™

REXX For PIE/CICS User Guide

Release 3.3.0

SYSTEMS, INC.

Document Number PIEPR0400-01

First Edition printed April 2000

This manual applies to PIE/CICS release 3.3.0 and to all subsequent releases of the product until otherwise indicated by new editions or updates to this publication.

UNICOM Systems, Inc.
UNICOM Plaza Suite 310
15535 San Fernando Mission Blvd.
Mission Hills, California 91345

Contents

About This Manual

This manual describes PIE/REXX, a high-level programming language available with Release 3.3.0 of PIE/CICS. This manual explains REXX programming concepts, syntax, and the elements of the language.

Audience

This book is intended for system administrators and end-users of PIE/CICS. Readers are expected to understand basic programming concepts of high-level languages. Specific prior knowledge of REXX programming techniques is not required.

How This Manual is Organized

This manual consists of three chapters and two appendixes. Listed below are the titles and a brief description of each chapter and appendix.

- Chapter 1 Introduction

  Explains the features of PIE/REXX and gives an overview of the process to write, debug, and run a program.
- Chapter 2 Basics of REXX Programming

  Describes the concepts of writing a PIE/REXX program.
- Chapter 3 PIE/REXX Commands and Functions

  Lists PIE/REXX keywords, functions, external routines, and special host commands.
- Appendix A Customer Service

  Describes procedures to report problems with PIE/CICS to UNICOM Systems, Inc. Customer Service.
- Appendix B Sample PIE/REXX Programs

  Describes several PIE/REXX programs included in the PIE/CICS SAMPLIB dataset.

Recommended Reading

The title and a brief description of all PIE/CICS manuals are shown in the following lists. Some manuals provide common information that applies to both the common and optional components of PIE/CICS. Other manuals pertain only to optional PIE/CICS components. These manuals need to be read only if these products are part of the PIE/CICS system installed at your site.

Common Manuals

These manuals provide common information that applies to both the shared and optional components of the PIE/CICS family.

- *PIE/CICS Installation Guide*

  Includes a series of procedures to install PIE/CICS.
- *PIE/CICS Release Notes*

  Describes new features or enhancements to PIE/CICS that are part of Release 3.3.0.
- *PIE/CICS Command Reference*

  Describes PIE/CICS Application and Environment commands.
- *PIE/CICS Customization Guide*

  Describes common procedures to adapt PIE/CICS to your site's requirements.
- *PIE/CICS Operation and Administration Guide*

  Describes common features or facilities that are available to all PIE/CICS products. Performance tuning techniques and implementing security also are described.
- *REXX for PIE/CICS User Guide*

  Describes how to write, compile, and execute SAA-compliant REXX programs that operate within a PIE/CICS external environment.
- *PIE/CICS Custom Menus Administration Guide*

  Describes how to create custom MultiCICS and Dynamic Menu screens that provide alternate language support.

Optional Manuals

These manuals describe optional PIE/CICS components.

- *PIE/CICS MultiCICS Administration Guide*

  Provides customization procedures and usage information to support multiple PIE/CICS sessions with MultiCICS.

- *PIE/CICS Dynamic Menus Administration Guide*

  Describes how to create custom PIE/CICS menus that provide extended security and enhanced transaction processing.

- *PIE/CICS NetGate Administration Guide*

  Explains how to access multiple VTAM applications through a PIE/CICS session with NetGate.

- *PIE/CICS NetMizer Administration Guide*

  Describes how to use NetMizer to optimize 3270-based data streams.

- *PIE/CICS Availability Plus Administration Guide*

  Explains how to use Availability Plus to distribute and balance work across multiple CICS regions.

- *PIE/CICS NonStop CICS Administration Guide*

  Describes how to use NonStop CICS to route work across CICS regions to balance the workload and minimize down time in the event of a region failure.

Syntax Conventions

A syntax diagram is included with each PIE/CICS command described in this manual. A syntax diagram shows the possible parameters, values, and variables associated with a command.

Syntax diagrams adhere to common conventions. The physical appearance of a diagram's elements indicates whether a command parameter, variable, or other values are required, optional, or included by default.

- An <u>underlined</u> parameter is the default assigned to the command.
- Command names are presented in MIXed case. The uppercase portion of a command name is the requisite abbreviated form. Lowercase letters represent the optional remainder of the command name that need not be specified to execute the command.
- An *italicized lowercase* parameter represents a value assigned by the user.
- A vertical bar (|) separates two or more mutually exclusive parameter values. Only one value can be specified for each parameter.
- Parameters enclosed within brackets [ ] are optional. Only one value can be specified to a parameter.
- Parameters values enclosed within braces { } are required. If unspecified, the parameter default is assigned to the command.
- `Monospace` type indicates a screen field or an example of a PIE/CICS command entered on the screen.

Chapter 1 Introduction

PIE/REXX is a high-level programming language integrated with PIE/CICS. You can write REXX programs that interface with both PIE/CICS and CICS.

PIE/REXX programs can extend the capability of your PIE/CICS system. For example, you can write PIE/REXX programs that allow you to:

- Retrieve and update information from any PIE/CICS session screens. You can perform enhanced cut and paste operations with REXX scripts that automatically cut and paste data between the fields of standardized screens.
- Start a REXX program by making it an option that users select from a menu.
- Start a REXX script automatically when you switch to another session
- Automate repetitive end-user procedures with a PIE/REXX script.
- Create a program that executes initial log on processing.

PIE/REXX programs can be executed automatically, manually, as scripts, or with standard PIE Application commands. Programs can also be started by assigning them to function keys or menu lines on a PIE/CICS screen.

REXX is regarded as a robust language that is easy to learn and use. PIE/REXX is no exception. Business users should be able to learn the language fairly quickly. Users who are familiar with REXX on other platforms should be able to adapt to PIE/REXX without difficulty. There are few differences between PIE/REXX and other versions of the language.

PIE/REXX provides a consistent, complete development environment. You can write, compile, test, and debug your programs within a PIE/CICS EDIT session. You do not have to use other applications during the process of coding, debugging, and verifying a program.

Features of PIE/REXX

PIE/REXX is compliant with IBM's SAA REXX. It is designed to be extensible with more commands and functions anticipated in future releases of PIE/CICS.

PIE/REXX's external environment is PIE/CICS. You can execute any PIE command from a PIE/REXX program. These commands include the Application commands CMD and TRAN, which execute CICS transactions, and use PIE/CICS variables. (You cannot use the PIE/CICS variable ZPSWD, for security reasons. Use the SCR_PUT_PASSWORD command instead.)

This version of PIE/REXX distributed with Release 3.3.0 of PIE/CICS supports the majority of SAA REXX keywords and built-in functions. It also provides specialized external functions to work specifically in the PIE/CICS environment. These specialized external functions allow you to manage PIE/CICS sessions and menus. Refer to " External Routines" on page 94 for more information.

PIE/REXX is resource efficient. It does not consume system resources waiting for terminal input from the user. It is pseudo-conversational without requiring special programming techniques. Also, PIE/REXX is compiled before it is executed and therefore executes more quickly than an interpreted language.

This release of PIE/REXX has the following deviations from SAA REXX standards:

- Only arithmetic integers are supported. Decimal numbers and numbers expressed in scientific notation are not supported.
- Only the external functions supplied with PIE/REXX are supported. You cannot write your own external functions.
- Compound variables are not supported.
- Handling abnormal conditions is not implemented for the SIGNAL keyword.

Installation and Customization

PIE/REXX is installed and customized as part of the normal PIE/CICS installation procedures. No additional procedures are required. If you are upgrading from a previous release of PIE/CICS, refer to the *PIE/CICS Release Notes* for any additional upgrade information that may pertain to PIE/REXX.

Creating PIE/REXX Programs

You create PIE/REXX programs with the PIE/CICS EDIT facility. To access the EDIT facility from the PIE/CICS Sessions menu, enter the following command:

```
EDIT [groupname.]progname
```

PIE EDIT is similar to ISPF EDIT and supports many of the same functions such as delete line (d or dd), insert line (i or i#), copy line (c or cc and a or b), and end (PF3).

PIE/REXX programs must be stored in the PIE/CICS Repository as text objects. They are stored there automatically when you create them with PIE EDIT. Also, you can upload REXX programs created in other environments to the Repository using the Repository Load Utility.

You can list the PIE/REXX programs stored in the Repository with the LIST TEXT command. You can edit, rename, or delete your programs from the Text List facility.

By default, the Text List facility displays all text objects stored in the Repository. You should establish a consistent naming convention to easily identify your PIE/REXX programs. For example, place them in a special group called REXX.

☞ You can batch compile PIE/REXX programs from a PDS. See “ Compiling a Program as a Batch Job” on page 9, for more information.

Refer to the following manuals for more information about these commands:

PIEEXEC	*PIE/CICS Command Reference*
EDIT	*PIE/CICS Operation and Administration Guide*, See page 67, “Text Utility”.” and the EDIT Help panel
LIST	*PIE/CICS Operation and Administration Guide*, See page 41, “List Utilities”.” and the LIST Help panel
Repository Load Utility	*PIE/CICS Operation and Administration Guide*, “Chapter 7 Repository Load Utility” on page 77".

PIE/REXX Debugging and Coding Tips

PIE/REXX programs are compiled and therefore operate differently than other versions of REXX that interpret program statements as they are executed. PIE/REXX does syntax checking when a program is compiled. All statements are compiled, including those coded after a final EXIT instruction.

- By default, PIE/REXX issues error messages in response to invalid program statements. Use the TRACE command to activate tracing and select the range of statements that the failing statement may be located.

  If there is a syntax error, a message is issued, and the program does not execute. You do not see trace statements for a program that failed because of syntax errors.
- Commenting out lines from a program can cause errors if these lines contain comment delimiters. See " Comments" on page 14.
- Compound variables are interpreted as simple variables. Numbers expressed as decimals or in scientific notation are interpreted as character strings, never as numbers.
- Place literal arguments in quotes to avoid unintentional variable substitution with command arguments.
- The performance of a REXX program improves if all literals are enclosed with quotes because PIE/REXX does not need to perform a variable check before executing the program.
- When several PIE commands execute sequentially within a PIE/REXX program, performance can be improved by *stacking* them; coding them as a single PIE command separated by delimiters. Use a unique command delimiter that is different than the PIE/CICS delimiter.
- PIE/REXX programs run within their own PIE session. Even when you execute a PIE/REXX program from a PF key, PIE/CICS opens a new session to run the program. A free session must be available to run a PIE/REXX program.

Executing a PIE/REXX Program

The REXX PIE Application command executes a PIE/REXX program. Batch-compiled programs are CICS programs that adhere to standard CICS conventions. See “ Compiling a Program as a Batch Job” on page 9 for more information.

If a program has been changed after being compiled, the REXX command automatically recompiles the program. You can request a program compilation by specifying either the REFRESH or COMPILE parameters with the REXX command. Compiling adds the program to the in-core object library and deletes any previous copy. PIE/REXX checks for syntax errors when it compiles a program.

REXX Command Syntax

The syntax of the REXX command is:

```
                                 |([SCRIPT] [REFRESH] [DEBUG]|
REXX [groupid.]progid args      {          (COMPILE          }
                                 |          (DELETE          |
```

groupid	PIE/REXX program group name. If omitted, PIE/REXX uses the standard hierarchical search: user ID, user’s group ID, and finally SYSTEM is assigned by default if a groupid is not found.
progid	Object name of the PIE/REXX program.
args	Optional arguments passed to the PIE/REXX program. The arguments can be multiple words, with a maximum string length of 256. This string can be retrieved with the ARG or PARSE keywords.
SCRIPT	Request to run the PIE/REXX program in script mode. In script mode, a PIE/REXX program can execute and interact with an application. If you omit SCRIPT, the program runs in standard mode. See “ Running REXX Programs as Scripts” on page 9 for more information.
REFRESH	Request to re-compile a PIE/REXX program and execute it.
DEBUG	Request to debug a REXX program and send diagnostic information to PIELOG. To set up PIELOG, see the *PIE/CICS Installation Guide*, “ Update the DCT” on page 10. This option is intended to be used with support from UNICOM Systems, Inc. Customer Service.
COMPILE	Request to compile a PIE/REXX program, but not run it.
DELETE	Request to delete a PIE/REXX program from memory. To delete the program from the Repository, go to the Text List and issue the D line command against the REXX program. See “Chapter 6 Text Utility” on page 67 of the *PIE/CICS Operation and Administration Guide*.

Examples of Starting a Program with the REXX Command

REXX is a PIE Application command. It executes as any other PIE/CICS Application command; from a menu or a PF key, from the CICS blank screen or a session command line using the escape string, or from a program using the PIEEXEC API.

A REXX program runs like a transaction and requires a free session. To run a REXX program in the special session, precede the command with ESCAPE. See the *PIE/CICS Command Reference* for more information about executing PIE Application commands and the PIEEXEC API.

- `REXX START`

  The START program executes from the default group in standard mode.
- `REXX START (SCRIPT`

  The START program executes from the default group in script mode.
- `REXX GRP7.START`

  The START program executes from GRP7 in standard mode.
- `REXX START (DELETE`

  The START program is deleted from memory
- `REXX START ACCNTREC`

  ACCNTREC is passed as an argument of the START program.

Examples of Starting a Program by Other Methods

- `==REXX CUT.REXX`
  Execute the CUT program from a MultiCICS session by entering an escape string before the REXX command:
- `SET PF01 'REXX CUT.REXX'`
  Assign the CUT program to a PF key in the user's profile. Assigning keys is done from the Keys screen of the PIE/CICS Profile utility.

-
  ```
  MVC  COMMAREA,CMD1
    C CICS XCTL PROGRAM('PEXEC') COMMAREA(COMMAREA)
                    LENGTH(=H'80') NOHANDLE

    ...
    CMD1 DC CL80'REXX CUT.REXX'
  ```
  Use the PIEEXEC API, XCTL to PROGRAM('PEXEC'), passing it the REXX command to execute the PIE/REXX program. For example, to execute the CUT program:

  `EXEProgram Updates and Compiles`

Pseudo Compiled REXX Programs

PIE/REXX programs are compiled for greater efficiency. They can be pseudo-compiled on their first execution or compiled as a batch job.

Before starting a program with the REXX command, PIE/REXX verifies if the program has changed since its last compilation. If there have been changes, PIE/REXX automatically re-compiles the program. The compiled pseudo code is saved in memory.This is the default method. You can manually compile a program with either the REXX REFRESH or REXX COMPILE commands.

When a PIE/REXX program is recompiled, the existing pseudo code is replaced with the updated code. However, users that are currently executing an earlier version of the program continue to run with the old copy. Users who execute the program after the change will use the new compilation of the program.

After a program is started, PIE/REXX scans for older copies of the program and deletes them if they are no longer in use. Normally, only one copy of a PIE/REXX program is kept in memory.

Interacting With an Application

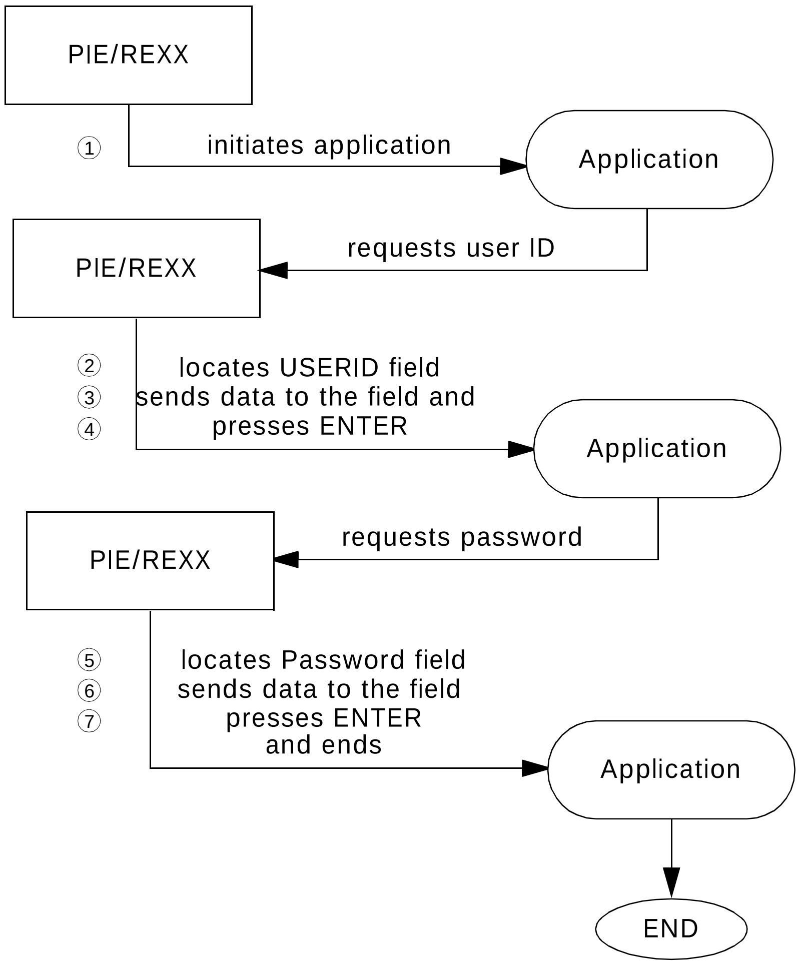
PIE/REXX programs can interact with an application pseudo-conversationally. It can search the session screen, place data into input fields, and then resume the application. Each time an application waits for terminal input, the PIE/REXX program regains control.

