z/OS
Version 2 Release 3

TSO/E REXX User's Guide

SA32-0982-30

Note

Before using this information and the product it supports, read the information in “Notices” on page 193.

This edition applies to Version 2 Release 3 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2019-02-16

Contents

List of Figures

List of Tables

About this document

This book describes how to use the TSO/E Procedures Language MVS/REXX processor (called the language processor) and the REstructured eXtended eXecutor (REXX) language. Together, the language processor and the REXX language are known as TSO/E REXX. TSO/E REXX is the implementation of the Systems Application Architecture® (SAA) Procedures Language on the MVS™ system.

Who should use this document

This book is intended for anyone who wants to learn how to write REXX programs. More specifically, the audience is programmers who may range from the inexperienced to those with extensive programming experience, particularly in writing CLISTs for TSO/E. Because of the broad range of experience in readers, this book is divided into two parts.

- Part 1, "Learning the REXX Language," on page 1 is for inexperienced programmers who are somewhat familiar with TSO/E commands and have used the Interactive System Productivity Facility/ Program Development Facility (ISPF/PDF) in TSO/E. Programmers unfamiliar with TSO/E should first read the *z/OS TSO/E Primer*. Experienced programmers new to REXX can also read this section to learn the basics of the REXX language.
- Part 2, "Using REXX," on page 91 is for programmers already familiar with the REXX language and experienced with the workings of TSO/E. It describes more complex aspects of the REXX language and how they work in TSO/E as well as in other MVS address spaces.

If you are a new programmer, you might want to concentrate on the first part. If you are an experienced TSO/E programmer, you might want to read the first part and concentrate on the second part.

How this document is organized

In addition to the two parts described in the preceding paragraphs, there are three appendixes at the end of the book.

- Appendix A, "Allocating Data Sets," on page 171 contains checklists for the tasks of creating and editing a data set and for allocating a data set to a system file.
- Appendix B, "Specifying Alternate Libraries with the ALTLIB Command," on page 181 describes using the ALTLIB command.
- Appendix C, "Comparisons Between CLIST and REXX," on page 183 contains tables that compare the CLIST language with the REXX language.

Terminology

Throughout this book a REXX program is called an exec to differentiate it from other programs you might write, such as CLISTs. The command to run an exec in TSO/E is the EXEC command. To avoid confusion between the two, this book uses lowercase and uppercase to distinguish between the two uses of the term "exec". References to the REXX program appear as `exec` and references to the TSO/E command appear as EXEC.

Purpose of each chapter

At the beginning of each chapter is a statement about the purpose of the chapter. Following that are headings and page numbers where you can find specific information.

Examples

Throughout the book, you will find examples that you can try as you read. If the example is a REXX keyword instruction, the REXX keyword is in uppercase. Information that you can provide is in lowercase. The following REXX keyword instruction contains the REXX keyword SAY, which is fixed, and a phrase, which can vary.

```
SAY 'This is an example of an instruction.'
```

Similarly, if the example is a TSO/E command, the command name and keyword operands, which are fixed, are in uppercase. Information that can vary, such as a data set name, is in lowercase. The following ALLOCATE command and its operands are in uppercase and the data set and file name are in lowercase.

```
"ALLOCATE DATASET(rexx.exec) FILE(sysexec) SHR REUSE"
```

This use of uppercase and lowercase is intended to make a distinction between words that are fixed and words that can vary. It does *not* mean that you must type REXX instructions and TSO/E commands with certain words in uppercase and others in lowercase.

Exercises

Periodically, you will find sections with exercises you can do to test your understanding of the information. Answers to the exercises are included when appropriate.

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS®, including the documentation available for z/OS TSO/E.

How to send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead "If you have a technical problem" on page xv.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function
If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

Feedback on IBM® Knowledge Center function
If your comment or question is about the IBM Knowledge Center functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Knowledge Center Support at ibmkc@us.ibm.com.

Feedback on the z/OS product documentation and content
If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS TSO/E REXX User's Guide, SA32-0982-30
- The section title of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Summary of changes

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line to the left of the change.

Summary of changes for TSO/E for Version 2 Release 3 (V2R3) and its updates

This information contains no technical changes for this release.

Summary of changes for TSO/E for Version 2 Release 2 (V2R2) and its updates

This information contains no technical changes for this release.

z/OS Version 2 Release 1 summary of changes

See the Version 2 Release 1 (V2R1) versions of the following publications for all enhancements related to z/OS V2R1:

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Part 1. Learning the REXX Language

The REXX language is a versatile general-purpose programming language that can be used by new and experienced programmers. This part of the book is for programmers who want to learn the REXX language. The chapters in this part cover the following topics.

- Chapter 1, "Introduction," on page 3 — The REXX language has many features that make it a powerful programming tool.
- Chapter 2, "Writing and Running a REXX Exec," on page 7 — Execs are easy to write and have few syntax rules.
- Chapter 3, "Using Variables and Expressions," on page 23 — Variables, expressions, and operators are essential when writing execs that do arithmetic and comparisons.
- Chapter 4, "Controlling the Flow Within an Exec," on page 39 — You can use instructions to branch, loop, or interrupt the flow of an exec.
- Chapter 5, "Using Functions," on page 57 — A function is a sequence of instructions that can perform a specific task and must return a value.
- Chapter 6, "Writing Subroutines and Functions," on page 65 — You can write internal and external routines that are called by an exec.
- Chapter 7, "Manipulating Data," on page 81 — Compound variables and parsing are two ways to manipulate data.

Note: Although you can write a REXX exec to run in a non-TSO/E address space in MVS, the chapters and examples in this part assume the exec will run in a TSO/E address space. If you want to write execs that run outside of a TSO/E address space, keep in mind the following exceptions to information in Part 1:

- An exec that runs outside of TSO/E cannot include TSO/E commands, unless you use the TSO/E environment service (see note).
- In TSO/E, several REXX instructions either display information on the terminal or retrieve information that the user enters at the terminal. In a non-TSO/E address space, these instructions get information from the input stream and write information to the output stream.
  - SAY — this instruction sends information to the output DD whose default is SYSTSPRT.
  - PULL — this instruction gets information from the input DD whose default is SYSTSIN.
  - TRACE — this instruction sends information to the output DD whose default is SYSTSPRT.
  - PARSE EXTERNAL — this instruction gets information from the input DD whose default is SYSTSIN.
- The USERID built-in function, instead of returning a user identifier, might return a stepname or jobname.

Note: You can use the TSO/E environment service, IKJTSOEV, to create a TSO/E environment in a non-TSO/E address space. If you run a REXX exec in the TSO/E environment you created, the exec can contain TSO/E commands, external functions, and services that an exec running in a TSO/E address space can use. That is, the TSO host command environment (ADDRESS TSO) is available to the exec. For more information about the TSO/E environment service and the different considerations for running REXX execs within the environment, see *z/OS TSO/E Programming Services*.

Chapter 1. Introduction

This chapter describes the REXX programming language and some of its features.

What is REXX?

REXX is a programming language that is extremely versatile. Aspects such as common programming structure, readability, and free format make it a good language for beginners and general users. Yet because the REXX language can be intermixed with commands to different host environments, provides powerful functions and has extensive mathematical capabilities, it is also suitable for more experienced computer professionals.

The TSO/E implementation of the REXX language allows REXX execs to run in any MVS address space. You can write a REXX exec that includes TSO/E services and run it in a TSO/E address space, or you can write an application in REXX to run outside of a TSO/E address space. For more information, see Chapter 13, "Using REXX in TSO/E and Other MVS Address Spaces," on page 159.

There is also a set of z/OS UNIX extensions to the TSO/E Restructured Extended Executor (REXX) language which enable REXX programs to access z/OS UNIX callable services. The z/OS UNIX extensions, called syscall commands, have names that correspond to the names of the callable services that they invoke—for example, access, chmod, and chown. For more information about the z/OS UNIX extensions, see *z/OS Using REXX and z/OS UNIX System Services*.

Features of REXX

In addition to its versatility, REXX has many other features, some of which are:

Ease of use

The REXX language is easy to read and write because many instructions are meaningful English words. Unlike some lower-level programming languages that use abbreviations, REXX instructions are common words, such as SAY, PULL, IF... THEN... ELSE..., DO... END, and EXIT.

Free format

There are few rules about REXX format. You need not start an instruction in a particular column, you can skip spaces in a line or skip entire lines, you can have an instruction span many lines or have multiple instructions on one line, variables do not need to be predefined, and you can type instructions in upper, lower, or mixed case. The few rules about REXX format are covered in "Syntax of REXX Instructions" on page 8.

Convenient built-in functions

REXX supplies built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations.

Debugging capabilities

When a REXX exec running in TSO/E encounters an error, messages describing the error are displayed on the screen. In addition, you can use the REXX TRACE instruction and the interactive debug facility to locate errors in execs.

Interpreted language

TSO/E implements the REXX language as an interpreted language. When a REXX exec runs, the language processor directly processes each language statement. Languages that are not interpreted must be compiled into machine language and possibly link-edited before they are run. You can use the IBM licensed product, IBM Compiler and Library for REXX/370, to provide this function.

Extensive parsing capabilities

REXX includes extensive parsing capabilities for character manipulation. This parsing capability allows you to set up a pattern to separate characters, numbers, and mixed input.

Components of REXX

The various components of REXX are what make it a powerful tool for programmers. REXX is made up of:

- Instructions — There are five types of instructions. All but commands are processed by the language processor.
  - Keyword
  - Assignment
  - Label
  - Null
  - Command (both TSO/E REXX commands and host commands)
- Built-in functions — These functions are built into the language processor and provide convenient processing options.
- TSO/E external functions — These functions are provided by TSO/E and interact with the system to do specific tasks for REXX.
- Data stack functions — A data stack can store data for I/O and other types of processing.

The SAA Solution

The SAA solution is based on a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications.

The SAA Procedures Language has been defined as a subset of the REXX language. Its purpose is to define a common subset of the language that can be used in several environments. TSO/E REXX is the implementation of the SAA Procedures Language on the MVS system.

The SAA solution:

- Defines a **common programming interface** you can use to develop applications that can be integrated with each other and transported to run in multiple SAA environments.
- Defines **common communications support** that you can use to connect applications, systems, networks, and devices.
- Defines a **common user access** that you can use to achieve consistency in panel layout and user interaction techniques.
- Offers some applications and application development tools written by IBM.

Several combinations of IBM hardware and software have been selected as SAA environments. These are environments in which IBM will manage the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

- MVS
  - TSO/E

  - CICS®
  - IMS™
- VM CMS
- Operating System/400® (OS/400®)
- Operating System/2® (OS/2)

Benefits of Using a Compiler

The IBM Compiler for REXX/370 (Program Number 5695-013) and the IBM Library for REXX/370 (Program Number 5695-014) provide significant benefits for programmers during program development and for users when a program is run. The benefits are:

- Improved performance
- Reduced system load
- Protection for source code and programs
- Improved productivity and quality
- Portability of compiled programs
- Checking for compliance to SAA

Improved Performance

The performance improvements that you can expect when you run compiled REXX programs depend on the type of program. A program that performs large numbers of arithmetic operations of default precision shows the greatest improvement. A program that mainly enters commands to the host shows minimal improvement because REXX cannot decrease the time taken by the host to process the commands.

Reduced System Load

Compiled REXX programs run faster than interpreted programs. Because a program has to be compiled only once, system load is reduced and response time is improved when the program is run frequently.

For example, a REXX program that performs many arithmetic operations might take 12 seconds to run interpreted. If the program is run 60 times, it uses about 12 minutes of processor time. The same program when compiled might run six times faster, using only about 2 minutes of processor time.

Protection for Source Code and Programs

Your REXX programs and algorithms are assets that you want to protect.

The Compiler produces object code, which helps you protect these assets by discouraging people from making unauthorized changes to your programs. You can distribute your REXX programs in object code only.

Load modules can be further protected by using a security server, such as RACF®.

Improved Productivity and Quality

The Compiler can produce source listings, cross-reference listings, and messages, which help you more easily develop and maintain your REXX programs.

The Compiler identifies syntax errors in a program before you start testing it. You can then focus on correcting errors in logic during testing with the REXX interpreter.

Portability of Compiled Programs

A REXX program compiled under MVS/ESA can run under CMS. Similarly, a REXX program compiled under CMS can run under MVS/ESA.

SAA Compliance Checking

The Systems Application Architecture (SAA) definitions of software interfaces, conventions, and protocols provide a framework for designing and developing applications that are consistent within and across several operating systems.

The SAA Procedures Language is a subset of the REXX language supported by the interpreter under TSO/E, and can be used in this operating environment.

To help you write programs for use in all SAA environments, the Compiler can optionally check for SAA compliance. With this option in effect, a warning message is issued for each non-SAA item found in a program.

For more information, see *IBM Compiler and Library for REXX/370; Introducing the Next Step in REXX Programming*.

Chapter 2. Writing and Running a REXX Exec

This chapter introduces execs and their syntax, describes the steps involved in writing and running an exec, and explains concepts you need to understand to avoid common problems.

Before You Begin

Before you can write a REXX program, called an exec, you need to create a data set to contain the exec. The data set can be either sequential or partitioned, but if you plan to create more than one exec, it is easier to create a REXX library as a partitioned data set (PDS) with execs as members.

To create a PDS, allocate a data set with your prefix (usually your user ID) as the first qualifier, any name as the second qualifier, and preferably "exec" as the third qualifier. You can allocate the PDS with the Utilities option in ISPF/PDF or with the TSO/E ALLOCATE command. For specific information about allocating a data set for an exec, see Appendix A, “Allocating Data Sets,” on page 171.

What is a REXX Exec?

A REXX exec consists of REXX language instructions that are interpreted directly by the REXX interpreter or compiled directly by a REXX language compiler and executed by a Compiler Runtime Processor. An exec can also contain commands that are executed by the host environment.

An advantage of the REXX language is its similarity to ordinary English. This similarity makes it easy to read and write a REXX exec. For example, an exec to display a sentence on the screen uses the REXX instruction SAY followed by the sentence to be displayed.

Example of a Simple Exec

```
/***************************** REXX ********************************/
 SAY 'This is a REXX exec.'
```

Note that this simple exec starts with a comment line to identify the program as a REXX exec. A comment begins with /* and ends with */. **To prevent incompatibilities with CLISTs, IBM recommends that all REXX execs start with a comment that includes the characters "REXX" within the first line (line 1) of the exec. Failure to do so can lead to unexpected or unintended results in your REXX exec.** More about comments and why you might need a REXX exec identifier appears later in “Null” on page 13.

When you run the exec, you see on your screen the sentence:

```
    This is a REXX exec.
```

Even in a longer exec, the instructions flow like ordinary English and are easy to understand.

Example of a Longer Exec

```
/***************************** REXX ********************************/
/*  This exec adds two numbers and displays their sum.             */
/*********************************************************************/

 SAY 'Please enter a number.'
 PULL number1
 SAY 'Now enter a number to add to the first number.'
 PULL number2
 sum = number1 + number2
 SAY 'The sum of the two numbers is' sum'.'
```

When you run the example, the exec interacts with you at the terminal. First you see on your screen:

```
   Please enter a number.
```

When you type a number, for example 42, and press the Enter key, the variable `number1` is assigned the value 42. You then see another sentence on the screen.

```
   Now enter a number to add to the first number.
```

When you enter another number, for example 21, the variable `number2` is assigned the value 21. Then the values in `number1` and `number2` are added and the total is assigned to `sum`. You see a final sentence on the screen displaying the sum.

```
  The sum of the two numbers is 63.
```

Before you actually try these examples, please read the next two sections:

- "Syntax of REXX Instructions" on page 8
- "Running an Exec" on page 14

Syntax of REXX Instructions

Some programming languages have rigid rules about how and where characters are entered on each line. For example, CLIST statements must be entered in uppercase, and assembler statements must begin in a particular column. REXX, on the other hand, has simple syntax rules. There is no restriction on how characters are entered and generally one line is an instruction regardless of where it begins or where it ends.

The Character Type of REXX Instructions

You can enter a REXX instruction in lowercase, uppercase, or mixed case. However, alphabetic characters are changed to uppercase, unless you enclose them in single or double quotation marks.

Using Quotation Marks in an Instruction

A series of characters enclosed in matching quotation marks is called a *literal string*. The following examples both contain literal strings.

```
SAY 'This is a REXX literal string.'        /* Using single quotes */
SAY "This is a REXX literal string."        /* Using double quotes */
```

You cannot enclose a literal string with one each of the two types of quotation marks. The following is *not* a correct example of an enclosed literal string.

```
SAY 'This is a REXX literal string."    /* Using mismatched quotes */
```

When you omit the quotation marks from a SAY instruction as follows:

```
SAY This is a REXX string.
```

you see the statement in uppercase on your screen.

```
      THIS IS A REXX STRING.
```

Note: If any word in the statement is the name of a variable that has already been assigned a value, REXX substitutes the value. For information about variables, see "Using Variables" on page 23.

If a string contains an apostrophe, you can enclose the literal string in double quotation marks.

```
SAY "This isn't a CLIST instruction."
```

You can also use two single quotation marks in place of the apostrophe, because a pair of single quotation marks is processed as one.

```
SAY 'This isn''t a CLIST instruction.'
```

Either way, the outcome is the same.

```
        This isn't a CLIST instruction.
```

The Format of REXX Instructions

The REXX language uses a free format. This means you can insert extra spaces between words and blank lines freely throughout the exec without causing an error. A line usually contains one instruction except when it ends with a comma (,) or contains a semicolon (;). A comma is the continuation character and indicates that the instruction continues to the next line. The comma, when used in this manner, also adds a space when the lines are concatenated. A semicolon indicates the end of the instruction and is used to separate multiple instructions on one line.

Beginning an instruction

An instruction can begin in any column on any line. The following are all valid instructions.

```
SAY 'This is a literal string.'
                     SAY 'This is a literal string.'
                                    SAY 'This is a literal string.'
```

This example appears on the screen as follows:

```
  This is a literal string.
  This is a literal string.
  This is a literal string.
```

Continuing an instruction

A comma indicates that the instruction continues to the next line. Note that a space is added between "extended" and "REXX" when it appears on the screen.

```
SAY 'This is an extended',
    'REXX literal string.'
```

This example appears on the screen as one line.

```
   This is an extended REXX literal string.
```

Also note that the following two instructions are identical and yield the same result when displayed on the screen:

```
SAY 'This is',
    'a string.'
```

is functionally identical to:

```
SAY 'This is' 'a string.'
```

These examples appear on the screen as:

```
   This is a string.
```

In the first example, the comma at the end of line 1 adds a space when the two lines are concatenated for display. In the second example, the space between the two separate strings is preserved when the line is displayed.

Continuing a literal string without adding a space

If you need to continue an instruction to a second or more lines but do not want REXX to add spaces when the line appears on the screen, use the concatenation operand (two single OR bars, ||).

```
SAY 'This is an extended literal string that is bro'||,
    'ken in an awkward place.'
```

This example appears on the screen as one line without adding a space within the word "broken".

```
   This is an extended literal string that is broken in an awkward place.
```

Also note that the following two instructions are identical and yield the same result when displayed on the screen:

```
SAY 'This is' ||,
    'a string.'
```

is functionally identical to:

```
SAY 'This is' || 'a string.'
```

These examples appear on the screen as:

```
   This isa string.
```

In the first example, the concatenation operator at the end of line 1 causes the deletion of any spaces when the two lines are concatenated for display. In the second example, the concatenation operator also concatenates the two strings without space when the line is displayed.

Ending an instruction

The end of the line or a semicolon indicates the end of an instruction. If you put more than one instruction on a line, you must separate each instruction with a semicolon. If you put one instruction on a line, it is best to let the end of the line delineate the end of the instruction.

```
SAY 'Hi!'; say 'Hi again!'; say 'Hi for the last time!'
```

This example appears on the screen as three lines.

```
   Hi!
   Hi again!
   Hi for the last time!
```

The following example demonstrates the free format of REXX.

Example of Free Format

```
/************************ REXX ***********************************/
SAY 'This is a REXX literal string.'
SAY              'This is a REXX literal string.'
  SAY 'This is a REXX literal string.'
SAY,
'This',
'is',
'a',
'REXX',
'literal',
'string.'

SAY'This is a REXX literal string.';SAY'This is a REXX literal string.'
SAY '     This is a REXX literal string.'
```

When the example runs, you see six lines of identical output on your screen followed by one indented line.

```
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
This is a REXX literal string.
     This is a REXX literal string.
```

Thus you can begin an instruction anywhere on a line, you can insert blank lines, and you can insert extra spaces between words in an instruction because the language processor ignores blank lines and spaces that are greater than one. This flexibility of format allows you to insert blank lines and spaces to make an exec easier to read.

Only when words are parsed do blanks and spaces take on significance. More about parsing is covered in “Parsing Data” on page 83.

Types of REXX Instructions

There are five types of REXX instructions: keyword, assignment, label, null, and command. The following example is an ISPF/PDF Edit panel that shows an exec with various types of instructions. A description of each type of instruction appears after the example. In most of the descriptions, you will see an edit line number (without the prefixed zeroes) to help you locate the instruction in the example.

```
EDIT ---- USERID.REXX.EXEC(TIMEGAME)------------------ COLUMNS 009 080
COMMAND ===>                                          SCROLL ===> HALF
****** ************************ TOP OF DATA ********************************
000001 /************************* REXX ***************************/
000002 /* This is an interactive REXX exec that asks a user for the*/
000003 /* time and then displays the time from the TIME command.   */
000004 /************************************************************/
000005 Game1:
000006
000007 SAY 'What time is it?'
000008 PULL usertime                 /* Put the user's response
000009                                  into a variable called
000010                                  "usertime"  */
000011 IF usertime = '' THEN           /* User didn't enter a time */
000012    SAY "O.K. Game's over."
000013 ELSE
000014    DO
000015      SAY "The computer says:"
000016      /* TSO system */  TIME     /* command */
000017    END
000018
000019 EXIT
****** ********************** BOTTOM OF DATA ***********************************
```

Keyword

A keyword instruction tells the language processor to do something. It begins with a REXX keyword that identifies what the language processor is to do. For example, SAY (line 7) displays a string on the screen and PULL (line 8) takes one or more words of input and puts them into the variable `usertime`.

IF, THEN (line 11) and ELSE (line 13) are three keywords that work together in one instruction. Each keyword forms a clause, which is a subset of an instruction. If the expression that follows the IF keyword is true, the instruction that follows the THEN keyword is processed. Otherwise, the instruction that follows the ELSE keyword is processed. If more than one instruction follows a THEN or an ELSE, the instructions are preceded by a DO (line 14) and followed by an END (line 17). More information about the IF/THEN/ELSE instruction appears in “Using Conditional Instructions” on page 39.

The EXIT keyword (line 19) tells the language processor to end the exec. Using EXIT in the preceding example is a convention, not a necessity, because processing ends automatically when there are no more instructions in the exec. More about EXIT appears in “EXIT Instruction” on page 53.

Assignment

An assignment gives a value to a variable or changes the current value of a variable. A simple assignment instruction is:

```
number = 4
```

In addition to giving a variable a straightforward value, an assignment instruction can also give a variable the result of an expression. An expression is something that needs to be calculated, such as an arithmetic expression. The expression can contain numbers, variables, or both.

```
number = 4 + 4

number = number + 4
```

In the first of the two examples, the value of `number` is 8. If the second example directly followed the first in an exec, the value of `number` would become 12. More about expressions is covered in “Using Expressions” on page 25.

Label

A label, such as `Game1:` (line 5), is a symbolic name followed by a colon. A label can contain either single- or double-byte characters or a combination of single- and double-byte characters. (Double-byte characters are valid only if you have included OPTIONS ETMODE as the first instruction in your exec.) A

label identifies a portion of the exec and is commonly used in subroutines and functions, and with the SIGNAL instruction. More about the use of labels appears in Chapter 6, “Writing Subroutines and Functions,” on page 65 and “SIGNAL Instruction” on page 55.

Null

A null is a comment or a blank line, which is ignored by the language processor but make an exec easier to read.

- Comments (lines 1 through 4, 8 through 11, 16)

  A comment begins with /* and ends with */. Comments can be on one or more lines or on part of a line. You can put information in a comment that might not be obvious to a person reading the REXX instructions. Comments at the beginning can describe the overall purpose of the exec and perhaps list special considerations. A comment next to an individual instruction can clarify its purpose.

  Note: To prevent incompatibilities with CLISTs, IBM recommends that all REXX execs start with a comment that includes the characters "REXX" within the first line (line 1) of the exec. Failure to do so can lead to unexpected or unintended results in your REXX exec. This type of comment is called the REXX exec identifier and immediately identifies the program to readers as a REXX exec and also distinguishes it from a CLIST. It is necessary to distinguish execs from CLISTs when they are both stored in the system file, SYSPROC. For more information about where and how execs are stored, see “Running an Exec Implicitly” on page 15.

- Blank lines (lines 6, 18)

  Blank lines help separate groups of instructions and aid readability. The more readable an exec, the easier it is to understand and maintain.

Command

An instruction that is not a keyword instruction, assignment, label, or null is processed as a command and is sent to a previously defined environment for processing. For example, the word "TIME" in the previous exec (line 16), even though surrounded by comments, is processed as a TSO/E command.

```
/* TSO system */ TIME   /* command */
```

More information about issuing commands appears in Chapter 8, “Entering Commands from an Exec,” on page 93.

Execs Using Double-Byte Character Set Names

You can use double-byte character set (DBCS) names in your REXX execs for literal strings, labels, variable names, and comments. Such character strings can be single-byte, double-byte, or a combination of both single- and double-byte names. To use DBCS names, you must code OPTIONS ETMODE as the first instruction in the exec. ETMODE specifies that those strings that contain DBCS characters are to be checked as being valid DBCS strings. DBCS characters must be enclosed within shift-out (X'0E') and shift-in (X'0F') delimiters. In the following example, the shift-out (SO) and shift-in (SI) delimiters are represented by the less than symbol (<) and the greater than symbol (>) respectively. [1] For example, <.S.Y.M.D> and <.D.B.C.S.R.T.N> represent DBCS symbols in the following examples.

Example 1:

The following is an example of an exec using a DBCS variable name and a DBCS subroutine label.

```
/*   REXX  */
OPTIONS 'ETMODE'          /* ETMODE to enable DBCS variable names  */
j = 1
<.S.Y.M.D> = 10            /* Variable with DBCS characters between
                              shift-out (<) and shift-in (>)        */
CALL <.D.B.C.S.R.T.N>     /* Invoke subroutine with DBCS name      */
:
```

[1] The SO and SI characters are non-printable.

```
<.D.B.C.S.R.T.N>:         /* Subroutine with DBCS name             */
DO i = 1 TO 10
  IF x.i = <.S.Y.D.M> THEN  /* Does x.i match the DBCS variable's
                               value?                               */
    SAY 'Value of the DBCS variable is : ' <.S.Y.D.M>
END
EXIT 0
```

Example 2:

The following example shows some other uses of DBCS variable names with the EXECIO stem option, as DBCS parameters passed to a program invoked through LINKMVS, and with built-in function, LENGTH.

```
/*   REXX   */
OPTIONS 'ETMODE'           /* ETMODE to enable DBCS variable names */

"ALLOC FI(INDD) DA('DEPTA29.DATA') SHR REU"

/*********************************************************************/
/*   Use EXECIO to read lines into DBCS stem variables               */
/*********************************************************************/

"EXECIO * DISKR indd (FINIS STEM <.d.b.c.s__.s.t.e.m>."

IF rc = 0 THEN          /* if good return code from execio          */

  /*******************************************************************/
  /*   Say each DBCS stem variable set by EXECIO                     */
  /*******************************************************************/

  DO i = 1 TO <.d.b.c.s__.s.t.e.m>.0

    SAY "Line " i "==> " <.d.b.c.s__.s.t.e.m>.i

  END

line1_<.v.a.l.u.e> = <.d.b.c.s__.s.t.e.m>.1  /* line 1 value */

line_len = length(line1_<.v.a.l.u.e>)  /* Length of line */

/*********************************************************************/
/* Invoke LINKMVS command "proca29" to process a line.               */
/* Two variable names are used to pass 2 parameters, one of          */
/* which is a DBCS variable name.  The LINKMVS host command          */
/* environment routine will look up the value of the two             */
/* variables and pass their values to the address LINKMVS            */
/* command, "proca29".                                               */
/*********************************************************************/

ADDRESS LINKMVS "proca29  line_len  line1_<.v.a.l.u.e>"

"FREE FI(INDD)"

EXIT 0
```

Running an Exec

After you have placed REXX instructions in a data set, you can run the exec *explicitly* by using the EXEC command followed by the data set name and the "exec" keyword operand, or *implicitly* by entering the member name. You can run an exec implicitly only if the PDS that contains it was allocated to a system file. More information about system files appears in the “Running an Exec Implicitly” on page 15.

Running an Exec Explicitly

The EXEC command runs non-compiled programs in TSO/E. To run an exec explicitly, enter the EXEC command followed by the data set name that contains the exec and the keyword operand "exec" to distinguish it from a CLIST.

You can specify a data set name according to the TSO/E data set naming conventions in several different ways. For example the data set name USERID.REXX.EXEC(TIMEGAME) can be specified as:

- A **fully-qualified data set**, which appears within quotation marks.

```
EXEC 'userid.rexx.exec(timegame)' exec
```

- A **non fully-qualified data set**, which has no quotation marks can eliminate your profile prefix (usually your user ID) as well as the third qualifier, exec.

```
EXEC rexx.exec(timegame) exec            /* eliminates prefix */
EXEC rexx(timegame) exec        /* eliminates prefix and exec */
```

For information about other ways to specify a data set name, see the EXEC command in *z/OS TSO/E Command Reference*.

You can type the EXEC command in the following places:

- At the READY prompt

```
READY
EXEC rexx.exec(timegame) exec
```

- From the COMMAND option of ISPF/PDF

```
------------------------------  TSO COMMAND PROCESSOR  -------------------------
ENTER TSO COMMAND OR CLIST BELOW:

===> exec rexx.exec(timegame) exec




ENTER SESSION MANAGER MODE ===> NO    (YES or NO)
```

- On the COMMAND line of any ISPF/PDF panel as long as the EXEC command is preceded by the word "tso".

```
------------------------------  EDIT - ENTRY PANEL  --------------------------
COMMAND ===> tso exec rexx.exec(timegame) exec

ISPF LIBRARY:
   PROJECT ===> PREFIX
   GROUP   ===> REXX     ===>            ===>            ===>
   TYPE    ===> EXEC
   MEMBER  ===> TIMEGAME          (Blank for member selection list)

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
   DATA SET NAME     ===>
   VOLUME SERIAL     ===>         (If not cataloged)

   DATA SET PASSWORD ===>         (If password protected)

   PROFILE NAME      ===>         (Blank defaults to data set type)

   INITIAL MACRO     ===>         LOCK       ===> YES  (YES, NO or NEVER)

   FORMAT NAME       ===>         MIXED MODE ===> NO   (YES or NO)
```

Running an Exec Implicitly

Running an exec implicitly means running an exec by simply entering the member name of the data set that contains the exec. Before you can run an exec implicitly, you must allocate the PDS that contains it to a system file (SYSPROC or SYSEXEC).

SYSPROC is a system file whose data sets can contain both CLISTs and execs. (Execs are distinguished from CLISTs by the REXX exec identifier, a comment at the beginning of the exec the first line of which includes the word "REXX".) SYSEXEC is a system file whose data sets can contain only execs. (Your installation might have changed the name to something other than SYSEXEC, but for the purposes of this book, we will call it SYSEXEC.) When both system files are available, SYSEXEC is searched before SYSPROC.

Allocating a PDS to a System File

To allocate the PDS that contains your execs to a system file, you need to do the following:

- Decide if you want to use the separate file for execs (SYSEXEC) or combine CLISTs and execs in the same file (SYSPROC). For information that will help you decide, see "Things to Consider When Allocating to a System File (SYSPROC or SYSEXEC)" on page 161.
- Use one of the following two checklists for a step-by-step guide to writing an exec that allocates a PDS to a system file.
  - "Checklist #3: Writing an Exec that Sets up Allocation to SYSEXEC" on page 176
  - "Checklist #4: Writing an Exec that Sets up Allocation to SYSPROC" on page 178

  After your PDS is allocated to the system file, you can then run an exec by simply typing the name of the data set member that contains the exec. You can type the member name in any of the following locations:

  - At the READY prompt

```
READY
 timegame
```

  - From the COMMAND option of ISPF/PDF

```
----------------------------  TSO COMMAND PROCESSOR  ------------------------
ENTER TSO COMMAND OR CLIST BELOW:

===> timegame




ENTER SESSION MANAGER MODE ===> NO    (YES or NO)
```

  - On the COMMAND line of any ISPF/PDF panel as long as the member name is preceded by "tso".

```
------------------------------  EDIT - ENTRY PANEL  --------------------------
COMMAND ===> tso timegame

ISPF LIBRARY:
   PROJECT ===> PREFIX
   GROUP   ===> REXX     ===>            ===>            ===>
   TYPE    ===> EXEC
   MEMBER  ===> TIMEGAME     (Blank for member selection list)

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
   DATA SET NAME     ===>
   VOLUME SERIAL     ===>    (If not cataloged)

   DATA SET PASSWORD ===>    (If password protected)

   PROFILE NAME      ===>    (Blank defaults to data set type)

   INITIAL MACRO     ===>    LOCK       ===> YES    (YES, NO or NEVER)

   FORMAT NAME       ===>    MIXED MODE ===> NO     (YES or NO)
```

To reduce the search time for an exec that is executed implicitly and to differentiate it from a TSO/E command, precede the member name with a %:

```
READY
 %timegame
```

When a member name is preceded by %, TSO/E searches a limited number of system files for the name, thus reducing the search time. Without the %, TSO/E searches several files before it searches SYSEXEC and SYSPROC to ensure that the name you entered is not a TSO/E command.

