Write
(X'F5')Erase/Write
(X'7E')Erase/Write Alternate
(X'F2')Read Buffer
(X'F6')Read Modified
(X'6E')Read Modified All
(X'6F')Erase all Unprotected
(X'F3')Write Structured Field
```

- b. For input, the first byte in TIOA is 3270 AID (Attention IDentifier).

```
AID
=====
(X'60')No AID generated
(X'E8')No AID generated - printer only
(X'88')Structured Field
(X'61')Read Partition
(X'7F')Trigger Action
(X'F0')Test Req and Sys Req
(X'F1') to (X'F9') F1 to F9
(X'7A') to (X'7C') F10 to F12
(X'C1') to (X'C9') F13 to F21
(X'4A') to (X'4C') F22 to F24
(X'6C')PA1
(X'6E')PA2
(X'6B')PA3
(X'6D')Clear
(X'6A')Clear partition
(X'7D')Enter
(X'7E')Selector Pen
(X'E6')Magnetic Readers: Operator ID reader
(X'E7')Magnetic Readers: Mag Reader Number
```

4. One 3270 session can have multiple session control blocks. The first control block cannot contain the user ID because of the way Global Recording creates repository records.
- a. To select an entire session, use EHS3COPY with a user-exit (see 'C' sample in EHS3SXC) to create a control file to copy in step 4.c. Specify COPY(NO) in this step. Calling a user-exit program takes more processing time. Use the parameters in the COPY statement to reduce processing overhead by filtering records before the utility calls the user-exit program.

```
CONTROL COPY(NO) EXIT(EHS3SXC) EXITTYPE(C)
COPY TERMINAL(filter-information) APPL(filter-information) USER(filter-
information)
```

**Note:** The COPY statement is optional. Also, in the //SCANINFO DD line, provide a "SCAN=string" (up to 75 bytes) card input or a "SELECT\_ALL" to select every record. You can specify multiple scan strings.

For user ID selection, specify:

```
COPY USER(userid)
```

For an actual copy job with COPY(YES) parameter, the USER parameter should not be used in most cases.

Example of COPY Statement syntax:

```
//SYSIN DD *
CONTROL COPY(YES) EXIT(NO)
COPY USER(SJPHXA1) -
SESSION_FROM(2011/04/06_14:30:00) TERMINAL(DHPR*)
/*
```

Notice that there is a hyphen indicating a continuation of the COPY USER line. Also there is a space between the SESSION\_FROM() and TERMINAL() parameters.

- b. The output from the exit program (see 'C' sample in EHS3SXC) goes to the //REPORT DD //CONTROL DD file.
- c. Run EHS3COPY with the control file created in step 4.a. If you want to modify the program, modify and compile the sample C program in SQQFSAMP. The compile JCL is in SQQFSAMP member EHS3CJC. The Hiperstation load library (SQQFLOAD) contains the EHS3SXC program.

## Preparing the JCL

Hiperstation provides a series of samples to aid you in running the utility and writing your own exit programs (Table 3-1). Locate these samples in the Hiperstation samples library (typically SQQFSAMP). Member \$EHS3IDX is an electronic index of these samples.

**Table 3-1.** Sample to Run EHS3COPY Utility and Aids for Authoring Exit Programs

Member	Description
EHS3CJA	JCL to assemble exit program written in Assembler
EHS3CJC	JCL to compile exit program written in C
EHS3CJCB	JCL to compile exit program written in COBOL
EHS3COPY	JCL to search and select/copy sessions
EHS3COP	JCL to copy sessions by terminal
EHS3RJA	JCL for using exit program written in Assembler
EHS3RJC	JCL for using exit program written in C
EHS3RJCB	JCL for using exit program written in COBOL
EHS3SCAN	JCL to search strings in a repository file
EHS3SXA	Source code for sample exit program written in Assembler
EHS3SXC	Source code for sample exit program written in C. <b>Note:</b> This sample is ready for immediate use. It has been compiled and linked. Execute this program with sample member EHS3COPY. Within EHS3COPY, on the SCAN statement, specify the text to use for filtering. All matching activity is copied to the new repository.