PIE/REXX programs receive and send data using a combination of PIE/REXX keywords and external functions. The special host command KEY sends a specified AID key (ENTER, PF1, etc.) to the application to simulate a key pressed at the terminal.

The following example shows a PIE/REXX program pseudo-conversing with an application.

```
   */ Sample PIE/REXX Program to sign on to TSO automatically */
①  OPTIONS 'SCRIPT=YES'
②  'ACCESS TS01'
③  call Scr_Find '*', 'ENTER USERID - ',,,'i'
   call Scr_Put_Field '*',ZUSERID
④  'KEY ENTER'
⑤  call Scr_Find '*', 'Password ====> ',,,'i'
⑥  call Scr_Put_Password '*'
⑦  'KEY ENTER'
```

The interaction between the PIE/REXX program and the application is shown in the following figure.


PIE/REXX
① initiates application
Application
requests user ID
PIE/REXX
② ③ ④ locates USERID field sends data to the field and presses ENTER
Application
requests password
PIE/REXX
⑤ ⑥ ⑦ locates Password field sends data to the field presses ENTER and ends
Application
END

PIE/CICS Escape String and Global Keys

During the period that a PIE/REXX program is running, the PIE escape string is disabled. If a user enters an escape string after the PARSE EXTERNAL, INPUT, DELAY, or WAIT commands, the escape string is simply regarded as a data string. PIE global keys are also disabled.

Users cannot switch out of a session while an application is running. In that case, you may want to define an AID key as a switching key. For example, you can set PF12 to switch to the Sessions menu. You could also check for a particular character string in a particular field.

There is a technique to release a user trapped in a never-ending script. If this occurs, and no PF key has been designated to switch out of the script, reset the user's log on using the User List utility. This returns the user to the Sessions menu. After that, cancel the session running the REXX script. Refer to " User List" on page 63 of the *PIE/CICS Operation and Administration Guide* for more information about the User List.

If no application is running, the escape string and PIE keys are processed by PIE/CICS as usual.

Compiling a Program as a Batch Job

Batch compiled programs become true CICS programs. You can XCTL to them with a commarea using the ARG and PARSE ARG commands to retrieve data. However, you cannot LINK to them. You must define these programs to CICS.

When you batch compile your PIE/REXX program, you can:

- Reduce the overhead of compiling a program on its first usage
- Secure programs as transactions using an external security system
- Write a REXX program to replace an existing transaction and continue to use the same tranid to execute the replacement.

☞ You cannot trace a batch compiled program. Also, error messages do not display the line number within the source code containing an error.

The following procedure describes the major steps to develop a PIE/REXX program with the batch compiler.

1. Create and test your PIE/REXX program using the PIE /CICS EDIT facility.

   Alternatively, you can create PIE/REXX programs using TSO EDIT. However, debugging may be more difficult because you cannot use the TRACE command and error messages do not list the program line numbers of the failing statements.
2. Load the program to a PDS using the Repository Load utility.
3. Edit and execute the JCL in member REXXCOMP of the CNTL dataset to compile the program.
4. Define the new program in CICS RDO. If you want to execute it using a transaction, define the transaction also.

Running REXX Programs as Scripts

A PIE/REXX program can execute in another application. Under normal operation (standard mode), the host application retains control until the REXX program ends; then control reverts to the REXX program. This is similar to the method used by REXX EXECs to initiate applications under TSO. The EXEC can do some processing before and after the application runs, but not while the application is running.

There are times, however, when you want a program to interact with an application. Sending data, receiving data, initiating functions, and issuing commands are examples of program-application interaction.

Program interaction is possible using PIE/REXX's script mode. In script mode, a PIE/REXX program can:

- Send and receive data between a REXX program and an application
- Send and receive data between the user and the REXX program
- Request processing in an application

You can use PIE/REXX's script mode to perform extensive application processing, programmatically. The PIE/REXX program can do anything a user can do from the terminal.Your PIE/REXX programs can automate any number of tasks to speed user processing.

Use the SCRIPT parameter of the REXX command to execute a PIE/REXX program in script mode. (See page 5.) Or place the command OPTIONS 'SCRIPT=YES' in your program. (See " OPTIONS" on page 42.)

User Interaction with Script Programs

Users cannot see the interaction between a PIE/REXX script and an application. PIE/REXX intercepts application output and places it into the session's screen buffer. When a PIE/REXX script searches and modifies the screen data, it does so using the buffer, not the screen itself.

You can let the user see what is happening by executing the SCR_RESHOW function, which sends the contents of the current screen buffer to the terminal. You can send a datastream with SCR_RESHOW to show the user that the PIE/REXX script is progressing normally. SCR_RESHOW is also useful in debugging PIE/REXX programs.

A PIE/REXX script can converse with users by the SAY and PARSE keywords. For example, the following program uses both commands to request a user ID be entered by the user.

```
/* Kill User */
say "Enter user to kill: "    /* Ask user to type in userid.    */
parse external USER .         /* Let user enter userid.         */
'list users' USER             /* Use LIST to see if the user
                                 is logged on.                  */
if ¬scr_find('*', "No user match") then do
                              /* If there's a match...          */
  call scr_find '*', '',,,'I'
                              /* Move cursor to 1st input field.*/
  call scr_find '*', '',,,'I' /* Move cursor to 2nd input field.*/
  call scr_put_field '*', 'F' /* Put 'F' beside userid.         */
   'KEY ENTER'                /* Simulate AID key being pressed.*/
   say 'RESPONSE:' scr_get_field('*', 4, 73)
   end
else
   say 'User' USER 'is not logged on.'
say 'Press ENTER to continue.'
'KEY PF3'                     /* End LIST COMMAND.              */
```

You can also use the INPUT, DELAY, and WAIT host commands to wait for input from the user or application. See " Special Host Commands" on page 118 for more information about these commands.

☞ User interaction during the execution of a REXX script should be kept to a minimum, because it undermines the purpose of automating a task. However, when interaction is necessary, these commands are available.

Use SCR_RESHOW, rather than SAY, to send a message that displays the progress of the script program to the user. SAY suspends the execution of the script with a *** prompt to allow the user to read the message. SCR_RESHOW does not affect REXX processing.

Returning from an Application

A PIE/REXX script starts only one application at a time. After a PIE/REXX program has started a CICS transaction or PIE Application command, it cannot execute other transactions or PIE Application commands until the first application ends. If an instruction attempts to do so, it will fail and the RC variable is set to -3.

Also, if a PIE/REXX program executes the KEY command when an application is not running, the command will fail and the RC variable is set to -3.

We suggest that you check the RC variable after invoking an application or executing the KEY command to ensure these commands have executed successfully.

Using Environment Commands with PIE/REXX Scripts

A PIE/REXX script can execute additional PIE Environment commands while an application is running. (Environment commands require "EC" or "ENV" before the command.) These commands are valid because they execute in a special PIE/CICS session, not in the session running the PIE/REXX script.

For example, if an application is running and the PIE/REXX program executes the statement

```
'EC SWITCH 12'
```

PIE/CICS immediately switches to session 12. The application and the PIE/REXX program are suspended until the user switches back to the REXX session. On return to the REXX session, the script resumes processing at the next instruction.

Environment commands can be very useful in combining multiple applications into one integrated, aggregate script. For example, you can start subordinate PIE/REXX scripts running separate applications in other sessions, look at the screen buffers in the other sessions, and copy data to or from these sessions.

Starting Scripts Automatically When Opening a Session

PIE/CICS provides the SWITCHR command to open a session and automatically start a supplied REXX script. Using SWITCHR, you can switch to a session and start the REXX script against an active application running in the session.

The format of SWITCHR is:

SWITCHR *session rexx_script rexx args*

session	Name or number of the session in which to execute the REXX script.
rexx_script	Name of the REXX script to start automatically when the session is opened.
rexx_args	Arguments passed to the REXX script that starts automatically in the session.

There are several considerations to use the SWITCHR command.

- You do not need to use the REXX OPTIONS keyword to identify the script. PIE assumes a script is always specified with the SWITCHR command.
- SWITCHR cannot be used to switch to a session that is currently running another REXX script.

Example Usage of SWITCHR Command

SPLIT and KEY are sample REXX scripts provided with PIE/CICS. These scripts allow the user to split the screen horizontally and run separate applications in each part. It does this by executing in its own session and using SWITCHR to control the two applications that it is splitting. This concept is expanding with the WINDOW sample which runs applications in windows within your 3270 screen. These REXX programs are supplied on an as is basis and are intended to be a help in creating your own REXX programs to do similar functions.

Other uses of using the SWITCHR command might include automating a repetitive task (e.g. performing some action to each member of a PDS under ISPF); also application integration can be done much more easily since you now no longer need to run the application under the control of REXX script for its complete duration. An example might be to add a PF key that initiates and navigate through an application to extract info (account number) it could then put that data into the current screen and navigate automatically to where the user wants to go next (so improving productivity).

Chapter 2 PIE/REXX Programming Basics

This chapter explains the fundamentals of PIE/REXX programming. No knowledge of SAA REXX is assumed. However, you may want to refer to other manuals, such as IBM's *Procedures Language MVS/REXX User's Guide* for more information about certain topics discussed in this chapter.

If you are familiar with SAA REXX, browse this chapter quickly or proceed immediately to "Chapter 3 Commands and Functions" on page 31."

PIE/REXX Program Elements

Tokens are the basic elements of PIE/REXX. They are similar to words and punctuation that compose a sentence; the individual components that together determine meaning of program statements.

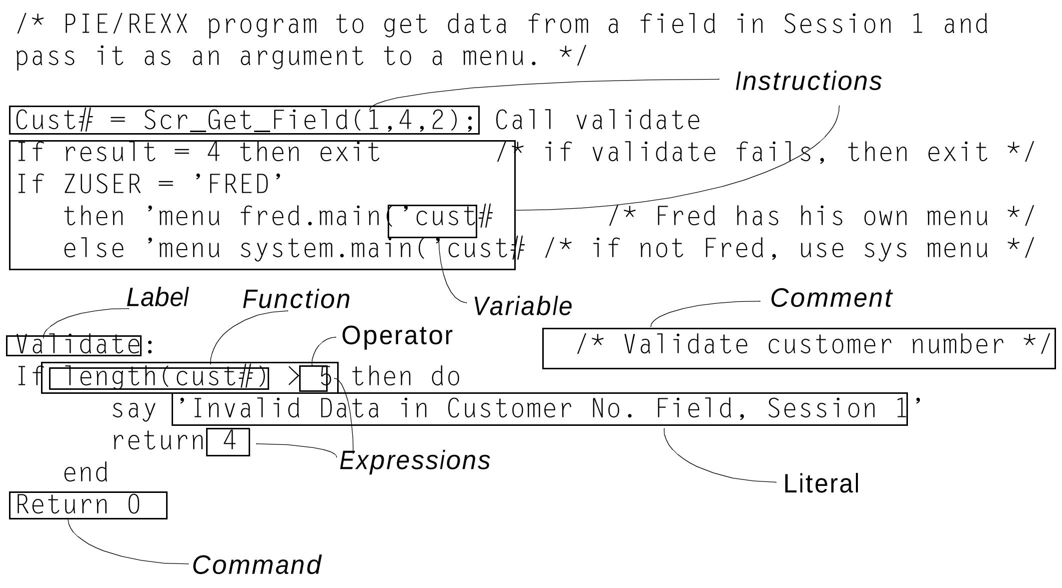
There are several types of REXX tokens:

- Comments are for clarity only. They are always ignored when the PIE/REXX program runs.
- Literals are constants that can be numbers, letters, or special characters. For example, "Hi there!", 234, and'E7'x are literals.
- Variables are names representing a value. For example, AMOUNT can stand for the number 395. The value of a variable can be a literal or the result of an operation or function. The type of data assigned to a variable can change from assignment to assignment.
- Operators (like +, -, =, >, ||) show the relationship between two tokens, for example, whether they should be added together or concatenated or compared.
- Commands perform special processing. They can be keywords (like CALL or SAY), functions (like Length(variable)), or host commands (like the PIE command MENU).
- Labels identify the beginning of a subroutine in your PIE/REXX program.

Tokens are combined to form expressions and instructions. An expression is a logical series of tokens that results in a string; for example, 4 + 2 is an expression that results in the string 6. An instruction performs an action, like assigning a variable or executing a command.

In the following sections of this chapter, each element of a REXX program is described in more detail.

Examples of Program Elements



```
/* PIE/REXX program to get data from a field in Session 1 and
pass it as an argument to a menu. */

Cust# = Scr_Get_Field(1,4,2); Call validate
If result = 4 then exit        /* if validate fails, then exit */
If ZUSER = 'FRED'
   then 'menu fred.main('cust#         /* Fred has his own menu */
   else 'menu system.main('cust# /* if not Fred, use sys menu */

Validate:                            /* Validate customer number */
 If length(cust#) > 5 then do
      say 'Invalid Data in Customer No. Field, Session 1'
      return 4
    end
Return 0
```

Comments

Comments can be entered anywhere within a PIE/REXX statement. A comment begins with /* and ends with */. For example:

```
   /* This is an example of a comment */
```

Comments are ignored during program execution. They do not affect operators, literals, or any other tokens. A comment can be any length.

PIE/REXX accepts nested comments. For example:

```
/* Primary comment /*nested comment */ with a nested comment */
```

Be careful when commenting out program lines that contain comment delimiters. For example, when you comment out

```
token, token /*comment*/
```

you must begin and end the line with comment delimiters:

```
/*token, token /*comment*/ */
```

Do not code the characters /* or */ in a comment, unless you have included a full nested comment, begun with /* and ended with */. If these characters are an essential part of your comment, you must separate them with at least one non-blank character.

☞ If comments are nested incorrectly, your program experiences unusual errors.

Literals

A literal is a constant whose value does not change during the execution of a program. A literal can be composed of characters or numbers.

Enclose character literals in quotes, ’ or ". For example:

```
'Enter customer number.'
"Enter customer number."
''    /* A null literal */
```

To code a single quote within single quotes, code two single quotes, ”. To code a double quote within double quotes, code two double quotes, "". For example:

```
'Five o''clock'
```

```
"PIE/REXX says, ""Hi!"""
```

Code an `X` or `x` after hexadecimal literals:

```
'08F9'X
"213B"x
'CC D3'x
"01 77 3D"X
```

Quotes are not required for a number literal. If you do put quotes around a number, you can code leading or trailing blanks. However, if you perform an arithmetic operation on a number literal with quotes, the result does not include leading or trailing blanks.

You can begin a number with a plus or minus sign.

```
365
'78543    '
-897650
"+354978654"
```

☞ PIE/REXX supports only integers between -2,147,483,648 and 2,147,483,647, inclusive. Numbers expressed as decimals, exponents, or with scientific notation are interpreted as character strings and are invalid for arithmetic operations.

The maximum length of a literal is 250 characters. Do not follow a literal with a left parenthesis, which indicates a function. For example, ‘ABC’ is incorrect.

Variables

Variables are names that stand for a value. The value can be a literal or the result of an expression. Variables may be up to 256 characters long. Their value is limited to 32,768 characters.

You can use any of the following characters in a variable:

A-Z, 0-9, @, #, $, ¢, !, ?, . and _ (underscore)

You can code a variable in both upper and lowercase. Differences in case are ignored. However, variables are translated to uppercase when the PIE/REXX program is executed.

Do not begin a variable with a number. For example, Month2 is a valid variable, but 2Month is not.