Exercises - Running the Example Execs

Create a PDS exec library using Checklist #1 or Checklist #2 in Appendix A, “Allocating Data Sets,” on page 171. Then try the example execs from the beginning of this chapter. Run them explicitly with the EXEC command and see if the results you get are the same as the ones in this book. If they are not, why aren't they the same?

Now write an exec to allocate your PDS to SYSPROC or SYSEXEC using “Checklist #3: Writing an Exec that Sets up Allocation to SYSEXEC” on page 176 or “Checklist #4: Writing an Exec that Sets up Allocation to SYSPROC” on page 178. Then run the example execs implicitly. Which way is easier?

Interpreting Error Messages

When you run an exec that contains an error, an error message often displays the line on which the error occurred and gives an explanation of the error. Error messages can result from syntax errors and from computational errors. For example, the following exec has a syntax error.

Example of an Exec with a Syntax Error

```
/************************* REXX ***********************************/
/* This is an interactive REXX exec that asks the user for a       */
/* name and then greets the user with the name supplied.  It       */
/* contains a deliberate error.                                    */
/*******************************************************************/

SAY "Hello! What's your name?"
PULL who                       /* Get the person's name.
IF who = '' THEN
   SAY 'Hello stranger'
ELSE
   SAY 'Hello' who
```

When the exec runs, you see the following on your screen:

```
  Hello! What's your name?
     7 +++ PULL who                    /* Get the person's name.IF who =
'' THEN SAY 'Hello stranger'ELSE   SAY 'Hello' who
IRX0006I Error running REXX.EXEC(HELLO), line 7: Unmatched "/*" or quote
***
```

The exec runs until it detects the error, a missing */ at the end of the comment. As a result, the SAY instruction displays the question, but doesn't wait for your response because the next line of the exec contains the syntax error. The exec ends and the language processor displays error messages.

The first error message begins with the line number of the statement where the error was detected, followed by three pluses (+++) and the contents of the statement.

```
     7 +++ PULL who                    /* Get the person's name.IF who =
'' THEN SAY 'Hello stranger'ELSE   SAY 'Hello' who
```

The second error message begins with the message number followed by a message containing the exec name, line where the error was found, and an explanation of the error.

```
IRX0006I Error running REXX.EXEC(HELLO), line 7: Unmatched "/*" or quote
```

For more information about the error, you can go to the message explanations in *z/OS TSO/E Messages*, where information is arranged by message number.

To fix the syntax error in this exec, add ***/** to the end of the comment on line 7.

```
PULL who                       /* Get the person's name.*/
```

Preventing Translation to Uppercase

As a rule, all alphabetic characters processed by the language processor are translated to uppercase before they are processed. These alphabetic characters can be from within an exec, such as words in a REXX instruction, or they can be external to an exec and processed as input. You can prevent this translation to uppercase in two ways depending on whether the characters are read as parts of instructions from within an exec or are read as input to an exec.

From Within an Exec

To prevent translation of alphabetic characters to uppercase from within an exec, simply enclose the characters in single or double quotation marks. Numbers and special characters, whether or not in quotation marks, are not changed by the language processor. For example, when you follow a SAY instruction with a phrase containing a mixture of alphabetic characters, numbers, and special characters, only the alphabetic characters are changed.

```
SAY The bill for lunch comes to $123.51!
```

results in:

```
THE BILL FOR LUNCH COMES TO $123.51!
```

Quotation marks ensure that information from within an exec is processed exactly as typed. This is important in the following situations:

- For output when it must be lowercase or a mixture of uppercase and lowercase.
- To ensure that commands are processed correctly. For example, if a variable name in an exec is the same as a command name, the exec ends in error when the command is issued. It is good programming practice to avoid using variable names that are the same as commands, but just to be safe, enclose all commands in quotation marks.

As Input to an Exec

When reading input from a terminal or when passing input from another exec, the language processor also changes alphabetic characters to uppercase before they are processed. To prevent translation to uppercase, use the PARSE instruction.

For example, the following exec reads input from the terminal screen and re-displays the input as output.

Example of Reading and Re-displaying Input:

```
/************************* REXX ***********************************/
/* This is an interactive REXX exec that asks a user for the name  */
/* of an animal and then re-displays the name.                     */
/******************************************************************/

SAY "Please type in the name of an animal."
PULL animal                              /* Get the animal name.*/
SAY animal
```

If you responded to the example with the word **tyrannosaurus**, you would see on your screen:

```
TYRANNOSAURUS
```

To cause the language processor to read input exactly as it is presented, use the PARSE PULL instruction.

```
PARSE PULL animal
```

Then if you responded to the example with **TyRannOsauRus**, you would see on the screen:

```
  TyRannOsauRus
```

Exercises - Running and Modifying the Example Execs

Write and run the preceding Example of Reading and Re-displaying Input. Try various input and observe the output. Now change the PULL instruction to a PARSE PULL instruction and observe the difference.

Passing Information to an Exec

When an exec runs, you can pass information to it in several ways, two of which are:

- Through terminal interaction
- By specifying input when invoking the exec.

Using Terminal Interaction

The PULL instruction is one way for an exec to receive input as shown by a previous example repeated here.

Example of an Exec that Uses PULL

```
/*************************** REXX ********************************/
/*  This exec adds two numbers and displays their sum.           */
/*****************************************************************/
 SAY 'Please enter a number.'
 PULL number1
 SAY 'Now enter a number to add to the first number.'
 PULL number2
 sum = number1 + number2
 SAY 'The sum of the two numbers is' sum'.'
```

The PULL instruction can extract more than one value at a time from the terminal by separating a line of input, as shown in the following variation of the previous example.

Variation of an Example that Uses PULL

```
/*************************** REXX ********************************/
/*  This exec adds two numbers and displays their sum.           */
/*****************************************************************/
 SAY 'Please enter two numbers.'
 PULL number1 number2
 sum = number1 + number2
 SAY 'The sum of the two numbers is' sum'.'
```

Note: For the PULL instruction to extract information from the terminal, the data stack must be empty. More information about the data stack appears in Chapter 11, “Storing Information in the Data Stack,” on page 125.

Specifying Values when Invoking an Exec

Another way for an exec to receive input is through values specified when you invoke the exec. For example to pass two numbers to an exec named "add", using the EXEC command, type:

```
EXEC rexx.exec(add) '42 21' exec
```

To pass input when running an exec implicitly, simply type values (words or numbers) after the member name.

```
add 42 21
```

These values are called an *argument.* For information about arguments, see “Passing Arguments” on page 21.

The exec "add" uses the ARG instruction to assign the input to variables as shown in the following example.

Example of an Exec that Uses the ARG Instruction

```
/***************************** REXX *********************************/
/*  This exec receives two numbers as input, adds them, and        */
/*  displays their sum.                                            */
/*****************************************************************/
 ARG number1 number2
 sum = number1 + number2
 SAY 'The sum of the two numbers is' sum'.'
```

ARG assigns the first number, 42, to `number1` and the second number, 21, to `number2`.

If the number of values is fewer or more than the number of variable names after the PULL or the ARG instruction, errors can occur as described in the following sections.

Specifying Too Few Values

When you specify fewer values than the number of variables following the PULL or ARG instruction, the extra variables are set to null. For example, you pass only one number to "add".

```
EXEC rexx.exec(add) '42' exec
```

The first variable following the ARG instruction, `number1`, is assigned the value 42. The second variable, `number2`, is set to null. In this situation, the exec ends with an error when it tries to add the two variables. In other situations, the exec might not end in error.

Specifying Too Many Values

When you specify more values than the number of variables following the PULL or ARG instruction, the last variable gets the remaining values. For example, you pass three numbers to "add".

```
EXEC rexx.exec(add) '42 21 10' exec
```

The first variable following the ARG instruction, `number1`, is assigned the value 42. The second variable gets both '21 10'. In this situation, the exec ends with an error when it tries to add the two variables. In other situations, the exec might not end in error.

To prevent the last variable from getting the remaining values, use a period (.) at the end of the PULL or ARG instruction.

```
ARG number1 number2 .
```

The period acts as a "dummy variable" to collect unwanted extra information. If there is no extra information, the period is ignored. You can also use a period as a place holder within the PULL or ARG instruction as follows:

```
ARG . number1 number2
```

In this case, the first value, 42, is discarded and `number1` and `number2` get the next two values, 21 and 10.

Preventing Translation of Input to Uppercase

Like the PULL instruction, the ARG instruction changes alphabetic characters to uppercase. To prevent translation to uppercase, precede ARG with PARSE as demonstrated in the following example.

Example of an Exec that Uses PARSE ARG

```
/*************************** REXX ********************************/
/*  This exec receives the last name, first name, and score of     */
/*  a student and displays a sentence reporting the name and       */
/*  score.                                                         */
/*****************************************************************/
 PARSE ARG lastname firstname score
 SAY firstname lastname 'received a score of' score'.'
```

Exercises - Using the ARG Instruction

The left column shows the input values sent to an exec. The right column is the ARG statement within the exec that receives the input. What value does each variable assume?

Input
Variables Receiving Input

1. 115 -23 66 5.8
ARG first second third

2. .2 0 569 2E6
ARG first second third fourth

3. 13 13 13 13
ARG first second third fourth fifth

4. Weber Joe 91
ARG lastname firstname score

5. Baker Amanda Marie 95
PARSE ARG lastname firstname score

6. Callahan Eunice 88 62
PARSE ARG lastname firstname score

ANSWERS

1. first = 115, second = -23, third = 66 5.8
2. first = .2, second = 0, third = 569, fourth = 2E6
3. first = 13, second = 13, third = 13, fourth = 13, fifth = *null*
4. lastname = WEBER, firstname = JOE, score = 91
5. lastname = Baker, firstname = Amanda, score = Marie 95
6. lastname = Callahan, firstname = Eunice, score = 88

Passing Arguments

Values passed to an exec are usually called arguments. Arguments can consist of one word or a string of words. Words within an argument are separated by blanks. The number of arguments passed depends on how the exec is invoked.

Passing Arguments Using the CALL Instruction or REXX Function Call

When you invoke a REXX exec using either the CALL instruction or a REXX function call, you can pass up to 20 arguments to an exec. Each argument must be separated by a comma.

Passing Arguments Using the EXEC Command

When you invoke a REXX exec either implicitly or explicitly using the EXEC command, you can pass either one or no arguments to the exec. Thus the ARG instruction in the preceding examples received only one argument. One argument can consist of many words. The argument, if present, will appear as a single string.

If you plan to use commas within the argument string when invoking a REXX exec using the EXEC command, special consideration must be given. For example, if you specify:

```
GETARG 1,2
```

or

```
ex 'sam.rexx.exec(getarg)' '1,2'
```

the exec receives a single argument string consisting of "1,2". The exec could then use a PARSE ARG instruction to break the argument string into the comma-separated values like the following:

```
PARSE ARG A ',' B
SAY 'A is ' A    /* Will say 'A is 1' */
SAY 'B is ' B    /* Will say 'B is 2' */
```

However, because commas are treated as separator characters in TSO/E, you cannot pass an argument string that contains a leading comma using the implicit form of the EXEC command. That is, if you invoke the exec using:

```
GETARG ,2
```

the exec is invoked with an argument string consisting of "2". The leading comma separator is removed before the exec receives control. If you need to pass an argument string separated by commas and the leading argument is null (that is, contains a leading comma), you must use the explicit form of the EXEC command. For example:

```
ex 'sam.rexx.exec(getarg)' ',2'
```

In this case, the exec is invoked with an argument string consisting of ",2".

For more information about functions and subroutines, see Chapter 6, “Writing Subroutines and Functions,” on page 65. For more information about arguments, see “Parsing Multiple Strings as Arguments” on page 87.

Chapter 3. Using Variables and Expressions

This chapter describes variables, expressions, and operators, and explains how to use them in REXX execs.

One of the most powerful aspects of computer programming is the ability to process variable data to achieve a result. The variable data could be as simple as two numbers, the process could be subtraction, and the result could be the answer.

```
answer = number1 - number2
```

Or the variable data could be input to a series of complex mathematical computations that result in a 3-dimensional animated figure.

Regardless of the complexity of a process, the premise is the same. When data is unknown or if it varies, you substitute a symbol for the data, much like the "x" and "y" in an algebraic equation.

```
x = y + 29
```

The symbol, when its value can vary, is called a *variable*. A group of symbols or numbers that must be calculated to be resolved is called an *expression*.

Using Variables

A variable is a character or group of characters that represents a value. A variable can contain either single- or double-byte characters, or a combination of single- and double-byte characters. (Double-byte characters are valid only if you include OPTIONS ETMODE as the first instruction of your exec.) The following variable `big` represents the value one million or 1,000,000.

```
big = 1000000
```

Variables can refer to different values at different times. If you assign a different value to `big`, it gets the value of the new assignment, until it is changed again.

```
big = 999999999
```

Variables can also represent a value that is unknown when the exec is written. In the following example, the user's name is unknown, so it is represented by the variable `who`.

```
SAY "Hello! What's your name?"

PARSE PULL who      /* Put the person's name in the variable "who" */
```

Variable Names

A variable name, the part that represents the value, is always on the left of the assignment statement and the value itself is on the right. In the following example, the word "variable1" is the variable name:

```
variable1 = 5
SAY variable1
```

As a result of the above assignment statement, `variable1` is assigned the value "5", and you see on the terminal screen:

```
   5
```

Variable names can consist of:

A...Z
uppercase alphabetic

a...z
lowercase alphabetic

0...9
numbers

@ # $ ¢ ? ! . _
special characters

X'41' ... X'FE'
double-byte character set (DBCS) characters. (ETMODE must be on for these characters to be valid in a variable name.)

Restrictions on the variable name are:

- The first character cannot be 0 through 9 or a period (.)
- The variable name cannot exceed 250 bytes. For names containing DBCS characters, count each DBCS character as two bytes, and count the shift-out (SO) and shift-in (SI) as one byte each.
- DBCS characters within a DBCS name must be delimited by SO (X'0E') and SI (X'0F'). Also note that:
  - SO and SI cannot be contiguous.
  - Nesting of SO / SI is not permitted.
  - A DBCS name cannot contain a DBCS blank (X'4040').
- The variable name should not be RC, SIGL, or RESULT, which are REXX special variables. More about special variables appears later in this book.

Examples of acceptable variable names are:

```
ANSWER    ?98B    X   Word3   number  the_ultimate_value
```

Also, if ETMODE is set on, the following are valid DBCS variable names, where < represents shift-out, and > represents shift-in, ‘.X’, ‘.Y’, and ‘.Z’ represent DBCS characters, and lowercase letters and numbers represent themselves.

```
<.X.Y.Z>   number_<.X.Y.Z>   <.X.Y>1234<.Z>
```

Variable Values

The value of the variable, which is the value the variable name represents, might be categorized as follows:

- A **constant**, which is a number that is expressed as:
  - An integer (12)
  - A decimal (12.5)
  - A floating point number (1.25E2)
  - A signed number (-12)
  - A string constant (' 12')
- A **string**, which is one or more words that may or may not be enclosed in quotation marks, such as:

```
This value is a string.
'This value is a literal string.'
```

- The **value from another variable**, such as:

```
variable1 = variable2
```

In the above example, `variable1` changes to the value of `variable2`, but `variable2` remains the same.

- An **expression**, which is something that needs to be calculated, such as:

```
variable2 = 12 + 12 - .6            /* variable2 becomes 23.4 */
```

Before a variable is assigned a value, the variable displays the value of its own name translated to uppercase. In the following example, if the variable new was not assigned a previous value, the word "NEW" is displayed.

```
SAY new                                          /* displays NEW */
```

Exercises - Identifying Valid Variable Names

Which of the following are valid REXX variable names?

1. 8eight
2. $25.00
3. MixedCase
4. nine_to_five
5. result

ANSWERS

1. Invalid, because the first character is a number
2. Valid
3. Valid
4. Valid
5. Valid, but it is a reserved variable name and we recommend that you use it only to receive results from a subroutine

Using Expressions

An expression is something that needs to be calculated and consists of numbers, variables, or strings, and one or more operators. The operators determine the kind of calculation to be done on the numbers, variables, and strings. There are four types of operators: arithmetic, comparison, logical, and concatenation.

Arithmetic Operators

Arithmetic operators work on valid numeric constants or on variables that represent valid numeric constants.

Types of Numeric Constants

12
A **whole number** has no decimal point or commas. Results of arithmetic operations with whole numbers can contain a maximum of nine digits unless you override the default with the NUMERIC DIGITS instruction. For information about the NUMERIC DIGITS instruction, see *z/OS TSO/E REXX Reference*. Examples of whole numbers are: 123456789 0 91221 999

12.5
A **decimal number** includes a decimal point. Results of arithmetic operations with decimal numbers are limited to a total maximum of nine digits (NUMERIC DIGITS default) before **and** after the decimal. Examples of decimal numbers are: 123456.789 0.888888888

1.25E2
A **floating point number** in exponential notation, is sometimes called scientific notation. The number after the "E" represents the number of places the decimal point moves. Thus 1.25E2 (also written as 1.25E+2) moves the decimal point to the right two places and results in 125. When an "E" is followed by a minus (-), the decimal point moves to the left. For example, 1.25E-2 is .0125.

Floating point numbers are used to represent very large or very small numbers. For more information about floating point numbers, see *z/OS TSO/E REXX Reference*.

-12
A **signed number** with a minus (-) next to the number represents a negative value. A plus next to a number indicates that the number should be processed as it is written. When a number has no sign, it is processed as a positive value.

The arithmetic operators you can use are as follows:

Operator
Meaning

+
Add

-
Subtract

*****
Multiply

/
Divide

%
Divide and return a whole number without a remainder

//
Divide and return the remainder only

******
Raise a number to a whole number power

-number
Negate the number

+number
Add the number to 0

Using numeric constants and arithmetic operators, you can write arithmetic expressions as follows:

```
7 + 2                       /*  result is 9    */
7 - 2                       /*  result is 5    */
7 * 2                       /*  result is 14   */
7 ** 2                      /*  result is 49   */
7 ** 2.5                    /*  result is an error  */
```

Division

Notice that three operators represent division. Each operator displays the result of a division expression in a different way.

/
Divide and express the answer possibly as a decimal number. For example:

```
7 / 2                       /* result is 3.5   */
6 / 2                       /* result is 3     */
```

%
Divide and express the answer as a whole number. The remainder is ignored. For example:

```
7 % 2                       /* result is 3     */
```

//
Divide and express the answer as the remainder only. For example:

```
7 // 2                       /* result is 1     */
```

Order of Evaluation

When you have more than one operator in an arithmetic expression, the order of numbers and operators can be critical. For example, in the following expression, which operation does the language processor perform first?

```
7 + 2 * (9 / 3) - 1
```

Proceeding from left to right, it is evaluated as follows:

- Expressions within parentheses are evaluated first.
- Expressions with operators of higher priority are evaluated before expressions with operators of lower priority.

Arithmetic operator priority is as follows, with the highest first:

Arithmetic Operator Priority

- +
Prefix operators

******
Power® (exponential)

*** / % //**
Multiplication and division

+ -
Addition and subtraction

Thus the preceding example would be evaluated in the following order:

1. Expression in parentheses

```
7 + 2 * (9 / 3) - 1
         \___/
           3
```

2. Multiplication

```
7 + 2 * 3 - 1
    \___/
      6
```

3. Addition and subtraction from left to right

```
7 + 6 - 1 = 12
```

Using Arithmetic Expressions

You can use arithmetic expressions in an exec many different ways. The following example uses several arithmetic operators to round and remove extra decimal places from a dollar and cents value.

Example Using Arithmetic Expressions

```
/****************************** REXX ******************************/
/* This exec computes the total price of an item including sales   */
/* tax rounded to two decimal places.  The cost and percent of the */
/* tax (expressed as a decimal number) are passed to the exec when */
/* it is run.                                                      */
/*******************************************************************/

 PARSE ARG cost percent_tax

 total = cost + (cost * percent_tax)       /* Add tax to cost.     */
 price = ((total * 100 + .5) % 1) / 100    /* Round and remove     */
                                           /* extra decimal places.*/
 SAY 'Your total cost is $'price'.'
```

Exercises - Calculating Arithmetic Expressions

1. What will the following program display on the screen?

   Exercise

   ```
   /***************************** REXX ****************************/
     pa = 1
     ma = 1
     kids = 3
     SAY "There are" pa + ma + kids "people in this family."
   ```

2. What is the value of:

   a. 6 - 4 + 1

   b. 6 - (4 + 1)

   c. 6 * 4 + 2

   d. 6 * (4 + 2)

   e. 24 % 5 / 2

ANSWERS

1. There are 5 people in this family.
2. The values are as follows:

   a. 3

   b. 1

   c. 26

   d. 36

   e. 2

Comparison Operators

Expressions that use comparison operators do not return a number value as do arithmetic expressions. Comparison expressions return either a true or false response in terms of 1 or 0 as follows:

1
True

0
False

Comparison operators can compare numbers or strings and ask questions, such as:

- Are the terms equal? (A = B)
- Is the first term greater than the second? (A > B)
- Is the first term less than the second? (A < B)

For example, if A = 4 and B = 3, then the results of the previous comparison questions are:

- (A = B) Does 4 = 3? **0** (False)
- (A > B) Is 4 > 3? **1** (True)
- (A < B) Is 4 < 3? **0** (False)

The more commonly used comparison operators are as follows:

Operator
Meaning

==
Strictly Equal

=
Equal

\ ==
Not strictly equal

\ =
Not equal

>
Greater than

<
Less than

> <
Greater than or less than (same as not equal)

> =
Greater than or equal to

\ <
Not less than

< =
Less than or equal to

\ >
Not greater than

Note: The not character, "¬", is synonymous with the backslash ("\"). The two characters may be used interchangeably according to availability and personal preference. This book uses the backslash ("\") character.

The Strictly Equal and Equal Operators

When two expressions are **strictly equal**, everything including the blanks and case (when the expressions are characters) is exactly the same.

When two expressions are **equal**, they are resolved to be the same. The following expressions are all true.

```
'WORD' = word              /* returns 1 */
'word ' \== word           /* returns 1 */
'word' == 'word'           /* returns 1 */
 4e2 \== 400               /* returns 1 */
 4e2 \= 100                /* returns 1 */
```

Using Comparison Expressions

Often a comparison expression is used in IF/THEN/ELSE instructions. The following example uses an IF/THEN/ELSE instruction to compare two values. For more information about this instruction, see "IF/THEN/ELSE Instructions" on page 39.

Example Using A Comparison Expression

```
/****************************** REXX ******************************/
/* This exec compares what you paid for lunch for two             */
/* days in a row and then comments on the comparison.             */
/******************************************************************/
SAY 'What did you spend for lunch yesterday?'
SAY 'Please do not include the dollar sign.'

PARSE PULL last

SAY 'What did you spend for lunch today?'
SAY 'Please do not include the dollar sign.'

PARSE PULL lunch

IF lunch > last THEN                      /* lunch cost increased */
   SAY "Today's lunch cost more than yesterday's."

ELSE                  /* lunch cost remained the same or decreased */
   SAY "Today's lunch cost the same or less than yesterday's."
```

Exercises - Using Comparison Expressions

1. In the preceding example of using a comparison expression, what appears on the screen when you respond to the prompts with the following lunch costs?

   Yesterday's Lunch
   Today's Lunch

   4.42
   3.75

   3.50
   3.50

   3.75
   4.42

2. What is the result (0 or 1) of the following expressions?

   a. "Apples" = "Oranges"
   b. " Apples" = "Apples"
   c. " Apples" == "Apples"
   d. 100 = 1E2
   e. 100 \= 1E2
   f. 100 \== 1E2

ANSWERS

1. The following sentences appear.

   a. Today's lunch cost the same or less than yesterday's.
   b. Today's lunch cost the same or less than yesterday's.
   c. Today's lunch cost more than yesterday's.

2. The expressions result in the following. Remember 0 is false and 1 is true.

   a. 0
   b. 1

c. 0 (The first " Apples" has a space.)

d. 1

e. 0

f. 1

Logical (Boolean) Operators

Logical expressions, like comparison expressions, return a true (1) or false (0) value when processed. Logical operators combine two comparisons and return the true (1) or false (0) value depending on the results of the comparisons.

The logical operators are:

Operator
Meaning

&

AND

Returns 1 if both comparisons are true. For example:

```
(4 > 2) & (a = a)   /* true, so result is 1  */
(2 > 4) & (a = a)   /* false, so result is 0 */
```

|

Inclusive OR

Returns 1 if at least one comparison is true. For example:

```
(4 > 2) | (5 = 3)   /* at least one is true, so result is 1 */
(2 > 4) | (5 = 3)   /* neither one is true, so result is 0 */
```

&&

Exclusive OR

Returns 1 if only one comparison (but not both) is true. For example:

```
(4 > 2) && (5 = 3)  /* only one is true, so result is 1 */
(4 > 2) && (5 = 5)  /* both are true, so result is 0 */
(2 > 4) && (5 = 3)  /* neither one is true, so result is 0 */
```

**Prefix **

Logical NOT

Returns the opposite response. For example:

```
\ 0                 /* opposite of 0, so result is 1 */
\ (4 > 2)           /* opposite of true, so result is 0 */
```

Using Logical Expressions

Logical expressions are used in complex conditional instructions and can act as checkpoints to screen unwanted conditions. When you have a series of logical expressions, for clarification, use one or more sets of parentheses to enclose each expression.

```
IF ((A < B) | (J < D)) & ((M = Q) | (M = D)) THEN …
```

The following example uses logical operators to make a decision.

Example Using Logical Expressions

```
/***************************** REXX ********************************/
/* This exec receives arguments for a complex logical expression   */
/* that determines whether a person should go skiing.  The first   */
/* argument is a season and the other two can be 'yes' or 'no'.    */
/***********************************************************************/

 PARSE ARG season snowing broken_leg

 IF ((season = 'winter') | (snowing ='yes')) & (broken_leg ='no')
    THEN SAY 'Go skiing.'
 ELSE
    SAY 'Stay home.'
```

When arguments passed to this example are "spring yes no", the IF clause translates as follows:

```
 IF ((season = 'winter') | (snowing ='yes')) & (broken_leg ='no') THEN
      \_____________/      \____________/       \_____________/
           false               true                true
              \_________________/                   /
                     true                          /
                       \_____________________________/
                                  true
```

As a result, when you run the exec, you see the message:

```
   Go skiing.
```

Exercises - Using Logical Expressions

A student applying to colleges has decided to pick ones according to the following specifications:

```
IF  (inexpensive | scholarship) & (reputable | nearby)  THEN
   SAY  "I'll consider it."
ELSE
   SAY  "Forget it!"
```

A college is inexpensive, did not offer a scholarship, is reputable, but is over 1000 miles away. Should the student apply?

ANSWER

Yes. The conditional instruction works out as follows:

```
IF  (inexpensive | scholarship) & (reputable | nearby)  THEN …
    \___________/ \___________/   \_________/ \______/
        true          false          true      false
           \____________/              \_________/
              true                        true
                \___________________________/
                           true
```

Concatenation Operators

Concatenation operators combine two terms into one. The terms can be strings, variables, expressions, or constants. Concatenation can be significant in formatting output.

The operators that indicate how to join two terms are as follows:

Operator
Meaning

blank
Concatenate terms and place one blank in between. Terms that are separated by more than one blank default to one blank when read. For example:

```
SAY true      blue  /* result is TRUE BLUE */
```

||
Concatenate terms and place no blanks in between. For example:

```
(8 / 2)||(3 * 3)    /* result is 49        */
```

abuttal
Concatenate terms and place no blanks in between. For example:

```
per_cent'%'         /* if per_cent = 50, result is 50%  */
```

Using Concatenation Operators

One way to format output is to use variables and concatenation operators as in the following example. A more sophisticated way to format information is with parsing and templates. Information about parsing appears in “Parsing Data” on page 83.

Example using Concatenation Operators

```
/****************************** REXX ******************************/
/* This exec formats data into columns for output.                */
/******************************************************************/
  sport = 'base'
  equipment = 'ball'
  column = '         '
  cost = 5

  SAY sport||equipment column '$' cost
```

The result of this example is:

```
        baseball         $ 5
```

Priority of Operators

When more than one type of operator appears in an expression, what operation does the language processor do first?

```
IF (A > 7**B) & (B < 3) | (A||B = C) THEN …
```

Like the priority of operators within the arithmetic operators, there is an overall priority that includes all operators. The priority of operators is as follows with the highest first.

Overall Operator Priority

\ or ¬ - +
Prefix operators

******
Power (exponential)

*** / % //**
Multiply and divide

+ -
Add and subtract

blank* || *abuttal
Concatenation operators

== = >< etc.
Comparison operators

&
Logical AND

| &&
Inclusive OR and exclusive OR

Thus the previous example presented again below:

```
IF (A > 7**B) & (B < 3) | (A||B = C) THEN …
```

given the following values:

- A = 8
- B = 2
- C = 10

would be evaluated as follows:

1. Convert variables to values

   ```
   IF (8 > 7**2) & (2 < 3) | (8||2 = 10) THEN …
   ```

2. Compute operations of higher priority within parentheses
   - Exponential operation

     ```
     IF (8 > 7**2) & (2 < 3) | (8||2 = 10)  THEN …
            \____/
              49
     ```

   - Concatenation operation

     ```
     IF (8 > 49) & (2 < 3) | (8||2 = 10)  THEN …
                            \____/
                              82
     ```

3. Compute all operations within parentheses from left to right

   ```
   IF (8 > 49) & (2 < 3) | (82 = 10)  THEN …
       \____/     \___/     \_____/
          0         1         0
   ```

4. Logical AND

   ```
   0    &    1    |     0
    \_______/
        0
   ```

5. Inclusive OR

```
0         |     0
 \_____________/
        0
```

Exercises - Priority of Operators

1. What are the answers to the following examples?

   a. 22 + (12 * 1)

   b. -6 / -2 > (45 % 7 / 2) - 1

   c. 10 * 2 - (5 + 1) // 5 * 2 + 15 - 1

2. In the example of the student and the college from “Exercises - Using Logical Expressions” on page 32, if the parentheses were removed from the student's formula, what would be the outcome for the college?

```
IF  inexpensive | scholarship & reputable | nearby  THEN
   SAY  "I'll consider it."
ELSE
   SAY  "Forget it!"
```

   Remember the college is inexpensive, did not offer a scholarship, is reputable, but is 1000 miles away.

ANSWERS

1. The results are as follows:

   a. 34 (22 + 12 = 34)

   b. 1 (true) (3 > 3 - 1)

   c. 32 (20 - 2 + 15 - 1)

2. I'll consider it.

   The & operator has priority, as follows, but the outcome is the same as the previous version with the parentheses.

```
IF  inexpensive | scholarship & reputable | nearby  THEN
    \_________/   \_________/   \_______/   \____/
       true          false        true      false
         \             \__________/          /
          \               false             /
           \_________________/             /
                  true                    /
                    \____________________/
                            true
```

Tracing Expressions with the TRACE Instruction

You can use the TRACE instruction to display how the language processor evaluates each operation of an expression as it reads it, or to display the final result of an expression. These two types of tracing are useful for debugging execs.