EHS3SXCB	Source code for sample exit program written in COBOL
EHS3XDA	Layout of data passed to the exit program from EHS3COPY (Assembler)
EHS3XDC	Layout of data passed to the exit program from EHS3COPY (C)
EHS3XDCB	Layout of data passed to the exit program from EHS3COPY (COBOL)

## JCL Examples

This section contains two examples of JCL that invokes EHS3COPY. The first is a simple scan and copy job. The second invokes a user-exit to scan the repository.



The following example invokes EHS3COPY to scan the repository and copies the activity from terminal VT15\* through VT18\* into a new repository. It also copies a user ID from the specified time frame with the specified terminal. The COPY USER(USERA1) example line continues to the next line. The hyphen denotes the continuation. Also notice that there is a space between the SESSION\_FROM() and TERMINAL() parameters.

```
//ST060 EXEC PGM=EHS3COPY,REGION=32M,PARM='POSIX(OFF)'/
/*
//STEPLIB DD DISP=SHR,DSN=COMPWARE.QQF160.SQQFLOAD
//REPOSIN DD DISP=SHR,DSN=CICSGOT.REPOSIT.AS0X01
//REPOSOUT DD DISP=SHR,DSN=CICSGOT.REPOSIT.ASOC04
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//CONTROL DD SYSOUT=*,DSORG=PS,RECFM=FB,LRECL=80
//SYSIN DD *
CONTROL COPY(YES) EXIT(NO)
COPY TERMINAL(VT15*)
COPY TERMINAL(VT16*)
COPY TERMINAL(VT17*)
COPY TERMINAL(VT18*)
COPY USER(USERA1) -
SESSION_FROM(2011/04/06_14:30:00) TERMINAL(DHPR*)
/*
```

The next example incorporates a user exit that scans the repository for sessions containing the text "green chev", then copies those sessions to a new repository.

```
/*..... JOB ...PLACE YOUR INSTALLATION'S JOBCARD HERE...
/*
/* BEFORE YOU SUBMIT THE JOB, PERFORM THE FOLLOWINGS:
/* (1) ADD THE PROPER JOB CARD.
/* (2) CHANGE QALOAD= assignment with your Hiperstation LOADLIB.
/* (3) CORRECT repository file names (REPOSIN and REPOSOUT)
/*
/* START of PROC definition
//REPOCOPY PROC QALOAD='COMPUWARE.QQF160.SQQFLOAD',
// REPOIN=,REPOOUT=
//DELETE EXEC PGM=IEFBR14
//DD1 DD DISP=(MOD,DELETE),DSN=&REPOOUT.,
// SPACE=(CYL,(1,1),RLSE),
// DSORG=PS,RECFM=VB,LRECL=32756,BLKSIZE=32760
/*
//SCAN EXEC PGM=EHS3COPY,REGION=0M,PARM='POSIX(OFF)'/
//STEPLIB DD DISP=SHR,DSN=&QALOAD.
//REPOSIN DD DISP=SHR,DSN=&REPOIN.
/*SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//CONTROL DD DSN=&&CONTROL,DISP=(NEW,PASS),
// SPACE=(CYL,(1,1)),
// DSORG=PS,RECFM=FB,LRECL=80
//REPORT DD DSN=&&REPORT,DISP=(NEW,PASS),
// SPACE=(CYL,(1,1)),
// DSORG=PS,RECFM=VB,LRECL=125
/*
//LISTRPT EXEC PGM=IDCAMS,REGION=1024K,
// COND=(4,LT,SCAN)
//INPUT DD DSN=*.SCAN.REPORT,DISP=(OLD,PASS)
//OUTPUT DD SYSOUT=*
//SYSPRINT DD DUMMY
/*
//LISTCNTL EXEC PGM=IDCAMS,REGION=1024K,
// COND=(4,LT,SCAN)
//INPUT DD DSN=*.SCAN.CONTROL,DISP=(OLD,PASS)
//OUTPUT DD SYSOUT=*
```

```

//SYSPRINT DD DUMMY
//*
//COPY EXEC PGM=EHS3COPY,REGION=OM,PARM='POSIX(OFF)/',
// COND=(4,LT,SCAN)
//STEPLIB DD DISP=SHR,DSN=&QALOAD.
//REPOSIN DD DISP=SHR,DSN=&REPOIN.