You can use PIE/CICS variables in PIE/REXX programs. Remove the ampersand (&) before the variable name; for example, code the PIE/CICS variable &ZUSER to simply ZUSER.

☞ PIE/REXX does not support compound variables; they are treated as simple variables. All other illegal variable names, including compound variables, are interpreted as literals. You cannot use the PIE/CICS variable ZPSWD, for security reasons. Use the SCR_PUT_PASSWORD external function instead.

Assigning Data to Variables

There are several ways to assign a value to a variable. The method you choose depends on how the assigned data is obtained.

If you are “creating” the data by associating a literal or performing an operation, use the following format:

```
variable = x
```

The variable must be the first token in the instruction. The value `x` can be a literal, variable, or an expression that requires evaluation. For example:

```
Var1 = 'Hi!'
Var1 = deposit + balance
Var1 = LENGTH(literal)
```

If you omit `x` ,

```
Var1 =
```

the variable is set to a null literal. However, your program will be easier to read if you code the null literal explicitly:

```
Var1 = ''
```

If the data is passed as an argument, you can assign data to several variables using the ARG and PARSE ARG keywords. For example,

```
ARG WORD1 WORD2
```

takes the arguments and assigns the first word to WORD1 and the rest of the string to WORD2. PARSE EXTERNAL reads input from the terminal and parses it to variables you supply.

PIE/CICS variables are assigned automatically. For example, the user ID is automatically assigned to the ZUSER variable. Likewise, the current time is assigned to the ZTIME variable.

If no value has been assigned to a variable, PIE/REXX uses the variable name itself, translated to uppercase, as its value. For example, if you code the variable Balance, but never assign anything to Balance, the value for Balance is BALANCE.

Operators

Operators specify the relationship between the elements of a program statement. Using operators, you can make decisions, create output for a terminal, manipulate numbers, and so forth. For example, let’s say the user ID is GEORGE. Then

```
Say ’HI ’ || ZUSER
```

writes ’HI GEORGE’ to the terminal. Or, let’s say that you want to do one thing when the value for x is less than 5 and another thing when its value is 5 or greater. So you code

```
If x < 5
     then instructiona
     else instructionb
```

There are several kinds of operators:

- Concatenating operators concatenate two strings.
- Arithmetic operators add, subtract, multiply, and so forth, two numbers.
- Comparative operators compare the relative value of two strings (is the first greater than, less than, equal to the second?).
- Boolean operators return a result of true or false, depending on whether two comparisons are true or false. For example, A=B & D>F is true only if both A=B and D>F.
- Unary operators affect only one token to the right of the operator. There are three unary operators for PIE/REXX: plus and minus before numbers and the Boolean Not, which means “not true” or “not false,” depending on the value of the token next to it.

Each operator type is detailed in the following chapter subsections.

☞ You may code comments and spaces between and around all operators and the tokens they relate.

Concatenating Strings

Concatenation is implied for literals and variables if there is no operator between them. For example, to print a dollar amount of the AMOUNT variable whose value is 395. you code:

```
'$'AMOUNT
```

The result is $395.

With implied concatenation, if one or more spaces separate tokens, there is a space between them in the result. For example, if you code

```
'Yellow'    'Moon'
```

you will get “Yellow Moon”.

Concatenation is *not* implied when there is a unary operator between the two tokens. For example,

```
'Result:' -TOTAL
```

is not valid. To concatenate these terms, place the unary operator and the token it modifies in parentheses:

```
'Result:' (-TOTAL)
```

To concatenate strings explicitly, place two vertical bars (||) between them. Spaces are ignored. For example,

```
'$' || AMOUNT
```

results in $395.

You must use || to concatenate two variables without a space between them:

```
AMOUNT1 || AMOUNT2
```

If AMOUNT1 is 5 and AMOUNT2 is 6, the result is 56.

Arithmetic Operators

The arithmetic operators are:

+	Add
-	Subtract
*	Multiply
/ and %	Divide
//	Modulus. Divide and return only the remainder (will retain positive or negative sign)
**	Exponentiation. Raise to the following power

☞ Arithmetic operators function only on numbers. To perform arithmetic on hex literals, first convert them to numbers using the built-in X2D function. Then, convert the result back to hex, if desired, using the D2X f unction. (If you do not convert them, hex values will be converted to character first, then calculated. If the hex value is not numeric, you receive an error.) Arithmetic operations can be performed only on integers; not decimals or numbers expressed as exponents or in scientific notation.

Examples

In the following examples, AMOUNT and TOTAL are variables with values of 300 and 1,000 respectively.

Operation	Result
4+5	9
AMOUNT + 5	305
AMOUNT*5	1,500
TOTAL/20	50
AMOUNT//8	4 (300 ÷ 8 = 37 $^4/_8$. Only the remainder is returned.)
TOTAL**2	1,000,000

Comparative Operators

Comparative operators compare the value of two strings. Comparisons return `1` for true conditions or `0` for false conditions.

There are two kinds of comparisons: `simple` and `strict`. In a simple comparison, PIE/REXX prepares the terms before comparing them. When either of the terms involves a character or hexadecimal literal, both terms are treated as character strings. If both terms are numeric, leading zeros are ignored—so 0123 equals 123. Leading and trailing blanks are ignored. Shorter literals are padded with blanks on the right.

In a strict comparison, the terms are compared as character strings from left to right, with no preparation. The values of numbers are not considered, only their leading characters (4 is `greater than` 10 because the leading character 4 is greater than 1). Leading zeros and blanks are considered (0123 is not equal to 123). To be equal, the two strings must be identical.

For both simple and strict comparisons, operators, functions, and variables are resolved before the comparison. Non-numeric characters (A-Z, #, <, ?, and so forth) are translated to their numeric equivalents in EBCDIC. Comparisons are case sensitive.

There are many comparative operators. They are combinations of several characters:

=	equal to
>	greater than
<	less than
\ and ¬	not

Simple Comparatives

=	True if both terms are equal.
\= and ¬= and < > and ><	True if the terms are not equal (if the first is greater or less than the second).
>	True if the first term is greater than the second.
>= and ¬< and \<	True if the first term is greater than or equal to (not less than) the second.
<	True if the first term is less than the second.
<= and ¬> and \>	True if the first term is less than or equal to (not greater than) the second.

☞ One way to assign a value to a variable is to code `variable` = x. That has the same syntax as the A = B comparison. To avoid ambiguities (is this an assignment or a comparison?), anytime you begin an instruction with `variable` = x, the instruction is considered to be an `assignment`. If the variable is not the first token in the instruction (there is another token before the comparison or the variable is after the equals sign), the instruction is considered a comparison.

If you want an “equality” comparison with a variable and some other token, you can place the variable second in the comparison.

```
expression = VAR1
```

If you want the variable in the first position, you can place a null literal before the variable:

```
'' VAR1 = expression
```

Or you can place the assignment in parentheses:

```
(VAR1 = expression
```

Strict Comparatives

==	True if both terms are strictly equal.
\== and ¬==	True if the terms are not strictly equal.
>>	True if the first term is strictly greater than the second.
>>= and ¬<< and \<<	True if the first term is strictly greater than or equal to (not less than) the second.
<<	True if the first term is strictly less than the second.
<<= and ¬>> and \>>	True if the first term is strictly less than or equal to (not greater than) the second.

Examples

In the following examples, AMOUNT and ITEM are variables with values of 300 and Widget No. 302.

Expression:	Result:
' ' == ' '	True. The strings are identical.
' ' == ''	False. The first string has two blanks, but the second has none.
'' '' = ""	True. This is not a strict comparison, so leading and trailing blanks are omitted. The terms are equal.

Expression:	Result:
AMOUNT > 500	False. 300 is less than 500.
7 - 2 < 20	True. 7 minus 2 is 5, which is less than 20.
7 - 2 << 20	False. 5 is not strictly less than 20, because 5 is greater than 2.
ITEM >> ’Widget No. 30’	True. “Widget No. 302” is strictly greater than “Widget No. 30.”
ITEM \= 'Widget No. 10'	True. The tokens are not equal.
AMOUNT=500	If these tokens begin an instruction, this example is not a comparison. (See the note on the previous page.) This statement assigns the variable AMOUNT to a new value, 500.
(AMOUNT=200)	These tokens form a comparison, no matter where they are placed in an instruction, because they are enclosed in parentheses. Since 300 does not equal 200, the result is false.

Boolean Operators

Boolean operators compare two expressions. The result of the comparison is either true (1) or false (0). They can compare two comparisons (A > B and C = D) or variables that equal 0 or 1 (presumably from a previous comparison) or functions that test a result.

&	(AND) True if both terms are true.
|	(Inclusive OR) True if either term is true.
&&	(Exclusive OR) True if one term is true and one term is false.

Examples of Boolean Operators

In the following examples, AMOUNT, TOTAL, and ITEM are variables with values of 300, 1000, and Widget No. 302.

Expression:	Result:
AMOUNT > 500 & ITEM = 'Widget No. 302'	False. The first term is false. Both must be true.
AMOUNT > 500 | ITEM = 'Widget No. 302'	True. The second term is true. The expression is true if either term is true.
AMOUNT > 500 && ITEM = 'Widget No. 302'	True. The first term is false and the second term is true. For an Exclusive OR, one term must be true and the other false.
AMOUNT < 500 | ITEM = 'Widget No. 302'	True. Both terms are true. Either qualifies the expression.

Unary Operators

Unary operators affect only one token to the right. There are three unary operators for PIE/REXX:. The + and - characters attribute the sign of numbers; whether the number is positive or negative. Plus means “do not change the sign.” Minus means “change the sign.” For example, if the variable NUMBER is -7, then +NUMBER is still -7, unchanged, and -NUMBER is +7, changed.

\ and ¬ The Boolean Not. These operators are valid only in front of tokens that evaluate to 0 or 1. The expression \1 means “not true.” The expression \0 means “not false.”

Interpretation of Operators Based Upon Precedence

Precedence rules must be obeyed when you code multiple operators in an expression. Unexpected results may occur otherwise. PIE/REXX interprets operators within an expression based upon the following precedence rules:

- Operators enclosed within parentheses are interpreted before other operators outside of the parentheses.
- Operators with higher precedence are interpreted before operators with lower precedence.

  For example, multiplication has higher precedence than addition. The expression 1+ 2 * 3 equals 7, rather than 9. Specific operator precedence is shown in the table shown on the following page.
- PIE/REXX evaluates expressions from left to right when operators have equivalent precedence.

Operator Precedence

Highest Precedence	\ ¬ - +	Unary operators
	**	Exponential power (Multiple powers are evaluated left to right)
	* / % //	Multiplication and division
	+ -	Addition and subtraction
	[blank or abutted] ||	Concatenation
	== \== ¬== >> >>= \<< ¬<< << <<= \>> ¬>> = ¬= \= <> >< > >= \< ¬< < <= \> ¬>	Comparatives
	&	Boolean AND
Lowest Precedence	| &&	Boolean OR and Exclusive OR

Examples of Operator Precedence

Expression:	Result:
6 + 4 / 2	8, division has precedence. It is done first.
(6 + 4) / 2	5, the operation in parentheses is done first.
2**2**3	64, not 256, because powers are evaluated from left to right.
-2**2**3	64, not -64, because unary operators are evaluated before powers.
AMOUNT > 500 & ITEM = 'Widget No. 302'	The expressions on either side of the & are evaluated before the &. The result is false.
1 + 1 << 20	The arithmetic operation is performed before the comparison. The result is false.

Commands

There are several types of commands: keywords, functions, subroutines internal functions, and host commands.

Keywords

Keywords mainly control program flow, for example branching instructions or calls to another PIE/REXX program. Some keywords perform an immediate action, like SAY which sends a line of output to the terminal and ARG which retrieves argument data.

```
Say 'Invalid Data in Customer No. Field, Session 1'
```

Some keywords have nested instructions, like IF and SELECT.

```
If total > 4 then say total   /* 'say total' is a
                                 nested instruction */
```

To be recognized, keywords must be the first token in the instruction. They cannot be followed immediately by = (signals an assignment) or: (signals a label). Blanks around keywords are ignored.

Functions

Functions test a condition or perform an action and then return a string as the result. For example, the function SIGN tests whether a number is positive or negative and returns 1 or 0 (true or false) as the result. The function LENGTH(string) returns the length of string, say 16. The function SET_CURSOR places the cursor on the terminal screen and returns 1 to indicate the operation was successful.

Built-In Functions typically manipulate, test, and return information about strings (for example, Length(cust#)).

- External Functions allow access to session screens, provide session information, and perform certain CICS programming commands (Scr_Get_Field(1,4,2), for example).

  Only the external functions supplied with PIE/REXX are supported as external function calls. You cannot create your own external functions. However, you can call your own PIE/REXX program by issuing the REXX command in your program. (See " Host Commands" on page 26, and " Executing a PIE/REXX Program" on page 5.)

Functions have special syntax requirements. The function name must be followed immediately by (, with no intervening blanks: label(arg). If there are no arguments, follow the function name with (): label(). The arguments are evaluated from left to right. Results are returned as a single string.

The string is returned in the position of the function in the REXX statement. You can think of the resulting string replacing the function in the statement. For example,

```
If length(cust#) > 5 ...
```

evaluates to

```
If 16 > 5 ...
```

You can put a function anywhere in an instruction that you can put a literal or variable. Wherever you put the function, its result must be processed as part of an instruction (see " Statements: Expressions and Instructions" on page 27). For example, it can be the argument for a keyword or the value in an assignment:

```
If length(cust#) > 5 ...
```

```
X = length(cust#)
```

Subroutines and Internal Functions

Subroutines and internal functions are routines in a PIE/REXX program that begin with a label and end with a RETURN statement. Example:

```
Validate:
If length(cust#) > 5 then do
      say 'Invalid Data in Customer No. Field, Session 1'
      return 4
   end
Return 0
```

Subroutines and internal functions differ depending on how they are invoked:

- If you invoke them using a function call—label(arg); they are internal functions. In this case, they must return a string. The string replaces the function in the instruction.
- If you invoke them using the CALL command—CALL label; they act as subroutines and may not return a string. If you are returning from a function call, the result (or return code) is placed in the special variable RESULT.

In either case, all previously assigned variables are available. Subroutines and internal functions can make nested calls to other routines.

All status information is saved and the program resumes executing when the routine returns.

Host Commands

PIE/REXX host commands execute in the host environment, which is PIE/CICS. Host commands are the PIE commands, both Application and Environment commands.

When a PIE/REXX program issues a PIE command, the program goes into a pseudo-conversational wait. After the command has executed, the PIE/REXX program resumes control. The return code for the command is stored in the special variable RC. (In standard mode, the return code is always 0.)

Before submitting a command to PIE, PIE/REXX will evaluate it as far as possible. If you want part of the statement to be submitted to PIE without evaluation, enclose it in quotes.

```
If ZUSER = 'Fred'
   then 'MENU fred.main('Cust#
   else 'Menu system.main('Cust#
```

☞ PIE Application commands and PIE/REXX programs interact differently depending on whether you are executing the PIE/REXX program in script or standard mode. See " Running REXX Programs as Scripts" on page 9, for more information.

When a PIE/REXX program executes several PIE commands in sequence, you can improve performance by stacking them; coding them as a single PIE command separated by a PIE command delimiter. (Remember not to us the semi-colon (;) as a delimiter, because that is a command separator for REXX. PIE delimiters are defined on the Profile Terminals screen.)

Distinguishing Command Types

There are occasions when different types of commands (keywords, functions, PIE commands) have the same name. When that is the case, PIE/REXX uses the following rules to determine the sequence that commands are executed:

- If the command has the format name(arg), it is considered to be a function. So the ARG function

  ARG(2)

  is distinguished from the ARG keyword

  ```
  ARG var1 var2
  ```
- Function types are searched in the following order: internal, built-in, external. If you put the function name in quotes—'Label'(arg)—PIE/REXX skips the search for an internal function and searches for built-in and external functions only.
- If the instruction begins with a PIE/REXX keyword, PIE/REXX executes the keyword.
- If the token is not a keyword, function, literal, or operator, it is passed to PIE/CICS as a PIE command.

Statements: Expressions and Instructions

PIE/REXX statements are composed of expressions and instructions.

- An expression is a logical series of tokens that result in a string. For example, 4 + 7 is an expression that results in the string 11. The built-in function Length(‘ABC’) is also an expression, with a result of 3. Even a literal or variable can be an expression: ’Mary’ and 345 and Result are all expressions when they are not part of an operation or function that creates a new string. The maximum length of the expression result is 32,768 characters.
- An instruction performs an action. It assigns a variable, executes a keyword, or issues a PIE command.