Tracing Operations

To trace operations within an expression, use the TRACE I (TRACE Intermediates) form of the TRACE instruction. All expressions that follow the instruction are then broken down by operation and analyzed as:

```
>V>   - Variable value - The data traced is the contents
        of a variable.
```

```
>L>   - Literal value - The data traced is a literal
        (string, uninitialized variable, or constant).
>O>   - Operation result - The data traced is the result
        of an operation on two terms.
```

The following example uses the TRACE I instruction.

```
EDIT ---- USERID.REXX.EXEC(SAMPLE) ---------------------- COLUMNS 009 080
COMMAND ===>                                              SCROLL ===> HALF
******* ************************** TOP OF DATA ****************************
000001 /************************* REXX ****************************/
000002 /*  This exec uses the TRACE instruction to show how an     */
000003 /*  expression is evaluated, operation by operation.        */
000004 /*********************************************************  */
000005  x = 9
000006  y = 2
000007  TRACE I
000008
000009  IF x + 1 > 5 * y THEN
000010     SAY 'x is big enough.'
000011  ELSE NOP                  /* No operation on the ELSE path */
******* ********************** BOTTOM OF DATA ****************************
```

When you run the example, you see on your screen:

```
 9 *-* IF x + 1 > 5 * y
   >V>   "9"
   >L>   "1"
   >O>   "10"
   >L>   "5"
   >V>   "2"
   >O>   "10"
   >O>   "0"
```

First you see the line number (`9 *-*`) followed by the expression. Then the expression is broken down by operation as follows:

```
>V>   "9"    (value of variable x)
>L>   "1"    (value of literal 1)
>O>   "10"   (result of operation x + 1)
>L>   "5"    (value of literal 5)
>V>   "2"    (value of variable y)
>O>   "10"   (result of operation 5 * y)
>O>   "0"    (result of final operation 10 > 10 is false)
```

Tracing Results

To trace only the final result of an expression, use the TRACE R (TRACE Results) form of the TRACE instruction. All expressions that follow the instruction are analyzed and the results are displayed as:

```
>>>   Final result of an expression
```

If you changed the TRACE instruction operand in the previous example from an I to an R, you would see the following results.

```
 9 *-* IF x + 1 > 5 * y
   >>>   "0"
```

In addition to tracing operations and results, the TRACE instruction offers other types of tracing. For information about the other types of tracing with the TRACE instruction, see *z/OS TSO/E REXX Reference*.

Exercises - Using the TRACE Instruction

Write an exec with a complex expression, such as:

```
IF (A > B) | (C < 2 * D) THEN …
```

Define A, B, C, and D in the exec and use the TRACE I instruction.

ANSWER

Possible Solution

```
/****************************** REXX ******************************/
/* This exec uses the TRACE instruction to show how an expression  */
/* is evaluated, operation by operation.                           */
/*****************************************************************/
 A = 1
 B = 2
 C = 3
 D = 4

 TRACE I

 IF (A > B) | (C < 2 * D) THEN
   SAY 'At least one expression was true.'
 ELSE
   SAY 'Neither expression was true.'
```

When this exec is run, you see the following:

```
 12 *-* IF (A > B) | (C < 2 * D)
    >V>   "1"
    >V>   "2"
    >O>   "0"
    >V>   "3"
    >L>   "2"
    >V>   "4"
    >O>   "8"
    >O>   "1"
    >O>   "1"
    *-*  THEN
13  *-*  SAY 'At least one expression was true.'
    >L>    "At least one expression was true."
At least one expression was true.
```

Chapter 4. Controlling the Flow Within an Exec

This chapter introduces instructions that alter the sequential execution of an exec and demonstrates how those instructions are used.

Generally when an exec runs, one instruction after another executes, starting with the first and ending with the last. The language processor, unless told otherwise, executes instructions sequentially.

You can alter the order of execution within an exec by using specific REXX instructions that cause the language processor to skip some instructions, repeat others, or jump to another part of the exec. These specific REXX instructions can be classified as follows:

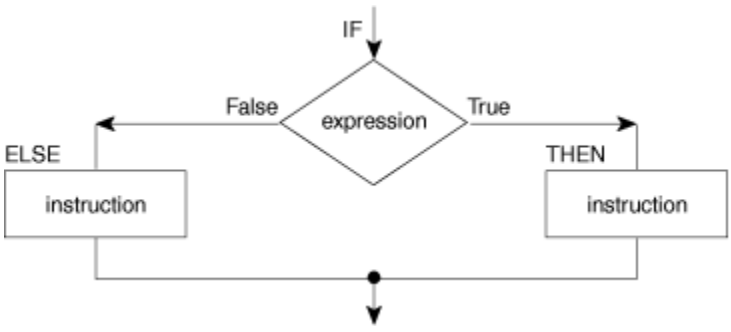
- Conditional instructions, which set up at least one condition in the form of an expression. If the condition is true, the language processor selects the path following that condition. Otherwise the language processor selects another path. The REXX conditional instructions are:
  - IF *expression*/THEN/ELSE
  - SELECT/WHEN *expression*/OTHERWISE/END.
- Looping instructions, which tell the language processor to repeat a set of instructions. A loop can repeat a specified number of times or it can use a condition to control repeating. REXX looping instructions are:
  - DO *expression*/END
  - DO FOREVER/END
  - DO WHILE *expression=true*/END
  - DO UNTIL *expression=true*/END
- Interrupt instructions, which tell the language processor to leave the exec entirely or leave one part of the exec and go to another part, either permanently or temporarily. The REXX interrupt instructions are:
  - EXIT
  - SIGNAL *label*
  - CALL *label*/RETURN

Using Conditional Instructions

There are two types of conditional instructions. IF/THEN/ELSE can direct the execution of an exec to one of two choices. SELECT/WHEN/OTHERWISE/END can direct the execution to one of many choices.

IF/THEN/ELSE Instructions

The examples of IF/THEN/ELSE instructions in previous chapters demonstrated the two-choice selection. In a flow chart, this appears as follows:



As a REXX instruction, the flowchart example looks like:

```
IF expression THEN instruction
              ELSE instruction
```

You can also arrange the clauses in one of the following ways to enhance readability:

```
IF expression THEN
   instruction
ELSE
   instruction
```

or

```
IF expression
  THEN
    instruction
  ELSE
    instruction
```

When you put the entire instruction on one line, you must separate the THEN clause from the ELSE clause with a semicolon.

```
IF expression THEN instruction; ELSE instruction
```

Generally, at least one instruction should follow the THEN and ELSE clauses. When either clause has no instructions, it is good programming practice to include NOP (no operation) next to the clause.

```
IF expression THEN
   instruction
ELSE NOP
```

If you have more than one instruction for a condition, begin the set of instructions with a DO and end them with an END.

```
IF weather = rainy THEN
   SAY 'Find a good book.'
ELSE
   DO
     SAY 'Would you like to play tennis or golf?'
     PULL answer
   END
```

Without the enclosing DO and END, the language processor assumes only one instruction for the ELSE clause.

Nested IF/THEN/ELSE Instructions

Sometimes it is necessary to have one or more IF/THEN/ELSE instructions within other IF/THEN/ELSE instructions. Having one type of instruction within another is called nesting. With nested IF instructions, it is important to match each IF with an ELSE and each DO with an END.

```
IF weather = fine THEN
   DO
      SAY 'What a lovely day!'
      IF tenniscourt = free THEN
         SAY 'Shall we play tennis?'
      ELSE NOP
   END
ELSE
   SAY 'Shall we take our raincoats?'
```

Not matching nested IFs to ELSEs and DOs to ENDs can have some surprising results. If you eliminate the DOs and ENDs and the ELSE NOP, as in the following example, what is the outcome?

Example of Missing Instructions

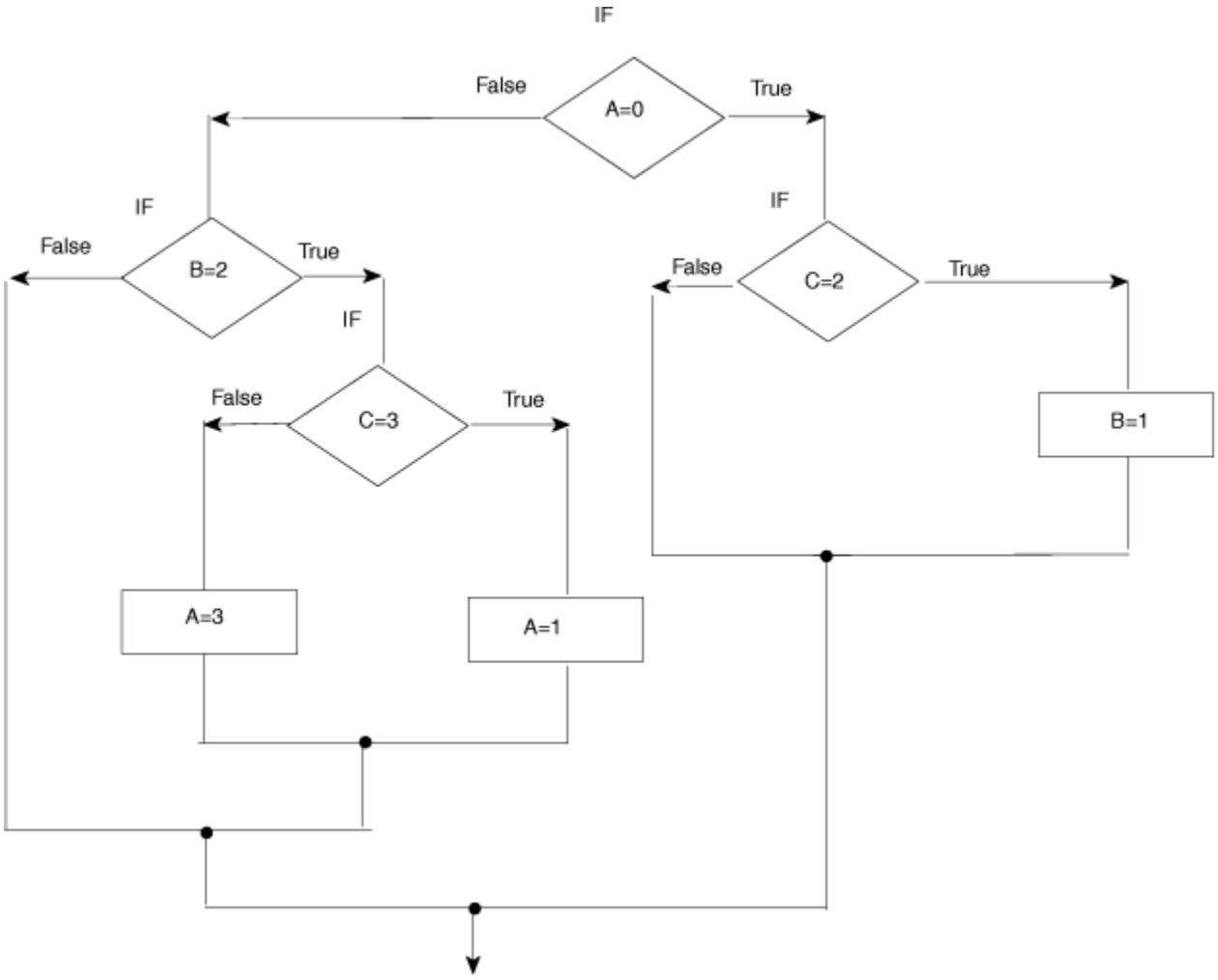
```
/******************************* REXX ***************************/
/* This exec demonstrates what can happen when you do not include  */
/* DOs, ENDs, and ELSEs in nested IF/THEN/ELSE instructions.       */
/****************************************************************/
 weather = 'fine'
 tenniscourt = 'occupied'

 IF weather = 'fine' THEN
    SAY 'What a lovely day!'
    IF tenniscourt = 'free' THEN
       SAY 'Shall we play tennis?'
 ELSE
    SAY 'Shall we take our raincoats?'
```

By looking at the exec you might assume the ELSE belongs to the first IF. However, the language processor associates an ELSE with the nearest unpaired IF. The outcome is as follows:

```
What a lovely day!
Shall we take our raincoats?
```

Exercise - Using the IF/THEN/ELSE Instruction

Write the REXX instructions for the following flowchart:



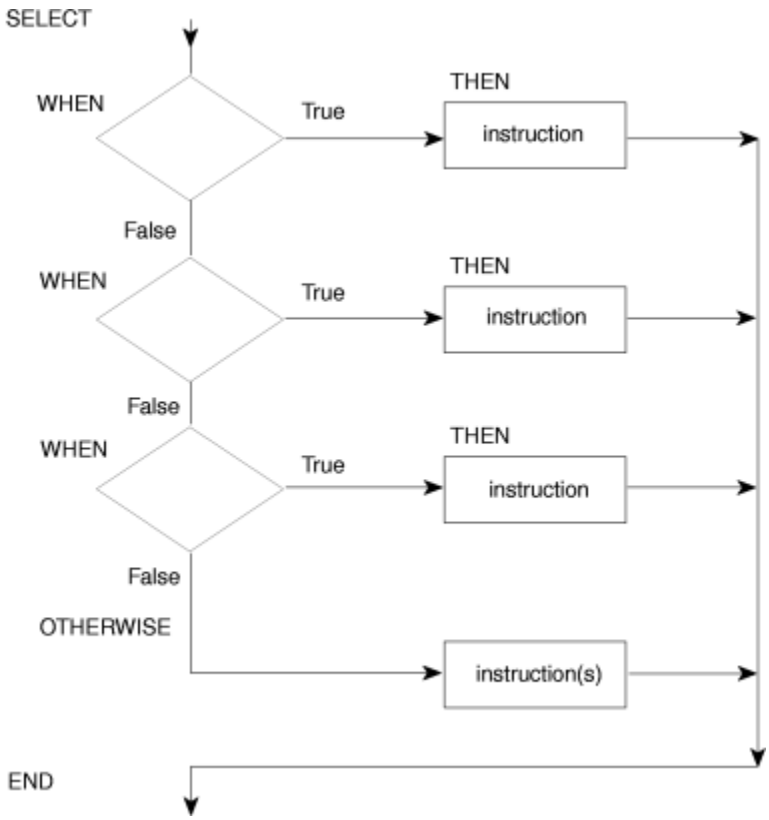
ANSWER

Possible Solution

```
IF A = 0 THEN
   IF C = 2 THEN
      B = 1
   ELSE NOP
ELSE
   IF B = 2 THEN
     IF C = 3 THEN
        A = 1
     ELSE
        A = 3
   ELSE NOP
```

SELECT/WHEN/OTHERWISE/END Instruction

To select one of any number of choices, use the SELECT/WHEN/OTHERWISE/END instruction. In a flowchart it appears as follows:



As a REXX instruction, the flowchart example looks like:

```
SELECT
   WHEN  expression  THEN  instruction
   WHEN  expression  THEN  instruction
   WHEN  expression  THEN  instruction
   :
   OTHERWISE
     instruction(s)
END
```

The language processor scans the WHEN clauses starting at the beginning until it finds a true expression. After it finds a true expression, it ignores all other possibilities, even though they might also be true. If no WHEN expressions are true, it processes the instructions following the OTHERWISE clause.

As with the IF/THEN/ELSE instruction, when you have more than one instruction for a possible path, begin the set of instructions with a DO and end them with an END. However, if more than one instruction follows the OTHERWISE keyword, DO and END are not necessary.

Example Using SELECT/WHEN/OTHERWISE/END

```
/******************************* REXX ***************************/
/* This exec receives input with a person's age and sex.  In     */
/* reply it displays a person's status as follows:               */
/*      BABIES    -  under 5                                     */
/*      GIRLS     -  female 5 to 12                              */
/*      BOYS      -  male 5 to 12                                */
/*      TEENAGERS -  13 through 19                               */
/*      WOMEN     -  female 20 and up                            */
/*      MEN       -  male 20 and up                              */
/****************************************************************/
 PARSE ARG age sex .

 SELECT
   WHEN age < 5 THEN               /* person younger than 5       */
     status = 'BABY'
   WHEN age < 13 THEN              /* person between 5 and 12     */
     DO
       IF sex = 'M' THEN           /* boy between 5 and 12        */
          status = 'BOY'
       ELSE                        /* girl between 5 and 12       */
          status = 'GIRL'
     END
   WHEN age < 20 THEN              /* person between 13 and 19    */
     status = 'TEENAGER'
   OTHERWISE
     IF sex = 'M' THEN             /* man 20 or older             */
       status = 'MAN'
     ELSE                          /* woman 20 or older           */
       status = 'WOMAN'
 END

 SAY 'This person should be counted as a' status '.'
```

Each SELECT must end with an END. Indenting each WHEN makes an exec easier to read.

Exercises - Using the SELECT/WHEN/OTHERWISE/END Instruction

"Thirty days hath September, April, June, and November; all the rest have thirty-one, save February alone ..."

Write an exec that provides the number of days in a month. First have the exec ask the user for a month specified as a number from 1 to 12 (with January being 1, February 2, and so forth). Then have the exec reply with the number of days. For month "2", the reply can be "28 or 29".

ANSWER

Possible Solution

```
/******************************** REXX ***************************/
/* This exec requests the user to enter a month as a whole number  */
/* from 1 to 12 and responds with the number of days in that       */
/* month.                                                          */
/*******************************************************************/

 SAY 'To find out the number of days in a month,'
 SAY 'Enter the month as a number from 1 to 12.'
 PULL month

 SELECT
   WHEN month = 9 THEN
     days = 30
   WHEN month = 4 THEN
     days = 30
   WHEN month = 6 THEN
     days = 30
   WHEN month = 11 THEN
     days = 30
   WHEN month = 2 THEN
     days = '28 or 29'
   OTHERWISE
     days = 31
 END

 SAY 'There are' days 'days in Month' month '.'
```

Using Looping Instructions

There are two types of looping instructions, **repetitive loops** and **conditional loops**. Repetitive loops allow you to repeat instructions a certain number of times, and conditional loops use a condition to control repeating. All loops, regardless of the type, begin with the DO keyword and end with the END keyword.

Repetitive Loops

The simplest loop tells the language processor to repeat a group of instructions a specific number of times using a constant following the keyword DO.

```
DO 5
  SAY 'Hello!'
END
```

When you run this example, you see five lines of `Hello!`.

```
  Hello!
  Hello!
  Hello!
  Hello!
  Hello!
```

You can also use a variable in place of a constant as in the following example, which gives you the same results.

```
number = 5
DO number
  SAY 'Hello!'
END
```

A variable that controls the number of times a loop repeats is called a **control variable**. Unless you specify otherwise, the control variable increases by 1 each time the loop repeats.

```
DO number = 1 TO 5
  SAY 'Loop' number
  SAY 'Hello!'
END
  SAY 'Dropped out of the loop when number reached' number
```

This example results in five lines of `Hello!` preceded by the number of the loop. The number increases at the bottom of the loop and is tested at the top.

```
  Loop 1
  Hello!
  Loop 2
  Hello!
  Loop 3
  Hello!
  Loop 4
  Hello!
  Loop 5
  Hello!
  Dropped out of the loop when number reached 6
```

You can change the increment of the control variable with the keyword BY as follows:

```
DO number = 1 TO 10 BY 2
  SAY 'Loop' number
  SAY 'Hello!'
END
  SAY 'Dropped out of the loop when number reached' number
```

This example has results similar to the previous example except the loops are numbered in increments of two.

```
  Loop 1
  Hello!
  Loop 3
  Hello!
  Loop 5
  Hello!
  Loop 7
  Hello!
  Loop 9
  Hello!
  Dropped out of the loop when number reached 11
```

Infinite Loops

What happens when the control variable of a loop cannot attain the last number? For example, in the following exec segment, `count` does not increase beyond 1.

```
DO count = 1 to 10
  SAY 'Number' count
  count = count - 1
END
```

The result is called an infinite loop because `count` alternates between 1 and 0 and an endless number of lines saying `Number 1` appear on the screen.

IMPORTANT - Stopping An Infinite Loop

When you suspect an exec is in an infinite loop, you can end the exec by pressing the attention interrupt key, sometimes labeled PA1. You will then see message IRX0920I. In response to this message, type HI for halt interpretation and press the Enter key. If that doesn't stop the loop, you can press the attention interrupt key again, type HE for halt execution, and press the Enter key.

HI will not halt an infinitely looping or long running external function, subroutine, or host command written in a language other than REXX and that was called by your exec. The HI condition is not checked by the REXX interpreter until control returns from the function, subroutine, or host command.

Example of EXEC1, an exec that calls an external function

```
/********************* REXX ***************************************/
/* Invoke a user-written external function, 'myfunct'.            */
/* not written in REXX.  For example, it might have been coded    */
/* in PL/I or assembler.                                          */
/**********************************************************************/
x = myfunct(1)
exit
```

If `myfunct` enters an infinite loop, pressing the attention interrupt key and entering HI will not stop `myfunct`. However, pressing the attention interrupt key and then entering HE will stop the function and the exec (EXEC1) that called it. HE does not automatically stop any exec that called EXEC1, unless you are running under ISPF. For more information about the HE condition, see *z/OS TSO/E REXX Reference*.

Note: HE does not alter the halt condition, which is raised by HI. If you entered HI before you entered HE (for example, you may have first issued HI and it failed to end your exec), the halt condition will remain set for the exec and all calling execs. HE will stop your exec, and then the halt condition, raised when you entered HI, will be recognized by any exec that called your exec.

DO FOREVER Loops

Sometimes you might want to purposely write an infinite loop; for instance, in an exec that reads records from a data set until it reaches end of file, or in an exec that interacts with a user until the user enters a particular symbol to end the loop. You can use the EXIT instruction to end an infinite loop when a condition is met, as in the following example. More about the EXIT instruction appears in “EXIT Instruction” on page 53.

Example Using a DO FOREVER Loop

```
/****************************** REXX ******************************/
/* This exec prints data sets named by a user until the user enters*/
/* a null line.                                                    */
/**********************************************************************/
   DO FOREVER
     SAY 'Enter the name of the next data set or a blank to end.'
     PULL dataset_name
     IF dataset_name = '' THEN
       EXIT
     ELSE
       DO
        "PRINTDS DA("dataset_name")"
         SAY dataset_name 'printed.'
       END
   END
```

This example sends data sets to the printer and then issues a message that the data set was printed. When the user enters a blank, the loop ends and so does the exec. To end the loop without ending the exec, use the LEAVE instruction, as described in the following topic.

LEAVE Instruction

TheLEAVE instruction causes an immediate exit from a repetitive loop. Control goes to the instruction following the END keyword of the loop. An example of using the LEAVE instruction follows:

Example Using the LEAVE Instruction

```
/******************************* REXX ***************************/
/* This exec uses the LEAVE instruction to exit from a DO FOREVER  */
/* loop that sends data sets to the printer.                       */
/*****************************************************************/
   DO FOREVER
     SAY 'Enter the name of the next data set.'
     SAY 'When there are no more data sets, enter QUIT.'
     PULL dataset_name
     IF dataset_name = 'QUIT' THEN
        LEAVE
     ELSE
        DO
          "PRINTDS DA("dataset_name")"
           SAY dataset_name 'printed.'
        END
   END
   SAY 'Good-bye.'
```

ITERATE Instruction

Anotherinstruction, ITERATE, stops execution from within the loop and passes control to the DO instruction at the top of the loop. Depending on the type of DO instruction, a control variable is increased and tested and/or a condition is tested to determine whether to repeat the loop. Like LEAVE, ITERATE is used within the loop.

```
DO count = 1 TO 10
  IF count = 8
    THEN
      ITERATE
    ELSE
      SAY 'Number' count
END
```

This example results in a list of numbers from 1 to 10 with the exception of number 8.

```
  Number 1
  Number 2
  Number 3
  Number 4
  Number 5
  Number 6
  Number 7
  Number 9
  Number 10
```

Exercises - Using Loops

1. What are the results of the following loops?

   a.
   ```
   DO digit = 1 TO 3
     SAY digit
   END
   SAY 'Digit is now' digit
   ```

   b.
   ```
   DO count = 10 BY -2 TO 6
     SAY count
   ```

```
END
SAY 'Count is now' count
```

c.
```
DO index = 10 TO 8
  SAY 'Hup! Hup! Hup!'
END
SAY 'Index is now' index
```

2. Sometimes an infinite loop can occur when input to end the loop doesn't match what is expected. For instance, in the previous example using the “LEAVE Instruction” on page 47, what happens when the user enters `Quit` and the PULL instruction is changed to a PARSE PULL instruction?

```
PARSE PULL dataset_name
```

ANSWERS

1. The results of the repetitive loops are as follows:

   a. 1
      2
      3
      Digit is now 4

   b. 10
      8
      6
      Count is now 4

   c. (blank)
      Index is now 10

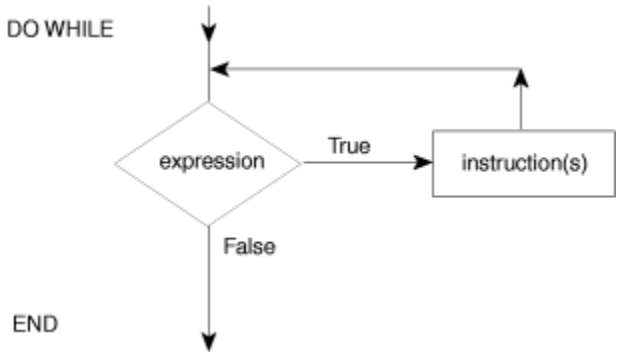
2. The user would be unable to leave the loop because "Quit" is not equal to "QUIT". In this case, omitting the PARSE keyword is preferred because regardless of whether the user enters "quit", "QUIT", or "Quit", the language processor translates the input to uppercase before comparing it to "QUIT".

Conditional Loops

There are two types of conditional loops, DO WHILE and DO UNTIL. Both types of loops are controlled by one or more expressions. However, DO WHILE loops test the expression before the loop executes the first time and repeat only when the expression is true. DO UNTIL loops test the expression after the loop executes at least once and repeat only when the expression is false.

DO WHILE Loops

DO WHILE loops in a flowchart appear as follows:



As REXX instructions, the flowchart example looks like:

```
DO WHILE  expression                    /* expression must be true */
   instruction(s)
END
```

Use a DO WHILE loop when you want to execute the loop while a condition is true. DO WHILE tests the condition at the top of the loop. If the condition is initially false, the loop is never executed.

You can use a DO WHILE loop instead of the DO FOREVER loop in the example using the "LEAVE Instruction" on page 47. However, you need to initialize the loop with a first case so the condition can be tested before you get into the loop. Notice the first case initialization in the beginning three lines of the following example.

Example Using DO WHILE

```
/******************************** REXX ***************************/
/* This exec uses a DO WHILE loop to send data sets to the system  */
/* printer.                                                        */
/*****************************************************************/
  SAY 'Enter the name of a data set to print.'
  SAY 'If there are no data sets, enter QUIT.'
  PULL dataset_name
  DO WHILE dataset_name \= 'QUIT'
     "PRINTDS DA("dataset_name")"
     SAY dataset_name 'printed.'
     SAY 'Enter the name of the next data set.'
     SAY 'When there are no more data sets, enter QUIT.'
     PULL dataset_name
  END
  SAY 'Good-bye.'
```

Exercise - Using a DO WHILE Loop

Write an exec with a DO WHILE loop that asks passengers on a commuter airline if they want a window seat and keeps track of their responses. The flight has 8 passengers and 4 window seats. Discontinue the loop when all the window seats are taken. After the loop ends, display the number of window seats taken and the number of passengers questioned.

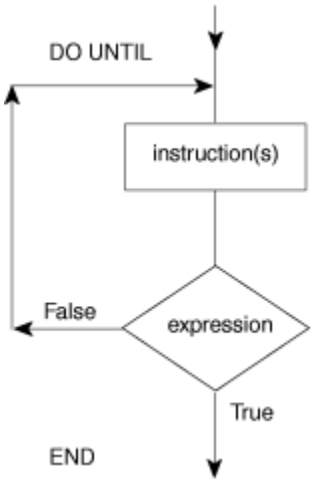
ANSWER

Possible Solution:

```
/******************************** REXX ***************************/
/* This exec uses a DO WHILE loop to keep track of window seats in */
/* an 8-seat commuter airline.                                     */
/*****************************************************************/

  window_seats = 0        /* Initialize window seats to 0          */
  passenger = 0           /* Initialize passengers to 0            */

  DO WHILE (passenger < 8) & (window_seats \= 4)

   /*****************************************************************/
   /* Continue while you have not questioned all 8 passengers and  */
   /* while all the window seats are not taken.                     */
   /*****************************************************************/

    SAY 'Do you want a window seat?  Please answer Y or N.'
    PULL answer
    passenger = passenger + 1
                      /* Increase the number of passengers by 1 */
    IF answer = 'Y' THEN
        window_seats = window_seats + 1
                     /* Increase the number of window seats by 1 */
    ELSE NOP
  END

  SAY window_seats 'window seats were assigned.'
  SAY passenger 'passengers were questioned.'
```

DO UNTIL Loops

DO UNTIL loops in a flowchart appear as follows:



As REXX instructions, the flowchart example looks like:

```
DO UNTIL  expression                  /* expression must be false */
   instruction(s)
END
```

Use DO UNTIL loops when a condition is not true and you want to execute the loop until the condition is true. The DO UNTIL loop tests the condition at the end of the loop and repeats only when the condition is false. Otherwise the loop executes once and ends. For example:

Example Using DO UNTIL

```
/******************************* REXX ***************************/
/* This exec uses a DO UNTIL loop to ask for a password.  If the   */
/* password is incorrect three times, the loop ends.               */
/*****************************************************************/
   password = 'abracadabra'
   time = 0
   DO UNTIL (answer = password) | (time = 3)
     SAY 'What is the password?'
     PULL answer
     time = time + 1
   END
```

Exercise - Using a DO UNTIL Loop

Change the exec in the previous exercise, “Exercise - Using a DO WHILE Loop” on page 49, from a DO WHILE to a DO UNTIL loop and achieve the same results. Remember that DO WHILE loops check for true expressions and DO UNTIL loops check for false expressions, which means their logical operators are often reversed.

ANSWER

Possible Solution

```
/******************************** REXX ***************************/
/* This exec uses a DO UNTIL loop to keep track of window seats in */
/* an 8-seat commuter airline.                                     */
/*******************************************************************/

  window_seats = 0        /* Initialize window seats to 0          */
  passenger = 0           /* Initialize passengers to 0            */

  DO UNTIL (passenger >= 8) | (window_seats = 4)

   /*****************************************************************/
   /* Continue until you have questioned all 8 passengers or until */
   /* all the window seats are taken.                               */
   /*****************************************************************/

    SAY 'Do you want a window seat?  Please answer Y or N.'
    PULL answer
    passenger = passenger + 1
                     /* Increase the number of passengers by 1  */
    IF answer = 'Y' THEN
       window_seats = window_seats + 1
                    /* Increase the number of window seats by 1 */
    ELSE NOP
  END
  SAY window_seats 'window seats were assigned.'
  SAY passenger 'passengers were questioned.'
```

Combining Types of Loops

You can combine repetitive and conditional loops to create a compound loop. The following loop is set to repeat 10 times while a certain condition is met, at which point it stops.

```
quantity = 20
DO number = 1 TO 10 WHILE quantity < 50
  quantity = quantity + number
  SAY 'Quantity = 'quantity '  (Loop 'number')'
END
```

The result of this example is as follows:

```
  Quantity = 21   (Loop 1)
  Quantity = 23   (Loop 2)
  Quantity = 26   (Loop 3)
  Quantity = 30   (Loop 4)
  Quantity = 35   (Loop 5)
  Quantity = 41   (Loop 6)
  Quantity = 48   (Loop 7)
  Quantity = 56   (Loop 8)
```

You can substitute a DO UNTIL loop, change the comparison operator from < to >, and get the same results.

```
quantity = 20
DO number = 1 TO 10 UNTIL quantity > 50
  quantity = quantity + number
  SAY 'Quantity = 'quantity '  (Loop 'number')'
END
```

Nested DO Loops

Like nested IF/THEN/ELSE instructions, DO loops can also be within other DO loops. A simple example follows:

```
DO outer = 1 TO 2
   DO inner = 1 TO 2
     SAY 'HIP'
   END
```

```
   SAY 'HURRAH'
END
```

The output from this example is:

```
   HIP
   HIP
   HURRAH
   HIP
   HIP
   HURRAH
```

If you need to leave a loop when a certain condition arises, use the LEAVE instruction followed by the control variable of the loop. If the LEAVE instruction is for the inner loop, you leave the inner loop and go to the outer loop. If the LEAVE instruction is for the outer loop, you leave both loops.

To leave the inner loop in the preceding example, add an IF/THEN/ELSE instruction that includes a LEAVE instruction after the IF instruction.

```
DO outer = 1 TO 2
   DO inner = 1 TO 2
     IF inner > 1 THEN
       LEAVE inner
     ELSE
       SAY 'HIP'
   END
   SAY 'HURRAH'
END
```

The result is as follows:

```
   HIP
   HURRAH
   HIP
   HURRAH
```

Exercises - Combining Loops

1. What happens when the following exec runs?

```
DO outer = 1 TO 3
  SAY                  /* Write a blank line                    */
  DO inner = 1 TO 3
    SAY 'Outer' outer 'Inner' inner
  END
END
```

2. Now what happens when the LEAVE instruction is added?

```
DO outer = 1 TO 3
  SAY                  /* Write a blank line                    */
  DO inner = 1 TO 3
    IF inner = 2 THEN
      LEAVE inner
    ELSE
      SAY 'Outer' outer 'Inner' inner
  END
END
```

ANSWERS

1. When this example runs, you see on your screen the following:

```
Outer 1  Inner 1
Outer 1  Inner 2
Outer 1  Inner 3

Outer 2  Inner 1
Outer 2  Inner 2
Outer 2  Inner 3

Outer 3  Inner 1
Outer 3  Inner 2
Outer 3  Inner 3
```

2. The result is one line of output for each of the inner loops.

```
Outer 1  Inner 1

Outer 2  Inner 1

Outer 3  Inner 1
```

Using Interrupt Instructions

Instructions that interrupt the flow of an exec can cause the exec to:

- Terminate (EXIT)
- Skip to another part of the exec marked by a label (SIGNAL)
- Go temporarily to a subroutine either within the exec or outside the exec (CALL/RETURN).