//REPOSOUT DD DISP=(NEW,CATLG,DELETE),
// DSN=&REPOOUT.,
// SPACE=(CYL,(1,1),RLSE),
// DSORG=PS,RECFM=VB,LRECL=32756,BLKSIZE=32760
//*SYSUDUMP DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//CONTROL DD SYSOUT=*,DSORG=PS,RECFM=FB,LRECL=80
//SYSIN DD DSN=*.SCAN.CONTROL,DISP=(OLD,PASS)
// PEND
//* END of PROC definition
//*
//SAMPLE EXEC REPOCOPY,
// REPOIN='REPOSITI',
// REPOOUT='REPOSITO'
//*
/* THE 1ST EHS3COPY (SCAN) IS SCANNING MESSAGES
/* THE 2ND EHS3COPY (COPY) IS SELECT/COPY
/*
//SCAN.SYSIN DD *
CONTROL COPY(NO) EXIT(EHS3SXC) EXITTYPE(C)
/*
/* SCANINFO file for user exit program EHS3SXC
/* Search strings (max length of 75) are provided
/* in this file prefixed by SCAN=
/* examples:
/* SCAN=USERAB
/* SCAN=green chev
/* SELECT_ALL in lieu of SCAN= will select all
//SCAN.SCANINFO DD *
SCAN=greenchev
//LISTRPT.SYSIN DD *
REPRO INFILE(INPUT) OUTFILE(OUTPUT)
//LISTCNTL.SYSIN DD *
REPRO INFILE(INPUT) OUTFILE(OUTPUT)
//

```

## Chapter 4.

# Guidelines for Successful Stress Testing

The last and perhaps most important step in a testing process is stress testing. Performing a stress test is a delicate and complex process that can consume people, system resources, and time. Through careful planning and using the proper tools, organizations can successfully perform consistent, repeatable stress testing to help ensure ongoing application and system quality.

---

## Planning the Stress Test

Because of the extensive resources required to perform a stress test, comprehensive planning is crucial. Do not proceed with testing until you have a test plan in place that has been approved by everyone, and you have acquired all of the necessary resources.

### Create a Test Plan

Solid planning up front helps you make the best use of your testing product. Be sure your test plan meets the following objectives:

- States your testing objectives.
- States the expected results.
- Specifies who is responsible for each phase of the test.
- Specifies what monitoring tools will be used.
- Specifies who will set up the testing environment.
- Specifies who will restore the testing environment after the test.
- Specifies the applications that will be used to perform the test.
- Specifies when the test is complete.
- Specifies who will analyze the test results.

Take into account that all components of the operating system, hardware, and application will be used in the test. To perform a DB2 stress test, for example, the application must be executed, the CICS or IMS regions must dispatch the work, and MVS must dispatch the CICS or IMS regions. Because any or all of these components could fail as a result of the test, include in your plan the appropriate technical specialists for these areas to help identify failures, any adverse effects on performance, and to resolve problems.

Stress tests can affect other applications executing in the environment. To ensure that results of the stress test can be measured while preserving the integrity of other applications, execute stress tests on evenings and weekends, or at other times when production activity is low.

### Sample Stress Test Plan

#### *State your testing objectives*

A customer has decided to simulate 800 users to a single CICS region.

*Test Description*

The accounting database will be exercised to determine how much data can be processed before response time is affected. Account add programs ACCT1234 and ACCT1235 will be exercised. The database can handle the current production load of 500 adds per minute, so the stress test begins at this point. This baseline test establishes baseline measurements from which future test results can be tracked. Each subsequent test increases the number of adds per minute by 50 until database response time diminishes to below the acceptable level of 1.0 seconds per transaction.

*Number of Hiperstation Batch Jobs Involved*

Prior tests indicate that each execution of the Hiperstation unattended batch (UAPB) job can drive 500 transactions per minute, which results in 300 accounting adds per minute. The test begins with two UAPB jobs until a rate of 600 accounting adds per minute has been achieved. At this point, a third UAPB job is added. These UAPB jobs execute on MVSPROD, which removes Hiperstation overhead from stress test statistics.