Expressions must be part of an instruction. Usually, you code expressions in instructions as arguments for keywords or in assignments:

```
If amount > 7
```

```
Var1 = deposit + balance
```

However, functions can sometimes be confusing. Functions are expressions, so they must be part of an instruction. Yet some functions perform actions just like keywords, for example, the SCR_RESHOW function which refreshes the terminal screen. We may be tempted to think of and code such functions as keywords. However, all functions are simply expressions, and they must be part of a full instruction. The two most common ways to execute functions are assigning them to variables:

```
x = SCR_RESHOW(‘*’)
```

and calling them with the CALL keyword:

```
CALL SCR_RESHOW ‘*’
```

Notice the example has no parentheses. Parentheses are not used with the CALL keyword. When you use CALL, the result is returned in the RESULT variable.

An instruction can contain nested instructions. For example:

```
If expression
   then instruction
   else instruction
```

Comments and labels are “instructions” when they stand by themselves.

Syntax

PIE/REXX statements can start in any column. Blank lines are permitted, which improves the readability of source programs.You do not have to code the word “REXX” on the first line.

You may code more than one instruction on a line. Use a semicolon to end an instruction.

```
Instruction 1;Instruction 2; ...
```

A semicolon is not required in the following cases.

- At line-end—

  ```
  Variable1=’This instruction is complete.’
  ```

  However, if your instruction ends with a comma, code a semicolon after it, even when the instruction ends at line-end:

  ```
  token,;
  ```

- After a label—

  ```
  Label:
  ```

- After the keywords ELSE, OTHERWISE, WHEN, and THEN

You cannot use a null clause (for example, an extra semicolon in an IF or SELECT statement) as a dummy instruction. You must use the NOP keyword.

```
If x=7
   then NOP
   else return
/* Your routine continues. */
```

Continuations

To continue an instruction or expression on another line, place a comma after the last token in the line to be continued:

```
token,
token
```

Do not split a literal or comment with a comma. Simply continue the literal or comment on the following line as if there were no line break. Do not separate the comment delimiters (/ and *).

```
/* Oh, doctor!
You can hang a star on that one! */
```

Instead of continuing comments and literals, you can separate them into two literals or comments. Literals are concatenated by default. So,

```
'Oh, doctor!',
'You can hang a star on that one!'
```

is the same as “Oh, doctor! You can hang a star on that one!”

Controlling Program Flow

Labels

Labels identify the destinations of branch operations within a program (CALL, SIGNAL) and internal functions. A label is a variable followed by a colon.

```
WHATSIT:
```

Labels are translated to uppercase before evaluation. Differences in case are ignored when the program is compiled.

Branches and Subroutines

To create a subroutine, identify it with a label and end it with RETURN or EXIT keywords. RETURN ends a subroutine and returns control back to the main program; it can also pass back a result. EXIT terminates a PIE/REXX program.

The following example returns 4 or 0 from the subroutine.

```
Validate:
If length(cust#) > 5 then do
      say 'Invalid Data in Customer No. Field, Session 1'
      return 4
   end
Return 0
```

You can access a subroutine using any of the following methods. The way you access a subroutine affects how the result, if there is one, is returned to the main program.

- Function call—label(arg)

  The subroutine must end with a RETURN statement and pass back a result. The result is derived from subroutine processing.

  ```
  If Validate() = 4 ...
  ```

- CALL—CALL label

  The subroutine need not return a string. If the subroutine ends with RETURN, the program resumes execution at the next instruction following the CALL instruction. The result from the subroutine is assigned to the special variable RESULT.

  ```
  Call Validate
  If RESULT = 4 ...
  ```

- SIGNAL—SIGNAL label

  The program branches to the subroutine and control does not return to the main program flow. No result is returned to the main program.

Loops

Create program loops with the DO keyword. You may conditionally change their flow with the ITERATE and LEAVE keywords. Refer to these keywords in “Chapter 3 Commands and Functions” on page 31 for more information.

Conditions

Use the IF and SELECT keywords to change the sequence that statements are executed within a program. Both keywords provide branching to other locations within a program based upon the outcome of a tested condition. Use IF when you have only one condition to test. Use SELECT to test any number of conditions. Refer to both keywords in “Commands and Functions”, beginning on page 31 for more information.

Chapter 3 Commands and Functions

This chapter describes PIE/REXX keywords, functions, routines, and commands. A syntax diagram and an example accompanies each PIE/REXX statement.

Keywords

Refer to *IBM REXX/MVS Reference* for more information about any keyword described in this chapter.

Keyword	Description
ARG	Retrieve arguments and assign them to variables
CALL	Call an internal or external routine
DO	Create a loop
EXIT	Exit the program
IF	Execute instructions based on a conditional test
ITERATE	Go to the next iteration of a DO loop
LEAVE	Exit a DO loop
OPTIONS	Special PIE/REXX processing requests
NOP	No operation
PARSE	Assign data to variables
RETURN	Return from a routine
SAY	Send a message to the terminal
SELECT	Execute instructions based on multiple of conditional tests
SIGNAL	Abnormal branch to a label with no return
TRACE	Control display of PIE/REXX functions and errors on the terminal
UPPER	Translate a string to uppercase

ARG

The ARG keyword retrieves program or subroutine parameters and assigns them to variables. It is a short form of PARSE UPPER ARG.

Argument strings are converted to uppercase as they are retrieved. To avoid uppercase translation, use the PARSE keyword.

Unless a subroutine, internal function, or method is being processed, the objects passed as parameters to the program are converted to string values and parsed into variables according to SAA REXX parsing rules. These rules are described in the IBM TSO/E Version 2 MVS/REXX Reference manual.

```
ARG template list
```

template list	Variable names. The template list is often a single template but can be several templates separated by commas. Use multiple templates when more than one argument string is passed. Place a comma between templates.

Examples

A function call passes the argument string “orange, apple, pear”. The internal function begins with the statement:

- `ARG Word1 Word2 Word3`
  The result is Word1=ORANGE, Word2=APPLE, Word3=PEAR. Arguments are translated to uppercase.
- `ARG Word1 Word2`
  The results are in Word1=ORANGE, Word2=APPLEPEAR.

  If you did not supply enough variables to parse the entire string, the remainder of the argument string is assigned to the last variable.
- `ARG Word1 Word2 Word3 Word4`
  The results are in Word1=ORANGE, Word2=APPLE, Word3=PEAR, Word4=””

  If there are more variables than arguments, extra variables are set to the null string.

If you expect more than one object to be available to the program or routine, you can use a comma in the parsing template list so each template is selected in turn.

-
  ```
  /* Function is called by CICSWORK ('data Y',2,3) */
  CICSWORK: Arg string, num1, num2
  ```

  results in STRING=DATA Y, NUM1=2, NUM2=3

CALL

The CALL keyword calls a subroutine or function. If a result is returned, it is assigned to the variable RESULT. If no result is returned, RESULT remains uninitialized.

```
CALL name [expression[,expression...]]
```

name	Name of the subroutine or function call.
expression	Up to 20 expressions to pass as arguments. Expressions are separated by commas.

Example

-
  ```
  CALL validate cust#
  If result = 4 ...
  ```

  The CALL instruction executes the Validate subroutine and passes the *cust#* argument. When the Validate subroutine returns, any result is assigned to the RESULT variable. The program continues at the IF instruction immediately following the CALL instruction.

DO

The DO keyword creates a l program loop that repeatedly executes a sequence of program statements based upon whether a tested condition is true or not. During looping, a control variable can be incremented or decremented by a range of values.

You can escape from a DO loop completely (for example, when you hit a certain condition) with the LEAVE keyword. You can force the next iteration of a loop (again, to handle a particular condition) with the ITERATE keyword. You can turn on loop protection with the OPTIONS keyword.

```
DO  [  simple-num-expr  ]
    [ variable=num-expr ]

    [[TO num-expr] [BY num-expr] [FOR num-expr]]
    [                 [FOREVER]                ]

    [WHILE boolean-expr]  [;]
    [UNTIL boolean-expr]

    instructions
    END
```

simple-num-expr	Number of times to repeat the DO loop. Specify a number or an expression that resolves to a number.
variable	Count conditions for the DO loop. If you also specify TO, the DO loop repeats until the count reaches the limit set by the TO *variable*. If you specify FOR, looping continues the specified number of times set by the FOR *variable*. If you specify BY, the loop count increments by the BY *variable*. For TO, BY, and FOR, specify a number or an expression that evaluates to a number. BY defaults to 1. If you specify FOREVER, looping continues until a WHEN or UNTIL parameter or a LEAVE instruction in the loop stops the loop. FOREVER is the default if you omit TO and FOR.
WHILE	Condition under which you want the loop to continue. Code an expression that evaluates to 0 or 1. The loop continue as long as the expression is true (1).
UNTIL	Condition that stops the DO loop. Code an expression that evaluates to 0 or 1. Looping stops when the expression is true (1).
instructions	Instructions included within the DO loop. You must begin the first instruction on a separate line or separate the DO and its parameters from the instruction with a semi-colon. For example:

DO i to 10
  instructions
end

or

DO i to 10; instructions;end

END	End of the loop.

Examples

A single program loop occurs if DO is specified without a loop counter or condition parameters (the first three listed parameters after the DO keyword). This kind of DO instruction is called a simple DO loop. A simple DO loop can be useful in an IF/THEN statement, when you want to process several instructions after THEN or ELSE.

Format:

```
DO
   instructions
END
```

Example:

```
If balance=0 then DO
     say 'Balance is 0. Cannot',
         'process transaction.'
     signal 'zero_bal'
end
```

A simple repetitive loop executes a specified number of times. The following example shows the code to read the next five queue records by looping five consecutive times.

Format:

```
DO simple-num-expr
   instructions
END
```

Example:

```
/* Read 5 queue items */
Do 5
   x = exec_cics_readq(queue,i)
end
```

A controlled repetitive loop keeps count using a variable. It begins the count with the number (or the result of the expression) after the variable. It increments the count by the number (or result) for BY. Looping continues until the variable reaches the TO number (or result). If you specify FOR, the loop will not execute more than FOR times.

Format:

```
DO variable=a [TO b]
   [FOR c] [BY d]
   instructions
END
```

Example:

```
/* Read 10 queue items, beginning
   with item#                    */
Do i=item# To item# + 9
     x = EXEC_CICS_READQ(queue,i)
end

/* Read 10 queue items, beginning
   with item#                    */
Do i=item# For 10
     x = EXEC_CICS_READQ(queue,i)
end
```

```
/* Read 10 queue items, beginning
   with item#                      */
Do i=item# To item# + 18 By 2
     x = EXEC_CICS_READQ(queue,i)
end
```

A conditional loop continues executing while or until a condition exists. You can use repetitors (simple-num-expr or variable, etc.) with conditional loops, if you like.

Format:

```
DO [repetitor]
                {WHILE a}
                {UNTIL b}
   instructions
END
```

Example:

```
/* Read queue items, through item
   # 150                           */
Do i=item# WHILE i <= 150
   x = exec_cics_readq(queue,i)
end

/* Read queue items, through item
   # 150                           */
Do i=item# UNTIL i >= 151
   x = exec_cics_readq(queue,i)
end
```

EXIT

The EXIT keyword unconditionally terminates the execution of a PIE/REXX program and passes a return value to the caller.

Only the current REXX program is terminated. A calling REXX program resumes execution from the point the current program was invoked.

```
EXIT
```

Example

-
  ```
  Validate:
  If length(cust#) > 5 then do
        say 'Invalid Data in Customer No. Field, Session 1'
        exit
     end
  Return 0
  ```

  This subroutine exits the program after encountering invalid data in the Customer Number field.

IF

The IF keyword conditionally processes a set of program instructions based upon the outcome of a boolean expression.

IF statements can be nested. If you have more than one condition to test, use the SELECT keyword instead. SELECT executes program instructions based upon the outcome of multiple conditional expressions.

☞ A semicolon is implied after THEN and ELSE.

```
IF boolean-expr [;]
     THEN [;] instruction
     [ELSE [;] instruction]
```

boolean-expr	Condition to test, in the form of an expression that evaluates to 0 or 1.
THEN	Instruction sequence to execute if the IF expression is true (1).
ELSE	Instruction sequence to execute if the IF expression is false (0). When you have nested IF/THEN statements, code ELSE for each level.
instruction	Command, assignment, or keyword instruction that are executed based upon the outcome of the IF/THEN or SELECT keywords. Only one instruction is permitted. To request more than one action, use a DO loop. The NOP keyword performs no action.

Examples

-
  ```
  If old# < 5
      then new# = old# + 5

      else new# = old#
  ```

  If the condition is true, the THEN instruction is executed. If it is false, the ELSE instruction is executed. Whether the condition is true or false, processing continues with the next instruction.

-
  ```
  Validate:
  If length(cust#) > 5 then do

        say 'Invalid Data in Customer No. Field, Session 1'

        return 4

     end

  Return 0
  ```

  The PIE/REXX Validate subroutine includes an IF/THEN statement with a DO loop. The DO loop is required to perform more than one instruction if the condition is true. There is no ELSE statement. Processing continues with the next instruction.

ITERATE

The ITERATE keyword alters program flow within a repetitive DO loop. Execution of the group of instructions stops, and control is passed to the DO instruction just as though the END clause has been encountered. The control variable is incremented and tested, and the group of instructions is processed again, unless the DO instruction ends the loop.

When ITERATE is processed, PIE/REXX considers the loop completed for that iteration and increments the counter and begins the DO loop for the next iteration.

If you want to end a DO loop completely, use LEAVE.

```
ITERATE [variable]
```

variable	Variable of the DO loop you want to iterate. Use this parameter to iterate a nested DO loop. All loops within the loop named will be iterated. If more than one loop uses this variable, PIE/REXX selects the innermost loop.

Example

-
  ```
  Do i=item# To item# + 9
       if i=100 iterate

       EXEC_CICS_READQ(queue,i)

       END
  ```

  This DO loop skips reading item 100. The DO counter increments and the loop continues with item 101.

LEAVE

The LEAVE keyword stops a DO loop. When LEAVE is processed, PIE/REXX considers the loop completed: no more iterations are processed and the counter is not incremented. PIE/REXX executes the next instruction after the DO statement.

```
LEAVE [variable]
```

variable	Variable of the DO loop you want to leave. Use this parameter to leave a nested DO loop. All loops within the named loop are bypassed. If more than one loop uses this variable, PIE/REXX selects the innermost loop.

Example

-
  ```
  Do i=item# To item# + 9
       say item#
       if i=100 leave
       EXEC_CICS_READQ(queue,i)
       END
  ```

  This DO loop ends when i = 100. The DO counter does not increment. The program continues with the next instruction.

NOP

The NOP keyword creates a dummy instruction that has no effect on program processing. It can be useful as the target of IF/THEN or SELECT/WHEN clauses that have no instructions associated with a tested condition.

```
NOP
```

Example

-
  ```
  If inventory > 50 then
   if inventory > 25 then say 'Inventory is dangerously low.',
                              'Time to reorder.'
   else nop
   else call Low_inventory inventory
  ```
  This example has a nested IF statement. The ELSE parameter is required, but no action is required. The NOP keyword moves the program to the next statement.

OPTIONS

The OPTIONS keyword requests special options for PIE/REXX processing. When you use an OPTIONS statement, place quotes around the entire argument string. Type the parameters in uppercase. Separate parameters with commas or blanks.

If an OPTIONS parameter is invalid, there is no error statement and the parameter is ignored.

```
OPTIONS '[STOR=n] [MAXSTOR=n] [SCRIPT=Y|N] [DEBUG=Y|N]
        [TRACE=TS|TD|TRM] [LOOP=OFF|n]'
```

STOR	Size of the initial storage allocation, in bytes, for the PIE/REXX stack and variable managers. The OPTIONS statement containing this parameter must be the first line of your PIE/REXX program. If you do not code STOR, 10,000 bytes is the default allocation. For large PIE/REXX programs or programs that use many variables, performance may improve if you increase the initial storage allocation using the STOR parameter. For small programs, you can reduce virtual storage usage by decreasing the storage allocation.
MAXSTOR	Maximum storage allocation for PIE/REXX stack and variable managers. A PIE/REXX program terminates with an error if this value is exceeded. The OPTIONS statement containing this parameter must be the first line of your PIE/REXX program.
SCRIPT	Program executes as a script or not. If coded as SCRIPT=Y, you do not need to put the SCRIPT option on the REXX command. The OPTIONS statement containing this parameter must be the first line of your PIE/REXX program.
DEBUG	Debugging information on the PIE/REXX executor. Use only with the support from a UNICOM Systems, Inc. Customer Service representative.
TRACE	Trace data collection: TD for transient data (queue name, PIEL), TS for temporary storage (queue name, @YZ + your unique PIE/CICS ID + the session ID), or TRM for terminal. Interactive trace commands are ignored if you specify TD or TS. TRM is the default.
LOOP	Maximum loop count number. PIE/REXX terminates a program and issues a message when the number of loops has been exceeded. Specify OFF to turn loop protection off. The default is LOOP=10000.