EXIT Instruction

The EXIT instruction causes an exec to unconditionally end and return to where the exec was invoked. If the exec was initiated from the PROC section of an ISPF selection panel, EXIT returns to the ISPF panel. If the exec was called by a program, such as another exec, EXIT returns to the program. More about calling external routines appears later in this chapter and in Chapter 6, “Writing Subroutines and Functions,” on page 65.

In addition to ending an exec, EXIT can also return a value to the invoker of the exec. If the exec was invoked as a subroutine from another REXX exec, the value is received in the REXX special variable RESULT. If the exec was invoked as a function, the value is received in the original expression at the point where the function was invoked. Otherwise, the value is received in the REXX special variable RC. The value can represent a return code and can be in the form of a constant or an expression that is computed.

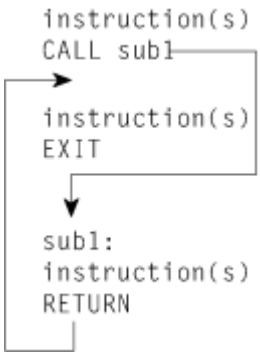
Example Using the EXIT Instruction:

```
/******************************** REXX ***************************/
/* This exec uses the EXIT instruction to end the exec and return  */
/* a value that indicates whether or not a job applicant gets the  */
/* job.  A value of 0 means the applicant does not qualify for     */
/* the job, but a value of 1 means the applicant gets the job.     */
/* The value is placed in the REXX special variable RESULT.        */
/*****************************************************************/
SAY 'How many months of experience do you have?  Please enter'
SAY 'the months as a number.'
PULL month

SAY 'Can you supply 3 references?  Please answer Y or N.'
PULL reference

SAY 'Are you available to start work tomorrow?  Please answer Y or N.'
PULL tomorrow

IF (month > 24) & (reference = 'Y') & (tomorrow = 'Y') THEN
  job = 1                                  /* person gets the job */
ELSE
  job = 0                            /* person does not get the job */

EXIT job
```

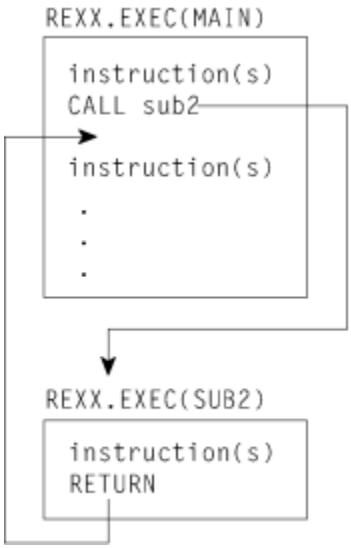
CALL/RETURN Instructions

The CALL instruction interrupts the flow of an exec by passing control to an internal or external subroutine. An internal subroutine is part of the calling exec. An external subroutine is another exec. The RETURN instruction returns control from a subroutine back to the calling exec and optionally returns a value.

When calling an internal subroutine, CALL passes control to a label specified after the CALL keyword. When the subroutine ends with the RETURN instruction, the instructions following CALL are executed.



When calling an external subroutine, CALL passes control to the exec name that is specified after the CALL keyword. When the external subroutine completes, you can use the RETURN instruction to return to where you left off in the calling exec.

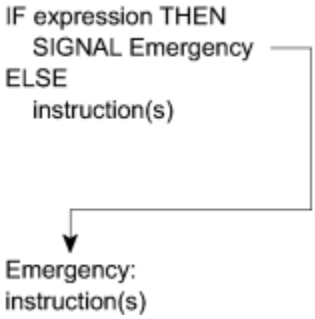


For more information about calling subroutines, see Chapter 6, "Writing Subroutines and Functions," on page 65.

SIGNAL Instruction

The SIGNAL instruction, like CALL, interrupts the normal flow of an exec and causes control to pass to a specified label. The label to which control passes can appear before or after the SIGNAL instruction. Unlike CALL, SIGNAL does not return to a specific instruction to resume execution. When you use SIGNAL from within a loop, the loop automatically ends; and when you use SIGNAL from an internal routine, the internal routine will not return to its caller.

In the following example, if the expression is true, then the language processor goes to the label `Emergency:` and skips all instructions in between.



SIGNAL is useful for testing execs or to provide an emergency course of action. It should not be used as a convenient way to move from one place in an exec to another. SIGNAL does not provide a way to return as does the CALL instruction described in "CALL/RETURN Instructions" on page 54.

For more information about the SIGNAL instruction, see "SIGL" on page 106, and *z/OS TSO/E REXX Reference*.

Chapter 5. Using Functions

This chapter defines what a function is and describes how to use the built-in functions.

What is a Function?

Afunction is a sequence of instructions that can receive data, process that data, and return a value. In REXX, there are several kinds of functions:

- Built-in functions — These functions are built into the language processor. More about built-in functions appears later in this chapter.
- User-written functions — These functions are written by an individual user or supplied by an installation and can be internal or external. An *internal function* is part of the current exec that starts at a label. An *external function* is a self-contained program or exec outside of the calling exec. More information about user-written functions appears in "Writing a Function" on page 72.
- Function packages — These are groups of functions and subroutines written by an individual user or supplied by an installation. They are link-edited into load modules and categorized as user, local, and system. TSO/E external functions are provided in a system function package. More information about TSO/E external functions appears in "TSO/E External Functions" on page 111.

Regardless of the kind of function, all functions return a value to the exec that issued the function call. To call a function, type the function name directly followed by one or more arguments within parentheses. **There can be no space between the function name and the left parenthesis.**

```
function(arguments)
```

A function call can contain up to 20 arguments separated by commas. Each argument can be one or more of the following.

- Blank

  ```
  function( )
  ```

- Constant

  ```
  function(55)
  ```

- Symbol

  ```
  function(symbol_name)
  ```

- Literal string

  ```
  function('With a literal string')
  ```

- Option recognized by the function

  ```
  function(option)
  ```

- Another function

  ```
  function(function(arguments))
  ```

- Combination of argument types

  ```
  function('With a literal string', 55, option)
  ```

When the function returns a value, and all functions must return values, the value replaces the function call. In the following example, the value returned is added to 7 and the sum is displayed.

```
SAY 7 + function(arguments)
```

A function call generally appears in an expression. Therefore a function call, like an expression, does not usually appear in an instruction by itself.

Example of a Function

Calculations represented by functions often require many instructions. For instance, the simple calculation for finding the highest number in a group of three numbers, might be written as follows:

Finding a Maximum Number

```
/***************************** REXX ********************************/
/* This exec receives three numbers from a user and analyzes which */
/* number is the greatest.                                         */
/*******************************************************************/

 PARSE ARG number1, number2, number3 .

 IF number1 > number2 THEN
    IF number1 > number3 THEN
       greatest = number1
    ELSE
       greatest = number3
 ELSE
    IF number2 > number3 THEN
       greatest = number2
    ELSE
       greatest = number3

 RETURN greatest
```

Rather than writing multiple instructions every time you want to find the maximum of a group of three numbers, you can use a built-in function that does the calculation for you and returns the maximum number. The function is called MAX and is used as follows:

```
MAX(number1,number2,number3,…)
```

To find the maximum of 45, -2, number, 199, and put the maximum into the symbol `biggest`, write the following instruction:

```
biggest = MAX(45,-2,number,199)
```

Built-In Functions

Over 50 functions are built into the language processor. The built-in functions fall into the following categories:

- Arithmetic functions

  These functions evaluate numbers from the argument and return a particular value.
- Comparison functions

  These functions compare numbers and/or strings and return a value.
- Conversion functions

  These functions convert one type of data representation to another type of data representation.
- Formatting functions

These functions manipulate the characters and spacing in strings supplied in the argument.

- String manipulating functions

  These functions analyze a string supplied in the argument (or a variable representing a string) and return a particular value.

- Miscellaneous functions

  These functions do not clearly fit into any of the other categories.

The following tables briefly describe the functions in each category. For a complete description of these functions, see *z/OS TSO/E REXX Reference*.

Arithmetic Functions

Function	Description
ABS	Returns the absolute value of the input number.
DIGITS	Returns the current setting of NUMERIC DIGITS.
FORM	Returns the current setting of NUMERIC FORM.
FUZZ	Returns the current setting of NUMERIC FUZZ.
MAX	Returns the largest number from the list specified, formatted according to the current NUMERIC settings.
MIN	Returns the smallest number from the list specified, formatted according to the current NUMERIC settings.
RANDOM	Returns a quasi-random, non-negative whole number in the range specified.
SIGN	Returns a number that indicates the sign of the input number.
TRUNC	Returns the integer part of the input number, and optionally a specified number of decimal places.

Comparison Functions

Function	Description
COMPARE	Returns 0 if the two input strings are identical. Otherwise, returns the position of the first character that does not match.
DATATYPE	Returns a string indicating the input string is a particular data type, such as a number or character.
SYMBOL	Returns this state of the symbol (variable, literal, or bad).

Conversion Functions

Function	Description
B2X	Returns a string, in character format, that represents the input binary string converted to hexadecimal. (Binary to hexadecimal)
C2D	Returns the decimal value of the binary representation of the input string. (Character to Decimal)

Function	Description
C2X	Returns a string, in character format, that represents the input string converted to hexadecimal. (Character to Hexadecimal)
D2C	Returns a string, in character format, that represents the input decimal number converted to binary. (Decimal to Character)
D2X	Returns a string, in character format, that represents the input decimal number converted to hexadecimal. (Decimal to Hexadecimal)
X2B	Returns a string, in character format, that represents the input hexadecimal string converted to binary. (Hexadecimal to binary)
X2C	Returns a string, in character format, that represents the input hexadecimal string converted to character. (Hexadecimal to Character)
X2D	Returns the decimal representation of the input hexadecimal string. (Hexadecimal to Decimal)

Formatting Functions

Function	Description
CENTER/ CENTRE	Returns a string of a specified length with the input string centered in it, with pad characters added as necessary to make up the length.
COPIES	Returns the specified number of concatenated copies of the input string.
FORMAT	Returns the input number, rounded and formatted.
JUSTIFY *	Returns a specified string formatted by adding pad characters between words to justify to both margins.
LEFT	Returns a string of the specified length, truncated or padded on the right as needed.
RIGHT	Returns a string of the specified length, truncated or padded on the left as needed.
SPACE	Returns the words in the input string with a specified number of pad characters between each word.
* Indicates a non-SAA built-in function provided only by TSO/E.	

String Manipulating Functions

Function	Description
ABBREV	Returns a string indicating if one string is equal to the specified number of leading characters of another string.
DELSTR	Returns a string after deleting a specified number of characters, starting at a specified point in the input string.
DELWORD	Returns a string after deleting a specified number of words, starting at a specified word in the input string.
FIND *	Returns the word number of the first word of a specified phrase found within the input string.

Function	Description
INDEX *	Returns the character position of the first character of a specified string found in the input string.
INSERT	Returns a character string after inserting one input string into another string after a specified character position.
LASTPOS	Returns the starting character position of the last occurrence of one string in another.
LENGTH	Returns the length of the input string.
OVERLAY	Returns a string that is the target string overlaid by a second input string.
POS	Returns the character position of one string in another.
REVERSE	Returns a character string, the characters of which are in reverse order (swapped end for end).
STRIP	Returns a character string after removing leading or trailing characters or both from the input string.
SUBSTR	Returns a portion of the input string beginning at a specified character position.
SUBWORD	Returns a portion of the input string starting at a specified word number.
TRANSLATE	Returns a character string with each character of the input string translated to another character or unchanged.
VERIFY	Returns a number indicating whether an input string is composed only of characters from another input string or returns the character position of the first unmatched character.
WORD	Returns a word from an input string as indicated by a specified number.
WORDINDEX	Returns the character position in an input string of the first character in the specified word.
WORDLENGTH	Returns the length of a specified word in the input string.
WORDPOS	Returns the word number of the first word of a specified phrase in the input string.
WORDS	Returns the number of words in the input string.
* Indicates a non-SAA built-in function provided only by TSO/E.	

Miscellaneous Functions

Function	Description
ADDRESS	Returns the name of the environment to which commands are currently being sent.
ARG	Returns an argument string or information about the argument strings to a program or internal routine.
BITAND	Returns a string composed of the two input strings logically ANDed together, bit by bit.
BITOR	Returns a string composed of the two input strings logically ORed together, bit by bit.
BITXOR	Returns a string composed of the two input strings eXclusive ORed together, bit by bit.
CONDITION	Returns the condition information, such as name and status, associated with the current trapped condition.

Function	Description
DATE	Returns the date in the default format (dd mon yyyy) or in one of various optional formats.
ERRORTEXT	Returns the error message associated with the specified error number.
EXTERNALS *	Returns the number of elements in the terminal input buffer. In TSO/E, this function always returns a 0.
LINESIZE *	Returns the current terminal line width minus 1.
QUEUED	Returns the number of lines remaining in the external data queue at the time when the function is invoked.
SOURCELINE	Returns either the line number of the last line in the source file or the source line specified by a number.
TIME	Returns the local time in the default 24-hour clock format (hh:mm:ss) or in one of various optional formats.
TRACE	Returns the trace actions currently in effect.
USERID *	Returns the TSO/E user ID, if the REXX exec is running in the TSO/E address space.
VALUE	Returns the value of a specified symbol and optionally assigns it a new value.
XRANGE	Returns a string of all 1-byte codes (in ascending order) between and including specified starting and ending values.
* Indicates a non-SAA built-in function provided only by TSO/E.	

Testing Input with Built-In Functions

Some of the built-in functions provide a convenient way to test input. When an interactive exec requests input, the user might respond with input that is not valid. For instance, in the example “Using Comparison Expressions” on page 30, the exec requests a dollar amount with the following instructions.

```
SAY 'What did you spend for lunch yesterday?'
SAY 'Please do not include the dollar sign.'
PARSE PULL last
```

If the user responds with a number only, the exec will process that information correctly. If the user responds with a number preceded by a dollar sign or with a word, such as `nothing`, the exec will return an error. To avoid getting an error, you can check the input with the DATATYPE function as follows:

```
DO WHILE DATATYPE(last) \= 'NUM'
  SAY 'Please enter the lunch amount again.'
  SAY 'The amount you entered was not a number without a dollar sign.'
  PARSE PULL last
END
```

Other useful built-in functions to test input are WORDS, VERIFY, LENGTH, and SIGN.

Exercise - Writing an Exec with Built-In Functions

Write an exec that checks a data set member name for a length of 8 characters. If a member name is longer than 8 characters, the exec truncates it to 8 and sends the user a message indicating the shortened name. Use the LENGTH and the SUBSTR built-in functions as described in *z/OS TSO/E REXX Reference*.

ANSWER

Possible Solution

```
/*************************** REXX ********************************/
/* This exec tests the length of a name for a data set member. If  */
/* the name is longer than 8 characters, the exec truncates the    */
/* extra characters and sends the user a message indicating the    */
/* shortened member name.                                          */
/*******************************************************************/
SAY 'Please enter a member name.'
PULL membername

IF LENGTH(membername) > 8 THEN  /* Name is longer than 8 characters*/
  DO
    membername = SUBSTR(membername,1,8)   /* Shorten the name to   */
                                          /* the first 8 characters*/
    SAY 'The member name you entered was too long.'
    SAY  membername 'will be used.'
  END
ELSE NOP
```

Chapter 6. Writing Subroutines and Functions

This chapter shows how to write subroutines and functions and compares their differences and similarities.

What are Subroutines and Functions?

Subroutines and functions are routines made up of a sequence of instructions that can receive data, process that data, and return a value. The routines can be:

Internal
The routine is within the current exec, marked by a label and used only by that exec.

External
A program or exec in a member of a partitioned data set that can be called by one or more execs. In order for an exec to call the routine, the exec and the routine must be allocated to a system file, for example SYSEXEC or SYSPROC, or be in the same PDS. For more information about allocating to a system file, see Appendix A, “Allocating Data Sets,” on page 171.

In many aspects, subroutines and functions are the same; yet they are different in a few major aspects, such as the way they are called and the way they return values.

- Calling a subroutine

  To call a subroutine, use the CALL instruction followed by the subroutine name (label or exec member name) and optionally followed by up to 20 arguments separated by commas. The subroutine call is an entire instruction.

  ```
  CALL  subroutine_name  argument1, argument2,...
  ```

  Issuing a CALL to internal label names for REXX subroutines and functions that are greater than eight characters, may have unintended results. Label names will be truncated to eight characters.

- Calling a function

  To call a function, use the function name (label or exec member name) immediately followed by parentheses that can contain arguments. There can be no space between the function name and the parentheses. The function call is part of an instruction, for example, an assignment instruction.

  ```
  x = function(argument1, argument2,...)
  ```

- Returning a value from a subroutine

  A subroutine does not have to return a value, but when it does, it sends back the value with the RETURN instruction.

  ```
  RETURN value
  ```

  The calling exec receives the value in the REXX special variable named RESULT.

  ```
  SAY 'The answer is' RESULT
  ```

- Returning a value from a function

  A function *must* return a value. When the function is a REXX exec, the value is returned with either the RETURN or EXIT instruction.

  ```
  RETURN value
  ```

The calling exec receives the value at the function call. The value replaces the function call, so that in the following example, x = value.

```
x = function(argument1, argument2,...)
```

When to Write Subroutines vs Functions

The actual instructions that make up a subroutine or a function can be identical. It is the way you want to use them in an exec that turns them into either a subroutine or a function. For example, the built-in function SUBSTR can be called as either a function or a subroutine. As a function, you invoke it as follows to shorten a word to its first eight characters:

```
x = SUBSTR('verylongword',1,8)          /* x is set to 'verylong' */
```

As a subroutine, you would get the same results with the following instructions:

```
CALL SUBSTR 'verylongword', 1, 8        /* x is set to 'verylong' */
x = RESULT
```

When deciding whether to write a subroutine or a function, ask yourself the following questions:

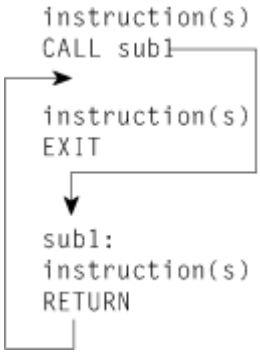
- Is a returned value optional? If so, write a subroutine.
- Do I need a value returned as an expression within an instruction? If so, write a function.

The rest of this chapter describes how to write subroutines, how to write functions, and finally summarizes the differences and similarities between the two.

Writing a Subroutine

A subroutine is a series of instructions that an exec invokes to perform a specific task. The instruction that invokes the subroutine is the CALL instruction. The CALL instruction may be used several times in an exec to invoke the same subroutine.

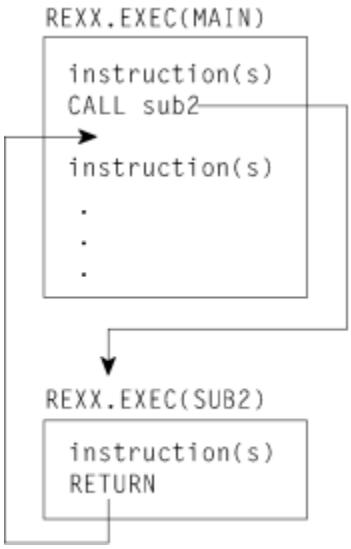
When the subroutine ends, it can return control to the instruction that directly follows the subroutine call. The instruction that returns control is the RETURN instruction.



Subroutines may be **internal** and designated by a label, or **external** and designated by the data set member name that contains the subroutine. The preceding example illustrates an internal subroutine named "sub1".

> **Note:**
>
> Because internal subroutines generally appear after the main part of the exec, when you have an internal subroutine, it is important to end the main part of the exec with the EXIT instruction.

The following illustrates an external subroutine named "sub2".



To determine whether to make a subroutine internal or external, you might consider factors, such as:

- Size of the subroutine. Very large subroutines often are external, whereas small subroutines fit easily within the calling exec.
- How you want to pass information. It is quicker to pass information through variables in an internal subroutine. This method is described in “Passing Information by Using Variables” on page 67.
- Whether the subroutine might be of value to more than one exec or user. If so, an external subroutine is preferable.

Passing Information to a Subroutine

An internal subroutine can share variables with its caller. Therefore you can use commonly shared variables to pass information between caller and internal subroutine. You can also use arguments to pass information to and from an internal subroutine. External subroutines, however, cannot share the same variables, and information must pass between them through arguments or some other external way, such as the data stack.

Passing Information by Using Variables

When an exec and its internal subroutine share the same variables, the value of a variable is what was last assigned, regardless of whether the assignment was in the main part of the exec or in the subroutine. In the following example, the value of `answer` is assigned in the subroutine and displayed in the main part of the exec. The variables `number1`, `number2`, and `answer` are shared.

Example of Passing Information in a Variable:

```
/****************************** REXX ******************************/
/* This exec receives a calculated value from an internal         */
/* subroutine and displays that value.                            */
/******************************************************************/

  number1 = 5
  number2 = 10
  CALL subroutine
  SAY answer                                    /* Displays 15 */
  EXIT

  subroutine:
  answer = number1 + number2
  RETURN
```

Using the same variables in an exec and its internal subroutine can sometimes create problems. In the following example, the main part of the exec and the subroutine use the same control variable, "i", for

their DO loops. As a result, the DO loop repeats only once in the main exec because the subroutine returns to the main exec with i = 6.

Example of a Problem Caused by Passing Information in a Variable:

```
/****************************** REXX ******************************/
/* NOTE: This exec contains an error.                             */
/* It uses a DO loop to call an internal subroutine and the       */
/* subroutine also uses a DO loop with same control variable as   */
/* the main exec.  The DO loop in the main exec repeats only once. */
/**********************************************************************/

  number1 = 5
  number2 = 10
  DO i = 1 TO 5
    CALL subroutine
    SAY answer                                    /* Displays 105 */
  END
  EXIT

  subroutine:
  DO i = 1 TO 5
    answer = number1 + number2
    number1 = number2
    number2 = answer
  END
  RETURN
```

To avoid this kind of problem in an internal subroutine, you can use:

- The PROCEDURE instruction as described in the next topic.
- Different variable names in a subroutine and pass arguments on the CALL instruction as described in “Passing Information by Using Arguments” on page 69.

Protecting Variables with the PROCEDURE Instruction

When you use the PROCEDURE instruction immediately after the subroutine label, all variables used in the subroutine become local to the subroutine and are shielded from the main part of the exec. You can also use the PROCEDURE EXPOSE instruction to protect all but a few specified variables.

The following two examples show the differing results when a subroutine uses the PROCEDURE instruction and when it doesn't.

Example Using the PROCEDURE Instruction

```
/****************************** REXX ******************************/
/* This exec uses a PROCEDURE instruction to protect the variables */
/* within its subroutine.                                         */
/**********************************************************************/
 number1 = 10
 CALL subroutine
 SAY number1 number2                     /* displays  10  NUMBER2 */
 EXIT

 subroutine: PROCEDURE
 number1 = 7
 number2 = 5
 RETURN
```

Example Without the PROCEDURE Instruction

```
/****************************** REXX ******************************/
/*  This exec does not use a PROCEDURE instruction to protect the  */
/*  variables within its subroutine.                               */
/******************************************************************/
 number1 = 10
 CALL subroutine
 SAY number1 number2                          /* displays 7  5  */
 EXIT

 subroutine:
 number1 = 7
 number2 = 5
 RETURN
```

Exposing Variables with PROCEDURE EXPOSE

To protect all but specific variables, use the EXPOSE option with the PROCEDURE instruction, followed by the variables that are to remain exposed to the subroutine.

Example Using PROCEDURE EXPOSE

```
/****************************** REXX ******************************/
/* This exec uses a PROCEDURE instruction with the EXPOSE option to*/
/* expose one variable, number1, in its subroutine.  The other     */
/* variable, number2, is set to null and displays its name in      */
/* uppercase.                                                      */
/******************************************************************/
 number1 = 10
 CALL subroutine
 SAY number1 number2                       /* displays 7  NUMBER2 */
 EXIT

 subroutine: PROCEDURE EXPOSE number1
 number1 = 7
 number2 = 5
 RETURN
```

For more information about the PROCEDURE instruction, see *z/OS TSO/E REXX Reference*.

Passing Information by Using Arguments

A way to pass information to either internal or external subroutines is through arguments. You can pass up to 20 arguments separated by commas on the CALL instruction as follows:

```
CALL subroutine_name  argument1, argument2, argument3,......
```

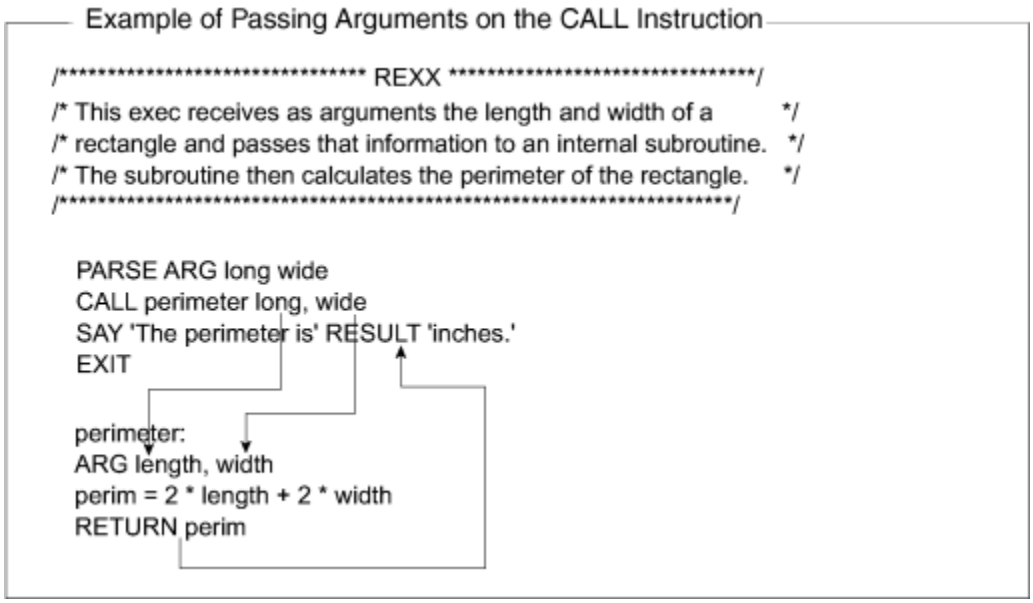
Using the ARG Instruction

The subroutine can receive the arguments with the ARG instruction. Arguments are also separated by commas in the ARG instruction.

```
ARG  arg1, arg2, arg3, .....
```

The names of the arguments on the CALL and the ARG instructions do not have to be the same because information is not passed by argument name but by position. The first argument sent becomes the first argument received and so forth. You can also set up a template in the CALL instruction, which is then used in the corresponding ARG instruction. For information about parsing with templates, see “Parsing Data” on page 83.

The following exec sends information to an internal subroutine that computes the perimeter of a rectangle. The subroutine returns a value in the variable `perim` that is specified after the RETURN instruction. The main exec receives the value in the special variable "RESULT".



Notice the positional relationships between `long` and `length`, and `wide` and `width`. Also notice how information is received from variable `perim` in the special variable RESULT.

Using the ARG Built-in Function

Another way for a subroutine to receive arguments is with the ARG built-in function. This function returns the value of a particular argument specified by a number that represents the argument position.

For instance, in the previous example, instead of the ARG instruction,

```
ARG length, width
```

you can use the ARG function as follows:

```
length = ARG(1)             /* puts the first argument into length */
width = ARG(2)              /* puts the second argument into width */
```

More information about the ARG function appears in *z/OS TSO/E REXX Reference*.

Receiving Information from a Subroutine

Although a subroutine can receive up to 20 arguments, it can specify only one expression on the RETURN instruction. That expression can be:

- A number

  ```
  RETURN 55
  ```

- One or more variables whose values are substituted or when no values were assigned, return their names

  ```
  RETURN value1 value2 value3
  ```

- A literal string

  ```
  RETURN 'Work complete.'
  ```

- An arithmetic, comparison, or logical expression whose value is substituted.

  ```
  RETURN 5 * number
  ```

Example - Writing an Internal and an External Subroutine

Write an exec that plays a simulated coin toss game of heads or tails between the computer and a user and displays the accumulated scores. Start off with the message, "This is a game of chance. Type 'heads', 'tails', or 'quit' and press the Enter key."

This means that there are four possible inputs:

- HEADS
- TAILS
- QUIT
- None of these three (not valid response).

Write an internal subroutine without arguments to check for valid input. Send valid input to an external subroutine that compares the valid input with a random outcome. Use the RANDOM built-in function as, RANDOM(0,1), and equate HEADS = 0, TAILS = 1. Return the result to the main program where results are tallied and displayed.

Good luck!

ANSWER

Possible Solution (Main Exec)

```
/*************************** REXX ********************************/
/* This exec plays a simulated coin toss game between the computer */
/* and a user.  The user enters heads, tails, or quit.  The user   */
/* is first checked for validity in an internal subroutine.        */
/* An external subroutine uses the RANDOM build-in function to     */
/* obtain a simulation of a throw of dice and compares the user    */
/* input to the random outcome.  The main exec receives            */
/* notification of who won the round.  Scores are maintained       */
/* and displayed after each round.                                 */
/*******************************************************************/
 SAY 'This is a game of chance.  Type "heads", "tails", or "quit"
 SAY '   and press ENTER.'
 PULL response
 computer = 0; user = 0        /* initialize scores to zero        */
 CALL check                    /* call internal subroutine, check  */
 DO FOREVER
   CALL throw response         /* call external subroutine, throw  */

   IF RESULT = 'machine' THEN  /* the computer won                 */
     computer = computer + 1   /* increase the computer score      */
   ELSE                        /* the user won                     */
     user = user + 1           /* increase the user score          */

   SAY 'Computer score = ' computer    '    Your score = ' user
   SAY 'Heads, tails, or quit?'
   PULL response
   CALL check                  /* call internal subroutine, check  */
 END
 EXIT
```

Possible Solution (Internal Subroutine named CHECK)

```
check:
/*********************************************************************/
/* This internal subroutine checks for valid input of "HEADS",       */
/* "TAILS", or "QUIT".  If the user entered anything else, the       */
/* subroutine tells the user that it is an invalid response and      */
/* asks the user to try again.  The subroutine keeps repeating       */
/* until the user enters valid input.  Information is returned to    */
/* the main exec through commonly used variables.                    */
/*********************************************************************/
 DO UNTIL outcome = 'correct'
   SELECT
     WHEN response = 'HEADS' THEN
       outcome = 'correct'
     WHEN response = 'TAILS' THEN
       outcome = 'correct'
     WHEN response = 'QUIT' THEN
       EXIT
     OTHERWISE
       outcome = 'incorrect'
       SAY "That's not a valid response.  Try again!"
       SAY "Heads, tails, or quit?"
       PULL response
   END
 END
 RETURN
```

Possible Solution (External Subroutine named THROW)

```
/****************************** REXX ******************************/
/* This external subroutine receives the valid input from the user,*/
/* analyzes it, gets a random "throw" from the computer and        */
/* compares the two values.  If they are the same, the user wins.  */
/* If they are different, the computer wins.  The outcome is then  */
/* returned to the calling exec.                                   */
/*********************************************************************/
 ARG input
 IF input = 'HEADS' THEN
   userthrow = 0                /* heads = 0  */
 ELSE
   userthrow = 1                /* tails = 1  */

 compthrow = RANDOM(0,1)        /* choose a random number between */
                                /* 0 and 1 */
 IF compthrow = userthrow THEN
   outcome = 'human'            /* user chose correctly           */
 ELSE
   outcome = 'machine'          /* user didn't choose correctly   */

 RETURN outcome
```

Writing a Function

A function is a series of instructions that an exec invokes to perform a specific task and return a value. As was described in Chapter 5, “Using Functions,” on page 57, a function may be built-in or user-written. An exec invokes a user-written function the same way it invokes a built-in function — by the function name immediately followed by parentheses with no blanks in between. The parentheses can contain up to 20 arguments or no arguments at all.

```
function(argument1, argument2,...)
```

or

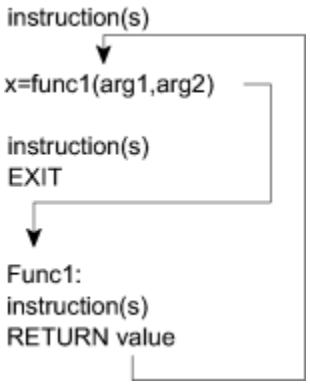
```
function()
```

Note:

A function requires a returned value because the function call generally appears in an expression.

```
x = function(arguments1, argument2,...)
```

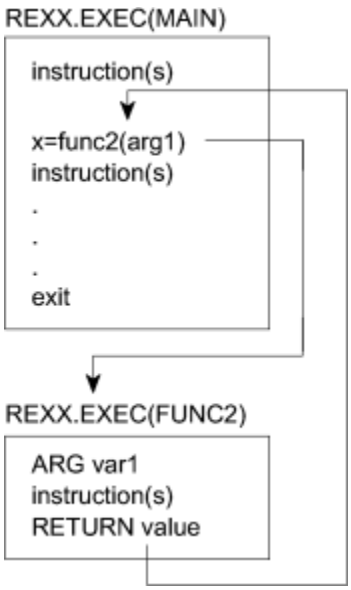
When the function ends, it may use the RETURN instruction to send back a value to replace the function call.