*Number of Tests*

The test begins with two UAPB jobs until a rate of 600 accounting adds per minute is achieved. At this point, a third UAPB job is added. These jobs execute on MVSPROD, which again removes Hiperstation overhead.

***State the Expected Test Results***

Test results are measured during each test execution. Measurements include:

- MVS paging rate
- CPU use
- CCS paging rate
- OSCORE use
- number of times a MAX and AMAX are reached
- transaction response time

Database measurements include:

- response time
- number of deadlocks
- number of deadly embraces

Channel busy and channel contention can also be measured.

***Specify the applications that will be used to perform the test***

The tests will execute in MVSTEST, CICS test region 1 using the ACCTADD database. CICS test region 1 should be configured to handle 1000 active transactions with a maximum of 1020. Storage size and OSCORE should reflect current production definition. Virtual terminal definitions for Hiperstation for VTAM should be added to VTAM and CICS test region 1. The ACCTADD database should be defined with enough threads to handle 600 concurrent users.

***Specify who is responsible for each phase of the test***

Each test begins at 8:00 p.m. and ends no later than 6:00 a.m. on Saturday and Sunday. One hour before and after the test is dedicated to setting up and restoring the testing environment.

***Specify who will set up the testing environment and restore it after the test***

The following staff is required to attend each test execution:

- MVS Specialist
- CICS Specialist

- VTAM Specialist
- Accounting Representative
- DBA Specialist
- Quality Assurance
- Test Manager

### *Specify when the test is complete*

Testing ends when response time drops below 1.0 seconds per transaction, or when 900 accounting adds per minute is achieved.

## Set Up the Test

Establish the testing environment to alleviate bottlenecks and skewing of test results. Take CICS tasks and storage, database threads, and VTAM storage into consideration. Also, because Hiperstation for VTAM uses virtual terminals to execute multiple terminal activity, the definitions for these terminals are required in VTAM and in the CICS region. These definitions should reflect the terminal type normally used in production. A database or file restore from a baseline is also required.

To set up a WebSphere MQ test, clear the queues accessed by the CICS region, and ensure that they are not available to applications other than Hiperstation for WebSphere MQ.

### *Overhead*

The main objective of stress testing is to load the system with enough data to simulate application performance requirements. Turn off Hiperstation parameters, such as comparing results and dubbing, that are not required during stress testing. This reduces overhead and frees Hiperstation to adequately stress system components.

Hiperstation for VTAM and Hiperstation for Mainframe Servers' APPC functions use the unattended playback batch job EHSBATCH to perform stress testing. Whenever possible, separate EHSBATCH from the environment being tested. This might involve executing EHSBATCH on a different CPU or LPAR. Removing the job from the environment removes any overhead that EHSBATCH incurs in terms of storage and CPU usage. It also allows a separate MVS to dispatch EHSBATCH tasks and manage the presentation of VTAM messages. The resulting analysis provides a true picture of environment performance.

If you are performing a WebSphere MQ test with Hiperstation for WebSphere MQ or TCP/IP tests with Hiperstation for Mainframe Servers, execute program TCPPBMN. This program normally runs on the same system as the CICS region or Web application being tested. You can run TCPPBMN on a different LPAR to reduce contention for resources.

### *Priority*

MVS dispatches work processed by CPUs based on priority. In most data processing environments, batch jobs have the lowest priority of all work on the machine because they are long-running tasks and use large amounts of resources. Because Hiperstation performs stress testing using EHSBATCH, the systems programmer must ensure its dispatching priority is immediately below VTAM.

TCPPBMN needs to be included with EHSBATCH as a named program. Region size should be 0M.

### *Region Size*

Set the region size for EHSBATCH to 0M. This allows MVS to give EHSBATCH as much storage as needed to perform the test.

*Multiple EHSBATCH Jobs*

If one EHSBATCH job does not simulate enough virtual terminals to stress the system enough, start more EHSBATCH jobs until the number of virtual terminal simulations is adequate.

*Multiple TCPPBMN Jobs*

Often an MQGROUP statement can be used to simulate a single WebSphere MQ user. If COUNT is used on the MQGROUP statement, then the value associated with COUNT indicates the number of users that TCPPBMN will simulate. For example, COUNT(10) simulates ten users. The maximum number of MQGROUP statements that TCPPBMN can play is approximately 400.