Example

- `OPTIONS 'STOR=20000 MAXSTOR=50000 LOOP=500'`
  This OPTIONS statement must come at the beginning of the program because it specifies allocation sizes. It allocates an initial storage amount of 20,000 bytes and limits total storage allocation to 50,000 bytes. It also sets the maximum number of program loops to 500.

PARSE

The PARSE keyword retrieves data strings and assigns them to variables. PARSE uses SAA REXX parsing rules. These rules are described in the IBM TSO/E Version 2 *MVS/REXX Reference* manual.

```
                ⎧          ARG           ⎫
                ⎪        EXTERNAL        ⎪
PARSE [UPPER]  ⎨VALUE expression WITH  ⎬ template listg
                ⎪        VAR name        ⎪
                ⎩                        ⎭
```

UPPER	Translates the argument to uppercase. If omitted, arguments are not translated to uppercase as they are parsed.
ARG	Retrieve arguments from a function or subroutine call.
EXTERNAL	Wait for and retrieve terminal input, usually after a SAY instruction. The user must press an AID key to continue program processing.
VALUE *expression* WITH	Evaluate the expression you specify and parse the result.
VAR	Parse the value of a variable.
template list	Variable names. Use multiple templates for PARSE ARG when more than one argument string is passed. Place a comma between templates.

Examples

- `PARSE ARG Word1 Word2 Word3`
  A function call passes the argument string "Apples Pears Oranges". The result is Word1=Apples, Word2=Pears, Word3=Oranges.
- `PARSE ARG Word1 Word2`
  If you did not supply enough variables to parse the whole string, the remainder of the string is assigned to the last variable. The result is Word1=Apples, Word2=PearsOranges.
- `PARSE ARG Word1 Word2 Word3 Word4`
  If there are more variables than arguments, the extra variables are set to the null string. The result is Word1=Apples, Word2=Pears, Word3=Oranges, Word4="".
- `PARSE UPPER ARG Word1 Word2 Word3`
  If you supply UPPER, the string is translated to uppercase. The result is Word1=APPLES, Word2=PEARS, Word3=ORANGES.
- `PARSE VAR NL1 SD ATL LA`
  PARSE VAR is useful for parsing out a variable that contains multiple words. For example, the variable NL1 has a value of "Padres Braves Dodgers". The result is SD=Padres, ATL=Braves, and LA=Dodgers.
- `PARSE EXTERNAL`

Retrieves terminal data after a SAY instruction. For example,

-
  ```
  SAY 'Type request'
  PARSE EXTERNAL Request
  ```
  Waits for a response to the SAY instruction and puts the response in the REQUEST variable.
- `PARSE VALUE`
  Evaluates an expression and parses the result. For example, suppose the function Scr_Get_Field('*',8,24) retrieves the value “Tony Gwynn” from a name field. Then the PARSE instruction
- `PARSE VALUE scr_get_field('*',8,24) fn ln`
  Parses the value, so that FN=Tony and LN=Gwynn.

RETURN

The RETURN keyword returns control from a subroutine or PIE/REXX program to the point of its invocation. The main program flow (or subroutine) resumes after the RETURN statement. RETURN can pass back a result.

```
RETURN [expression]
```

expression	Result passed back from a subroutine or another PIE/REXX program. If you are returning from a subroutine after a CALL, the result is placed in the RESULT variable. If you are returning from a function call, the result returned is the result of the function.

Example

-
  ```
  Validate:
  If length(cust#) > 5 then do
        say 'Invalid Data in Customer No. Field, Session 1'
        return 4
     end
  Return 0
  ```

  The Validate subroutine has two RETURN instructions. A different result is passed back to the calling program depending on the outcome of the condition tested by the IF THEN statement.

SAY

The SAY keyword writes a message to the default output stream. Typically, the message is sent to the session terminal running the PIE/REXX program. If you are requesting a response, follow the SAY instruction with a PARSE EXTERNAL instruction.

PIE/REXX SAY works like SAY in TSO/E REXX. It clears the screen before its first execution and writes the first output to the top of the screen. After that, it maintains a line count and writes further SAY instructions to the next line in sequence. Thus it works something like the TSO READY prompt in TSO.

We recommend that you do not issue SAY commands in conjunction with screen manager external functions or EXEC_CICS_SENDMAP or EXEC_CICS_SENDERASE. The screen handling of these commands may conflict with the SAY keyword.

SAY displays unprintable characters as colons (:).

```
SAY expression
```

expression	Message sent to the terminal. The message can be of any length. If you omit *expression*, the null string is written.

Example

- `SAY 'It is' ZTIME||'.'`
  Writes the time on the terminal. For example, “It is 10:22:11.”

SELECT

The SELECT keyword sequentially tests multiple conditions and executes instructions associated with the first tested true condition.

Each boolean-expr after a WHEN is evaluated in turn and must result in 0 or 1. If the result is 1, the instruction following the associated THEN is processed and control then passes to END. If the result is 0, control passes to the next WHEN clause. If none of the WHEN expressions evaluates to 1, control passes to the instructions after OTHERWISE.

☞ A semicolon is implied after WHEN, THEN, and OTHERWISE.

```
SELECT
    WHEN boolean-expr [;] THEN [;] instruction
     ...
    [OTHERWISE [;] instruction]
    END
```

WHEN	Test condition. If the expression is true (1), PIE/REXX performs the instruction associated with the paired THEN statement. After that, control is passed to END. If the expression is false (0), control passes to the next WHEN statement.
THEN	Instruction to perform if the WHEN expression is true.
OTHERWISE	Instruction executed if none of the WHEN expressions are true. If you nest SELECT statements, code OTHERWISE at each level.
instruction	Any command, assignment, or keyword instruction, including IF or SELECT keywords. Only one instruction is permitted with each THEN statement. To request more than one instruction, use a DO loop. The NOP keyword performs no action.
END	Statement that closes a sequence of WHEN-THEN SELECT statements.

Examples

-
  ```
  Select
         when zuser = 'FRED' then 'menu fred.main('cust#
         when zuser = 'SALLY' then 'menu sally.main('cust#
         when zuser = 'GEORGE' then 'menu george.main('cust#
         otherwise 'menu system.main('cust#
         end
  ```

  This example checks for three user IDs and issues a personal menu for each of them. Otherwise, a default menu is issued for all other users.

SIGNAL

The SIGNAL keyword makes an abnormal branch to another part of a PIE/REXX program. Like GOTO in other languages, SIGNAL does not return. Unlike GOTO, SIGNAL invalidates control variables for the DO loops and routines it branches from—so you cannot return without errors.

```
SIGNAL labelname
```

labelname	Branch label, which is regarded as a constant.

Example

-
  ```
  If result=0
     then signal zero_bal
     else ...
      ...
  Zero_bal:
  ```

  This program branches to the Zero_bal routine when the variable result is 0. The program instruction sequence does return to the main program flow.

TRACE

The TRACE keyword controls statement tracing during the processing of a PIE/REXX program. When tracing is active, PIE/REXX displays information about every statement executed in a program. By default, tracing is off.

TRACE displays unprintable characters as colons (:).

The OPTIONS keyword determines where trace data is written. If it is written to a transient data or temporary storage queue, interactive trace commands (`Trace ? ...`) are ignored. See “ OPTIONS” on page 42, for more information.

```
                  ┌       option      ┐
TRACE [!] [?]     │                   │
                  │ VALUE  expression │
                  └                   ┘
```

!	Traces expressions, but does not execute them. For example, `Trace!C` traces host commands but does not execute them. Further TRACE! instructions toggle this feature off and on.
?	Turns on interactive trace. A PIE/REXX program pauses for input after most traced statements. The only valid response during interactive trace is `TRACE O`, for TRACE OFF. When interactive trace is on, subsequent trace statements in the program are ignored until you turn off interactive trace.
option	Type of tracing performed within a program. Only the first letter of the option is required. Valid options are:

All	All statements are traced before execution
Commands	External commands are traced.
Error	External commands causing an error are traced after execution and the return code is displayed.
Failure	Same as Normal
Normal	External commands causing an error are traced after execution. This is the default.
Off	No Tracing. Subsequent TRACE statements in the program can reinstate tracing.
Results	All statements are traced before execution. The final values of expressions and values assigned using ARG and PARSE are also traced.

VALUE *expression*	These parameters dynamically determine the trace type. Specify an expression that evaluates to one of the valid trace option characters: A, C, E, F, I, N, O, or R. You can omit the keyword VALUE if the expression starts with an operator or parenthesis.

Examples

- Trace ?E
  Turns on interactive trace for external commands causing an error.
- Trace !A
  Traces but inhibits execution of host commands.

Now suppose that we are writing a PIE/REXX program and that we want tracing on. At the top of the program we make the assignment:

- `Trace_type = R`
  Then in later routines, we set up tracing using the TRACE VALUE instruction
- `TRACE VALUE Trace_type`
  When we have finished debugging the program, we can turn off tracing for all routines by changing the initial assignment:
- `Trace_type = N`

UPPER

The UPPER keyword converts one or more variable values to all uppercase characters.

```
UPPER variable [variable ...]
```

variable	Variable whose value is converted to uppercase.

Example

-
  ```
  UPPER Test
  If Test='ABC' ...
  ```

  First, the value of the Test variable is converted to uppercase. Then, the value of the Test variable is compared to an uppercase string. Using this sequence of statements avoids irrelevant case conflicts in the comparison.

Built-In Functions

Refer to the IBM TSO/E Version 2 *MVS/REXX Reference* manual for additional information about any of the built-in functions described in this section.

ABBREV	Test whether one string is a leading substring of another
ABS	Return the absolute value of a number
ARG	Return an argument string, the number of arguments, or whether a particular argument exists
BITAND	Logically AND two strings
BITOR	Logically OR two strings
BITXOR	Logically exclusive-OR two strings
CENTER	Center a string in a new string of desired length
COMPARE	Compare two strings
COPIES	Create a string with *n* copies of another string
C2D	Convert a character string to its numeric value
C2X	Convert a character string to its hexadecimal value
DATATYPE	Test the data type of a string
DELSTR	Delete characters from a string
DELWORD	Delete words from a string
D2C	Convert a number to EBCDIC
D2X	Convert a number to hexadecimal
INSERT	Insert a new string into a target string
LASTPOS	Return the position of the last occurrence of a substring in a string
LEFT	Return the leftmost *n* characters of a string
LENGTH	Return the length of a string
MAX	Return the largest number from a list
MIN	Return the smallest number from a list
OVERLAY	Overlay a target string with a new string
POS	Return the position of a substring in a larger string
REVERSE	Reverse a string
RIGHT	Return the rightmost *n* characters of a string
SIGN	Test the sign of a number
SPACE	Replace the spaces in a string with n pad characters
STRIP	Remove leading or trailing characters or both from a string
SUBSTR	Return n characters from a string
SUBWORD	Return n words from a string
TRANSLATE	Change characters in a string to other characters
VERIFY	Test whether a string contains only the characters you specify

WORD	Return the nth word in a string
WORDINDEX	Return the position of the first character in the *n*th word of a string
WORDLENGTH	Return the length of the nth word of a string
WORDPOS	Return the word position of a substring in a larger string
WORDS	Return the number of words in a string
XRANGE	Return a hexadecimal string that increments from a beginning value to an ending value
X2C	Convert a hexadecimal string to EBCDIC
X2D	Convert a hexadecimal string to a decimal string

When you code functions, code the parentheses shown in the format boxes. The arguments for all functions are positional. If you omit the argument for a position but code an argument in a later position, place a comma as a place holder for the unused argument.

When the function format calls for a string, you can specify any expression that evaluates to a string. That expression can be a literal, variable, operation, or even function.

ABBREV

The ABBREV function tests whether one string is a leading substring of another and the substring is not less than the minimum length.

ABBREV returns 1 if the conditions tested are true. It returns 0 if any condition is false.

```
ABBREV(full,short[,length])
```

full	Full-length string.
short	Substring.
length	Expected length of the substring.

Examples

Function:	Result:
ABBREV('Gehrig','Ge',2)	1
ABBREV('Gehrig','Gu')	0
ABBREV('Gehrig','Ge',3)	0

ABS

The ABS function returns the absolute value of a number. The result has no sign.

```
ABS(number)
```

number	Valid REXX number.

Examples

Function:	Result:
ABS(22)	22
ABS(-75)	75

ARG

The ARG function returns either the number of arguments, the value of a specific argument, or whether a particular argument has been included or omitted.

☞ Arguments passed with the REXX command are passed as a single string, even when multiple words are separated by blanks. You may parse those words with the ARG or PARSE keywords.

```
ARG([n[,option]])
```

n	Returns an argument string, number of arguments to return. If you omit n, the number of arguments is returned. If you specify an option, ARG tests whether the nth argument exists.
option	Specify E to return 1 if the nth argument exists or 0 if it does not. Specify O to return 1 if the nth argument is omitted or 0 if it exists.

Example

The following examples test the argument “SMITH TN123” with the ARG function.

Function:	Result:
ARG()	2, the number of arguments
ARG(2)	TN123, the second argument
ARG(3,E)	0, the third argument does not exist.

BITAND

The BITAND function compare two strings in binary, with logical AND. BITAND converts the strings to binary, compares them using AND logic, and converts the result to EBCDIC.

```
BITAND(string1,string2[,pad])
```

string1	First operand of the AND comparison
string2	Second operand of the AND comparison. The default is null.
pad	Pad character appended to the shorter string. The string is padded on the right. If pad is omitted, the comparison stops at the end of the shorter string. The remainder of the longer string is appended to the result.

Examples

Function	Result
BITAND('EF 7E'x,'6E 5A'x)	'>!' EBCDIC, '6E 5A'x
BITAND('AB6*','abL%')	'abK<' EBCDIC, '8182D24C'x

BITOR

The BITOR function compares two binary strings and returns the logical OR of its arguments in EBCDIC.BITOR converts the strings to binary, compares them using OR logic, and converts the result to EBCDIC.

```
BITOR(string1,string2[,pad])
```

string1	First operand of the OR operation.
string2	Second operand of the OR operation. The default is the null string.
pad	Pad character appended to the shorter string. The string is padded on the right. If pad is omitted, the comparison stops at the end of the shorter string. The remainder of the longer string is appended to the result.

Example

Function	Result
BITOR('20 C0'x,'41 01'x)	'/A' EBCDIC, '61C1'x
BITOR('abcd','wx&&')	'xxLM' EBCDIC, 'A7A7D3D4'x

BITXOR

The BITXOR function compares two binary strings and returns the logical Exclusive OR of its arguments in EBCDIC.

```
BITXOR(string1,string2[,pad])
```

string1	First operand of the exclusive OR operation.
string2	Second operand of the exclusive OR operation. The default is the null string.
pad	Pad character for the shorter string. The string will be padded on the right. If pad is omitted, the comparison stops when the shorter string is exhausted. The remainder of the longer string is appended to the result.

Example

Function	Result
BITOR('20 C0'x,'41 01'x)	'/A' EBCDIC, '61C1'x
BITOR('vwcd','*;&&')	'98LM' EBCDIC, 'F9F8D3D4'x

CENTER

The CENTER function aligns a string in the middle of a field of a specified width. The CENTER function is also useful for extracting the central characters of a string.

The spelling of this function can be either CENTER or CENTRE.

```
⎧CENTER⎫
⎨      ⎬(string,length[,‘pad’])
⎩CENTRE⎭
```

string	String to be centered on the field.
length	Number of characters in the width of the field. If the string is longer than the width of the field, it is truncated evenly at both ends.
pad	Pad character to place before and after `string` to make up the desired length. The default pad character is a blank space.