Functions may be **internal** and designated by a label, or **external** and designated by the data set member name that contains the function. The previous example illustrates an internal function named "func1".

> Because internal functions generally appear after the main part of the exec, when you have an internal function, it is important to end the main part of the exec with the EXIT instruction.

The following illustrates an external function named "func2".



To determine whether to make a function internal or external, you might consider factors, such as:

- Size of the function. Very large functions often are external, whereas small functions fit easily within the calling exec.

- How you want to pass information. It is quicker to pass information through variables in an internal function. This method is described in the next topic under "Passing Information by Using Variables" on page 74.
- Whether the function might be of value to more than one exec or user. If so, an external function is preferable.
- Performance. The language processor searches for an internal function before it searches for an external function. For the complete search order of functions, see "Search Order for Functions" on page 124.

Passing Information to a Function

When an exec and its internal function share the same variables, you can use commonly shared variables to pass information between caller and internal function. The function does not need to pass arguments within the parentheses that follow the function call. However, all functions, both internal and external, must return a value.

Passing Information by Using Variables

When an exec and its internal function share the same variables, the value of a variable is what was last assigned, regardless of whether the assignment was in the main part of the exec or in the function. In the following example, the value of `answer` is assigned in the function and displayed in the main part of the exec. The variables `number1`, `number2`, and `answer` are shared. In addition, the value of `answer` replaces the function call because `answer` follows the RETURN instruction.

Example of Passing Information in a Variable

```
/****************************** REXX ******************************/
/* This exec receives a calculated value from an internal         */
/* function and displays that value.                              */
/******************************************************************/

  number1 = 5
  number2 = 10
  SAY add()                                  /* Displays 15      */
  SAY answer                                 /* Also displays 15 */
  EXIT

  add:
  answer = number1 + number2
  RETURN answer
```

Using the same variables in an exec and its internal function can sometimes create problems. In the following example, the main part of the exec and the function use the same control variable, "i", for their DO loops. As a result, the DO loop repeats only once in the main exec because the function returns to the main exec with i = 6.

Example of a Problem Caused by Passing Information in a Variable

```
/****************************** REXX ******************************/
/* This exec uses an instruction in a DO loop to call an internal  */
/* function.  A problem occurs because the function also uses a DO */
/* loop with the same control variable as the main exec.  The DO   */
/* loop in the main exec repeats only once.                        */
/******************************************************************/

  number1 = 5
  number2 = 10
  DO i = 1 TO 5
    SAY add()                                    /* Displays 105 */
  END
  EXIT

  add:
  DO i = 1 TO 5
    answer = number1 + number2
    number1 = number2
    number2 = answer
  END
  RETURN answer
```

To avoid this kind of problem in an internal function, you can use:

- The PROCEDURE instruction as described in the next topic.
- Different variable names in a function.

Protecting Variables with the PROCEDURE Instruction

When you use the PROCEDURE instruction immediately following the function label, all variables used in the function become local to the function and are shielded from the main part of the exec. You can also use the PROCEDURE EXPOSE instruction to protect all but a few specified variables.

The following two examples show the differing results when a function uses the PROCEDURE instruction and when it doesn't.

Example Using the PROCEDURE Instruction

```
/****************************** REXX ******************************/
/* This exec uses a PROCEDURE instruction to protect the variables */
/* within its function.                                            */
/******************************************************************/
 number1 = 10
 SAY pass() number2                     /* Displays  7  NUMBER2 */
 EXIT

 pass: PROCEDURE
 number1 = 7
 number2 = 5
 RETURN number1
```

Example Without the PROCEDURE Instruction

```
/******************************* REXX ***************************/
/* This exec does not use a PROCEDURE instruction to protect the   */
/* variables within its function.                                  */
/*******************************************************************/
 number1 = 10
 SAY pass() number2                              /* displays 7  5  */
 EXIT

 pass:
 number1 = 7
 number2 = 5
 RETURN number1
```

Exposing Variables with PROCEDURE EXPOSE

To protect all but specific variables, use the EXPOSE option with the PROCEDURE instruction, followed by the variables that are to remain exposed to the function.

Example Using PROCEDURE EXPOSE

```
/******************************* REXX ***************************/
/* This exec uses a PROCEDURE instruction with the EXPOSE option to*/
/* expose one variable, number1, in its function.                  */
/*******************************************************************/
 number1 = 10
 SAY pass() number1                               /* displays 5  7 */
 EXIT

 pass: PROCEDURE EXPOSE number1
 number1 = 7
 number2 = 5
 RETURN number2
```

For more information about the PROCEDURE instruction, see *z/OS TSO/E REXX Reference*.

Passing Information by Using Arguments

A way to pass information to either internal or external functions is through arguments. You can pass up to 20 arguments separated by commas in a function call.

```
function(argument1,argument2,argument3,.........)
```

Using the ARG Instruction

The function can receive the arguments with the ARG instruction. Arguments are also separated by commas in the ARG instruction.

```
ARG arg1,arg2,arg3 .......
```

The names of the arguments on the function call and the ARG instruction do not have to be the same because information is not passed by argument name but by position. The first argument sent becomes the first argument received and so forth. You can also set up a template in the function call, which is then used in the corresponding ARG instruction. For information about parsing templates, see “Parsing Data” on page 83.

The following exec sends information to an internal function that computes the perimeter of a rectangle. The function returns a value in the variable `perim` that is specified after the RETURN instruction. The main exec uses the value in `perim` to replace the function call.

Example of an Internal Function

```
/********************************* REXX ***********************************/
/* This exec receives as arguments the length and width of a          */
/* rectangle and passes that information to an internal function      */
/* named perimeter. The function then calculates the perimeter of     */
/* the rectangle.                                                     */
/**********************************************************************/

PARSE ARG long wide
SAY 'The perimeter is' perimeter(long,wide) 'inches.'
EXIT

perimeter:
ARG length, width
perim = 2 * length + 2 * width
RETURN perim
```

Notice the positional relationships between `long` and `length`, and `wide` and `width`. Also notice that information is received from variable `perim` to replace the function call.

Using the ARG Built-in Function

Another way for a function to receive arguments is with the ARG built-in function. This built-in function returns the value of a particular argument specified by a number that represents the argument position.

For instance, in the previous example, instead of the ARG instruction,

```
ARG length, width
```

you can use the ARG function as follows:

```
length = ARG(1)             /* puts the first argument into length */
width = ARG(2)              /* puts the second argument into width */
```

More information about the ARG function appears in *z/OS TSO/E REXX Reference*.

Receiving Information from a Function

Although a function can receive up to 20 arguments in a function call, it can specify only one expression on the RETURN instruction. That expression can be a:

- Number

  ```
  RETURN 55
  ```

- One or more variables whose values are substituted or when no values were assigned, return their names

  ```
  RETURN value1 value2 value3
  ```

- Literal string

  ```
  RETURN 'Work complete.'
  ```

- Arithmetic, comparison, or logical expression whose value is substituted.

  ```
  RETURN 5 * number
  ```

Exercise - Writing a Function

Write a function named "AVG" that receives a list of numbers separated by blanks, and computes their average as a decimal number. The function is called as follows:

```
AVG(number1 number2 number3 ...)
```

Use the WORDS and WORD built-in functions. For more information about these built-in functions, see *z/OS TSO/E REXX Reference*.

ANSWER

Possible Solution

```
/****************************** REXX ******************************/
/* This function receives a list of numbers, adds them, computes  */
/* their average and returns the average to the calling exec.     */
/******************************************************************/

 ARG numlist           /* receive the numbers in a single variable */

 sum = 0               /* initialize sum to zero                   */

 DO n = 1 TO WORDS(numlist)   /* Repeat for as many times as there */
                              /* are numbers                       */

    number = WORD(numlist,n)  /* Word #n goes to number            */
    sum = sum + number        /* Sum increases by number           */
 END

 average = sum / WORDS(numlist)   /* Compute the average           */

 RETURN average
```

Summary of Subroutines and Functions

SUBROUTINES	FUNCTIONS
Invoked by using the CALL instruction followed by the subroutine name and optionally up to 20 arguments.	**Invoked** by specifying the function's name immediately followed by parentheses that optionally contain up to 20 arguments.
Can be **internal** or **external** • Internal – Can pass information by using common variables – Can protect variables with the PROCEDURE instruction – Can pass information by using arguments • External – Must pass information by using arguments – Can use the ARG instruction or the ARG built-in function to receive arguments	Can be **internal** or **external** • Internal – Can pass information by using common variables – Can protect variables with the PROCEDURE instruction – Can pass information by using arguments • External – Must pass information by using arguments – Can use the ARG instruction or the ARG built-in function to receive arguments
Uses the RETURN instruction to return to the caller.	Uses the RETURN instruction to return to the caller.
Might return a value to the caller.	*Must* return a value to the caller.

SUBROUTINES	FUNCTIONS
Returns a value by placing it into the REXX special variable RESULT.	**Returns a value** by replacing the function call with the value.

Chapter 7. Manipulating Data

This chapter describes how to use compound variables and stems, and shows various ways of parsing using templates.

Using Compound Variables and Stems

Sometimes it is useful to store groups of related data in such a way that the data can be easily retrieved. For example, a list of employee names can be stored in an array and retrieved by number. An array is an arrangement of elements in one or more dimensions, identified by a single name. You could have an array called `employee` that contains names as follows:

```
EMPLOYEE
   (1) Adams, Joe
   (2) Crandall, Amy
   (3) Devon, David
   (4) Garrison, Donna
   (5) Leone, Mary
   (6) Sebastian, Isaac
```

In some computer languages, you access an element in the array by the number of the element, such as, employee(1), which retrieves Adams, Joe. In REXX, you use compound variables.

What is a Compound Variable?

Compound variables are a way to create a one-dimensional array or a list of variables in REXX. Subscripts do not necessarily have to be numeric. A compound variable contains at least one period with characters on both sides of it. The following are examples of compound variables.

```
FRED.5
Array.Row.Col
employee.name.phone
```

The first variable in a compound variable always remains a symbol with no substitution. The remaining variables in a compound variable take on values previously assigned. If no value was previously assigned, the variable takes on the uppercase value of the variable name.

```
first = 'Fred'
last = 'Higgins'
employee = first.last
                            /* EMPLOYEE is assigned FIRST.Higgins */
SAY employee.first.middle.last
                          /* Displays EMPLOYEE.Fred.MIDDLE.Higgins */
```

You can use a DO loop to initialize a group of compound variables and set up an array.

```
DO i = 1 TO 6
   SAY 'Enter an employee name.'
   PARSE PULL employee.i
END
```

If you entered the same names used in the previous example of an array, you would have a group of compound variables as follows:

```
employee.1 = 'Adams, Joe'
employee.2 = 'Crandall, Amy'
employee.3 = 'Devon, David'
employee.4 = 'Garrison, Donna'
employee.5 = 'Leone, Mary'
employee.6 = 'Sebastian, Isaac'
```

When the names are in the group of compound variables, you can easily access a name by its number, or by a variable that represents its number.

```
name = 3
SAY employee.name                       /* Displays 'Devon, David' */
```

For more information about compound variables, see *z/OS TSO/E REXX Reference*.

Using Stems

When working with compound variables, it is often useful to initialize an entire collection of variables to the same value. You can do this easily with a **stem**. A stem is the first variable name and first period of the compound variable. Thus every compound variable begins with a stem. The following are stems:

```
FRED.
Array.
employee.
```

You can alter all the compound variables in an array through the stem. For example, to change all employee names to Nobody, issue the following assignment instruction:

```
employee. = 'Nobody'
```

As a result, all compound variables beginning with employee., whether or not they were previously assigned, return the value Nobody. Compound variables that are assigned after the stem assignment are not affected.

```
SAY employee.5                          /* Displays 'Nobody'       */
SAY employee.10                         /* Displays 'Nobody'       */
SAY employee.oldest                     /* Displays 'Nobody'       */

employee.new = 'Clark, Evans'
SAY employee.new                        /* Displays 'Clark, Evans' */
```

You can use stems with the EXECIO command when reading to and writing from a data set. For information about the EXECIO command, see “Using EXECIO to Process Information to and from Data Sets” on page 142. You can also use stems with the OUTTRAP external function when trapping command output. For information about OUTTRAP, see “Using the OUTTRAP Function” on page 115.

Exercises - Using Compound Variables and Stems

1. After these assignment instructions, what is displayed in the following SAY instructions?

```
a = 3                               /* assigns '3' to variable 'A'  */
b = 4                               /*         '4' to          'B'  */
c = 'last'                          /*      'last' to          'C'  */
a.b = 2                             /*         '2' to         'A.4' */
a.c = 5                             /*         '5' to      'A.last' */
x.a.b = 'cv3d'                      /*      'cv3d' to       'X.3.4' */
```

   a. SAY a
   b. SAY B
   c. SAY c
   d. SAY a.a
   e. SAY A.B
   f. SAY b.c
   g. SAY c.a
   h. SAY a.first
   i. SAY x.a.4

2. After these assignment instructions, what is displayed?

```
hole.1 = 'full'
hole. = 'empty'
hole.s = 'full'
```

   a. SAY hole.1

   b. SAY hole.s

   c. SAY hole.mouse

ANSWERS

1. a. 3

   b. 4

   c. last

   d. A.3

   e. 2

   f. B.last

   g. C.3

   h. A.FIRST

   i. cv3d

2. a. empty

   b. full

   c. empty

Parsing Data

Parsing in REXX is separating data into one or more variable names. An exec can parse an argument to break it up into smaller parts or parse a string to assign each word to a variable name. Parsing is also useful to format data into columns.

Instructions that Parse

There are several REXX instructions and variations of instructions that parse data.

PULL Instruction

In earlier chapters PULL was described as an instruction that reads input from the terminal and assigns it to one or more variables. If however, the data stack contains information, the PULL instruction takes information from the data stack; and when the data stack is empty, PULL takes information from the terminal. For information about the data stack, see Chapter 11, “Storing Information in the Data Stack,” on page 125. PULL changes character information to uppercase and assigns it to one or more variable names. When PULL is followed by more than one variable, it parses the information into the available variables.

```
SAY 'What is the quote for the day?'    /* user enters "Knowledge  */
                                        /*  is power."            */
PULL word1 word2 word3
                                  /* word1 contains 'KNOWLEDGE'  */
                                  /* word2 contains 'IS'         */
                                  /* word3 contains 'POWER.'     */
```

The PARSE PULL instruction assigns information, without altering it, to variable names.

```
SAY 'What is the quote for the day?'    /* user enters "Knowledge  */
                                        /*  is power."            */
```

```
PARSE PULL word1 word2 word3
                                    /* word1 contains 'Knowledge'  */
                                    /* word2 contains 'is'         */
                                    /* word3 contains 'power.'     */
```

PARSE UPPER PULL causes the same result as PULL in that it changes character information to uppercase before assigning it to one or more variables.

ARG Instruction

The ARG instruction takes information passed as arguments to an exec, function, or subroutine, and puts it into one or more variable names. Before character information is put into a variable name, ARG changes it to uppercase. When ARG is followed by more than one variable name, it parses the information into the available variable names. For example, if an exec named USERID.REXX.EXEC(QUOTE) can receive arguments, you can invoke the exec with the EXEC command and the three arguments as follows:

```
EXEC rexx.exec(quote) 'Knowledge is power.' exec
```

The exec receives the arguments with the ARG instruction as follows:

```
ARG word1 word2 word3
                                    /* word1 contains 'KNOWLEDGE'  */
                                    /* word2 contains 'IS'         */
                                    /* word3 contains 'POWER.'     */
```

The PARSE ARG instruction assigns information, without altering it, to variable names.

```
PARSE ARG word1 word2 word3
                                    /* word1 contains 'Knowledge'  */
                                    /* word2 contains 'is'         */
                                    /* word3 contains 'power.'     */
```

PARSE UPPER ARG causes the same result as ARG in that it changes character information to uppercase before assigning it to one or more variables.

PARSE VAR Instruction

The PARSE VAR instruction parses a specified variable into one or more variable names that follow it. If the variable contains character information, it is not changed to uppercase.

```
quote = 'Knowledge is power.'
PARSE VAR quote word1 word2 word3
                                    /* word1 contains 'Knowledge'  */
                                    /* word2 contains 'is'         */
                                    /* word3 contains 'power.'     */
```

The PARSE UPPER VAR instruction changes character information to uppercase before putting it into the variables.

```
quote = 'Knowledge is power.'
PARSE UPPER VAR quote word1 word2 word3
                                    /* word1 contains 'KNOWLEDGE'  */
                                    /* word2 contains 'IS'         */
                                    /* word3 contains 'POWER.'     */
```

For more information about parsing instructions, see *z/OS TSO/E REXX Reference*.

PARSE VALUE ... WITH Instruction

The PARSE VALUE ... WITH instruction parses a specified expression, such as a literal string, into one or more variable names that follow the WITH subkeyword. If the literal string contains character information, it is not changed to uppercase.

```
PARSE VALUE 'Knowledge is power.' WITH word1 word2 word3
                                    /* word1 contains 'Knowledge'  */
                                    /* word2 contains 'is'         */
                                    /* word3 contains 'power.'     */
```

The PARSE UPPER VALUE instruction changes character information to uppercase before assigning it to the variable names.

```
PARSE UPPER VALUE 'Knowledge is power.' WITH word1 word2 word3
                                    /* word1 contains 'KNOWLEDGE'  */
                                    /* word2 contains 'IS'         */
                                    /* word3 contains 'POWER.'     */
```

Ways of Parsing

Parsing separates data by comparing the data to a template (or pattern of variable names). Separators in a template can be a blank, string, variable, or number that represents column position.

Blank

The simplest template is a group of variable names separated by blanks. Each variable name gets one word of data in sequence except for the last, which gets the remainder of the data. The last variable name might then contain several words and possibly leading and trailing blanks.

```
PARSE VALUE 'Value    with Blanks.' WITH  pattern type
                            /* pattern contains 'Value'          */
                            /* type contains '   with Blanks.'   */
```

When there are more variables than data, the extra variables are set to null.

```
PARSE VALUE 'Value with Extra Variables.' WITH  data1 data2 data3 data4 data5
                              /* data1 contains 'Value'          */
                              /* data2 contains 'with'           */
                              /* data3 contains 'Extra'          */
                              /* data4 contains 'Variables.'     */
                              /* data5 contains ''               */
```

A period in a template acts as a place holder. The data that corresponds to the period is not assigned to a variable name. You can use a period as a "dummy variable" within a group of variables or at the end of a template to collect unwanted information.

```
PARSE VALUE 'Value with Periods in it.' WITH  pattern . type .
                              /* pattern contains 'Value'        */
                              /* type contains 'Periods'         */
              /* the periods replace the words "with" and "in it." */
```

String

You can use a string in a template to separate data as long as the data includes the string as well. The string becomes the point of separation and is not included as data.

```
phrase = 'To be, or not to be?'       /* phrase containing comma   */
PARSE VAR phrase part1 ',' part2      /* template containing comma */
                                      /*   as string separator     */
                              /* part1 contains 'To be'          */
                              /* part2 contains ' or not to be?' */
```

In this example, notice that the comma is not included with 'To be' because the comma is the string separator.

Variable

When you do not know in advance what string to specify as separator in a template, you can use a variable enclosed in parentheses. The variable value must be included in the data.

```
separator = ','
phrase = 'To be, or not to be?'
PARSE VAR phrase part1 (separator) part2
                              /* part1 contains 'To be'          */
                              /* part2 contains ' or not to be?' */
```

Again, in this example, notice that the comma is not included with 'To be' because the comma is the string separator.

Number

You can use numbers in a template to indicate the column at which to separate data. An unsigned integer indicates an absolute column position and a signed integer indicates a relative column position.

- Absolute column position

  An unsigned integer or an integer prefixed with an equal sign (=) in a template separates the data according to absolute column position. The first segment starts at column 1 and goes up to, but does not include, the information in the column number specified. The subsequent segments start at the column numbers specified.

```
quote = 'Ignorance is bliss.'
        ....+....1....+....2

PARSE VAR quote part1 5 part2
                            /* part1 contains 'Igno'             */
                            /* part2 contains 'rance is bliss.'  */
```

  This example could have also been coded as follows. Note the explicit use of the column 1 indicator prior to *part1* that was implied in the previous example and the use of the *=5 part2* to indicate the absolute position, column 5.

```
quote = 'Ignorance is bliss.'
        ....+....1....+....2

PARSE VAR quote 1 part1 =5 part2
                            /* part1 contains 'Igno'             */
                            /* part2 contains 'rance is bliss.'  */
```

  When a template has more than one number, and a number at the end of the template is lower than an earlier number, parse loops back to the beginning of the data.

```
quote = 'Ignorance is bliss.'
        ....+....1....+....2

PARSE VAR quote part1 5 part2 10 part3 1 part4
                          /* part1 contains 'Igno'               */
                          /* part2 contains 'rance'              */
                          /* part3 contains ' is bliss.'         */
                          /* part4 contains 'Ignorance is bliss.' */
```

  When each variable in a template has column numbers both before and after it, the two numbers indicate the beginning and the end of the data for the variable.

```
quote = 'Ignorance is bliss.'
        ....+....1....+....2

PARSE VAR quote 1 part1 10 11 part2 13 14 part3 19 1 part4 20
                          /* part1 contains 'Ignorance'          */
                          /* part2 contains 'is'                 */
                          /* part3 contains 'bliss'              */
                          /* part4 contains 'Ignorance is bliss.' */
```

- Relative column position

  A signed integer in a template separates the data according to relative column position, that is, a starting position relative to the starting position of the preceding part. A signed integer can be either positive (+) or negative (-) causing the part to be parsed to shift either to the right (with a +) or to the left (with a -). *part1* starts at column 1, the preceding 1 is not coded but implied. In the following example, therefore, the *+5 part2* causes part2 to start in column 1+5=6, the *+5 part3* causes part3 to start in column 6+5=11, and so on.

```
quote = 'Ignorance is bliss.'
        ....+....1....+....2

PARSE VAR quote part1 +5 part2 +5 part3 +5 part4
```

```
                                /* part1 contains 'Ignor'          */
                                /* part2 contains 'ance '          */
                                /* part3 contains 'is bl'          */
                                /* part4 contains 'iss.'           */
```

The use of the minus sign is similar to the use of the plus sign in that it is used to identify a relative position in the data string. The minus sign is used to "back up" (move to the left) in the data string. In the following example, therefore, the *part1* causes part1 to start in column 1 (implied), the *+10 part2* causes part2 to start in column 1+10=11, the *+3 part3* causes part3 to start in column 11+3=14, and the *-3 part4* causes part4 to start in column 14-3=11.

```
quote = 'Ignorance is bliss.'
         ....+....1....+....2

PARSE VAR quote part1 +10 part2 +3 part3 -3 part4
                                /* part1 contains 'Ignorance '     */
                                /* part2 contains 'is '            */
                                /* part3 contains 'bliss.'         */
                                /* part4 contains 'is bliss.'      */
```

- Variables

  You can define and use variables to provide further flexibility of a PARSE VAR instruction. Define the variable prior to the parse instruction, such as the `movex` variable in the following example. With the PARSE instruction, enclose the variable in parenthesis, in place of a number. This variable *must be an unsigned integer*. Therefore, use a sign outside the parenthesis to indicate how REXX is to interpret the unsigned integer. REXX substitutes the numeric value for the variable as follows:

```
quote = 'Ignorance is bliss.'
         ....+....1....+....2

movex = 3                       /* variable position               */
PARSE VAR quote part5 +10 part6 +3 part7 -(movex) part8
                                /* part5 contains 'Ignorance '     */
                                /* part6 contains 'is '            */
                                /* part7 contains 'bliss.'         */
                                /* part8 contains 'is bliss.'      */
```

  Note: The variable `movex` in the previous example must be an unsigned integer. Always code a sign prior to the parenthesis to indicate how the integer is to be interpreted. If you do not, the variable will be interpreted as a string separator. Valid signs are:

  – A plus sign (+) indicates column movement to the right
  – A minus sign (-) indicates column movement to the left
  – An equal sign (=) indicates an absolute column position.

For more information about parsing, see *z/OS TSO/E REXX Reference*.

Parsing Multiple Strings as Arguments

When passing arguments to a function or a subroutine, you can specify multiple strings to be parsed. Arguments are parsed with the ARG, PARSE ARG, and PARSE UPPER ARG instructions.

To pass multiple strings, separate each string with a comma. This comma is not a string separator as illustrated in“String” on page 85, although you can also use a string separator within an argument template.

The following example passes three arguments separated by commas to an internal subroutine. The first argument consists of two words "String One" that are parsed into three variable names. The third variable name is set to null because there is no third word. The second and third arguments are parsed entirely into variable names `string2` and `string3`.

```
CALL sub2 'String One', 'String Two', 'String Three'
:
EXIT

sub2:
PARSE ARG word1 word2 word3, string2, string3
```

```
                                /* word1 contains 'String'         */
                                /* word2 contains 'One'            */
                                /* word3 contains ''               */
                                /* string2 contains 'String Two'   */
                                /* string3 contains 'String Three' */
```

For more information about passing multiple arguments, see *z/OS TSO/E REXX Reference*.

Exercise - Practice with Parsing

What are the results of the following parsing examples?

1.
```
quote = 'Experience is the best teacher.'
PARSE VAR quote word1 word2 word3
```

   - a) word1 =
   - b) word2 =
   - c) word3 =

2.
```
quote = 'Experience is the best teacher.'
PARSE VAR quote word1 word2 word3 word4 word5 word6
```

   - a) word1 =
   - b) word2 =
   - c) word3 =
   - d) word4 =
   - e) word5 =
   - f) word6 =

3.
```
PARSE VALUE 'Experience is the best teacher.' WITH word1 word2 . . word3
```

   - a) word1 =
   - b) word2 =
   - c) word3 =

4.
```
PARSE VALUE 'Experience is the best teacher.' WITH v1 5 v2
            ....+....1....+....2....+....3.
```

   - a) v1 =
   - b) v2 =

5.
```
quote = 'Experience is the best teacher.'
        ....+....1....+....2....+....3.

PARSE VAR quote v1 v2 15 v3 3 v4
```

   - a) v1 =
   - b) v2 =
   - c) v3 =
   - d) v4 =

6.
```
quote = 'Experience is the best teacher.'
        ....+....1....+....2....+....3.

PARSE UPPER VAR quote 15 v1 +16 =12 v2 +2 1 v3 +10
```

   - a) v1 =
   - b) v2 =
   - c) v3 =

7.
```
quote = 'Experience is the best teacher.'
         ....+....1....+....2....+....3.

PARSE VAR quote 1 v1 +11 v2 +6 v3 -4 v4
```

   - a) v1 =
   - b) v2 =
   - c) v3 =
   - d) v4 =

8.
```
first = 7
quote = 'Experience is the best teacher.'
         ....+....1....+....2....+....3.

PARSE VAR quote 1 v1 =(first) v2 +6 v3
```

   - a) v1 =
   - b) v2 =
   - c) v3 =

9.
```
quote1 = 'Knowledge is power.'
quote2 = 'Ignorance is bliss.'
quote3 = 'Experience is the best teacher.'
CALL sub1 quote1, quote2, quote3
EXIT

sub1:
PARSE ARG word1 . . , word2 . . , word3 .
```

   - a) word1 =
   - b) word2 =
   - c) word3 =

ANSWERS

1. • a) word1 = Experience
   - b) word2 = is
   - c) word3 = the best teacher.
2. • a) word1 = Experience
   - b) word2 = is
   - c) word3 = the
   - d) word4 = best
   - e) word5 = teacher.
   - f) word6 = ''
3. • a) word1 = Experience
   - b) word2 = is
   - c) word3 = teacher.
4. • a) v1 = Expe
   - b) v2 = rience is the best teacher.
5. • a) v1 = Experience
   - b) v2 = is
   - c) v3 = the best teacher.
   - d) v4 = perience is the best teacher.
6. • a) v1 = THE BEST TEACHER
   - b) v2 = IS

- c) v3 = EXPERIENCE

7.
   - a) v1 = 'Experience '
   - b) v2 = 'is the'
   - c) v3 = ' best teacher.'
   - d) v4 = ' the best teacher.'
8.
   - a) v1 = 'Experi'
   - b) v2 = 'ence i'
   - c) v3 = 's the best teacher.'
9. a) word1 = Knowledge
   b) word2 = Ignorance
   c) word3 = Experience

Part 2. Using REXX

In addition to being a versatile general-purpose programming language, REXX can interact with TSO/E, MVS, APPC/MVS, and ISPF, which expands its capabilities. This part of the book is for programmers already familiar with the REXX language and experienced in TSO/E. The chapters in this part cover the following topics.

- Chapter 8, "Entering Commands from an Exec," on page 93 — A REXX exec can issue different types of host commands within the same exec.
- Chapter 9, "Diagnosing Problems Within an Exec," on page 105 — Several debugging options are available in an exec.
- Chapter 10, "Using TSO/E External Functions," on page 111 — TSO/E external functions are provided to interact with the system to do specific tasks.
- Chapter 11, "Storing Information in the Data Stack," on page 125 — The data stack is useful in I/O and other types of special processing.
- Chapter 12, "Processing Data and Input/Output Processing," on page 141 — You can process information to and from data sets by using the EXECIO command.
- Chapter 13, "Using REXX in TSO/E and Other MVS Address Spaces," on page 159 — You can run execs in other MVS address spaces besides TSO/E foreground and background.

Note: Although you can write a REXX exec to run in a non-TSO/E address space in MVS, the chapters and examples in this part, unless otherwise stated, assume the exec will run in a TSO/E address space. If you want to write execs that run outside of a TSO/E address space, keep in mind the following exceptions to information in this part of the book.

- An exec that runs outside of a TSO/E address space cannot include TSO/E commands, ISPF commands, or ISPF/PDF edit commands. An exec that runs outside of a TSO/E address space can include TSO/E commands if you use the TSO/E environment service (see note).
- An exec that runs outside of TSO/E cannot include most of the TSO/E external functions. For information about the functions you can use in TSO/E and non-TSO/E address spaces, see "Services Available to REXX Execs" on page 159.
- In TSO/E, several REXX instructions either display information on the terminal or retrieve information that the user enters at the terminal. In a non-TSO/E address space, these instructions get information from the input stream and write information to the output stream.
  - SAY — this instruction sends information to the output DD whose default is SYSTSPRT.
  - PULL — this instruction gets information from the input DD whose default is SYSTSIN.
  - TRACE — this instruction sends information to the output DD whose default is SYSTSPRT.
  - PARSE EXTERNAL — this instruction gets information from the input DD whose default is SYSTSIN.
- An exec that runs outside of TSO/E cannot interact with CLISTs.

Note: You can use the TSO/E environment service, IKJTSOEV, to create a TSO/E environment in a non-TSO/E address space. If you run a REXX exec in the TSO/E environment you created, the exec can contain TSO/E commands, external functions, and services that an exec running in a TSO/E address space can use. That is, the TSO host command environment (ADDRESS TSO) is available to the exec with some limitations. For more information about the TSO/E environment service, limitations on the environment it creates, and the different considerations for running REXX execs within the environment, see *z/OS TSO/E Programming Services*.

Chapter 8. Entering Commands from an Exec

This chapter describes how to issue TSO/E commands and other types of commands from a REXX exec.

Types of Commands

A REXX exec can issue many types of commands. The two main categories of commands are:

- TSO/E REXX commands - Commands provided with the TSO/E implementation of the language. These commands do REXX-related tasks in an exec, such as:
  - Control I/O processing of information to and from data sets (EXECIO)
  - Perform data stack services (MAKEBUF, DROPBUF, QBUF, QELEM, NEWSTACK, DELSTACK, QSTACK)
  - Change characteristics that control the execution of an exec (EXECUTIL and the immediate commands)
  - Check for the existence of a host command environment (SUBCOM).