Often a SOCKETS statement can be used to simulate a single TCP/IP user. If the COUNT statement appears on the SOCKETS statement, then the value associated with COUNT indicates the number of users that TCPPBMN will simulate. For example, COUNT(10) simulates ten users. The maximum number of SOCKETS that TCPPBMN can play is approximately 400.

If modifications to the test scripts are required, make the changes before you use the script in a stress test.

## Execute the Test

When you run one huge stress test that includes all terminals, applications, queues, and databases, the large number of components makes it difficult to determine causes if the test fails. Not knowing the exact cause of the problem could lead to improper decisions based on the stress test results.

Start your stress test with a small number of virtual terminals within one EHSBATCH job. This determines if the test plan executes well and the test environment is set up properly.

Start your WebSphere MQ or TCP/IP stress test with a small number of messages within the TCPPBMN job. Confirm that the test environment is set up properly, and that a single script can be played multiple times. Often, in WebSphere MQ messages, one or more fields make that message unique (for example, the MsgID field in the message descriptor).

*Work from Small to Large*

As each test completes successfully, incrementally increase the number of virtual terminals/messages or TCP connections and retest. Change the number of simulated users by changing the TERM parameter (for 3270 playbacks) or the COUNT parameter (for WebSphere MQ and TCP/IP playbacks). You may need to start an additional job to obtain increased volume.

*Journals*

Do not invoke journals. When performing stress tests, the main objective is to lead the system according to requirements. Writing the results of the playback to external files could prevent Hiperstation from sending data quickly enough to stress a system component.

*Caching*

Turn caching on. Caching reduces storage and overhead used by Hiperstation and makes storage and CPU cycles available for stress testing.

---

## Using the Stress Test Results

The VTAM Playback Summary Report lets you view the results of your stress test, such as number of transactions, response time, think time, and comparison checks. The success or failure of the test should tie back to the definition of the completed test as defined in the test plan. Results can be used in a number of ways:

- to determine performance levels
- to reveal unexpected results
- to make improvements

For WebSphere MQ and TCP/IP playbacks, use one or more COLLECT statements to gather pertinent information. Normally these are transaction or message counts. The COLLECT statements direct TCPPBMN to create a file containing the statistics requested by the COLLECT statements. This file can be used to generate reports.

### Determine Performance Levels

For example, a number of stress tests are performed on a DB2 database to identify the number of DB2 calls the database can handle before performance degrades.

The results of each test can be posted to a spreadsheet that monitors the number of DB2 calls, number of transactions, and transaction response times. This collective information can be used to perform multiple stress tests in which the number of calls is raised incrementally with each test until transaction performance begins to suffer. These results can then be included in the stress testing documentation as well as forwarded to management for review.

### Reveal Unexpected Results

Because stress testing can exercise multiple components, it is important that you have a way to identify specific areas that fail as a result of the stress test. In the previous DB2 example, increasing the number of database calls requires that you increase the number of CICS transactions. This puts additional stress not only on the database, but also on CICS and MVS, which supports the CICS region. A situation could arise where the CICS region has problems handling the additional volume, which would prohibit the increased database calls from being presented to the database. Stress testing often yields unexpected results, and these results should be documented and presented as part of the overall stress testing results.

### Make Improvements

Stress test results often indicate bottlenecks or areas that contributed to the failure of a stress test. This information can be used as the starting point for a plan of action for areas that need improvement. The plan might also indicate who owns these non-compliant areas, and who is responsible for correcting them. The results of the stress test and any action plans can be used to prepare for the next stress test and to compare test results to ensure there is no regression.