Examples

Function	Result
CENTER(’Debits’,10)	’ Debits ’
CENTER(’Debits’,10, ‘*’)	'**Debits**'
CENTER('Debits',9, ‘*’)	'*Debits**'
CENTER('Debits',3)	'ebi'

COMPARE

The COMPARE function returns the index of the first mismatch between two input strings. COMPARE returns a 0 if the two strings are identical.

```
COMPARE(string1,string2[,pad])
```

string1	First string to be compared.
string2	Second string to be compared.
pad	Pad character appended to the shorter of the two strings before the comparison to make them equal length. The shorter string is padded on the right

Examples

Function	Result
COMPARE('Dimaggio','Dimaggio')	0
COMPARE('Dimaggio','Dime')	4
COMPARE('Dimaggio ','Dimaggio',)	0

COPIES

The COPIES function returns a concatenation of the specified number of string copies.

```
COPIES(string,n)
```

string	String to be copied.
n	Number of string copies.

Example

Function	Result
COPIES('Work',3)	WorkWorkWork

C2D

The C2D function returns the value of a character string interpreted as a decimal number. C2D converts the character string to binary and then to decimal.

```
C2D(string[,n])
```

string	String to be converted to a decimal number.
n	Length of the result. C2D returns a signed binary number using the rules of two’s complement. If you omit *n*, C2D returns an unsigned binary number. g

Examples

Function	Result
C2D(’F’)	198
C2D(’F’,1)	-58
C2D(’HI’)	51401
C2D(’A1’x)	161

C2X

The C2X function returns the EBCDIC value of an input character string interpreted as a hexadecimal number.

```
C2X(string)
```

string	String to be converted to a hexadecimal number.

Examples

Function	Result
C2X('F')	C6
C2X('A1')	C1F1
C2X('A1'x)	A1

DATATYPE

The DATATYPE function tests the data type of a string. If the type parameter is specified, the string is compared to the data type. If there is a match, DATAYPE returns 1. If the data does not match, DATATYPE returns 0 instead.

```
DATATYPE(string[,type])
```

string	String compared
type	Data type used in the comparison of a string. If the string matches the type, DATATYPE returns a 1. Otherwise, it returns a 0 for all mismatched comparisons.

If you omit *type*, DATATYPE returns NUM for numbers and CHAR for any string with alphabetic characters.

A	Alphanumeric (letters and numbers).
B	Binary or Bits (0 and 1).
L	Lowercase alphabetic (a-z).
M	Mixed case alphabetic (A-Z and a-z).
N	Number (0-9).
S	Symbols (A-Z, a-z, 0-9, and @ # $ % ¢ . ! ? _). Symbols are restricted to the characters allowed in a variable name.
U	Uppercase alphabetic (A-Z).
X	hexadecimal (A-F and 0-9 and blanks between pairs). Can be a null string.

Examples

Function	Result
DATATYPE(123)	NUM
DATATYPE('FF 0C')	CHAR
DATATYPE('Musial',L)	0
DATATYPE('Musial',M)	1
DATATYPE('FF 0C',X)	1

DELSTR

The DELSTR function deletes a specified numb er of characters from a string.

```
DELSTR(string,position[,number])
```

string	String from which characters are deleted.
position	String position *n* from the starting point of the string to begin deleting characters.
number	Number of characters to delete. If omitted, DELSTR deletes all characters to the right of the starting *n* string position.

Examples

Function	Result
DELSTR('Robinson',5)	Robin
DELSTR('Robinson',5,1)	Robison

DELWORD

The DELWORD function delete words from a string. DELWORD deletes the substring of a string that starts at the nth word and is of length number. If the length of the substring deletion is not specified, all remaining words in the string are deleted from the nth position.

```
DELWORD(string,word[,number])
```

string	String from which words are deleted.
word	Number of the word from the first word in the string to begin deleting.
number	Number of words to delete. If omitted, DELWORD deletes all words from the starting word to the end of the string.

Examples

Function	Result
DELWORD('You can hang a star on that one',6)	'You can hang a star '
DELWORD('You can hang a star on that one',6,2)	'You can hang a star one'

D2C

The D2C function converts a decimal number to an EBCDIC character. D2C first converts the number into binary and then assigns the EBCDIC character associated with that number.

```
D2C(number[,n])
```

number	Decimal number to be converted to an EBCDIC character.
n	Length of the resulting string. If the result is shorter than *n*, it will be sign extended ('00'x or 'FF'x) on the left. If the result is longer, it is truncated on the left.

Examples

Function	Result
D2C(196)	'D'
D2C(196,2)	' D'
D2C(220)	':', an unprintable character

D2X

The D2X function converts a decimal number to hexadecimal. This string is the hexadecimal representation of the whole number.The result does not include spaces.

```
D2X(number[,n])
```

number	Decimal number to be converted to hexadecimal.
n	Length of the resulting hexadecimal number. If the result is shorter than *n*, it will be sign extended (’00’x or ’FF’x) on the left. If the result is longer, it is truncated on the left.

Examples

Function	Result
D2X(196)	’C4’
D2X(196,2)	’C4’
D2X(196,4)	’00C4’
D2X(220)	’DC’

INSERT

The INSERT function inserts a new string into a target string.

```
INSERT(new,target[,n][,length][,pad])
```

new	String inserted within a target string.
target	Target string.
n	Position after which to begin inserting the new string. The default is 0; the inserted string is inserted beginning at position 1 of the *target* string.
length	Length of *new* inserted string. INSERT pads or truncates new on the right if *length* is greater or less than the character length of *new* respectively.
pad	Pad character. The default is a blank space.

Examples

Function	Result
INSERT('aaa','bbbbbb')	aaabbbbbb
INSERT('aaa','bbbbbb',2)	bbaaabbbb
INSERT('aaa','bbbbbb',10)	bbbbbb aaa
INSERT('aaa','bbbbbb',2,5)	bbaaa bbbb
INSERT('abc','dddddd',2,1)	ddadddd

LASTPOS

The LASTPOS function returns the starting position of the last occurrence of a substring within a string. If the substring is not located within the string, LASPOS returns 0.

```
LASTPOS(substring,string,last)
```

substring	Substring searched for within a string.
string	String searched for a substring.
last	Last character position searched within a string for the specified substring. If you omit *last*, the entire string is searched.

Examples

Function	Result
LASTPOS(’w’,’Gwynn’)	2
LASTPOS(’n’,’Gwynn’)	5
LASTPOS(’n’,’Gwynn’,4)	4
LASTPOS(23,1234)	2
LASTPOS(7,1234)	0

LEFT

The LEFT function returns the leftmost *n* characters of a string.

```
LEFT(string,n[,pad])
```

string	String that characters are selected.
n	Number of characters to return. The returned value is padded on the right if *n* exceeds the length of *string.*
pad	Pad character added to the right of the returned value if *n* exceeds the length of *string*.

Examples

Function	Result
LEFT('Ruth',3)	'Rut'
LEFT('Ruth',7)	'Ruth '
LEFT('Ruth',7,'%')	'Ruth%%%'

LENGTH

The LENGTH function returns the number of characters within a string.

```
LENGTH(string)
```

string	String to return the length of.

Examples

Function	Result
LENGTH('Williams')	8
LENGTH(7)	1

MAX

The MAX function returns the largest number from a group of numbers.

```
MAX(number[,number ...])
```

number	Numbers within a group that are compared for the largest value.Up to 20 numbers can be specified with the MAX function. To specify more than 20, nest MAX statements.

Examples

Function	Result
MAX(7,12,35,20)	35
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21))	21

MIN

The MIN function returns the smallest number from a group of numbers.

```
MIN(number[,number ...])
```

number	Numbers within a group that are compared for the smallest value.Up to 20 numbers can be specified with the MIN function. To specify more than 20, nest MAIN statements.

Examples

Function	Result
MIN(7,12,35,20)	7
MIN(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MIN(20,21))	1

OVERLAY

The OVERLAY function overlays the contents of a target string with a new string.

```
OVERLAY(new,target[,n][,length][,pad])
```

new	String whose contents overlay the *target* string.
target	Target string that is overlaid by the contents of the *new* string.
n	Starting character position within *target* that is overlaid. The default is 1, which is the leftmost character in the *target* string.
length	Number of characters inserted by the *new* string. The *new* string is padded or truncated on the right if *length* is greater or shorter than the number of characters in *new* respectively.
pad	Pad character added to the right of the *new* string if the *length* parameters exceeds the number of characters

Examples

Function	Result
OVERLAY('z','abcd')	zbcd
OVERLAY('z','abcd',3)	abzd
OVERLAY('z','abcd',2,2)	az d
OVERLAY('xyz','abcd',2,1)	axcd

POS

The POS function returns the first position of a substring in a larger string. POS returns 0 if the substring is not located within a string.

```
POS(substring,string[,first])
```

substring	Substring searched for within *string*
string	String whose contents are compared with the value of *substring*.
first	Starting position within *string* to begin the search for *substring.* The default is 1, which is the leftmost character.

Examples

Function	Result
POS('r','Aaron')	3
POS('a','Aaron')	2, comparisons are case sensitive
POS('a','Aaron',3)	0
POS(23,1234)	2

REVERSE

The REVERSE function reverses the order of characters within a string.

```
REVERSE(string)
```

string	String whose characters are arranged in reverse order.

Examples

Function	Result
REVERSE('Mays')	syaM
REVERSE(1234)	4321

RIGHT

The RIGHT function returns the rightmost *n* characters of a string.

```
RIGHT(string,length[,pad])
```

string	String from which characters are selected.
n	Number of characters to return. The returned value is padded on the left if *n* exceeds the length of *string.*
pad	Pad character added to the left of the returned value if *n* exceeds the length of *string*.

Examples

Function	Result
RIGHT('Bench',3)	'nch'
RIGHT('Bench',7)	' Bench'
RIGHT('Bench',7,'%')	'%%Bench'

SIGN

The SIGN function tests the sign of a number. It returns -1 if the number is negative, 0 if the number is 0, and 1 if the number is positive.

```
SIGN(number)
```

number	Number to test for sign value.

Examples

Function	Result
SIGN(-127)	-1
SIGN(127)	1
SIGN(0)	0

SPACE

The SPACE function replaces blank spaces within a string with *n* pad characters. Leading and trailing blanks are removed.

```
SPACE(string[,n][,pad])
```

string	String whose blank spaces are replaced by pad characters.
n	Number of pad characters that replace each blank space. The default is 1.
pad	Pad character to replace blank spaces within a string. The default is a space.

Examples

Function	Result
SPACE('Two floors down',3,'+')	'Two+++floors+++down'
SPACE('Two floors down',,'+')	'Two+floors+down'
SPACE('Two floors down ')	'Two floors down'

STRIP

The STRIP function removes leading or trailing characters from a string.

```
STRIP(string[,option][,char])
```

string	String whose leading or training blank spaces are removed.	
option	Leading or trailing blanks removed from a string.	
	B	Both leading and trailing blanks from a string. The default.
	L	Leading blanks are removed from a string.
	T	Trailing blanks are removed from as string.
char	Leading or trailing character to remove from a string. The default is a blank space.	

Examples

Function	Result
STRIP(' Clark ')	'Clark'
STRIP(' Clark ',T)	' Clark'
STRIP('::::Clark::::',,':')	'Clark'

SUBSTR

The SUBSTR function returns *n* characters from a string, beginning at a position you specify.

```
SUBSTR(string,position[,n][,pad])
```

string	String to return characters from.
position	Starting character position within a string to return characters.
n	Number of characters returned from a string. The returned string is padded on the right if *n* exceeds the number of characters available from the string. If omitted, the remainder of the string is returned from the right of the *position* parameter.
pad	Pad character added to the right of the returned string if *n* exceeds the number of available characters within the string.

Examples

Function	Result
SUBSTR('supercalifragilisticexpealidocious',7,3)	ali
SUBSTR('hi',2,3,'*')	i**

SUBWORD

The SUBWORD function returns *n* words from a string.

```
SUBWORD(string,word[,n])
```

string	String to return words from.
word	Starting position within *string* to return words. By default, the leftmost word of a string is 1.
n	Number of words to return. If omitted, the remainder of the words to the right of *word* are returned.

Example

Function	Result
SUBWORD('You can hang a star on that one',3)	'hang a star on that one'
SUBWORD('You can hang a star on that one',3,3)	'hang a star '

TRANSLATE

The TRANSLATE function changes characters in a string to other characters. TRANSLATE looks up each character of a string in an input table. If the string character is not listed within the input table, the character remains unchanged. Otherwise, characters are replaced by the corresponding character found in the same position of the output table.

The output table is extended by the pad character if it is shorter than the input table.

```
TRANSLATE(string[,output-table][,input-table][,pad])
```

string	String whose characters are translated to other characters.
output-table	Characters that replace *string* characters listed in *input-table*. The translated character is found in the same position of the original character listed in *input-table*.
input-table	Characters you want to translate from, in order corresponding to-characters. It defaults to XRANGE('00'x,'FF'x). If you do not specify a character, the original string character remains unchanged. If you omit both translation tables, the string is translated to uppercase.
pad	Pad character returned as a translated character in a string if the output table is shorter than the input table. The default is a blank.

Example

Function	Result
TRANSLATE('moon', 'bte', 'mno')	'beet'
TRANSLATE('ABCDEF','ZZZZZZ','ABCDEF')	'ZZZZZZZ'
TRANSLATE('ABCABC','ZYX','ABC')	'ZYXZYX'
TRANSLATE('ABC','Z','ABC')	'Z '
TRANLATE('ABC','Z','ABC',*	'Z**'

VERIFY

The VERIFY function tests whether a string contains only the characters you specify.

```
VERIFY(string,reference[,option][,start])
```

string	String is searched for characters specified with the *reference* parameter.
reference	Characters you want to test against.
option	Specify how you want VERIFY to respond.

N	Nomatch (the default) returns 0 if all characters in *string* are in *reference* or if there is a mismatch, the position of the first mismatch character.
M	Match returns the position of the first character in *string* that is also in *reference* or if no characters match, 0.

start	Starting character position in *string* to begin the search.

Examples

Function	Result
VERIFY('supercalifragilisticexpealidocious','acdefgiloprstux')	0
VERIFY('supercalifragilisticexpealidocious','acdefgiloprstux',M)	1
VERIFY('Gwynn','G')	2
VERIFY('Gwynn','yn',,3)	0

WORD

The WORD function returns the *n*th word in a string.

```
WORD(string,n)
```

string	String from which the *n*th word is selected.
n	Numerical position of the word within the string to return. If there are fewer than *n* words, WORD returns a null string.

Examples

Function	Result
WORD('You can hang a star on that one',5)	'star'
WORD('You can hang a star on that one',9)	'', null string

WORDINDEX

The WORDINDEX function returns the column position of the first character in the *n*th word in a string.

```
WORDINDEX(string,n)
```

string	Input string
n	Number of the word to return the column position of the first character. If there are fewer than *n* words, WORD returns 0.

Examples

Function	Result
WORDINDEX('You can hang a star on that one',2)	5
WORDINDEX('You can hang a star on that one',9)	0

WORDLENGtH

The WORDLENGTH function returns the number of characters of the *n*th word in a string.

```
WORDLENGTH(string,n)
```

string	String to check.
n	Number of the word in the string whose length is returned by WORDLENGTH. By default, the leftmost word in a string is 1. If *string* is composed of less than *n* words, WORD returns a 0.

Examples

Function	Result
WORDLENGTH('You can hang a star on that one',2)	3
WORDLENGTH('You can hang a star on that one',9)	0

WORDPOS

The WORDPOS function returns the starting word position of a phrase in a string. If the phrase is not found, WORDPOS returns 0. Multiple blanks are treated like single blanks. Otherwise, the complete phrase must be found in the string.

```
WORDPOS(phrase,string[,start])
```

phrase	Substring to search for.
string	String to search.
start	Starting word position in the *string* to begin the phrase search. If omitted, WORDPOS begins with the first word in *string*.

Examples

Function	Result
WORDPOS('You can','You can hang a star on that one')	1
WORDPOS('You can','You can hang a star on that one',2)	0
WORDPOS('on that one','You can hang a star on that one')	6

WORDS

The WORDS function returns the number of words in a string.

```
WORDS(string)
```

string	String to check.

Examples

Function	Result
WORDS('You can hang a star on that one')	8
WORDS('Oh, doctor!')	2

XRANGE

The XRANGE function returns a string that increments (in hex) from a beginning value to an ending value.

```
XRANGE([start][,end])
```

start	Beginning of the range. It defaults to ’00’x.
end	End of the range. It defaults to ’FF’x.