  More information about these TSO/E REXX commands appears throughout the book where the related task is discussed
- Host commands - The commands recognized by the host environment in which an exec runs. A REXX exec can issue various types of host commands as discussed in the remainder of this chapter.

When an exec issues a command, the REXX special variable RC is set to the return code. An exec can use the return code to determine a course of action within the exec. Every time a command is issued, RC is set. Thus RC contains the return code from the most recently issued command.

Issuing TSO/E Commands from an Exec

Like a CLIST, a REXX exec can contain TSO/E commands to be executed when the exec runs. An exec can consist of nothing but TSO/E commands, such as an exec that sets up a user's terminal environment by allocating the appropriate libraries of data sets, or the exec can contain commands intermixed with REXX language instructions.

Using Quotations Marks in Commands

Generally, to differentiate commands from other types of instructions, enclose the command within single or double quotation marks. When issuing TSO/E commands in an exec, it is recommended that you enclose them in double quotation marks. If the command is not enclosed within quotation marks, it will be processed as an expression and might end in error. For example, a word immediately followed by a left parenthesis is processed by the language processor as a function call. Several TSO/E commands, one of which is ALLOCATE, require keywords followed by parentheses.

```
"ALLOC DA(NEW.DATA) LIKE(OLD.DATA) NEW"
```

If the ALLOCATE command in the example above was not enclosed in quotation marks, the parentheses would indicate to the language processor that DA and LIKE were function calls, and the command would end in an error.

Many TSO/E commands use single quotation marks within the command. For example, the EXEC command encloses an argument within single quotation marks, and other commands, such as ALLOCATE, require single quotation marks around fully-qualified data set names.

```
EXEC myrexx.exec(add) '25 78 33' exec

ALLOC DA('USERID.MYREXX.EXEC') F(SYSEXEC) SHR REUSE
```

As REXX instructions, these commands can be entirely enclosed in double quotation marks and still retain the single quotation marks for the specific information within the command. For this reason, it is recommended that, as a matter of course, you enclose TSO/E commands with double quotation marks.

```
"EXEC myrexx.exec(add) '25 78 33' exec"

"ALLOC DA('USERID.MYREXX.EXEC') F(SYSEXEC) SHR REUSE"
```

Remember that data set names beginning with your prefix (usually your user ID) can be specified without the prefix and without quotation marks.

```
"ALLOC DA(MYREXX.EXEC) F(SYSEXEC) SHR REUSE"
```

More about data sets names and when to enclose them in quotation marks is covered in the next topic.

Passing Data Set Names as Arguments

How you pass a data set name as an argument depends on the way you specify the data set name and whether you invoke the exec explicitly or implicitly.

Ways to specify the data set name are controlled by the TSO/E naming conventions, which define fully-qualified and non fully-qualified data sets. A fully-qualified data set name specifies all three qualifiers including the prefix and must appear within a set of quotation marks.

```
'userid.myrexx.exec'
```

A non fully-qualified data set name can eliminate the prefix and is not enclosed within quotation marks.

```
myrexx.exec
```

If you use the EXEC command to *explicitly* invoke an exec, the EXEC command processor requires a set of single quotation marks around the argument. When passing a non fully-qualified data set name as an argument, you need not add additional quotation marks. The following EXEC command is issued at the READY prompt and passes the data set name REXX.INPUT as an argument to the exec contained in MYREXX.EXEC(TEST2). Both data sets are specified as non fully-qualified data set names.

```
READY
 EXEC myrexx.exec(test2) 'rexx.input' exec
```

When passing a fully-qualified data set name as an argument with the EXEC command, you must include more than one set of quotation marks; one to indicate it is a fully-qualified data set and one to indicate it is the argument to be passed. Because TSO/E commands process two sets of single quotation marks as one and do not recognize double quotation marks as does the language processor, you must use three sets of single quotation marks. The following EXEC command passes USERID.REXX.INPUT as an argument expressed as a fully-qualified data set name.

```
READY
 EXEC myrexx.exec(test2) 'userid.rexx.input'' exec
```

When passing a non fully-qualified data set name as an argument while *implicitly* invoking the exec, you need no quotation marks.

```
READY
 test2 rexx.input
```

To pass a fully-qualified data set name as an argument while implicitly invoking an exec, enclose the data set name in a single set of quotation marks.

```
READY
 test2 'userid.rexx.input'
```

Using Variables in Commands

When a variable is used in a TSO/E command, the variable cannot be within quotation marks if its value is to be substituted. Only variables outside quotation marks are processed by the language processor. For example, the variable name is assigned the data set name MYREXX.EXEC. When name is used in a LISTDS command, it must remain outside the quotation marks placed around the command.

```
name = myrexx.exec
"LISTDS" name "STATUS"
```

When a variable represents a fully-qualified data set name, the name must be enclosed in two sets of quotation marks to ensure that one set of quotation marks remains as part of the value.

```
name = "'project.rel1.new'"
"LISTDS" name "STATUS"
```

Another way to ensure that quotation marks appear around a fully-qualified data set name when it appears as a variable is to include them as follows:

```
name = project.rel1.new
"LISTDS '"name"' STATUS"
```

Causing Interactive Commands to Prompt the User

If your TSO/E profile allows prompting, when you issue an interactive command without operands, you are prompted for operands. For example, when you issue the LISTDS command from READY, you are prompted for a data set name.

```
 READY
listds
 ENTER DATA SET NAME -
```

To have TSO/E commands prompt you when the commands are issued from within an exec, you can do one of two things:

- Run the exec explicitly with the EXEC command and use the PROMPT operand.

  ```
  EXEC mynew.exec(create) exec prompt
  ```

- Use the PROMPT function within the exec. Because PROMPT is a function, it is used as an expression within an instruction, such as an assignment instruction or a SAY instruction. To turn prompting on, write:

  ```
  saveprompt = PROMPT('ON')    /* saveprompt is set to the previous
                                                setting of PROMPT */
  ```

  To turn prompting off, write:

  ```
  x = PROMPT('OFF')    /* x is set to the previous setting of PROMPT */
  ```

  To find out the prompting status, write:

  ```
  SAY PROMPT()                      /* displays either "ON" or "OFF" */
  ```

  To reset prompting to a specific setting saved in variable *saveprompt*, write:

  ```
  x = prompt(saveprompt)
  ```

Note:

Neither of these options can override a NOPROMPT operand in your TSO/E profile. Your TSO/E profile controls prompting for all commands issued in your TSO/E session whether the commands are issued in line mode, in ISPF, in an exec, or in a CLIST. To display your profile, issue the PROFILE command. To change a profile from NOPROMPT to PROMPT, issue:

```
PROFILE PROMPT
```

Prompting by commands also depends on whether there are elements in the data stack. If the data stack contains an element, the user at the terminal is not prompted because the data stack element is used in response to the prompt. For more information about the data stack, see Chapter 11, “Storing Information in the Data Stack,” on page 125.

Invoking Another Exec as a Command

Previously, this book discussed how to invoke another exec as an external routine (Chapter 6, “Writing Subroutines and Functions,” on page 65). You can also invoke an exec from another exec explicitly with the EXEC command or implicitly by member name. Like an external routine, an exec invoked explicitly or implicitly can return a value to the caller with the RETURN or EXIT instruction. Unlike an external routine, which passes a value to the special variable RESULT, the invoked exec passes a value to the REXX special variable RC.

Invoking Another Exec with the EXEC Command

To explicitly invoke another exec from within an exec, issue the EXEC command as you would any other TSO/E command. The called exec should end with a RETURN or EXIT instruction, ensuring that control returns to the caller. The REXX special variable RC is set to the return code from the EXEC command. You can optionally return a value to the caller on the RETURN or EXIT instruction. When control passes back to the caller, the REXX special variable RC is set to the value of the expression returned on the RETURN or EXIT instruction.

For example, to invoke an exec named MYREXX.EXEC(CALC) and pass it an argument of four numbers, you could include the following instructions:

```
"EXEC myrexx.exec(calc) '24 55 12 38' exec"
SAY 'The result is' RC
```

'Calc' might contain the following instructions:

```
ARG number1 number2 number3 number4
answer = number1 * (number2 + number3) - number4
RETURN answer
```

You might want to invoke an exec with the EXEC command rather than as an external routine when the exec is not within the same PDS as the calling exec, or when the PDSs of the two execs are not allocated to either SYSEXEC or SYSPROC.

Invoking Another Exec Implicitly

To implicitly invoke another exec from within an exec, type the member name either with or without %. Because it is treated as a command, enclose the member name and the argument, if any, within quotation marks. As with any other implicitly invoked exec, the PDSs containing the calling exec and the called exec must be allocated to either SYSEXEC or SYSPROC. Remember that a % before the member name reduces the search time because fewer files are searched.

For example, to implicitly invoke an exec named MYREXX.EXEC(CALC) and send it an argument of four numbers, you could include the following instructions.

```
"%calc 24 55 12 38"
SAY 'The result is' RC
```

'Calc' might contain the following instructions:

```
ARG number1 number2 number3 number4
answer = number1 * (number2 + number3) - number4
RETURN answer
```

Issuing Other Types of Commands from an Exec

A REXX exec in TSO/E can issue TSO/E commands, APPC/MVS calls, MVS module invocations, ISPF commands, and ISPF/PDF EDIT commands. If you have TSO/E CONSOLE command authority and an extended MCS console session is active, you can also issue MVS system and subsystem commands in a REXX exec. Each type of invocation is associated with a different *host command environment.*

What is a Host Command Environment?

An environment for executing commands is called a host command environment. Before an exec runs, an active host command environment is defined to handle commands issued by the exec. When the language processor encounters a command, it passes the command to the host command environment for processing.

When a REXX exec runs on a host system, there is at least one default environment available for executing commands.

The default host command environments available in TSO/E REXX are as follows:

- **TSO** - the environment in which TSO/E commands and TSO/E REXX commands execute in the TSO/E address space.
- **MVS** - the environment in which TSO/E REXX commands execute in a non-TSO/E address space.
- **LINK** - an environment that links to modules on the same task level.
- **LINKMVS** - an environment that links to modules on the same task level. This environment allows you to pass multiple parameters to an invoked module, and allows the invoked module to update the parameters. The parameters you pass to the module include a length identifier.
- **LINKPGM** - an environment that links to modules on the same task level. This environment allows you to pass multiple parameters to an invoked module, and allows the invoked module to update the parameters. The parameters you pass to the module do not include a length identifier.
- **ATTACH** - an environment that attaches modules on a different task level.
- **ATTCHMVS** - an environment that attaches modules on a different task level. This environment allows you to pass multiple parameters to an invoked module, and allows the invoked module to update the parameters. The parameters you pass to the module include a length identifier.
- **ATTCHPGM** - an environment that attaches modules on a different task level. This environment allows you to pass multiple parameters to an invoked module, and allows the invoked module to update the parameters. The parameters you pass to the module do not include a length identifier.
- **ISPEXEC** - the environment in which ISPF commands execute.
- **ISREDIT** - the environment in which ISPF/PDF EDIT commands execute.
- **CONSOLE** - the environment in which MVS system and subsystem commands execute. To use the CONSOLE environment, you must have TSO/E CONSOLE command authority and an extended MCS console session must be active. You use the TSO/E CONSOLE command to activate an extended MCS console session. See *z/OS TSO/E System Programming Command Reference*, for more information about using the CONSOLE command.
- **CPICOMM** - the environment that allows you to invoke the SAA common programming interface (CPI) Communications calls.
- **LU62** - the environment that allows you to invoke the APPC/MVS calls that are based on the SNA LU 6.2 architecture. These calls are referred to as APPC/MVS calls throughout the book.

- **APPCMVS** - the environment that allows you to access MVS/APPC callable services related to server facilities and for the testing of transaction programs.

In a non-TSO/E environment, TSO/E REXX provides the following host command environments:

- MVS (the initial host command environment)
- LINK
- LINKMVS
- LINKPGM
- ATTACH
- ATTCHMVS
- ATTCHPGM
- CPICOMM
- LU62
- APPCMVS

From TSO/E READY mode, TSO/E REXX provides the following host command environments:

- TSO (the initial host command environment)
- MVS
- LINK
- LINKMVS
- LINKPGM
- ATTACH
- ATTCHMVS
- ATTCHPGM
- CONSOLE
- CPICOMM
- LU62
- APPCMVS

In ISPF, TSO/E REXX provides the following host command environments:

- TSO (the initial host command environment)
- MVS
- LINK
- LINKMVS
- LINKPGM
- ATTACH
- ATTCHMVS
- ATTCHPGM
- ISPEXEC
- ISREDIT
- CONSOLE
- CPICOMM
- LU62
- APPCMVS

Note: These lists of host command environments represent the defaults. Your installation may have added or deleted environments.

The default host command environment for execs running in TSO/E and ISPF is TSO. Thus all commands are sent to TSO/E for processing, unless the exec changes the host command environment.

When an exec runs in an MVS environment, TSO/E command processors and services are not available to it. For more information, see “Services Available to REXX Execs” on page 159. In an MVS host command environment, you can issue many of the TSO/E REXX commands, such as EXECIO, MAKEBUF, and NEWSTACK.

APPC/MVS Host Command Environments

The CPICOMM environment enables you to invoke the SAA CPI Communications calls and the LU62 and APPCMVS environments enable you to invoke APPC/MVS calls. You can write transaction programs in the REXX language, using the LU62, CPICOMM, or APPCMVS host command environments, to issue APPC calls to a partner transaction program. The CPICOMM host command environment allows transaction programs written in the REXX language to be ported across SAA environments. The LU62 host command environment allows you to use specific features of MVS in conversations with transaction programs on other systems. APPCMVS allows you to access APPC/MVS callable services related to server facilities and for the testing of transaction programs. Each of these host command environments enable REXX programs to communicate with other programs on the same MVS system, different MVS systems, or different operating systems in an SNA network.

The following APPC/MVS calls are supported under the APPCMVS host command environment:

- ATBCUC1 (Cleanup_TP(Unauthorized))
- ATBGTE2 (Get_Event)
- ATBPOR2 (Post_on_Receipt)
- ATBQAQ2 (Query_Allocate_Query)
- ATBRAL2 (Receive_Allocate)
- ATBRFA2 (Register_for_Allocate)
- ATBRJC2 (Reject_Conversation)
- ATBSAQ2 (Set_Allocate_Queue_Attributes)
- ATBSCA2 (Set_Conversation_Accounting_Information)
- ATBSTE2 (Set_Event_Notification)
- ATBTEA1 (Accept_Test)
- ATBTER1 (Register_Test)
- ATBTEU1 (Unregister_Test)
- ATBURA2 (Unregister_for_Allocates)
- ATBVERS (MVS_Version_Check)

The following SAA CPI Communications calls are supported under the CPICOMM host command environment:

- CMACCP (Accept_Conversation)
- CMALLC (Allocate)
- CMCFM (Confirm)
- CMCFMD (Confirmed)
- CMDEAL (Deallocate)
- CMECS (Extract_Conversation_State)
- CMECT (Extract_Conversation_Type)
- CMEMN (Extract_Mode_Name)
- CMEPLN (Extract_Partner_LU_Name)
- CMESL (Extract_Sync_Level)
- CMFLUS (Flush)

- CMINIT (Initialize_Conversation)
- CMPTR (Prepare_To_Receive)
- CMRCV (Receive)
- CMRTS (Request_To_Send)
- CMSCT (Set_Conversation_Type)
- CMSDT (Set_Deallocate_Type)
- CMSED (Set_Error_Direction)
- CMSEND (Send_Data)
- CMSERR (Send_Error)
- CMSF (Set_Fill)
- CMSLD (Set_Log_Data)
- CMSMN (Set_Mode_Name)
- CMSPLN (Set_Partner_LU_Name)
- CMSPTR (Set_Prepare_To_Receive_Type)
- CMSRC (Set_Return_Control)
- CMSRT (Set_Receive_Type)
- CMSSL (Set_Sync_Level)
- CMSST (Set_Send_Type)
- CMSTPN (Set_TP_Name)
- CMTRTS (Test_Request_To_Send_Received)

The SAA CPI Communications calls are described in *SAA Common Programming Interface Communications Reference*.

The following APPC/MVS calls are supported under the LU62 host command environment:

- ATBALC2 (Allocate)
- ATBALLC (Allocate)
- ATBCFM (Confirm)
- ATBCFMD (Confirmed)
- ATBDEAL (Deallocate)
- ATBFLUS (Flush)
- ATBGETA (Get_Attributes)
- ATBGETC (Get_Conversation)
- ATBGETP (Get_TP_Properties)
- ATBGETT (Get_Type)
- ATBGTA2 (Get_Attribute)
- ATBPTR (Prepare_To_Receive)
- ATBRCVI (Receive_Immediate)
- ATBRCVW (Receive_And_Wait)
- ATBRTS (Request_To_Send)
- ATBSEND (Send_Data)
- ATBSERR (Send_Error)

Note: The numeric suffix within the service name indicates the MVS release in which the service was introduced and thereby also available in all subsequent releases, as follows:

none
MVS SP4.2 service. For example, ATBGETA

1
MVS SP4.2.2 service. For example, ATBTEA1

2
MVS SP4.3 service. For example, ATBALC2

Therefore, your z/OS base control program (BCP) must be at least at the indicated level to take advantage of these services.

The parameters for these services and the requirements for using them in APPC/MVS transaction programs are described in *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Examples Using APPC/MVS Services

The following example illustrates the syntax for invoking an SAA CPI Communications call under the CPICOMM host command environment:

CPICOMM Example

```
/* REXX */
ADDRESS CPICOMM 'CMALLC conversation_id return_code'
if return_code = CM_OK then say 'OK!'
                       else say 'Why not?'
```

The following example illustrates the syntax for invoking an APPC/MVS call under the LU62 host command environment:

LU62 Example

```
/* REXX */
ADDRESS LU62 'ATBDEAL conversation_id deallocate_type',
                     'notify_type return_code'
```

Whenever you issue an SAA CPI Communications call or APPC/MVS call from a REXX program, the entire call must be enclosed in single or double quotes.

SAA CPI Communications calls and APPC/MVS calls can use pseudonyms rather than integer values. In the CPICOMM example, instead of comparing the variable return_code to an integer value of 0, the example compares return_code to the pseudonym value CM_OK. The integer value for CM_OK is 0. TSO/E provides two pseudonym files, one for the LU62 host command environment and one for the CPICOMM host command environment. These files define the pseudonyms and their integer values. The LU62 pseudonym file is REXAPPC1, and the CPICOMM pseudonym file is REXAPPC2. Both files are found in SYS1.SAMPLIB. You can include this information from the pseudonym files in your REXX execs.

For more information about host command environments and pseudonym files, refer to *z/OS TSO/E REXX Reference*.

Changing the Host Command Environment

You can change the host command environment either from the default or from whatever environment was previously established. To change the host command environment, use the ADDRESS instruction followed by the name of an environment.

The ADDRESS instruction has two forms: one affects all commands issued after the instruction, and one affects only a single command.

- **All commands**

When an ADDRESS instruction includes only the name of the host command environment, all commands issued afterward within that exec are processed as that environment's commands.

```
ADDRESS ispexec     /* Change the host command environment to ISPF */
"edit DATASET("dsname")"
```

The ADDRESS instruction affects only the host command environment of the exec that uses the instruction. When an exec calls an external routine, the host command environment reverts back to the default environment, regardless of the host command environment of the exec that called it. Upon return to the original exec, the host command environment that was previously established by an ADDRESS instruction is resumed.

- **Single command**

  When an ADDRESS instruction includes both the name of the host command environment and a command, only that command is affected. After the command is issued, the former host command environment becomes active again.

```
/* Issue one command from the ISPF host command environment        */
 ADDRESS ispexec "edit DATASET("dsname")"
/* Return to the default TSO host command environment              */
 "ALLOC DA("dsname") F(SYSEXEC) SHR REUSE"
```

Note: Keywords, such as DATASET, within an ISPF command must be in uppercase when used in a REXX instruction.

Determining the Active Host Command Environment

To find out what host command environment is currently active, use the ADDRESS built-in function.

```
x = ADDRESS()
```

In this example, x is set to the active host command environment, for example, TSO.

Checking if a Host Command Environment is Available

To check if a host command environment is available before trying to issue commands to that environment, issue the TSO/E REXX SUBCOM command followed by the name of the host command environment, such as ISPEXEC.

```
SUBCOM ISPEXEC
```

If the environment is present, the REXX special variable RC returns a 0. If the environment is not present, RC returns a 1. For example, when editing a data set, before trying to use ISPF/PDF edit, you can find out if ISPEXEC is available as follows:

```
ARG dsname
SUBCOM ISPEXEC
IF RC=0 THEN
  ADDRESS ISPEXEC "SELECT PGM(ISREDIT)"    /* select ISPF/PDF edit */
ELSE
  "EDIT" dsname                          /* use TSO/E line mode edit */
```

Examples Using the ADDRESS Instruction

ADDRESS Example 1

```
/****************************** REXX ******************************/
/* This exec must be run in ISPF.  It asks users if they know the  */
/* PF keys, and when the answer is a variation of "no", it displays*/
/* the panel with the PF key definitions.                          */
/*******************************************************************/
 SAY 'Do you know your PF keys?'

 PULL answer .
 IF answer = 'NO' | answer = 'N' THEN
    ADDRESS ispexec "display PANEL(ispopt3c)"
 ELSE
   SAY 'O.K. Never mind.'
```

ADDRESS Example 2

```
/****************************** REXX ******************************/
/* This exec must be run in ISPF.  It blanks out previous data set */
/* name information from the fields of an ISPF panel named newtool.*/
/* It then displays the panel to the user.                         */
/*******************************************************************/
 ADDRESS ispexec
 CALL blankem                       /* Call an internal subroutine */

 IF RC = 0 THEN
     "display PANEL(newtool)"
 ELSE
     "setmsg MSG(nt001)"                /* Send an error message. */

 EXIT

 blankem:
   'vget (ZUSER)'
   ntgroup = '
   nttype = '
   ntmem = '
 RETURN RC
```

ADDRESS Example 3

```
/****************************** REXX ******************************/
/* This exec must be run in ISPF.  It displays panel named newtool */
/* and gets the name of a data set from input fields named ntproj, */
/* ntgroup, nttype, and ntmem.  If no member name is specified (the*/
/* data set is sequential) the data set name does not include it.  */
/* If a member name is specified, the member is added to data set  */
/* name.  The fully-qualified data set name is then inserted into a*/
/* TRANSMIT command that includes single quotation marks and the   */
/* destination, which was received from an input field named ntdest*/
/*******************************************************************/
 ADDRESS ispexec
 "DISPLAY PANEL(newtool)"

 ADDRESS tso      /* re-establish the TSO host command environment */
 IF ntmem = '' THEN                      /* member name is blank */
   DO
     dsname = ntproj'.'ntgroup'.'nttype
     "TRANSMIT" ntdest "DA('"dsname"')"
   END
 ELSE
   DO
     dsname = ntproj'.'ntgroup'.'nttype'('ntmem')'
     "TRANSMIT" ntdest "DA('"dsname"')"
   END
```

ADDRESS Example 4

To link to or attach a logoff routine named MYLOGOFF and pass it the level of TSO/E installed, you can issue the following instructions from an exec.

```
ADDRESS LINK 'MYLOGOFF' SYSVAR(SYSTSOE)
```

or

```
ADDRESS ATTACH 'MYLOGOFF' SYSVAR(SYSTSOE)
```

Chapter 9. Diagnosing Problems Within an Exec

This chapter describes how to trace command output and other debugging techniques.

Debugging Execs

When you encounter an error in an exec, there are several ways to locate the error.

- The TRACE instruction displays how the language processor evaluates each operation. For information about using the TRACE instruction to evaluate expressions, see “Tracing Expressions with the TRACE Instruction” on page 35. For information about using the TRACE instruction to evaluate host commands, see the next section, “Tracing Commands with the TRACE Instruction” on page 105.
- Special variables, RC and SIGL, are set by the system to indicate:
  - The return code from a command - (RC)
  - The line number from which there was a transfer of control because of a function call, a SIGNAL instruction, or a CALL instruction - (SIGL)
- The TSO/E command EXECUTIL TS (Trace Start) and EXECUTIL TE (Trace End) control the interactive debug facility as do various options of the TRACE instruction. For more information about interactive debug, see “Tracing with the Interactive Debug Facility” on page 107.

Tracing Commands with the TRACE Instruction

The TRACE instruction has many options for various types of tracing, two of which are "commands" or "c" and "error" or "e".

TRACE C

When you specify "trace c" in an exec, any command that follows is traced before it is executed, then it is executed, and the return code from the command is displayed.

When an exec without "trace c" issues an incorrect TSO/E command, the exec ends with a TSO/E error message. For example, a LISTDS command specifies an incorrect data set name.

```
"LISTDS ?"
```

This example results in the following error message.

```
MISSING DATA SET NAME
INVALID KEYWORD, ?
***
```

If an exec includes "trace c" and again incorrectly issues the LISTDS command, the exec displays the line number and the command, executes it, and displays the error message and the return code from the command, as follows:

```
   3 *-* "LISTDS ?"
     >>>   "LISTDS ?"
MISSING DATA SET NAME
INVALID KEYWORD, ?
     +++ RC(12) +++
***
```

TRACE E

When you specify "trace e" in an exec, any host command that results in a nonzero return code is traced after it executes and the return code from the command is displayed.

If an exec includes "trace e" and again issues the previous incorrect LISTDS command, the exec displays error messages, the line number and the command, and the return code from the command, as follows:

```
  MISSING DATA SET NAME
  INVALID KEYWORD, ?
     3 *-* "LISTDS ?"
       +++ RC(12) +++
  ***
```

For more information about the TRACE instruction, see *z/OS TSO/E REXX Reference*.

Using REXX Special Variables RC and SIGL

As mentioned earlier, the REXX language has three special variables — RC, SIGL, and RESULT. These variables are set by the system during particular situations and can be used in an expression at any time. If the system did not set a value, a special variable displays its name, as do other variables in REXX. You can use two of these special variables, RC and SIGL, to help diagnose problems within execs.

RC

RC stands for return code and is set every time a command is issued. When a command ends without error, RC is usually set to 0. When a command ends in error, RC is set to whatever return code is assigned to that error.

For example, the previous incorrect LISTDS command is issued followed by the RC special variable in a SAY instruction.

```
"LISTDS ?"
SAY 'The return code from the command is' RC
```

This results in the following:

```
  MISSING DATA SET NAME
  INVALID KEYWORD, ?
  The return code from the command is 12
  ***
```

The RC variable can be especially useful in an IF instruction to determine which path an exec should take.

```
'ALLOC DA('dsname') F(SYSPROC) SHR REUSE'
IF RC \= 0 THEN
  CALL error1
ELSE NOP
```

Note: The value of RC is set by every command and might not remain the same for the duration of an exec. When using RC, make sure it contains the return code of the command you want to test.

SIGL

The SIGL special variable is used in connection with a transfer of control within an exec because of a function, or a SIGNAL or CALL instruction. When the language processor transfers control to another routine or another part of the exec, it sets the SIGL special variable to the line number from which the transfer occurred.

```
000001  /* REXX */
:
000005  CALL routine
:
000008
000009  routine:
000010  SAY 'We came here from line' SIGL      /* SIGL is set to 3 */
000011  RETURN
```

If the called routine itself calls another routine, SIGL is reset to the line number from which the most recent transfer occurred.

SIGL and the SIGNAL ON ERROR instruction can help determine what command caused an error and what the error was. When SIGNAL ON ERROR is included in an exec, any host command that returns a nonzero return code causes a transfer of control to a routine named "error". The error routine runs regardless of other actions that would normally take place, such as the display of error messages.

```
000001  /* REXX */
000002  SIGNAL ON ERROR
000003  "ALLOC DA(new.data) LIKE(old.data)"
:
000008  "LISTDS ?"
:
000011  EXIT
000012
000013  ERROR:
000014  SAY 'The return code from the command on line' SIGL 'is' RC
000015  /* Displays:
000016       The return code from the command on line 5 is 12      */
```

For more information about the SIGNAL instruction, see *z/OS TSO/E REXX Reference*.

Tracing with the Interactive Debug Facility

The interactive debug facility permits a user to interactively control the execution of an exec. A user can view the tracing of various types of instructions separated by pauses as the exec runs. During a pause, a user can continue to the next traced instruction, insert instructions, re-execute the previous instruction, and change or terminate interactive tracing.

Starting Interactive Tracing

You can start interactive tracing with either the ? option of the TRACE instruction or with the TSO/E EXECUTIL TS command. When interactive tracing is initiated with the TRACE instruction, interactive tracing is not carried over into external routines that are called but is resumed when the routines return to the traced exec. When interactive trace is initiated by the EXECUTIL TS command, interactive trace continues in all external routines called unless a routine specifically ends tracing.

? Option of the TRACE Instruction

One way to start interactive tracing is to include in an exec the TRACE instruction followed by a question mark and a trace option. For example, TRACE ?I (TRACE ?Intermediates). The question mark must precede the option with no blanks in between. Interactive tracing then begins for the exec but not for external routines the exec calls.

The following example includes a TRACE ?R (TRACE ?Results) instruction to interactively trace the result of each instruction.

Example of Interactive Trace

```
/********************************* REXX ***************************/
/* This exec receives as arguments the destination and the name    */
/* of a data set.  It then interactively traces the transmitting   */
/* that data set to the destination and the returning of a message */
/* that indicates whether the transmit was successful.             */
/*********************************************************************/
  TRACE ?R
  ARG dest dsname .
  "TRANSMIT" dest "DA("dsname")"
  IF RC = 0 THEN
    SAY 'Transmit successful.'
  ELSE
    SAY 'Return code from transmit was' RC
```

If the arguments passed to this exec were "node1.mel" and a sequential data set named "new.exec", the interactively traced results would be as follows with each segment separated by a pause.

```
     8 *-* ARG dest dsname .
       >>>   "NODE1.MEL"
       >>>   "NEW.EXEC"
       >.>   ""
+++ Interactive trace.    TRACE OFF to end debug, ENTER to continue. +++
```

```
     9 *-* "TRANSMIT" dest "DA("dsname")"
       >>>   "TRANSMIT NODE1.MEL DA(NEW.EXEC)"
0 message and 20 data records sent as 24 records to NODE1.MEL
Transmission occurred on 05/20/1989 at 14:40:11.
```

```
    10 *-* IF RC = 0
       >>>   "1"
```

```
       *-* THEN
    11 *-* SAY 'Transmit successful.'
       >>>   "Transmit successful."
Transmit successful.
```

EXECUTIL TS Command

Another way to start interactive tracing is to issue the EXECUTIL TS (trace start) command or cause an attention interrupt and type TS. The type of interactive tracing begun is equivalent to that of the TRACE ?R instruction, except that tracing continues through all routines invoked unless it is specifically ended. For information about ending interactive trace, see “Ending Interactive Trace” on page 109.

The EXECUTIL TS command can be issued from several environments; it affects only the current exec and the execs it invokes. Like other TSO/E commands, EXECUTIL TS can be issued from within an exec, from READY mode, and from an ISPF panel.

- From Within an Exec

  You can issue the EXECUTIL TS command from within an exec.

  ```
  ⋮
  "EXECUTIL TS"
  ⋮
  EXIT
  ```

  The exec is then interactively traced from the point in the exec at which the command was issued. Any other execs that the exec invokes are also interactively traced.

  You can also issue EXECUTIL TS from within a CLIST to initiate tracing in execs that the CLIST invokes.

- From READY Mode

  You can issue the command from READY mode.

  ```
  READY
  executil ts
  ```

  The next exec invoked from READY mode is then interactively traced. If that exec invokes another exec, the invoked exec is also interactively traced.

- From an ISPF Panel

  You can also issue EXECUTIL TS from the ISPF COMMAND option or from the command line of an ISPF panel.

  ```
  ----------------------------  TSO COMMAND PROCESSOR  -------------------------
  ENTER TSO COMMAND OR CLIST BELOW:

  ===> executil ts
  ```

  ```
  ----------------------------  ALLOCATE NEW DATA SET  --------------------------
  COMMAND ===> tso executil ts
  ```

The next exec invoked from ISPF is then interactively traced. If that exec calls another exec, the called exec is also interactively traced. If you are in split screen mode in ISPF, an exec run from the opposite screen is not interactively traced because each side of a split screen is a different environment.

To begin interactive trace after pressing the attention interrupt key, sometimes labeled PA1, enter TS (trace start) after the message that the attention facility displays.

```
ENTER HI TO END, A NULL LINE TO CONTINUE, OR AN IMMEDIATE COMMAND+
 ts
```

The type of tracing is the same as that initiated by issuing the EXECUTIL TS command.

Options Within Interactive Trace

When you are operating in the interactive debug facility, you have several options during the pauses that occur between each traced instruction. You can:

- Continue tracing by entering a null line
- Type one or more additional instructions to be processed before the next instruction is traced
- Enter an equal sign (=) to re-execute the last instruction traced
- End interactive tracing as described in the next topic.