# Index

## Numerics

- 3270 repository filtering and copy utility, 3-1
  - control statement, 3-1
  - JCL examples, 3-6
  - prepare JCL, 3-6
  - tips for user exits, 3-4
  - user-written exit programs, 3-3
- 3270 variables, 1-9

## A

- accessibility
  - font and font size, xiii
  - keyboard shortcuts, xiv
  - windows accessibility features, xiii
- ACTUAL variables, 1-19, 1-21
- ACTUAL\_COMPCODE variable, 1-20
- ACTUAL\_REASON variable
  - REXX variables, 1-44
- <ADDCALL> tag, 1-35
- Adobe Reader, ix
- ALLOC command, 1-35–1-36
- ALLOC return codes, 1-37
- ALLOCATE command, 1-24
- ALLOCATE statement, 1-14
- <ALTER> statement, 1-34
- analyze application screens, 1-27
- APPC
  - general REXX variables, 1-17
  - script data modification example, 1-38
  - See also* REXX variables
- APPC testing
  - REXX variables - application data, 1-14
  - REXX variables - environment data, 1-14
  - REXX variables, error determination, 1-13
- APPC\_ACTUAL variable, 1-15
- APPC\_ACTUAL\_LENGTH variable, 1-16
- APPC\_EXPECTED variable, 1-15
- APPC\_EXPECTED\_LENGTH variable, 1-16
- assignment variables, 1-6
- authoring exit programs, 3-6

## B

- build script statements using <EXECUTE>, 1-33

## C

- CALL parameter list, 2-2
- CALL statement, 1-8
- COMPRESULT variable, 1-10

- COLUMNS variable, 1-10
- command syntax, -xi
- common programming interface, 2-1
  - call interface, 2-1
  - COPY3270, 2-8
  - DD requirements, 2-2
  - EXECUTE, 2-5
  - HSCPI, 2-1
  - LOGOFF, 2-4
  - LOGON, 2-3
  - parameters, 2-2
  - RECEIVE, 2-6
  - REPL3270, 2-9
  - RETRIEVE, 2-7
  - REXX EXEC, sample, 2-9
  - SEND, 2-5
  - session IDs, 2-2
  - TYPE, 2-6
  - verbs, 2-2
- completion codes, 1-20
- Compuware FrontLine customer support website, xv
- CONNECTION\_NBR variable, 1-18
- constants, script language, 1-4
- <CONTENT> tag, 1-14
- control statement
  - 3270 repository filtering and copy utility, 3-1
- CONVCORR variable, 1-14
- COPY statement, 3-2
- COPY3270, common programming interface, 2-8
- correlation ID
  - replace during playback, 1-40
  - request queue, 1-46
  - return actual and expected, 1-21
- count variable, initialize, 1-43
- CPI (common programming interface), 2-1
- CPICRC variable, 1-13
- custom return codes, 1-8
- CYCLE variable, 1-10, 1-17

## D

- <DELCALL> tag, 1-35
- documentation, ix
  - master index, ix
- domain destination, 1-13
- domain traveler, 1-27

## E

- EHS3COPY
  - batch utility, 3-1
  - control statements, 3-1
- EHSBATCH jobs, 4-4
- environment data REXX variables, 1-14
- ERRMSGNO variable, 1-13
- error determination REXX variables, 1-13
- examples
  - EHS3COPY JCL, 3-6
  - EXECIO statement, 1-44
  - modify APPC script data, 1-38

- modify MQ script to test requestor, 1-40
- REPLACE tag, 1-46
- REXX EXEC, 2-9
- stress test plan, 4-1
- writing exit programs, 3-3
- EXECIO statement, 1-44
- <EXECUTE> statement, 1-33
- <EXECUTE> tag, 1-34
- EXECUTE verb, 2-5
- execution units, 1-3
- exit programs, writing, 3-3
- EXPECTED variable, 1-21
- expressions, script language, 1-5
- external file, read data from, 1-44
- external sources, retrieve data, 1-6

## F

- file I/O from a script, 1-24
- font and font size, accessibility, xiii