Examples

Function	Result
XRANGE()	’0001020304050607...FDFEFF’x
XRANGE(’0C’x,’1A’x)	’0C0D0E0F1112131415161718191A’x
XRANGE(’FD’x,’02’x)	’FDFEFF000102’x
XRANGE(’a’,’f’)	’abcdef’
XRANGE(’r’,’s’)	’r::::::::s’ (’99’x to ’A2’x)

X2C

The X2C function converts a hexadecimal string to a character string. X2Cconverts the hexadecimal string to binary and then to the corresponding EBCDIC value.

```
X2C(hex)
```

hex	Hexadecimal string to convert. Don't place an x after the value. If you do, X2C converts each character to its EBCDIC hexadecimalrepresentation, before converting to EBCDIC. Leading and trailing spaces are not allowed, though you may code spaces between two hexadecimal numbers.

Examples

Function	Result
X2C('AA C4 86 4D')	':Df('
X2C('5C 7C')	'*@'
X2C('F5 C3')	'5C'
X2C('F5 C3'x)	'*'; 'F5 C3' converts to '5C', then '5C' converts to '*'.

X2D

The X2D function converts a hexadecimal string to its decimal value. If the decimal value is out of the PIE/REXX number range, you will get an error message.

```
X2D(hex,n)
```

hex	hexadecimal string to convert. Don't place an x after the value. If you do, X2D converts each character to its EBCDIC hexadecimalrepresentation, before converting to decimal. Leading and trailing spaces are not allowed, though you may code spaces between two hexadecimal numbers.
n	Length of the result. X2D truncates the result on the left or pad it with 0 characters. The resulting string is signed and calculated using the rules of two's complement. If you omit *n*, the resulting string is unsigned.

Examples

Function	Result
X2D('AA C4')	43716
X2D('F0 F1')	61681
X2D('F0 F1'x)	1
X2D('AA C4 86 4D')	Error

External Routines

External routines can be executed either with a function call or a CALL statement. If you execute them as a function call, you must include them in a full instruction, for example, make them part of an assignment.

External Routines/Functions

EXEC_CICS_DELETEQ	Delete a temporary storage queue
EXEC_CICS_LINK	Perform a CICS link to a program
EXEC_CICS_READQ	Return the temporary storage queue record requested
EXEC_CICS_READQTD	Return a transient data queue record
EXEC_CICS_SENDMAP	Send a map to the terminal
EXEC_CICS_SENDERASE	Clear the screen
EXEC_CICS_WRITEQ	Write a temporary storage queue record
EXEC_CICS_WRITEQTD	Write a transient data queue record
SCR_FIND	Locate a string in a session screen
SCR_GET_AID	Return the last entered AID key for a session
SCR_GET_CURSOR_COL	Return the column position of the cursor
SCR_GET_CURSOR_ROW	Return the row position of the cursor
SCR_GET_FIELD	Return a string from the screen
SCR_PUT_FIELD	Place a string in the specified position on a screen
SCR_PUT_PASSWORD	Send the user’s password to a dark field in the screen
SCR_RESHOW	Resend a session’s screen to the terminal
SCR_SET_CURSOR	Place the cursor at the specified position on a screen
SES_INPUT	Wait for user input
SES_LOCATE	Return a session ID
SES_NAME	Return a session name
SES_STATUS	Return the status of a session
SES_TITLE	Return a session title

All arguments are positional. If you omit the argument, code a comma as a place holder to maintain the correct sequence of subsequent arguments in the function string.

EXEC_CICS_DELETEQ

The EXEC_CICS_DELETEQ routine deletes the contents of a temporary storage queue. A value of 1 is returned if the queue exists and is deleted. A 0 is returned if the queue does not exist.

```
EXEC_CICS_DELETEQ(queue[,tsqsub])
```

queue	Name of a temporary storage queue whose contents are deleted.
tsqsub	Status of temporary storage queue substitution. See " TSQFLDS and TERMASK: Substitute Temporary Storage Queues" on page 186 of the Customization Reference for more information about TS queue substitution.

T	Temporary storage queue substitution is allowed.
N	Temporary storage queue substitution is prohibited.

Example

-
  ```
  Queue='WXYZ' || ZTERM
  CALL EXEC_CICS_DELETEQ queue
  ```

  These instructions delete a user's temporary storage queue for transaction WXYZ. The queue name is built by concatenating the transaction code and the PIE/CICS variable ZTERM, which retrieves the terminal ID.

EXEC_CICS_LINK

The EXEC_CICS_LINK function performs a CICS link to a program. Control returns to the PIE/REXX program after the linked program ends.

```
EXEC_CICS_LINK(program,commarea)
```

program	Name of the linked program.
commarea	Name of the commarea whose contents are passed to the linked program.

Example

- `Funct = EXEC_CICS_LINK(PCTSPGMM,upload_parms)`
  This function links to the PIE/CICS upload utility and passes the information in upload_parms.

EXEC_CICS_READQ

The EXEC_CICS_READQ function reads an item from a temporary storage queue. If CICS gives an ITEMERR response, it returns a null string. If the queue has no more records, it returns a null string. For any other CICS error response, an error message is received.

```
EXEC_CICS_READQ(queue[,item#][,tsqsub])
```

queue	Name of a temporary storage queue.
item#	Number of the item to be read from the queue. If an item number is not specified, READQ reads the next sequential item.
tsqsub	Status of temporary storage queue substitution. See “ TSQFLDS and TERMASK: Substitute Temporary Storage Queues” on page 186 of the Customization Reference for more information on TS queue substitution.

T	Temporary storage queue substitution is allowed.
N	Temporary storage queue substitution is prohibited.

Example

-
  ```
  Queue='WXYZ' || ZTERM
  Do i=item# To item# + 9

        Funct = EXEC_CICS_READQ(queue,i)

        END
  ```

  This DO loop reads the next ten items in the queue. EXEC_CICS_READQ is executed as a function.

  The queue name is built by concatenating the transaction code and the PIE/CICS variable ZTERM (which retrieves the terminal ID).

EXEC_CICS_READQTD

The EXEC_CICS_READQTD function reads a transient data queue record. It returns the next record in the queue. If the queue has no more records, it returns a null string. For any other CICS error response, an error message is received.

```
EXEC_CICS_READQTD(queue)
```

queue	Name of a transient data queue.

Example

-
  ```
  Do 10
    Funct = EXEC_CICS_READQTD(TDQ1)
    END
  ```
  This DO loop reads the next ten items from the TDQ1 queue.

EXEC_CICS_SENDMAP

The EXEC_CICS_SENDMAP function sends a BMS map. It returns 1 if the send was successful, 0 if it was not.

This function sends the map only, not the data for the map. To send data, use the SCR_PUT_FIELD function instead.

```
EXEC_CICS_SENDMAP(mapname[,mapset])
```

mapname	Name of the BMS map to send.
mapset	BMS mapset containing this map.

Example

- `CALL EXEC_CICS_SENDMAP PC@@0PO`
  This instruction sends the PC@@0PO map to the terminal.

EXEC_CICS_SENDERASE

The EXEC_CICS_SENDERASE function clear the current session screen. It always returns 1.

```
EXEC_CICS_SENDERASE
```

EXEC_CICS_WRITEQ

The EXEC_CICS_WRITEQ function writes a temporary storage queue item. It returns 1 if the write was successful, 0 if CICS gives an ITEMERR response. For any other CICS error response, an error message is received.

```
EXEC_CICS_WRITEQ(queue,string[,item#][,location][,tsqsub])
```

queue	Name of a temporary storage queue.
string	String written as a record to a temporary storage queue.
item#	Item number to be written. If the item exists, it is overwritten. If you do not specify an item number, WRITEQ writes a new item at the end of the queue.
loc	Storage location (MAIN or AUX.) to write the queue, AUX is the default. If the queue already exists, loc is ignored.
tsqsub	Temporary storage queue substitution specification.

T	Allow temporary storage queue substitution.
N	Prohibit temporary storage queue substitution. See “ TSQFLDS and TERMASK: Substitute Temporary Storage Queues” on page 186 of the Customization Reference for more information on TS queue substitution.

Example

-
  ```
  Queue='WXYZ' || ZTERM
  Funct = EXEC_CICS_WRITEQ(queue,'12345 DOE JOHN')
  ```

  This WRITEQ instruction writes a record containing, “12345 DOE JOHN”, at the end of this user's WXYZ queue. The queue name is built by concatenating the transaction code and the PIE/CICS variable ZTERM (which retrieves the terminal ID).

EXEC_CICS_WRITEQTD

The EXEC_CICS_WRITEQTD function writes a transient data queue record. It returns 1 if the write was successful. For any other CICS error response, an error message is returned.

```
EXEC_CICS_WRITEQTD(queue,string)
```

queue	Name of a transient data queue.
string	String written as a record to a transient data queue.

Example

- `Funct = EXEC_CICS_WRITEQTD(PIEL,'12345 DOE JOHN')`
  This WRITEQTD instruction writes a record containing, “12345 DOE JOHN”, at the end of the PIEL queue.

SCR_FIND

The SCR_FIND function locates a string on a session screen. It returns 1 if the string is found and places the cursor at the first position of the string on the screen. If the string is not found, it leaves the cursor in its current position and returns 0.

```
SCR_FIND(id,string[,row,col][,opt])
```

id	Session ID. If the session is not allocated, an error message is received. If you specify '*' or the null string, PIE/REXX accesses the current session.
row,col	Location of the field on the screen to begin the search. The default is the current cursor position.
string	String to search for. If you specify a null string, SCR_FIND returns a 1 and moves the cursor as specified in `opt`.
opt	Cursor screen position after the string is found.

S	Cursor is placed on the first character of the string after it is found. This is the default.
I	Cursor is placed at the beginning of the next input field.
F	Cursor is placed at the beginning of the next field after the string, regardless of the field type.

Example

- `CALL SCR_FIND '*','===>'`
  This instruction searches the current session screen for '===>'.

SCR_GET_AID

The SCR_GET_AID function returns the last entered AID key for a session. The AID value is a five-character string, padded with blanks on the right, for example, ’PF1 ’.

```
SCR_GET_AID(id)
```

id	Session ID. If the session is not allocated, an error message is received. If you specify ’*’ or the null string, SCR_GET_AID accesses the current session.

Example

- `CALL SCR_GET_AID 7`
  This instruction returns the last entered AID key for session 7.

SCR_GET_CURSOR_COL

The SCR_GET_CURSOR_COL function returns the current column position of the cursor in a session.

```
SCR_GET_CURSOR_COL(id)
```

id	Session ID. If the session is not allocated, an error message is received. If you specify ’*’ or the null string, PIE/REXX accesses the current session.

Example

- `CALL SCR_GET_CURSOR_COL ‘*’`
  The cursor column position of the current session is returned.

SCR_GET_CURSOR_ROW

The SCR_GET_CURSOR_ROW function returns the row position of the cursor in a session.

```
SCR_GET_CURSOR_ROW(id)
```

id	Session ID. If the session is not allocated, you will get an error message. If you specify '*' or the null string, PIE/REXX will access the current session.

Example

- `CALL SCR_GET_CURSOR_ROW ‘’`
  This instruction returns the cursor row position for the current session.

SCR_GET_FIELD

The SCR_GET_FIELD function retrieves data from a field on a session screen.

```
SCR_GET_FIELD(id[,row,col][,length])
```

id	Session ID. If the session is not allocated, an error message is received. If you specify '*' or the null string, PIE/REXX accesses the current session.
row,col	Beginning location of the field on the screen. The default is the current cursor position.
length	Length of the string to return. The string may span lines on the screen. If the requested length exceeds the right boundary of the screen, the returned string is padded with blanks on the right. If length is not supplied, SCR_GET_FIELD return all characters up to the next attribute byte (that is, the rest of the screen field).

Example

- `CALL SCR_GET_FIELD 5,,,8`
  This instruction retrieves 8 characters from the screen in session 5. It begins retrieving data from the current cursor position.

SCR_PUT_FIELD

The SCR_PUT_FIELD function writes a string to a field of a session screen. It returns 1 if the update was successful, 0 if it was not.

After making changes, execute the KEY command to process your change, or execute SCR_RESHOW to refresh the session screen.

```
SCR_PUT_FIELD(id,string[,row,col][,upd][,pro][,clr][,hl])
```

id	Session ID. If the session is not allocated, an error message is received. If you specify '*' or the null string, PIE/REXX accesses the current session.
string	String to be written to the screen. If the string exceeds the boundaries of an attribute field, it is truncated to fit within the data space of the field (unless upd is A).
row,col	Beginning location of the field on the screen. The default is the current cursor position.
upd	Update code to control updatable fields. The following codes restrict updates to:

U	Unprotected fields, the default. The MDT attribute will be set on for the field updated.
D	Dark attribute fields.
P	Protected fields.
A	All fields. SCR_PUT_FIELD writes the entire string to the screen, even when the string overwrites attribute codes.

pro

P	Updated field is protected.
U	Updated field is unprotected.
Omit	Updated field maintains its current status. If you change the status, SCR_PUT_FIELD will specify an attribute option immediately following the string to reset the rest of the field to its previous status.

clr Color code for the field.

B	blue
G	Green
P	Pink
R	Red
T	Turquoise
V	Violet
W	White

	Y	Yellow
	D	as is
hl	Highlight code for the field.	
	F	Flashing
	R	Reverse video
	U	Underline
	N	As is (neutral)

Example

-
  ```
  CALL SCR_PUT_FIELD 8,total,,,,,'T'
  CALL SCR_RESHOW 8
  ```

  The SCR_PUT_FIELD instruction writes the value for total on the session 8 screen, beginning at the current cursor position. The color of the string is changed to turquoise. The SCR_RESHOW instruction refreshes the screen.

SCR_PUT_PASSWORD

SCR_PUT_PASSWORD writes a user password to a dark screen field. If the field is not dark, an error message is received. If the password exceeds the length of the screen field, it is truncated on the right to the length of the field.

```
SCR_PUT_FIELD(id[,row,col])
```

id	Session ID. If the session is not allocated, an error message is received. If you specify ’*’ or the null string, PIE/REXX accesses the current session.
row,col	Beginning row and column position of the password field. If omitted, the password is entered at the current cursor position.

Example

- `CALL SCR_PUT_PASSWORD 8`
  The SCR_PUT_PASSWORD instruction writes the password to the session 8 screen at the current cursor position.

SCR_RESHOW

The SCR_RESHOW function refreshes a session screen. It makes changes made with SCR_PUT_FIELD and SCR_PUT_CURSOR visible to the user. You can also use it to send your own datastream.

SCR_RESHOW always returns 1.

```
SCR_RESHOW(id[,datastream[,wcc]])
```

id	Session ID. If the session is not allocated, an error message is received. If you specify '*' or the null string, PIE/REXX accesses the current session.
datastream	3270 datastream to send.
wcc	Indicate whether the datastream contains write control characters. Specify C if it does or N if it does not.

Example

-
  ```
  CALL SCR_PUT_FIELD *,total,,,,,'T'
  CALL SCR_RESHOW *
  ```

  The Reshow instruction refreshes the current session screen after a SCR_PUT_FIELD.

SCR_SET_CURSOR

The SCR_SET_CURSOR function places the cursor at the specified position. It always returns 1.

```
SCR_SET_CURSOR(id,row,col)
```

id	Session ID. If the session is not allocated, you will get an error message. If you specify '*' or the null string, PIE/REXX accesses the current session.
row,col	Screen row and column position where the cursor is placed.

Example

- `CALL SCR_SET_CURSOR '*',20,12`
  This instruction places the cursor at row 20 and column 12 on the current session screen.

SES_INPUT

The SES_INPUT function waits for user input. A PIE/REXX program goes into a pseudo-conversational wait for input from the terminal.

SES_INPUT returns 1 after the user hits an AID key.

☞ SES_INPUT is for use in standard mode only. For script mode, use the INPUT command instead. See “ INPUT” on page 120.

```
SES_INPUT()
```

Examples

-
  ```
  CALL SENDMAP ‘USERMAP’
  CALL SES_INPUT ‘’
  DATA = SCR_GET_FIELD(‘’,10,10)
  ```

  This example sends a map, waits for input from the user, and then assigns the field at row 10, column 10, to the DATA. variable

SES_LOCATE

The SES_LOCATE function returns the ID of a requested session or the maximum number of sessions allowed. SES_LOCATE returns 0 if the session is not found,

```
SES_LOCATE(type[,string])
```

type Type of session you are looking for.