Continuing Interactive Tracing

To continue tracing through an exec, simply press the Enter key to enter a null line during the pause between each traced instruction. The next traced instruction then appears on the screen. Repeatedly pressing the Enter key, therefore, takes you from pause point to pause point until the exec ends.

Typing Additional Instructions to be Processed

During the pause between traced instructions, you can enter one or more instructions that are processed immediately. The instruction can be any type of REXX instruction including a command or invocation to another exec or CLIST. You can also enter a TRACE instruction, which alters the type of tracing. After you enter the instruction, you might need to press the Enter key again to resume tracing.

```
TRACE L    /* Makes the language processor pause at labels only */
```

The instruction can also change the course of an exec, such as by assigning a different value to a variable to force the execution of a particular branch in an IF THEN ELSE instruction. In the following example, RC is set by a previous command.

```
IF RC = 0 THEN
  DO
    instruction1
    instruction2
  END
ELSE
  instructionA
```

If during normal execution, the command ends with other than a 0 return code, the ELSE path will be taken. To force taking the IF THEN path during interactive trace, you can change the value of RC as follows during a pause.

```
RC = 0
```

Re-executing the Last Instruction Traced

You can re-execute the last instruction traced by entering an equal sign (=) with no blanks. The language processor then re-executes the previously traced instruction with values possibly modified by instructions, if any were entered during the pause.

Ending Interactive Trace

You can end interactive tracing in one of the following ways:

- Use the TRACE OFF instruction.
- Let the exec run until it ends.
- Use the TRACE ? instruction.
- Issue the EXECUTIL TE command.

TRACE OFF

The TRACE OFF instruction ends tracing as stated in the message displayed at the beginning of interactive trace.

```
+++ Interactive trace.    TRACE OFF to end debug, ENTER to continue. +++
```

You can enter the TRACE OFF instruction only during a pause while interactively tracing an exec.

End the Exec

Interactive tracing automatically ends when the exec that initiated tracing ends. You can cause the exec to end prematurely by entering the EXIT instruction during a pause. The EXIT instruction causes the exec and interactive tracing both to end.

TRACE ?

The question mark prefix before a TRACE option can end interactive tracing as well as begin it. The question mark reverses the previous setting for interactive tracing.

While interactively tracing an exec, you can also enter the TRACE ? instruction with any operand to discontinue the interactive debug facility but continue the type of tracing specified by the operand.

EXECUTIL TE

The EXECUTIL TE (Trace End) command ends interactive tracing when issued from within an exec or when entered during a pause while interactively tracing an exec.

For more information about the EXECUTIL command, see *z/OS TSO/E REXX Reference*.

Chapter 10. Using TSO/E External Functions

This chapter shows how to use TSO/E external functions and describes function packages.

TSO/E External Functions

In addition to the built-in functions, TSO/E provides external functions that you can use to do specific tasks. Some of these functions perform the same services as control variables in the CLIST language.

The TSO/E external functions are:

- GETMSG - returns in variables a system message issued during an extended MCS console session. It also returns in variables associated information about the message. The function call is replaced by a function code that indicates whether the call was successful.
- LISTDSI - returns in variables the data set attributes of a specified data set. The function call is replaced by a function code that indicates whether the call was successful.
- MSG - controls the display of TSO/E messages. The function returns the previous setting of MSG.
- MVSVAR - uses specific argument values to return information about MVS, TSO/E, and the current session.
- OUTTRAP - traps lines of TSO/E command output into a specified series of variables. The function call returns the variable name specified.
- PROMPT - sets the prompt option on or off for TSO/E interactive commands. The function returns the previous setting of prompt.
- SETLANG - retrieves and optionally changes the language in which REXX messages are displayed. The function returns the previous language setting.
- STORAGE - retrieves and optionally changes the value in a storage address.
- SYSCPUS - returns in a stem variable information about all CPUs that are on-line.
- SYSDSN - returns OK if the specified data set exists; otherwise, it returns an appropriate error message.
- SYSVAR - uses specific argument values to return information about the user, terminal, language, exec, system, and console session.

Following are brief explanations about how to use the TSO/E external functions. For complete information, see *z/OS TSO/E REXX Reference*.

Using the GETMSG Function

The GETMSG function retrieves a system message issued during an extended MCS console session. The retrieved message can be either a response to a command or any other system message, depending on the message type you specify.

The message text and associated information are stored in variables, which can be displayed or used within the REXX exec. The function call is replaced by a function code that indicates whether the call was successful. See *z/OS TSO/E REXX Reference* for more information about the syntax, function codes, and variables for GETMSG. You must have CONSOLE command authority to use the GETMSG function. Before you issue GETMSG, you must:

- Use the TSO/E CONSPROF command to specify the types of messages that are not to be displayed at the terminal. The CONSPROF command can be used before you activate a console session and during a console session if values need to be changed.
- Use the TSO/E CONSOLE command to activate an extended MCS console session.

The GETMSG function can be used only in REXX execs that run in the TSO/E address space.

Using the LISTDSI Function

You can use the LISTDSI (list data set information) function to retrieve detailed information about a data set's attributes. The attribute information is stored in variables, which can be displayed or used within instructions. The function call is replaced by a function code that indicates whether the call was successful.

The LISTDSI function can be used only in REXX execs that run in the TSO/E address space.

To retrieve the attribute information, include the data set name within parentheses after LISTDSI. When you specify a fully-qualified data set name, be sure to enclose it in two sets of quotation marks as follows; one set to define it as a literal string to REXX and the other to indicate a fully-qualified data set to TSO/E.

```
x = LISTDSI("'proj5.rexx.exec'")    /* x is set to a function code */
```

or

```
x = LISTDSI('proj5.rexx.exec'')  /* x is set to a function code */
```

When you specify a data set name that begins with your prefix (usually your user ID), you can use one set of quotation marks to define it as a literal string or no quotation marks. TSO/E adds your prefix to the data set name whether or not it is enclosed within a set of quotation marks.

```
x = LISTDSI('my.data')              /* x is set to a function code */
```

```
x = LISTDSI(my.data)                /* x is set to a function code */
```

When you specify a variable that was previously set to a data set name, do *not* enclose the variable in quotation marks. Quotation marks would prevent the data set name from being substituted for the variable name.

```
variable = 'my.data'
x = LISTDSI(variable)
```

You cannot use LISTDSI with the *filename* parameter if the filename is allocated to a data set

- which exists more than once with the same name on different volumes, and
- which is already in use

because in this case the system may not retrieve information for the data set you wanted. After LISTDSI executes, the function call is replaced by one of the following function codes:

Function Code	Meaning
0	Normal completion
4	Some data set information is unavailable. All data set information other than directory information can be considered valid.
16	Severe error occurred. None of the variables containing information about the data set can be considered valid.

The following variables are set to the attributes of the data set specified.

Variable	Contents
SYSDSNAME	Data set name
SYSVOLUME	Volume serial ID
SYSUNIT	Device unit on which volume resides
SYSDSORG	Data set organization: PS, PSU, DA, DAU, IS, ISU, PO, POU, VS

Variable	Contents
SYSRECFM	Record format; three-character combination of the following: U, F, V, T, B, S, A, M
SYSLRECL	Logical record length
SYSBLKSIZE	Block size
SYSKEYLEN	Key length
SYSALLOC	Allocation, in space units
SYSUSED	Allocation used, in space units
SYSUSEDPAGES	Used space of a partitioned data set extended (PDSE) in 4K pages.
SYSPRIMARY	Primary allocation in space units
SYSSECONDS	Secondary allocation in space units
SYSUNITS	Space units: CYLINDER, TRACK, BLOCK
SYSEXTENTS	Number of extents allocated
SYSCREATE	Creation date: Year/day format, for example: 1985/102
SYSREFDATE	Last referenced date: Year/day format, for example: 1985/107 (Specifying DIRECTORY causes the date to be updated.)
SYSEXDATE	Expiration date: Year/day format, for example: 1985/365
SYSPASSWORD	Password indication: NONE, READ, WRITE
SYSRACFA	RACF indication: NONE, GENERIC, DISCRETE
SYSUPDATED	Change indicator: YES, NO
SYSTRKSCYL	Tracks per cylinder for the unit identified in the SYSUNIT variable
SYSBLKSTRK	Blocks per track for the unit identified in the SYSUNIT variable
SYSADIRBLK	Directory blocks allocated - returned only for partitioned data sets when DIRECTORY is specified
SYSUDIRBLK	Directory blocks used - returned only for partitioned data sets when DIRECTORY is specified
SYSMEMBERS	Number of members - returned only for partitioned data sets when DIRECTORY is specified
SYSREASON	LISTDSI reason code
SYSMSGLVL1	First-level message if an error occurred
SYSMSGLVL2	Second-level message if an error occurred
SYSDSSMS	Information about the type of a data set provided by DFSMS/MVS.
SYSDATACLASS	SMS data class name
SYSSTORCLASS	SMS storage class name

Variable	Contents
SYSMGMTCLASS	SMS management class name

Using the MSG Function

The MSG function can control the display of TSO/E messages. When the MSG function is not used, both error and non-error messages are displayed as an exec runs. These messages can interfere with output, especially when the exec's output is a user interface, such as a panel.

The MSG function can be used only in REXX execs that run in the TSO/E address space.

To prevent the display of TSO/E messages as an exec runs, use the MSG function followed by the word "OFF" enclosed within parentheses.

```
status = MSG('OFF')    /* status is set to the previous setting of */
                       /*  MSG and sets the current setting to OFF */
```

To resume the display of TSO/E messages, substitute the word "ON" for "OFF".

To find out if messages will be displayed, issue the MSG function followed by empty parentheses.

```
status = MSG()       /* status is set to ON or OFF  */
```

Using the MVSVAR Function

The MVSVAR function retrieves information about MVS, TSO/E, and the current session, such as the symbolic name of the MVS system, or the security label of the TSO/E session. The information retrieved depends on the argument specified.

To retrieve the information, use the MVSVAR function immediately followed by an argument value enclosed in parentheses. For example, to find out the APPC/MVS logical unit (LU) name, use the MVSVAR function with the argument SYSAPPCLU.

```
appclu = MVSVAR('SYSAPPCLU')
```

The MVSVAR function is available **in any MVS address space.** Compare this to the SYSVAR function which also retrieves system information but can only be used in REXX execs that run in the TSO/E address space.

Many of the MVSVAR arguments retrieve the same information as do CLIST control variables.

The following table lists the items of information that are available for retrieval by MVSVAR.

Argument Value	Description
SYSAPPCLU	the APPC/MVS logical unit (LU) name
SYSDFP	the level of MVS/Data Facility Product (MVS/DFP)
SYSMVS	the level of the base control program (BCP) component of z/OS
SYSNAME	the name of the system your REXX exec is running on, as specified in the SYSNAME statement in SYS1.PARMLIB member IEASYSxx
SYSSECLAB	the security label (SECLABEL) name of the TSO/E session
SYSSMFID	identification of the system on which System Management Facilities (SMF) is active
SYSSMS	indicator whether DFSMS/MVS is available to your REXX exec
SYSCLONE	MVS system symbol representing its system name

Argument Value	Description
SYSPLEX	the MVS sysplex name as found in the COUPLExx or LOADxx member of SYS1.PARMLIB
SYMDEF	symbolic variables of your MVS system

Using the OUTTRAP Function

The OUTTRAP function puts lines of command output into a series of numbered variables, each with the same prefix. These variables save the command output and allow an exec to process the output. Specify the variable name in parentheses following the function call.

```
SAY 'The OUTTRAP variable name is' OUTTRAP('var')
/* Displays the variable name in which command output is trapped.  */
```

In this example, the variable `var` becomes the prefix for the numbered series of variables. Var1, var2, var3, and so on, receive a line of output each. If you do not set a limit to the number of output lines, the numbering of variables continues as long as there is output. Output from the most recent command is placed after the previous command's output. The total number of lines trapped is stored in var0.

```
x = OUTTRAP('var')
"LISTC"
SAY 'The number of lines trapped is' var0
```

To limit the number of lines of output saved, you can specify a limit, for example 5, after the variable name.

```
x = OUTTRAP('var',5)
```

This results in up to 5 lines of command output stored in var1, var2, var3, var4, var5; and var0 contains the number 5. Subsequent lines of command output are not saved.

The following example traps output from two commands and then displays the member names from a partitioned data set named MYNEW.EXEC. The stem variable includes a period, which causes the lines of output to be stored in a series of compound variables. For more information about compound variables, see “Using Compound Variables and Stems” on page 81.

```
x = OUTTRAP('var.')
"LISTC"
SAY 'The number of lines trapped is' var.0    /* could display 205 */
lines = var.0 + 1
"LISTDS mynew.exec MEMBERS"
SAY 'The number of lines trapped is' var.0    /* could display 210 */
DO i = lines TO var.0
  SAY var.i                                  /* displays 5 members */
END
```

To turn trapping off, reissue the OUTTRAP function with the word "OFF".

```
x = OUTTRAP('OFF')       /* turns trapping OFF */
```

The OUTTRAP function can be used only in REXX execs that run in the TSO/E address space.

The OUTTRAP function does not trap all lines of command output from all TSO/E commands. For more information, see *z/OS TSO/E REXX Reference*.

Using the PROMPT Function

When your profile allows for prompting, the PROMPT function can set the prompting option on or off for interactive TSO/E commands, or it can return the type of prompting previously set. When prompting is on, execs can issue TSO/E commands that prompt the user for missing operands.

The PROMPT function can be used only in REXX execs that run in the TSO/E address space.

To set the prompting option on, use the PROMPT function followed by the word "ON" enclosed within parentheses.

```
x = PROMPT('ON')     /* x is set to the previous setting of prompt */
                     /*      and sets the current setting to ON    */
```

To set prompting off, substitute the word "OFF" for "ON".

To find out if prompting is available for TSO/E interactive commands, use the PROMPT function followed by empty parentheses.

```
x = PROMPT()         /* x is set to ON or OFF */
```

The PROMPT function overrides the NOPROMPT operand of the EXEC command, but it cannot override a NOPROMPT operand in your TSO/E profile. To display your profile, issue the PROFILE command. To change a profile from NOPROMPT to PROMPT, issue:

```
PROFILE PROMPT
```

Using the SETLANG Function

You can use the SETLANG function to determine the language in which REXX messages are currently being displayed and to optionally change the language. If you do not specify an argument, SETLANG returns a 3-character code that indicates the language in which REXX messages are currently being displayed. Table 1 on page 116 shows the language codes that replace the function call and the corresponding language for each code.

You can optionally specify one of the language codes on the function call to change the language in which REXX messages are displayed. In this case, SETLANG sets the language to the code specified and returns the language code of the previous language setting. The language codes you can specify on SETLANG depend on the language features that are installed on your system.

Table 1: Language Codes for SETLANG Function That Replace the Function Call

Language Code	Language
CHS	Simplified Chinese
CHT	Traditional Chinese
DAN	Danish
DEU	German
ENP	US English-all uppercase
ENU	US English-mixed case (uppercase and lowercase)
ESP	Spanish
FRA	French
JPN	Japanese
KOR	Korean
PTB	Brazilian Portuguese

To find out the language in which REXX messages are currently being displayed, issue the SETLANG function followed by empty parentheses:

```
curlang=SETLANG()        /* curlang is set to the 3-character     */
                         /* code of the current language setting. */
```

To set the language to Japanese for subsequent REXX message displays, issue the SETLANG function followed by the 3-character code, JPN, enclosed within parentheses:

```
oldlang=SETLANG(JPN)     /* oldlang is set to the previous        */
                         /* language setting.                     */
                         /* The current setting is set to JPN.    */
```

The SETLANG function can be used in REXX execs that run in any MVS address space.

Using the STORAGE Function

You can use the STORAGE function to retrieve data from a particular address in storage. You can also use the STORAGE function to place data into a particular address in storage.

The STORAGE function can be used in REXX execs that run in any MVS address space.

Using the SYSCPUS Function

The SYSCPUS function places, in a stem variable, information about those CPUs that are on-line.

The SYSCPUS function runs **in any MVS address space.**

Example:

Consider a system with two on-line CPUs. Their serial numbers are FF0000149221 and FF1000149221. Assuming you issue the following sequence of statements

```
/* REXX */
x = SYSCPUS('cpus.')
SAY '0, if function performed okay: ' x
SAY 'Number of on-line CPUs is ' cpus.0
DO i = 1 TO CPUS.0
  SAY 'CPU' i ' has CPU info ' cpus.i
END
```

you get the following output:

```
0, if function performed okay: 0
Number of on-line CPUs is 2
CPU 1 has CPU info FF0000149221
CPU 2 has CPU info FF1000149221
                  /* ↑     ↑                          */
                  /* |     4 digits = model number   */
                  /* 6 digits       = CPU ID         */
```

Using the SYSDSN Function

The SYSDSN function determines if a specified data set is available for your use. If the data set is available for your use, it returns "OK".

```
available = SYSDSN('myrexx.exec')
 /* available could be set to "OK"  */
```

When a data set is not correct as specified or when a data set is not available, the SYSDSN function returns one of the following messages:

- `MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED`
- `MEMBER NOT FOUND`

- DATASET NOT FOUND
- ERROR PROCESSING REQUESTED DATASET
- PROTECTED DATASET
- VOLUME NOT ON SYSTEM
- UNAVAILABLE DATASET
- INVALID DATASET NAME, *data-set-name:*
- MISSING DATASET NAME

After a data set is available for use, you may find it useful to get more detailed information. For example, if you later need to invoke a service that requires a specific data set organization, then use the LISTDSI function. For a description of the LISTDSI function, see "Using the LISTDSI Function" on page 112.

When you specify a fully-qualified data set, be sure to use two sets of quotation marks as follows; one set to define a literal string to REXX and the other set to indicate a fully-qualified data set to TSO/E.

```
x = SYSDSN("'proj5.rexx.exec'")
```

or

```
x = SYSDSN('proj5.rexx.exec'')
```

When you specify a data set that is not fully-qualified and begins with your prefix (usually your user ID), you can use one set of quotation marks or none at all. TSO/E adds your prefix to the data set name whether or not it is enclosed within a set of quotation marks.

```
x = SYSDSN('myrexx.exec')
```

or

```
x = SYSDSN(myrexx.exec)
```

When you specify a variable that was previously set to a data set name, do *not* enclose the variable in quotation marks. Quotation marks would prevent the data set name from being substituted for the variable name.

```
variable = 'myrexx.exec'
x = SYSDSN(variable)
```

The following example uses the SYSDSN function together with the LISTDSI function to test whether a data set exists and whether it is a partitioned data set:

```
DO FOREVER
  SAY 'Enter a Data Set Name'
  PARSE UPPER PULL dsname
  IF SYSDSN(dsname) ¬= 'OK' THEN ITERATE
  FC = LISTDSI(dsname)
  IF SYSDSORG ¬= 'PO' THEN ITERATE
  SAY 'Okay: ' dsname 'is ' SYSDSORG
  LEAVE
END
```

The SYSDSN function can be used only in REXX execs that run in the TSO/E address space.

Using the SYSVAR Function

The SYSVAR function retrieves information about MVS, TSO/E, and the current session, such as levels of software available, your logon procedure, and your user ID. The information retrieved depends on the argument specified.

To retrieve the information, use the SYSVAR function immediately followed by an argument value enclosed in parentheses. For example, to find out the name of the logon procedure of your current session, use the SYSVAR function with the argument SYSPROC.

```
proc = SYSVAR(sysproc)
```

The SYSVAR function can be used only in REXX execs that run in the TSO/E address space.

Many of the SYSVAR arguments retrieve the same information as do CLIST control variables. The following tables divide the argument values into categories pertaining to user, terminal, language, exec, system, and console session information.

User Information

Argument Value	Description
SYSPREF	Prefix as defined in user profile
SYSPROC	SYSPROC returns the current procedure name (either the LOGON procedure name, the Started Task procedure name, or 'INIT' for a batch job). For more information, see *z/OS TSO/E REXX Reference*.
SYSUID	User ID of current session

Terminal Information

Argument Value	Description
SYSLTERM	Number of lines available on screen
SYSWTERM	Width of screen

Language Information

Argument Value	Description
SYSPLANG	Primary language for translated messages
SYSSLANG	Secondary language for translated messages
SYSDTERM	Whether DBCS is supported for this terminal
SYSKTERM	Whether Katakana is supported for this terminal

Exec Information

Argument Value	Description
SYSENV	Whether exec is running in foreground or background
SYSICMD	Name by which exec was implicitly invoked
SYSISPF	Whether ISPF is available for exec
SYSNEST	Whether exec was invoked from another exec or CLIST. Invocation could be implicit or explicit.
SYSPCMD	Name of most recently executed command
SYSSCMD	Name of most recently executed subcommand

System Information

Argument Value	Description
SYSCPU	Number of CPU seconds used during session in the form: *seconds.hundredths of seconds*
SYSHSM	Level of Data Facility Hierarchical Storage Manager (DFHSM) installed
SYSJES	Name and level of JES installed
SYSLRACF	Level of RACF installed
SYSRACF	Whether RACF is available
SYSNODE	Network node name of the installation's JES
SYSSRV	Number of system resource manager (SRM) service units used during session
SYSTERMID	Terminal ID of the terminal where the REXX exec was started
SYSTSOE	Level of TSO/E installed in the form: *version release modification_number*

Console Session Information

Argument Value	Description
SOLDISP	Whether solicited messages (command responses) should be displayed at terminal
UNSDISP	Whether unsolicited messages should be displayed at terminal
SOLNUM	The number of solicited messages (command responses) to be held in message table
UNSNUM	The number of unsolicited messages to be held in message table
MFTIME	Whether time stamp should be displayed with messages
MFOSNM	Whether originating system name should be displayed with messages
MFJOB	Whether originating job name or job ID should be displayed with messages
MFSNMJBX	Whether system name and job name should be excluded from display of retrieved messages

Additional Examples

Example 1 - Using the LISTDSI and SYSDSN Functions:

```
/***************************** REXX ******************************/
/* This exec reallocates a data set with more space.  It receives  */
/* as arguments the names of a base data set and a new data set.   */
/* It uses the SYSDSN function to ensure the base data set exists, */
/* uses the LISTDSI function to set variables with attributes of   */
/* the base data set, doubles the primary space variable and then  */
/* uses the variables as input to the ALLOCATE command to          */
/* reallocate a new data set.                                      */
/*******************************************************************/

 PARSE ARG baseds newds            /* Receive the data set names   */
                                   /*   with quotes, if any.       */
 IF SYSDSN(baseds) = 'OK' THEN
   DO                              /* If the base data set exists, */
     x = LISTDSI(baseds)           /*  use the LISTDSI function.   */
     IF x = 0 THEN                 /* If the function code is 0,   */
       CALL alc                    /*  call an internal subroutine.*/
     ELSE
       DO                          /* Else, display the system     */
         SAY sysmsglvl1            /* messages and codes for LISTDS*/
         SAY sysmsglvl2
         SAY 'Function code from LISTDSI is' x
         SAY 'Sysreason code from LISTDSI is' sysreason
       END
   END
 ELSE
   SAY 'Data set' baseds 'not found.'
 EXIT

 alc:
 newprimary = 2 * sysprimary       /* Compute new primary space.   */
 "ALLOC DA("newds") NEW SPACE("newprimary sysseconds") LIKE("baseds")"
                                   /* Allocate the new data set.   */
 IF RC = 0 THEN               /* If return code from allocate is 0 */
   SAY 'Data set' newds 'was allocated.'
 ELSE
   SAY 'Data set' newds 'was not allocated. Return code was' RC
```

Example 2 Part 1 - Using the OUTTRAP Function:

```
/*************************** REXX ********************************/
/* This exec adds a data set to the front of the data sets in the  */
/* SYSPROC concatenation.  It first asks for the name of the data  */
/* set to add, then it finds all data sets currently allocated to  */
/* SYSPROC, adds the new data set to the beginning and re-allocates*/
/* the concatenation to SYSPROC.                                   */
/*******************************************************************/
 SAY 'Enter the fully-qualified data set name you want added'
 SAY 'to the beginning of the SYSPROC concatenation.  Do NOT'
 SAY 'place quotation marks around the data set name.'

 PULL addname .

 x = OUTTRAP('name.')/*Begin trapping lines of output from commands*/
                /* Output goes to variables beginning with 'name.'*/

 "LISTA ST"      /* List the status of your currently allocations  */
 found = 'NO'    /* Set the found flag to no    */
 i = 1           /* Set the index variable to 1 */

/*******************************************************************/
/* Loop through the lines of trapped command output to find lines  */
/* 9 characters long or longer. Check those lines for the word     */
/* SYSPROC until it is found or until all lines have been checked. */
/* If SYSPROC is found, the index is decreased one and the name of */
/* the first data set concatenated to SYSPROC is stored in variable*/
/* "concat".                                                       */
/*******************************************************************/
 DO WHILE (found = 'NO') & (i <= name.0)
   IF LENGTH(name.i) >= 9 THEN
     IF SUBSTR(name.i,3,7) = 'SYSPROC' THEN
       DO
         found = 'YES'
         i = i - 1
         concat = "'"name.i"'"
       END
     ELSE
       i = i + 1
   ELSE
     i = i + 1
 END
```

Example 2 Part 2 - Using the OUTTRAP Function:

```
/*******************************************************************/
/* When SYSPROC is found, loop through data sets until another file*/
/* name is encountered or until all lines are processed.  Append   */
/* data set names to the one in variable "concat".                 */
/*******************************************************************/
 IF found = 'YES' THEN
   DO WHILE (i + 3) <= name.0
     i = i + 3
       IF SUBSTR(name.i,1,3) = '   ' THEN
         DO
           i = i - 1
           concat = concat",'"name.i"'"
         END
       ELSE
         i = name.0

   END
 ELSE NOP

/* Allocate the new concatenation to SYSPROC */
 "ALLOC F(sysproc) DA('"addname"',"concat") SHR REUSE"
```

Example 3 - Using the OUTTRAP Function:

```
/****************************** REXX ******************************/
/* This exec lists datasets allocated to a ddname that is passed  */
/* as an argument when the exec is invoked.  It uses the OUTTRAP  */
/* function to trap output from the LISTA STATUS command.  It then */
/* loops through the output looking for a match to the input ddname*/
/* When match is found, the exec will SAY the name of all datasets */
/* allocated to that ddname.                                      */
/*                                                                */
/* The LISTA STATUS command produces output of the form           */
/*                                                                */
/*       DATASET-NAME-ALLOCATED-TO-DDNAME                         */
/*     DDNAME   DISP                                              */
/*                                                                */
/* In this output when the area for DDNAME is blank, then the data */
/* set is allocated to the previous DDNAME that was not blank. This*/
/* condition is one of the tests in the program below.            */
/*                                                                */
/******************************************************************/

ARG ddname .

x = OUTTRAP('ddlist.')         /* start output trapping into DDLIST*/
"LISTA STATUS"                 /* issue the LISTA command          */
x = OUTTRAP('OFF')             /* turn off output trapping         */

done = 'NO'                    /* initialize loop control variable */

DO i = 1 TO ddlist.0 WHILE done = 'NO'

   IF (words(ddlist.i) = 2) & ddname = word(ddlist.i,1) THEN
      DO                       /* if there is a DDNAME & it matches*/
      firstdataset = i - 1     /* back up to first dataset name    */
      SAY ddlist.firstdataset  /* Give the first dataset allocated */
      DO j = i+1 TO ddlist.0 BY 2 WHILE done = 'NO'
        next = j + 1
        IF (next <= ddlist.0) & (words(ddlist.next)\=1) THEN
           done = 'YES'        /* if we reach the end of the command
                                  output, or the next DDNAME, we are
                                  done                             */
        ELSE
           SAY ddlist.j        /* Give the next dataset allocated  */
      END
      END

END
If done = 'NO' then            /* If the DDNAME is not allocated   */
    say "The DDNAME" ddname "is not allocated."
                               /* Then say so                      */
EXIT 0
```

Function Packages

A **function package** is a group of external routines (functions and subroutines) that are accessed more quickly than external routines written in interpreted REXX. Routines in a function package must be written in a programming language that produces object code, which can be link-edited into a load module. The routine must also support the system interface for function packages. Some programming languages that meet these qualifications are assembler, COBOL, and PL/I.

There are three types of function packages.

- User packages — User-written external functions that are available to an individual. These functions are searched before other types of function packages and are often written to replace the other types of function packages.
- Local packages — Application or system support functions that are generally available to a specific group of users. Local packages are searched after user packages.
- System packages — Functions written for system-wide use, such as the TSO/E external functions. System packages are searched after user and local packages.

Function packages written by a user or an installation must be pre-loaded at logon time. The default name for the user packages is IRXFUSER, and the default name for the local package is IRXFLOC. Other function packages can be named in a parameter block set up by a system programmer.

For more information about function packages, see *z/OS TSO/E REXX Reference*.

Search Order for Functions

When the language processor encounters a function call, if defaults have not been changed, it goes through the following search order:

- Internal functions — Labels in the exec that issued the function call are searched first (unless the label is in quotation marks in the function call).
- Built-in functions — The built-in functions are next in the search order.
- Function packages — User, local, and system function packages, in that order, are searched.
- Load libraries — Functions stored in a load library are next in the search order.
- External function — An external function and its caller must either be members in the same PDS or members of PDSs allocated to a system library, such as SYSEXEC or SYSPROC.

Chapter 11. Storing Information in the Data Stack

This chapter describes how to use the REXX data stack to store information. Also, this chapter describes how to add a buffer to a data stack and how to create a private data stack in TSO/E.

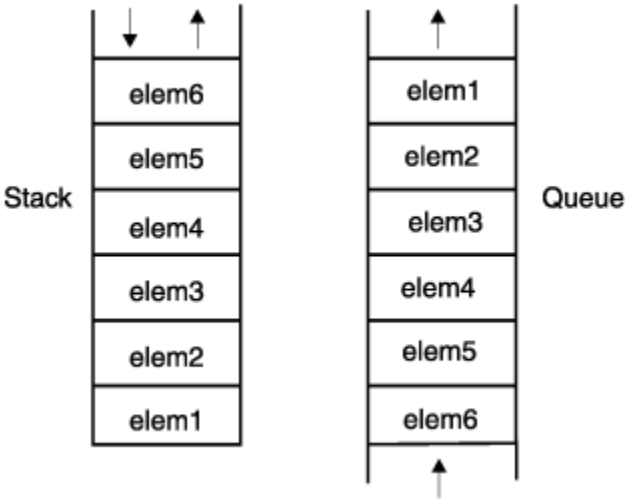
What is a Data Stack?

REXX in TSO/E uses an expandable data structure called a *data stack* to store information. The data stack combines characteristics of a conventional stack and queue.

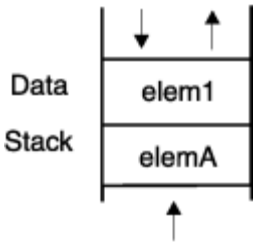
Stacks and queues are similar types of data structures used to temporarily hold data items (elements) until needed. When elements are needed, they are removed from the top of the data structure. The basic difference between a stack and a queue is where elements are added (as shown in the following figure). Elements are added to the top of a stack and to the bottom of a queue.

Using a *stack*, the last element added to the stack (elem6) is the first removed. Because elements are placed on the top of a stack and removed from the top, the newest elements on a stack are the ones processed first. The technique is called LIFO (last in first out).

Using a *queue*, the first element added to the queue (elem1) is the first removed. Because elements are placed on the bottom of a queue and removed from the top, the oldest elements on a queue are the ones processed first. The technique is called FIFO (first in first out).



As shown in the following figure, the data stack that REXX uses combines the techniques used in adding elements to stacks and queues. Elements can be placed on the top or the bottom of a data stack. Removal of elements from the data stack, however, occurs from the top of the stack only.



Manipulating the Data Stack

There are several REXX instructions that manipulate the data stack. Two instructions add elements to the data stack and another removes elements from the data stack.

Adding Elements to the Data Stack

You can store information on the data stack with two instructions, PUSH and QUEUE.

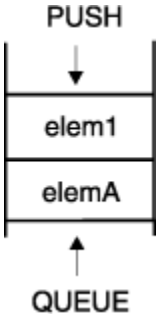
- PUSH - puts one item of data on the top of the data stack. There is virtually no limit to the length of the data item.

```
elem1 = 'String 1 for the data stack'
PUSH elem1
```

- QUEUE - puts one item of data on the bottom of the data stack. Again, there is virtually no limit to the length of the data item.

```
elemA = 'String A for the data stack'
QUEUE elemA
```

If the two preceding sets of instructions were in an exec, the data stack would appear as follows:



Note: Some people find it less confusing when adding elements in a particular order to the data stack, to consistently use the same instruction, either PUSH or QUEUE, but not both.