- FREE command, 1-35–1-36
- FrontLine, x, xv
- function calls, script language, 1-5

## G

- GET\_QUEUE\_NAME variable, 1-23
- GETVSAMR command, 1-26, 1-35–1-36
- GROUP statement, 1-27
- GRPCYCLE variable, 1-10, 1-17

## H

- help, Hiperstation customer support, xv
- HSCMDS command environment, 1-35
  - ALLOC command, 1-36
  - FREE command, 1-36
  - GETVSAMR command, 1-36
  - return codes, 1-37
- HSCPI function definition, 2-1
- HTML documentation files, x

## I

- <INBOUND> group of verbs, 1-14
- index, master, ix
- input group, 1-2
- inquiry program, test, 1-40
- installation requirements, 1-1

## K

- keyboard shortcuts, accessibility, xiv

## L

- language constructs, 1-4
- LASTAID variable, 1-10
- LASTKEY variable, 1-11
- LEAVE statement, 1-4
- LOGMODE variable, 1-14
- LOGOFF common programming interface, 2-4
- LOGON common programming interface, 2-3
- Logon CPI verb return codes, 2-4
- LUNAME variable, 1-11
- LUWINST variable, 1-14
- LUWNAME variable, 1-14
- LUWSEQ variable, 1-14

## M

- message ID, return expected and actual, 1-21
- MQ playback, REXX variables, 1-20
- MQ script application flow, 1-40
- MQ, REXX variables, 1-20
- MQGROUP statement, 4-4
- MQGROUP\_COUNT\_NBR variable, 1-23
- MQGROUP\_REPEAT\_NBR variable, 1-23

## N

- notation conventions, screens and commands, x

## O

- OBJECT\_ID variable, 1-22
- operators, valid script language, 1-5
- OPNRTNCD variable, 1-14
- <OUTBOUND> group of verbs, 1-14
- output group, 1-2
- OUTPUTNUM variable, 1-11

## P

- parameters, common programming interface, 2-2
- playback, APPC REXX variables, 1-14
- PORT variable, 1-11, 1-17
- predefined REXX variables, 1-17
- PUT\_QUEUE\_NAME variable, 1-23

## R

- reason codes, return expected and actual, 1-22
- RECEIVE, common programming interface, 2-6
- RECEIVER variable, 1-14
- REPL3270, common programming interface, 2-9
- REPLACE tag, example of, 1-46
- REPLAY\_ID variable, 1-18, 1-22
- request parameter list (RPL), 1-13
- retrieve screen data, 1-6

RETRIEVE, common programming interface, 2-7

return codes

- custom, 1-8
- Logon CPI verb, 2-4

REXX

- analyze application screens, 1-27
- Log, 1-27
- sample REXX EXEC, 2-9
- statement of compliance, 1-1

REXX statements

- ADDRESS, 1-35
- EXIT, 1-9
- IF, 1-9
- LEAVE, 1-4
- RETURN, 1-9
- SAY, 1-8–1-9, 1-14
- SELECT, 1-4

REXX variables, 1-9, 1-24

- 3270, 1-9
- ACTUAL variables, 1-19, 1-21
- ACTUAL\_COMPCODE variable, 1-20
- APPC application data, 1-14
- APPC environment data, 1-14
- APPC error determination, 1-13
- APPC\_ACTUAL, 1-15
- APPC\_ACTUAL\_LENGTH, 1-16
- APPC\_EXPECTED, 1-15
- APPC\_EXPECTED\_LENGTH, 1-16
- COMPRESULT, 1-10
- COLUMNS, 1-10
- CONNECTION\_NBR, 1-18
- CYCLE, 1-10, 1-17
- EXPECTED variable, 1-21
- general variables, APPC, 1-17
- GET\_QUEUE\_NAME, 1-23
- GRPCYCLE, 1-10, 1-17
- Hiperstation predefined variables, 1-2
- LASTAID, 1-10
- LASTKEY, 1-11
- LUNAME, 1-11
- MQ, 1-20
- MQGROUP\_COUNT\_NBR, 1-23
- MQGROUP\_REPEAT\_NBR, 1-23
- OBJECT\_ID, 1-22
- OUTPUTNUM, 1-11
- perform file I/O from a script, 1-24
- PORT, 1-11, 1-17
- predefined, 1-9, 1-17, 1-20
- PUT\_QUEUE\_NAME, 1-23
- REPLAY\_ID, 1-18, 1-22
- ROWS, 1-11
- SCREEN, 1-11
- SCREENEXPECT, 1-12
- SCREENEXPECT\_SIZE, 1-12