P	Previous session, the session used just before the current session
C	Current session
O	Oldest session
M	Maximum number of sessions allowed
N	Name, the first session whose name matches the supplied string
T	Title, the first session whose title matches the supplied string

string Name or title of the session, if you specified N or T for type.

Examples

- `CALL SES_LOCATE C`
  This instruction returns the ID of the current session.
- `CALL SES_LOCATE N,'WXYZ'`
  This instruction returns the ID of the session named WXYZ.

SES_NAME

SES_NAME returns the name of a session as given on the `Name` field of the Sessions menu. A null string is returned if the session does not exist.

```
SES_NAME(id)
```

id	Session ID. If the session is not allocated, you will get an error message. If you specify '*' or the null string, PIE/REXX will access the current session.

Example

- `CALL SES_NAME 3`
  This instruction returns the name of session 3.

SES_STATUS

SES_STATUS returns the status of a session. It returns OPEN, ALLOCATED (allocated but not open), or FREE (de-allocated and closed).

```
SES_STATUS(id)
```

id	Session ID. If the session is not allocated, you will get an error message. If you specify '*' or the null string, PIE/REXX will access the current session.

Example

- `CALL SES_STATUS 3`
  This instruction returns the status of session 3.

SES_TITLE

SES_TITLE returns the title of a session listed in the `Title` field of the Sessions menu. A null string is returned if the session does not exist.

```
SES_TITLE(id)
```

id	Session ID. If the session is not allocated, you receive an error message. If you specify '*' or the null string, PIE/REXX returns the title of the current session.

Example

- `CALL SES_TITLE 3`
  This instruction returns the title of session 3 as it is listed on the `Title` field of the Sessions menu.

Special Host Commands

The special host commands are valid in script mode only.

DELAY	Wait the specified number of seconds for terminal input
INPUT	Wait for terminal input
KEY	Send an AID key to the terminal
WAIT	Wait for input from the application

DELAY

The DELAY command waits a specified period of time for terminal input from the user or an application. DELAY automatically reshows the current screen during the wait period.

If there is a response within the specified delay period, the screen buffer is updated with any changes made to the screen data by the user. However, these changes are not sent to the application until the REXX program issues the KEY command. Using KEY after DELAY extracts data from all changed input fields and sends the changes to the application.

If there is no response within the DELAY period, control returns to the REXX script.

```
DELAY n
```

n	Number of seconds to wait for input from a user or application. The maximum is 359999.

Example

- `DELAY 60`
  This instruction waits 60 seconds for a response from the user or an application.

INPUT

The INPUT command waits for input from the user or application. INPUT automatically reshows the current screen. The program waits indefinitely for a response.

If the user changes data shown on a screen, the screen buffer is updated with the changes. However, these changes are not sent to the application until the REXX KEY command is executed. Using the INPUT and KEY command sequence extracts data from all changed input fields and sends it to the application.

```
INPUT
```

Example

-
  ```
  INPUT
  KEY ENTER
  ```

  The INPUT instruction waits for a user or application response—a user response is intended. After the response is received, the program sends it to the application with the KEY command. The application responds as if the user pressed ENTER.

KEY

The KEY command resumes an application waiting for input. The KEY command simulates a user pressing an AID key, such as ENTER, CLEAR, or PF3, from the terminal. KEY extracts any changed data from input fields in the screen buffer and passes it to the application with the resumption of the application.

```
KEY key
```

key	AID key passed to an application. Possible AID keys are:
	ENTER
	CLEAR
	PA01-PA03
	PF01-PF24

Example

- `KEY PF03`
  This instruction passes PF3 to the application to exit from the current screen or end the application.

WAIT

The WAIT command suspends program execution until the screen is updated by the user or an application. WAIT is useful for applications that update the terminal' screen automatically with the EXEC CICS START command. CICS performance monitoring products like Omegamon and TMON use EXEC CICS START to automatically refresh the screen.

WAIT accepts a response from either the user or the application. If the user changes the data on the screen, those changes are discarded. WAIT does not reshow the current application screen.

```
WAIT
```

Example

- `WAIT`
  The program is placed in a wait state pending a screen refresh, either as an automatic refresh by the application or an AID key pressed by the user.

Appendix A Customer Support

This appendix describes how to get help from Customer Service when you experience a problem with a UNICOM Systems, Inc. product. This appendix includes separate sections that describe several diagnostic suggestions to rule out user errors and the information you should have ready before reporting the problem.

Contacting Customer Service

UNICOM Systems, Inc. Customer Service can be reached by the following methods:

Voice	818-838-0606
Fax	818-838-0776
E-mail	support@unicomsi.com

A Support and Services web page provides Customer Service information about all of UNICOM Systems, Inc.'s products. Use the following URL to browse the Support and Services web page:

http://www.unicomsi.com/support/index.html

The Support and Services web page provides an online form to report a problem with a UNICOM Systems, Inc. product. Use the following URL to complete and submit a Technical Support Request form:

http://www.unicomsi.com/support/index.html

Normal business hours are from 6:00 a.m. to 5:00p.m. Pacific Standard Time, Monday through Friday. Emergency customer service is available 24 hours a day, 7 days a week.

An answering service receives customer service calls beyond normal business hours. You may leave a message if it is not an urgent problem. A customer service representative will return your call at the start of the next business day.

Requests for urgent support outside of normal business hours are answered immediately. A customer service representative will be summoned to return your call. Leave a phone number where you can be reached. If you have not received a return call from a Customer Service representative within an hour of reporting the problem, please call back. Our customer service representative may be experiencing difficulties returning your call.

International customers should contact their local distributor to report any problems with a UNICOM Systems, Inc. product.

Troubleshooting Suggestions

This section describes several troubleshooting suggestions to diagnose common errors that can cause PIE/CICS problems. Before calling Customer Service, follow these suggestions to rule out the possibility these errors are causing your PIE/CICS problem.

- Run the Installation Verification Program with the **P#IV** transaction. Browse the PIECIVP temporary storage queue to see if it contains error messages that suggest problems with the allocation of CICS programs, transactions, maps, or files.
- Verify any recent changes to your site’s operating system, CICS, or other products are fully compatible with PIE/CICS.
- Verify that all load modules are at the same release level if a new release of PIE/CICS was installed over a previous release.
- Check that all modules were reassembled after upgrading PIE/CICS or applying maintenance to CICS.
- Verify that all PIE/CICS system tables were reassembled after applying maintenance to CICS or upgrading to another release.
- Verify that all users have current PIE/CICS passwords and entered them correctly.
- Examine your CICS logs, MVS console, and PIE/CICS logs for error messages from not only PIE/CICS, but any other product that runs concurrently with PIE/CICS.

Describing the Problem

Gather the following information about your system environment before reporting a problem to UNICOM Systems, Inc. Customer Service:

Operating system release and PUT Level ___________________________________

VTAM system release and PUT Level ______________________________________

PIE/CICS release____

Date of PIE/CICS distribution tape ____

Gather the following information about your CICS system before reporting a problem to UNICOM Systems, Inc. Customer Service:

CICS release and PUT Level ____

CICS configuration MRO/ISC etc._______________________________________

Real or virtual terminal ____

Before calling UNICOM Systems, Inc. Customer Service, get answers to the following questions.

What PIE/CICS products were active when the problem occurred?

Availability Plus
Dynamic Menus
MultiCICS
NetGate
NetMizer
NonStop CICS

Is the problem occurring in the TOR or AOR? ______________________________

Is the problem occurring in a production or test region? ______________________

What is the severity of the problem? ______________________________________

What are the major symptoms of the problem? ______________________________

____

Is the problem re-creatable under specific conditions?________________________

Has the problem occurred more than once? ________________________________

Were changes made to CICS or PIE/CICS immediately prior to the occurrence of the problem? ____

What other software products were running when the problem occurred? _______

____

Is a diagnostic message produced when the problem occurs? If so, what is the ID and text of the messages? ____

Does an abend occur? If so, what are the abend and return codes? ___________

Is a dump produced when the problem occurs? If so, what kind of dump is it? __

____

Please try to be as accurate and complete as possible in answering these questions. Your problem can be resolved more quickly if a customer service representative has all of the pertinent information needed to find a solution.

Appendix B. Sample PIE/REXX Programs

This appendix describes sample REXX programs stored as text members in the PIE/CICS Repository. They are assigned to the SYSTEM group within the Repository.

Enhanced Cut

The following program copies a text block from a screen to temporary storage.

```
/*************************************************************/
/*   REXX Program CUT.REXX: program to allow a block of text to */
/*                          be cut from a screen and placed in  */
/*                          a TS queue to be accessed for paste */
/*                          etc.                                */
/*       This program will either mark the corner for a block   */
/*       cut or will cut the block of data from the prior       */
/*       marked corner to the supplied corner.                  */
/*                                                              */
/*       To invoke you can do one of the following:             */
/*                                                              */
/*       "CMD REXX CUT.REXX r1 c1 r2 c2"                        */
/*       - this will open a new temporary session while cutting */
/*       "ESCAPE CMD REXX CUT.REXX r1 c1 r2 c2"                 */
/*       - this will use session zero to do the cut             */
/*                                                              */
/*       r1, c1, r2, c2 are optional row column values to do    */
/*       the cut with. If not supplied the current cursor       */
/*       location is used if only one row column value is given */
/*       (as is the case if you use the current location) then  */
/*       the first invocation saves the value so that the       */
/*       second invocation will perform the cut to the new      */
/*       cursor location.                                       */
/*************************************************************/
/*------------------------------------*/
/* extract any optional row column    */
/* values                             */
/*------------------------------------*/
parse arg r1 c1 r2 c2 rest

/*------------------------------------*/
/* get the prior session to do the    */
/* cut from                           */
/*------------------------------------*/
cut_sess = ses_locate('p')
```

```
/*-----------------------------------*/
/* set the TS queue name to cut to    */
/*-----------------------------------*/
ts_queue = ztsqprf||C||zlterm

/*-----------------------------------*/
/* no row column values supplied so   */
/* use the current cursor location    */
/* for one corner                     */
/*-----------------------------------*/
if r1 = '' then do
    r1=scr_get_cursor_row(cut_sess)
    c1=scr_get_cursor_col(cut_sess)
    end

/*-----------------------------------*/
/* only one set of row column values  */
/* supplied so see if we have saved   */
/* a second.                          */
/*-----------------------------------*/
if r2 = '' then do
    /*                                */
    /* go set the second cut values   */
    /* if they didn't get set we are  */
    /* done                           */
    /*                                */
    call set_cut_to
    if r2='' then exit
    end

/*-----------------------------------*/
/* validate the row column values     */
/*-----------------------------------*/
if datatype(r1)\='NUM' | datatype(c1)\='NUM' | ,
    datatype(r2)\='NUM' | datatype(c2)\='NUM' then do
    say 'Invalid row/column value supplied'
    exit
    end

/*-----------------------------------*/
/* cut the data and save in the ts    */
/* queue                              */
/*-----------------------------------*/
call cut_data

exit

/*************************************/
/* this subroutine will extract the    */
/* cut to row column values from the   */
/* control TS queue. if there is no    */
/* queue or if the session has changed */
/* then return with the row column     */
/* not set                             */
/*************************************/
set_cut_to:
    /*                                 */
    /* get the TS queue record for the */
    /* cut and extract the session id  */
    /* for the last cut                */
    /*                                 */
    ts_rec=exec_cics_readq(ts_queue,1,n)
    saved_sess = substr(ts_rec,1,2)
    /*                                 */
    /* if we don't match session ID    */
    /* (they won't match if no prior   */
    /* cut was done) then save the     */
    /* values we have cut here         */
    /*                                 */
    if cut_sess \= saved_sess then do
       call set_ts_control cut_sess,r1,c1
       end
    /*                                 */
    /* otherwise set the cut to values */
    /*                                 */
    else do
       r2 = substr(ts_rec,3,2)
       c2 = substr(ts_rec,5,2)
       end

    return

/*************************************/
/* this subroutine will cut the data  */
/* block specified by r1,c1 and r2,c2 */
/* into the ts queue                  */
/*************************************/
cut_data:
    /*                                 */
    /* first save the control record   */
    /*                                 */
    call set_ts_control cut_sess,r1,c1
    /*                                 */
    /* check to see if we are looping  */
    /* forward or backward             */
    /*                                 */
    from_row = r1
    to_row = r2
    if r1>r2 then do
       from_row = r2
       to_row = r1
       end
    /*                                    */
    /* loop thru the rows cutting the     */
    /* text and saving.                   */
    /*                                    */
    do i=from_row to to_row
       /*                                 */
       /* get a line of text from the     */
       /* screen between the supplied     */
       /* columns                         */
       /*                                 */
       if c1<c2 then cut_text=scr_get_field(cut_sess,i,c1,c2-c1+1)
       else cut_text=scr_get_field(cut_sess,i,c2,c1-c2+1)
       /*                                 */
       /* save the text in the ts queue */
       /*                                 */
       call exec_cics_writeq ts_queue,cut_text,,,n
       end
    return
       end
    /*                                    */
    /* loop thru the rows cutting the     */
    /* text and saving.                   */
    /*                                    */
    do i=from_row to to_row
       /*                                 */
       /* get a line of text from the     */
       /* screen between the supplied     */
       /* columns                         */
       /*                                 */
       if c1<c2 then cut_text=scr_get_field(cut_sess,i,c1,c2-c1+1)
       else cut_text=scr_get_field(cut_sess,i,c2,c1-c2+1)
       /*                                 */
       /* save the text in the ts queue */
       /*                                 */
       call exec_cics_writeq ts_queue,cut_text,,,n
       end
    return
/****************************************/
/* this subroutine will build the       */
/* control record for the ts queue      */
/* with the session id and cut row      */
/* column value                         */
/****************************************/
set_ts_control:
    parse arg session,row,column
    ts_rec = substr(session,1,2)||substr(row,1,2)||,
             substr(column,1,2)
    call exec_cics_deleteq ts_queue,n
    call exec_cics_writeq ts_queue,ts_rec,,,n
    return
```

Enhanced Paste

The following program pastes a block of text data stored in a temporary storage queue to the current screen. Use it following a PIE/REXX cut operation.

```
/*****************************************************************/
/*  REXX Program PASTE.REXX: program to allow a block of text   */
/*                           to be pasted from a TS queue to    */
/*                           the current screen                 */
/*                                                               */
/*       This program will paste a block of text from a TS      */
/*       queue either to the supplied location, or to the       */
/*       cursor location                                         */
/*       To invoke you can do one of the following:              */
/*                                                               */
/*       "CMD REXX PASTE.REXX r1 c1"                             */
/*       - this will open a new temporary session while pasting */
/*       "ESCAPE CMD REXX PASTE.REXX r1 c1"                      */
/*       - this will use session zero to do the paste            */
/*                                                               */
/*       r1, c1 are optional row column values to paste into.   */
/*       If not supplied the current cursor location is          */
/*       used.  The block will be pasted within unprotected     */
/*       fields. This can be modified if required by changing   */
/*       the options on the SCR_PUT_FIELD request.               */
/*                                                               */
/*****************************************************************/
/*-------------------------------------*/
/* extract any optional row column     */
/* values                              */
/*-------------------------------------*/
parse arg r1 c1 rest

/*-------------------------------------*/
/* get the prior session to do the     */
/* paste to                            */
/*-------------------------------------*/
paste_sess = ses_locate('p')

/*-------------------------------------*/
/* set the TS queue name to paste from */
/*-------------------------------------*/
ts_queue = ztsqprf||C||zlterm

/*-------------------------------------*/
/* no row column values supplied so    */
/* use the current cursor location     */
/* for one corner                      */
/*-------------------------------------*/
if r1 = '' then do
```

```
    r1=scr_get_cursor_row(paste_sess)
    c1=scr_get_cursor_col(paste_sess)
    end

/*------------------------------------*/
/* validate the row column values     */
/*------------------------------------*/
if datatype(r1)\='NUM' | datatype(c1)\='NUM' then do
    say 'Invalid row/column value supplied'
    exit
    end

/*------------------------------------*/
/* paste the data into the screen     */
/*------------------------------------*/
ts_rec = exec_cics_readq(ts_queue,2,n)
do while ts_rec \== ''
    call scr_put_field paste_sess,ts_rec,r1,c1
    r1=r1+1
    ts_rec = exec_cics_readq(ts_queue,,n)
    end
```

Index