- SCREENIN, 1-11
- SCREENSIZE, 1-12
- SCREENXATR, 1-12
- SCRIPT\_REPEAT\_NBR, 1-19, 1-23
- SOCKETS\_COUNT\_NBR, 1-18
- SOCKETS\_REPEAT\_NBR, 1-18
- TCP/IP variables, 1-17
- TCPIP\_ACTUAL, 1-17
- TCPIP\_ACTUAL\_LENGTH, 1-18
- TCPIP\_EXPECTED, 1-17
- TCPIP\_EXPECTED\_LENGTH, 1-18
- TPFNAME, 1-13
- use as input to the screen, 1-24
- use logic to control script flow, 1-27

RLOG, 1-27

ROWS variable, 1-11

RPL6CCST variable, 1-13

RPL6RCSC variable, 1-13

RPL6RPCPR variable, 1-13

RPL6SNSI variable, 1-13

RPL6WHAT variable, 1-13

RPLFBK2 variable, 1-13

RPLRTNCD variable, 1-13

## S

SAY statement, 1-27

SCREEN variable, 1-11, 1-27

SCREENEXPECT variable, 1-12

SCREENEXPECT\_SIZE variable, 1-12

SCREENIN variable, 1-11

SCREENSIZE variable, 1-12

SCREENXATR variable, 1-12

script editing examples

- test MQ requester application, 1-40

script language, 1-1

- ALLOC command, 1-36
- ALTER statements, 1-34
- assignment, 1-6
- build statements, 1-33
- capabilities, 1-1
- comments, 1-6
- compliance with REXX, 1-1
- conditional statements, 1-4
- constants, 1-4
- custom return codes, 1-8
- expressions, 1-5
- FREE command, 1-36
- function calls, 1-5
- GETVSAMR command, 1-36
- HSCMDS command environment, 1-35
- iterative statements, 1-3
- language statements, 1-7
- modular scripts, 1-7
- multiple transaction statements, 1-3
- operators, 1-5
- retrieve data from external sources, 1-6
- script reports, 1-8
- statements, 1-2
- terminal input substitution, 1-7
- variables, 1-4

SCRIPT\_REPEAT\_NBR variable, 1-19, 1-23

scripts

- modify APPC script data, 1-38

SECUSER variable, 1-14

SELECT statement, 1-4

selection utility, 3270, 3-1

<SEND\_DATA> tag, 1-14

<SEND\_EXPEDITED\_DATA> tag, 1-14

SEND, common programming interface, 2-5

SENDER variable, 1-14

session IDs, common programming interface, 2-2

SOCKETS replay, 1-18

SOCKETS statement, 4-4

SOCKETS\_COUNT\_NBR variable, 1-18

SOCKETS\_REPEAT\_NBR variable, 1-18

stress test

- caching, 4-4
- create test plan, 4-1
- execute, 4-4

- guidelines, 4-1
- improvements, 4-5
- journals, 4-4
- multiple terminal sessions, 1-1
- objectives, 4-1
- performance levels, 4-5
- planning, 4-1
- region size, 4-3
- results, 4-5
- sample, 4-1
- See also* test
- set up, 4-3
- unexpected results, 4-5

string concatenation, 1-7

syntax

- conventions, xi
- diagrams, xi

## T

TCP/IP

- REXX variables, 1-17
- See also* REXX variables

TCPIP\_ACTUAL variable, 1-17

TCPIP\_ACTUAL\_LENGTH variable, 1-18

TCPIP\_EXPECTED variable, 1-17

TCPIP\_EXPECTED\_LENGTH variable, 1-18

TCPPBMN jobs, 4-4

test

- inquiry program, 1-40
- script modifications, 4-4
- See also* stress test
- set up, 4-3

tokens, script language, 1-4

- assignment, 1-6
- constants, 1-4
- expressions, 1-5
- function calls, 1-5
- operators, 1-5
- variables, 1-4

TPFNAME variable, 1-13

TPNAME variable, 1-14

TYPE, common programming interface, 2-6

## U

utility, filter and copy, 3-1

## V

variables

- initialize count variables, 1-43
- REXX, 1-24
- script language, 1-4
- See also* REXX variables
- substitution, 1-24
- terminal input substitution, 1-7

verbs, common programming interface, 2-2

VSAM-GETVSAMR, 1-26

VTAM Playback Summary Report, 4-5