Removing Elements from the Stack

To remove information from the data stack, use the PULL and PARSE PULL instructions, the same instructions used previously in this book to extract information from the terminal. (When the data stack is empty, PULL removes information from the terminal.)

- PULL and PARSE PULL - remove one element from the top of the data stack.

```
PULL stackitem
```

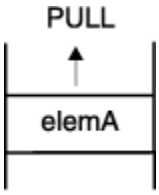
Using the examples from “Adding Elements to the Data Stack” on page 126, the variable `stackitem` then contains the value of elem1 with the characters translated to uppercase.

```
SAY stackitem              /* displays STRING 1 FOR THE DATA STACK */
```

When you add PARSE to the preceding instruction, the value is not translated to uppercase.

```
PARSE PULL stackitem
SAY stackitem              /* displays String 1 for the data stack */
```

After either of the preceding examples, the data stack appears as follows:



Determining the Number of Elements on the Stack

The QUEUED built-in function returns the total number of elements on a data stack. For example, to find out how many elements are on the data stack, you can use the QUEUED function with no arguments.

```
SAY QUEUED()         /* displays a decimal number */
```

To remove all elements from a data stack and display them, you can use the QUEUED function as follows:

```
number = QUEUED()
DO number
  PULL element
  SAY element
END
```

Exercise - Using the Data Stack

Write an exec that puts the letters T, S, O, E on the data stack in such a way that they spell "TSOE" when removed. Use the QUEUED built-in function and the PULL and SAY instructions to help remove the letters and display them. To put the letters on the stack, you can use the REXX instructions PUSH, QUEUE, or a combination of the two.

ANSWER

Possible Solution 1

```
/******************************* REXX ***************************/
/*  This exec uses the PUSH instruction to put the letters T,S,O,E,*/
/*  on the data stack in reverse order.                           */
/*******************************************************************/

 PUSH 'E'                              /*************************/
 PUSH 'O'                              /* Data in stack is:     */
 PUSH 'S'                              /*    (fourth push)    T  */
 PUSH 'T'                              /*    (third push)     S  */
                                       /*    (second push)    O  */
 number = QUEUED()                     /*    (first push)     E  */
 DO number                             /*************************/
   PULL stackitem
   SAY stackitem
 END
```

Possible Solution 2

```
/******************************** REXX ***************************/
/* This exec uses the QUEUE instruction to put the letters T,S,O,E,*/
/* on the data stack in that order.                                */
/*******************************************************************/

 QUEUE 'T'                              /*************************/
 QUEUE 'S'                              /*  Data in stack is:    */
 QUEUE 'O'                              /*     (first queue)    T  */
 QUEUE 'E'                              /*     (second queue)   S  */
                                        /*     (third queue)    O  */
 DO QUEUED()                            /*     (fourth queue)   E  */
   PULL stackitem                       /*************************/
   SAY stackitem
 END
```

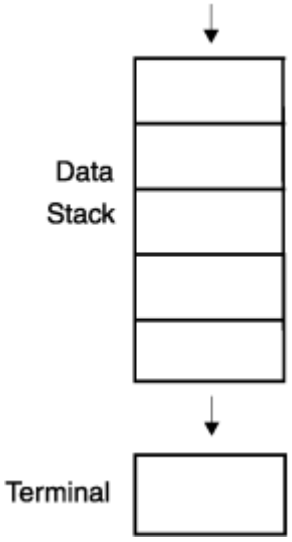
Possible Solution 3

```
/******************************** REXX ***************************/
/* This exec uses the PUSH and QUEUE instructions to put T,S,O,E   */
/* on the data stack.                                              */
/*******************************************************************/

 PUSH 'S'                               /*************************/
 QUEUE 'O'                              /*  Data in stack is:    */
 PUSH 'T'                               /*     (second push)    T  */
 QUEUE 'E'                              /*     (first push)     S  */
                                        /*     (first queue)    O  */
 DO QUEUED()                            /*     (second queue)   E  */
   PULL stackitem                       /*************************/
   SAY stackitem
 END
```

Processing of the Data Stack

You can think of a data stack as a temporary holding place for information. Every TSO/E REXX user has a separate data stack available for each REXX environment that is initialized. REXX environments are initialized at the READY prompt, when you enter ISPF, and again when you split the screen in ISPF.

When an exec issues a PULL instruction, and when it issues an interactive TSO/E command, the data stack is searched first for information and if that is empty, information is retrieved from the terminal.



Some types of input that can be stored on the data stack are:

- Data for the PULL and PARSE PULL instructions

  When an exec issues a PULL instruction, the language processor first goes to the data stack and pulls off the top element. If the data stack is empty, the language processor goes to the terminal for input. When the data stack is empty, the PULL instruction can be used with the SAY instruction to interact with a user at the terminal.

  Note: To prevent the language processor from searching the data stack, you can issue the PARSE EXTERNAL instruction instead of PULL. PARSE EXTERNAL gets input directly from the terminal and bypasses the data stack.
- Responses to commands

  A TSO/E interactive command (such as LISTDS, TRANSMIT, and ALLOCATE) can prompt a terminal user for information. Similarly, user responses can be put on the data stack by an exec for the command's use.
- Commands to be issued after the exec ends

  When an exec ends, all elements remaining on the data stack are processed before the READY mode message is displayed. These remaining elements are treated as TSO/E commands to be issued. If the element is not a TSO/E command (or an implicit exec or CLIST to be run), you see the message:

```
COMMAND command_name NOT FOUND
```

- Information the EXECIO command reads from and writes to data sets when performing I/O.

  For information about the EXECIO command and how it uses the data stack, see "Using EXECIO to Process Information to and from Data Sets" on page 142.

Using the Data Stack

The data stack has some unique characteristics, such as:

- It can contain a virtually unlimited number of data items of virtually unlimited size.
- It can contain commands to be issued after the exec ends.
- It can pass information between REXX execs and other types of programs in a TSO/E or non-TSO/E address space.

Because of the data stack's unique characteristics, you can use the data stack specifically to:

- Store a large number of data items for a single exec's use.
- Pass a large number of arguments or an unknown number of arguments between a routine (subroutine or function) and the main exec.
- Pass responses to an interactive command that can run after the exec ends.
- Store data items from an input data set, which were read by the EXECIO command. For information about the EXECIO command, see "Using EXECIO to Process Information to and from Data Sets" on page 142.
- Share information between an exec and any program running in MVS. For more information about running REXX execs in MVS, see Chapter 13, "Using REXX in TSO/E and Other MVS Address Spaces," on page 159.
- Execute subcommands of a TSO/E command issued from a REXX exec.

Passing Information Between a Routine and the Main Exec

You can use the data stack to pass information from an exec to an external routine without using arguments. The exec pushes or queues the information on the stack and the routine pulls it off and uses it as in the following example.

Example of Using the Data Stack to Pass Information

```
/***************************** REXX ********************************/
/* This exec helps an inexperienced user allocate a new PDS.  It   */
/* prompts the user for the data set name and approximate size,    */
/* and queues that information on the data stack.  Then it calls   */
/* an external subroutine called newdata.                          */
/*******************************************************************/

 message = 'A data set name for a partitioned data set has three',
'qualifiers separated by periods.  The first qualifier is usually',
'a user ID.  The second qualifier is any name.  The third qualifier',
'is the type of data set, such as "exec".  Generally the user ID',
'is assumed, so you might specify a data set name as MYREXX.EXEC.',
'A new data set name cannot be the same as an existing data set',
'name.  Please type a name for the new data set or type QUIT to end.'

 SAY 'What is the new data set name?  If you are unsure about'
 SAY 'naming data sets, type ?.  To end, type QUIT.'

 PULL name
 DO WHILE (name = '?') | (name = 'QUIT')
   IF name = '?' THEN
     DO
       SAY message
       PULL name
     END
   ELSE
     EXIT
 END

 SAY 'Approximately how many members will the data set have:'
 SAY '6  12  18  24  30  36  42  48  54  60?'

 PULL number
 QUEUE name
 QUEUE number
 CALL newdata

 IF RESULT > 0 THEN
   SAY 'An error prevented' name 'from being allocated.'
 ELSE
   SAY 'Your data set' name 'has been allocated.'
```

Example of the External Subroutine NEWDATA

```
/***************************** REXX ********************************/
/* This external subroutine removes the data set name and the      */
/* number of members from the stack and then issues the ALLOCATE   */
/* command.                                                        */
/*******************************************************************/

 PULL name
 PULL number

 "ALLOCATE DATASET("name") NEW SPACE(50,20) DIR("number%6") DSORG(PO)",
 "RECFM(V,B) LRECL(255) BLKSIZE(5100)"

 RETURN RC    /* The return code from the TSO/E command sets the   */
              /* REXX special variable, RC, and is returned to the */
              /* calling exec. A 0 return code means no errors.    */
```

Passing Information to Interactive Commands

When your TSO/E profile allows prompting, most TSO/E commands prompt you for missing operands. For example, the TRANSMIT command prompts you for a node and user ID when you do not include the destination with the command.

An exec can put responses to command prompts on the data stack. Because of the information search order, the data stack supplies the necessary information instead of a user at the terminal.

For example, the following exec puts the TRANSMIT command and its operands on the data stack. When the exec completes, the TSO/E data stack service continues to get input from the data stack. Thus the TRANSMIT command is issued after the exec ends.

Issuing Subcommands of TSO/E Commands

To execute subcommands of a TSO/E command in a REXX exec, you must place the subcommands onto the data stack before you issue the TSO/E command.

Example of Passing Information from the Stack to a Command

```
/****************************** REXX ******************************/
/* This exec prompts a user for a node and gets the user ID from a */
/* built in function.  It then calls an external subroutine to     */
/* check if the user's job is finished.                            */
/* The TRANSMIT command and its operands, including a message with */
/* the status of the job, are queued on the data stack to run after*/
/* the exec terminates.                                            */
/*****************************************************************/

 SAY 'What is your node?'
 PULL node
 id = USERID()
 dest = node'.'id

 CALL jobcheck userid /* Go to a subroutine that checks job status */

 IF RESULT = 'done' THEN
    note = 'Your job is finished.'
 ELSE
    note = 'Your job is not finished.'

 QUEUE 'transmit'
 QUEUE dest 'line'     /* Specify that the message be in line mode */
 QUEUE note
 QUEUE ''           /* Insert a null to indicate line mode is over */
```

Creating a Buffer on the Data Stack

When an exec calls a routine (subroutine or function) and both the exec and the routine use the data stack, the stack becomes a way to share information. However, execs and routines that do not purposely share information from the data stack, might unintentionally do so and end in error. To help prevent this, TSO/E provides the MAKEBUF command that creates a buffer, which you can think of as an extension to the stack, and the DROPBUF command that deletes the buffer and all elements within it.

Although the buffer does not prevent the PULL instruction from accessing elements placed on the stack before the buffer was created, it is a way for an exec to create a temporary extension to the stack. The buffer allows an exec to:

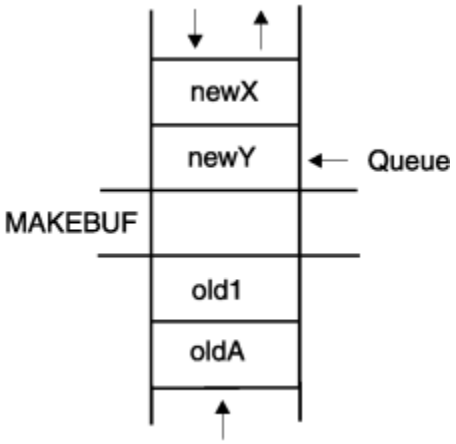
1. Use the QUEUE instruction to insert elements in FIFO order on a stack that already contains elements.
2. Have temporary storage that it can delete easily with the DROPBUF command.

An exec can create multiple buffers before dropping them. Every time MAKEBUF creates a new buffer, the REXX special variable RC is set with the number of the buffer created. Thus if an exec issues three MAKEBUF commands, RC is set to 3 after the third MAKEBUF command.

Note: To protect elements on the stack, an exec can create a new stack with the NEWSTACK command. For information about the NEWSTACK command, see “Protecting elements in the data stack” on page 136.

Creating a Buffer with the MAKEBUF Command

To create a buffer on the data stack before adding more elements to the stack, use the TSO/E REXX MAKEBUF command. All elements added to the data stack after the MAKEBUF command are placed in the buffer. Below the buffer are elements placed on the stack before the MAKEBUF command.



Instructions that could be used to create the illustrated buffer are as follows:

```
'MAKEBUF'
PUSH 'newX'
QUEUE 'newY'
```

Removing Elements from a Stack with a Buffer

The buffer created by MAKEBUF does not prevent an exec from accessing elements below it. After an exec removes the elements added after the MAKEBUF command, then it removes elements added before the MAKEBUF command was issued.

Using the previous illustration, when the exec issues three PULL instructions, the following elements are removed from the data stack.

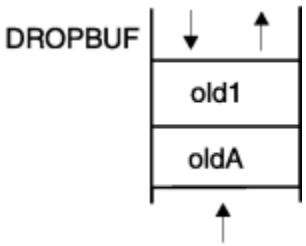
```
newX
newY
old1
```

To prevent a routine from accessing elements below the buffer, you can use the QUEUED built-in function as follows:

```
olditems = QUEUED()
'MAKEBUF'
PUSH …
QUEUE …
DO WHILE QUEUED() > olditems  /* total items > old number of items */
   PULL ….
   …
END
'DROPBUF'
```

Dropping a Buffer with the DROPBUF Command

When an exec has no more need for a buffer on the data stack, it can use the TSO/E REXX DROPBUF command to remove the buffer (and its contents). The DROPBUF command removes the most recently created buffer.



To drop a specific buffer on the data stack and all buffers created after it, issue the DROPBUF command with the number of the buffer. The first MAKEBUF creates buffer 1, the second creates buffer 2, and so on. For example, if an exec issued three MAKEBUF commands that created three buffers, when you issue DROPBUF 2, the second and third buffers and all elements within them are removed.

To remove all elements from the entire data stack including elements placed on the data stack before buffers were added, issue DROPBUF 0. DROPBUF 0 creates an empty data stack and should be used with caution.

Note: When an element is removed below a buffer, the buffer disappears. Thus when elements are unintentionally removed below a buffer, the corresponding DROPBUF command might remove the incorrect buffer and its elements. To prevent an exec from removing elements below a buffer, use the QUEUED built-in function or use the NEWSTACK command as described in "Protecting elements in the data stack" on page 136.

Finding the Number of Buffers with the QBUF Command

To find out how many buffers were created with the MAKEBUF command, use the TSO/E REXX QBUF command. QBUF returns in the REXX special variable RC, the number of buffers created.

```
'MAKEBUF'
⋮
'MAKEBUF'
⋮
'QBUF'
SAY 'The number of buffers is' RC                    /* RC = 2 */
```

QBUF returns the total number of buffers created, not just the ones created by a single exec. Thus if an exec issued two MAKEBUF commands and called a routine that issued two more, when the routine issues a QBUF command, RC returns the total number of buffers created, which is four.

Finding the Number of Elements In a Buffer

To find out how many elements are in the most recently created buffer, use the TSO/E REXX QELEM command. QELEM returns in the REXX special variable RC, the number of elements in the most recently created buffer.

```
PUSH A
'MAKEBUF'
PUSH B
PUSH C
'QELEM'
SAY 'The number of elements is' RC                   /* RC = 2 */
```

QELEM does not return the number of elements on a data stack with no buffers created by the MAKEBUF command. If QBUF returns 0, no matter how many elements are on the stack, QELEM also returns 0.

For more information about these stack commands, see *z/OS TSO/E REXX Reference*.

Exercises - Creating a Buffer on the Data Stack

1. What are the results of the following instructions?

a. What is item?

```
QUEUE A
QUEUE B
'MAKEBUF'
QUEUE C
PULL item
```

b. What is element?

```
PUSH 'a'
PUSH 'b'
'MAKEBUF'
PUSH 'c'
PUSH 'd'
'DROPBUF'
PARSE PULL element
```

c. What is stackitem?

```
QUEUE a
'MAKEBUF'
QUEUE b
'MAKEBUF'
QUEUE c
'DROPBUF'
PULL stackitem
```

d. What is RC?

```
PUSH A
'MAKEBUF'
PUSH B
CALL sub1
'QBUF'
SAY RC
EXIT

sub1:
'MAKEBUF'
RETURN
```

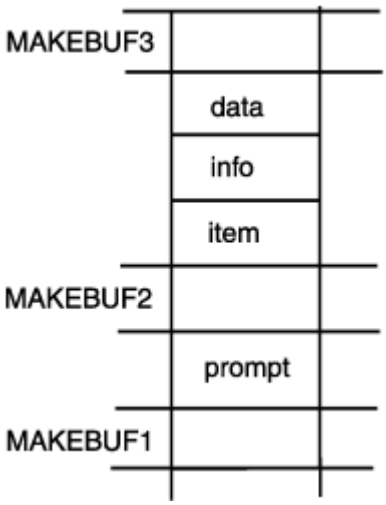
e. What is RC?

```
QUEUE A
'MAKEBUF'
PUSH B
PUSH C
'MAKEBUF'
PUSH D
'QELEM'
SAY RC
```

f. What is RC?

```
QUEUE A
QUEUE B
QUEUE C
'QELEM'
SAY RC
```

2. Given the data stack below and the instructions that created it, what are the results of the subsequent instructions that follow?




```
'MAKEBUF'
QUEUE 'prompt'
'MAKEBUF'
QUEUE 'data'
QUEUE 'info'
QUEUE 'item'
'MAKEBUF'
```

a. What is returned to the function?

```
SAY QUEUED()
```

b. What is RC?

```
'QBUF'
SAY RC
```

c. What is RC?

```
'QELEM'
SAY RC
```

d. What are both RCs and the result of the QUEUED() function?

```
'DROPBUF 2'
'QBUF'
SAY RC
'QELEM'
SAY RC
SAY QUEUED()
```

ANSWERS

1. a. C
   b. b
   c. B (b was changed to uppercase because it was queued without quotes and pulled without PARSE.)
   d. 2
   e. 1
   f. 0
2. a. 4
   b. 3
   c. 0
   d. 1, 1, 1

Protecting elements in the data stack

In certain environments, particularly MVS, where multiple tasks run at the same time, it is often important for an exec to isolate stack elements from other execs.

Similarly, an exec in TSO/E might want to protect stack elements from a routine (subroutine or function) that it calls. For example, if an exec puts elements on the data stack for its own use and then calls a subroutine that issues an interactive TSO/E command, such as ALLOCATE, the command goes to the data stack first for input to the command. Because the stack input is incorrect for the command prompt, the exec ends in error.

```
EXEC1

PUSH prompt1
PUSH prompt2
CALL sub1
7invellip.
EXIT

SUB1:

'MAKEBUF'
'ALLOCATE'
⋮
```

Figure 1: Example of an interactive command error

Even though the subroutine in the preceding example starts with the MAKEBUF command, the stack elements are used because MAKEBUF does not protect elements previously placed on the stack.

To protect elements on the data stack, you can create a new data stack with the TSO/E REXX NEWSTACK command. Read the next section to see how the exec in the previous example can safely issue an interactive TSO/E command.

To delete the new data stack and all elements in it, use the TSO/E REXX DELSTACK command. Execs can create multiple stacks before deleting them.

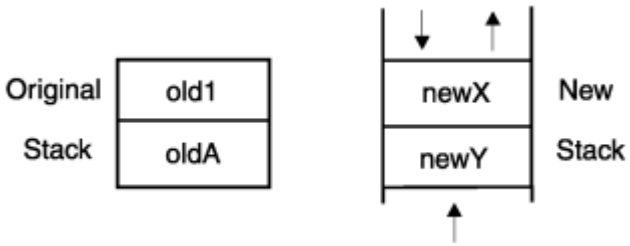
Note: Before an exec returns to its caller, the called exec should issue a DELSTACK command for each NEWSTACK command it issued, unless the called exec intends for the caller to also use the new data stack.

Creating a New Data Stack with the NEWSTACK Command

The TSO/E REXX NEWSTACK command creates a private data stack that is completely isolated from the original data stack. The elements on the original data stack cannot be accessed by an exec or the routines that it calls until a DELSTACK command is issued. When there are no more elements in the new data stack, information is taken from the terminal.

Note: When you issue the NEWSTACK, it is your responsibility to issue a corresponding DELSTACK command.

All elements added to the data stack after the NEWSTACK command are placed in the new data stack. The original stack contains the elements placed on the stack before the NEWSTACK command.



Instructions that could be used to create the illustrated new stack are as follows:

```
PUSH 'oldA'
PUSH 'old1'
'NEWSTACK'
QUEUE 'newY'
PUSH 'newX'
```

In the Example of an Interactive Command Error, the MAKEBUF command did not protect the elements in the stack. If you substitute the NEWSTACK command for the MAKEBUF command, the elements become inaccessible.

Example of using NEWSTACK with an Interactive Command:

```
EXEC1

PUSH prompt1
PUSH prompt2
CALL sub1
⋮
EXIT

SUB1:

'NEWSTACK'
'ALLOCATE'
⋮
```

Note: To have an interactive command prompt the user for input from the terminal, run an exec explicitly with the EXEC command and specify `prompt` or include the PROMPT(on) function within the exec. For more information, see “Causing Interactive Commands to Prompt the User” on page 95.

Deleting a Private Stack with the DELSTACK Command

When an exec wants to delete the new stack and remove all elements placed on the new stack, it can issue the TSO/E REXX DELSTACK command. The DELSTACK command removes the most recently created data stack. If no stack was previously created with the NEWSTACK command, DELSTACK **removes all the elements from the original stack.**

Finding the Number of Stacks

To find out how many stacks exist, use the TSO/E REXX QSTACK command. QSTACK returns in the REXX special variable RC, the total number of stacks including the original data stack.

```
'NEWSTACK'
⋮
'NEWSTACK'
⋮
'QSTACK'
SAY 'The number of stacks is' RC                  /* RC contains 3 */
```

QSTACK returns the total number of stacks, not just the ones created for a single exec. Thus if an exec issued two NEWSTACK commands and called a routine that issued two more, when the routine issues a QSTACK command, RC returns the total number of stacks, which is five.

For more information about these commands, see *z/OS TSO/E REXX Reference*.

Additional Examples

Data Stack Example 1:

```
/******************************** REXX ***************************/
/* This exec tests several of the stack functions to see how they  */
/* work together. It uses the NEWSTACK and DELSTACK commands, puts */
/* an element on the stack that exceeds 255 characters, uses the   */
/* LENGTH built-in function to see how long the element is, uses   */
/* QUEUED built-in function to see how many items are on the stack,*/
/* and then issues more PULL instructions than are elements on the */
/* stack.                                                          */
/*****************************************************************/
 element = 'Attention please!  This is a test.'
 PUSH element

 'NEWSTACK'  /* Create a new stack and protect elements previously */
             /* placed on the stack */

 longitem = 'SAA is a definition -- a set of software interfaces,',
  'conventions, and protocols that provide a framework for designing',
  'and developing applications with cross-system consistency.',
  'The Systems Application Architecture defines a common programming',
  'interface you can use to develop applications, and defines common',
  'communications support that you can use to connect those',
  'applications.'

 SAY 'The length of the element is' LENGTH(longitem) 'characters.'
                   /* The length of the element is 379 characters. */
 QUEUE longitem

 PULL anyitem
 SAY anyitem           /* Displays the longitem quote in uppercase */

 SAY 'There are' QUEUED() 'number of elements on the stack.'
                                 /* The QUEUED function returns 0 */

 PULL emptyitem /* Pull an element from an empty stack. Results in */
                /* a blank screen and PULL waits for terminal      */
                /* input.  To end the wait, press ENTER.           */

 'DELSTACK'   /* Remove the new stack and return to original stack.*/

 PULL anyitem
 SAY anyitem        /* Displays ATTENTION PLEASE!  THIS IS A TEST. */
```

Data Stack Example 2:

```
/******************************** REXX ***************************/
/* This exec runs another exec implicitly and then sends a message */
/* when the called exec finishes.  It receives as an argument the  */
/* name of a PDS member to run.  It activates the system procedure */
/* file SYSEXEC, allocates the data set to SYSEXEC, pushes some    */
/* commands on the data stack and then implicitly executes the exec*/
/*********************************************************************/

 ARG dsn

 "EXECUTIL SEARCHDD(yes)"   /* Establish the system library SYSEXEC*/

 PUSH "SEND 'Sequence over' USER(*)"  /* Put a message on the stack*/
 PUSH "TIME"                          /* Push the time command     */
 PUSH "FREE F(SYSEXEC)"             /* Push command to free SYSEXEC*/

 PARSE VAR dsn name '(' member  /* Separate the data set name from */
                                          /* the member name. */

 "ALLOC DA("name") F(SYSEXEC) SHR REUSE"

 execname = STRIP(member,t,')') /* Remove the last parentheses from*/
                                /* the member name.               */

 PUSH '%'execname               /* Put the member name on the stack*/

/*********************************************************************/
/* The output from this exec depends on the exec that it runs.     */
/* Output can be as follows:                                       */
/*                                                                 */
/*TIME-01:23:56 PM.CPU-00:00:23 SERVICE-297798 SESSION-04:15:20 MAY*/
/*12,1989                                                          */
/* Sequence over USERID                                           */
/* READY                                                          */
/*********************************************************************/
```

Chapter 12. Processing Data and Input/Output Processing

This chapter describes dynamic modification of a single REXX expression and I/O processing of data sets.

Types of Processing

The word "processing" is used here to mean the performance of operations and calculations on data. Normal processing of instructions in REXX occurs every time the language processor evaluates an expression. This chapter describes two special types of REXX processing:

- Dynamic modification of a single REXX expression

  The INTERPRET instruction evaluates an expression and then treats it as a REXX instruction.
- Processing information to and from data sets

  The TSO/E REXX EXECIO command in an exec reads information from a data set to the data stack (or a list of variables) and writes information from the data stack (or list of variables) back to a data set.

Dynamic Modification of a Single REXX Expression

Typically REXX expressions are evaluated and the result replaces the expression. For example, the arithmetic expression "5 + 5" is evaluated as "10".

```
answer = 5 + 5                          /* answer gets the value 10 */
```

If the arithmetic expression is in quotation marks, the expression is evaluated as a string.

```
answer = '5 + 5'                    /* answer gets the value 5 + 5 */
```

To both evaluate and execute an expression, you can use the INTERPRET instruction.

Using the INTERPRET Instruction

The INTERPRET instruction not only evaluates an expression, but also treats it as an instruction after it is evaluated. Thus if a combination of the previous examples were used with the INTERPRET instruction, answer becomes "10".

```
answer = 5 + 5
INTERPRET 'say' answer '"= 5 + 5"'          /* displays 10 = 5 + 5 */
```

You can also group a number of instructions within a string, assign the string to a variable, and use the INTERPRET instruction to execute the instructions assigned to the variable.

```
action = 'DO 3; SAY "Hello!"; END'
INTERPRET action                              /* results in:
                                                        Hello!
                                                        Hello!
                                                        Hello!   */
```

Because the INTERPRET instruction causes dynamic modification, use it very carefully. For more information about the INTERPRET instruction, see *z/OS TSO/E REXX Reference*.

Using EXECIO to Process Information to and from Data Sets

An exec uses the EXECIO command to perform the input and output (I/O) of information to and from a data set. The information can be stored in the data stack for serialized processing or in a list of variables for random processing.

When to Use the EXECIO Command

The various operands and combination of operands of the EXECIO command permit you to do many types of I/O. For example, you can use the EXECIO command to:

- Read information from a data set
- Write information to a data set
- Open a data set without reading or writing any records
- Empty a data set
- Copy information from one data set to another
- Copy information to and from a list of compound variables
- Add information to the end of a sequential data set
- Update information in a data set one line at a time

Using the EXECIO Command

EXECIO reads information from a data set with either the DISKR or DISKRU operands. Using these operands, you can also open a data set without reading its records. Refer to “Reading Information from a Data Set” on page 142 for more information about the DISKR and DISKRU operands. EXECIO writes information to a data set with the DISKW operand. Using this operand, you can also open a data set without writing records or empty an existing data set. Refer to “Writing Information to a Data Set” on page 144 for more information on the DISKW operand.

Before an exec can use the EXECIO command to read from or write to a data set, the data set must meet the following requirements. An I/O data set must be:

- Either sequential or a single member of a PDS.
- Previously allocated with the appropriate attributes for its specific purpose. Some examples of the various uses of EXECIO and the type of data set allocation appropriate for these uses are shown in and after “Copying Information From One Data Set to Another” on page 147.

If you use EXECIO to read information from a data set and to the data stack, the information can be stored in FIFO or LIFO order on the data stack. FIFO is the default. If you use EXECIO to read information from a data set and to a list of variables, the first data set line is stored in variable1, the second data set line is stored in variable2, and so on. Data read into a list of variables can be accessed randomly. After the information is in the data stack or in a list of variables, the exec can test it, copy it to another data set, or update it before returning it to the original data set.

Reading Information from a Data Set

To read information from a data set to the data stack or to a list of variables, use EXECIO with either the DISKR or DISKRU operand. A typical EXECIO command to read all lines from the data set allocated to the ddname MYINDD, might appear as:

```
"EXECIO * DISKR  myindd (FINIS"
```

The rest of this topic describes the types of information you can specify with EXECIO DISKR and EXECIO DISKRU. For further information, see *z/OS TSO/E REXX Reference*.

How to specify the number of lines to read

To open a data set without reading any records, put a zero immediately following the EXECIO command and specify the OPEN operand.

```
"EXECIO 0  DISKR mydd (OPEN"
```

To read a specific number of lines, put the number immediately following the EXECIO command.

```
"EXECIO 25 …"
```

To read the entire data set, put an asterisk immediately following the EXECIO command.

```
"EXECIO  * …"
```

When all the information is on the data stack, either queue a null line to indicate the end of the information, or if there are null lines throughout the data, assign the built-in QUEUED() function to a variable to indicate the number of items on the stack.

How to read the data set

Depending on the purpose you have for the input data set, use either the DISKR or DISKRU operand.

- DISKR - Reading Only

  To initiate I/O from a data set that you want to read only, use the DISKR operand with the FINIS option. The FINIS option closes the data set after the information is read. Closing the data set allows other execs to access the data set and the ddname.

  ```
  "EXECIO  *  DISKR … (FINIS"
  ```

  Note: Do not use the FINIS option if you want the next EXECIO statement in your exec to continue reading at the line immediately following the last line read.

- DISKRU - Reading and Updating

  To initiate I/O to a data set that you want to both read and update, use the DISKRU operand without the FINIS option. Because you can update only the last line that was read, you usually read and update a data set one line at a time, or go immediately to the single line that needs updating. The data set remains open while you update the line and return the line with a corresponding EXECIO DISKW command.

  ```
  "EXECIO  1  DISKRU …"
  ```

  More about using DISKRU appears in “Updating Information in a Data Set” on page 149.

How to access the data set

An I/O data set must first be allocated to a ddname. The ddname need not exist previously. In fact, it might be better to allocate it to a new ddname, such as MYINDD, in order not to interfere with previously established allocations. You can allocate before the exec runs, or you can allocate from within the exec with the ALLOCATE command as shown in the following example.

```
"ALLOC DA(io.data) F(myindd) SHR REUSE"
"EXECIO  *  DISKR  myindd  (FINIS"
```

Option of specifying a starting line number

If you want to start reading at other than the beginning of the data set, specify the line number at which to begin. For example, to read all lines to the data stack starting at line 100, add the following line number operand.

```
"EXECIO  *  DISKR  myindd  100  (FINIS"
```

To read just 5 lines to the data stack starting at line 100, write the following:

```
"EXECIO  5  DISKR  myindd  100  (FINIS"
```

To open a data set at line 100 without reading lines to the data stack, write the following:

```
"EXECIO  0  DISKR  myindd  100  (OPEN"
```

Options for DISKR and DISKRU

Options you can use are:

- OPEN - To open a data set. When you specify OPEN with EXECIO 0, it opens the data set and positions the file position pointer before the first record.

  ```
  "EXECIO 0 DISKR myindd (OPEN"
  ```

  Note: If the data set is already open, no operation is performed for OPEN.
- FINIS - To close the data set after reading it. Closing the data set allows other execs to access it and its ddname. It also resets the current positional pointer to the beginning of the data set.
- STEM - To read the information to either a list of compound variables that can be indexed, or a list of variables appended with numbers. Specifying STEM with a variable name ensures that a list of variables (not the data stack) receives the information.

  ```
  "EXECIO  *  DISKR  myindd  (STEM newvar."
  ```

  In this example, the list of compound variables has the stem `newvar.` and lines of information or records from the data set are placed in variables `newvar.1`, `newvar.2`, `newvar.3`, and so forth. The number of items in the list of compound variables is placed in the special variable `newvar.0`.

  Thus if 10 lines of information are read into the newvar variables, `newvar.0` contains the number 10, indicating that 10 records have been read. Furthermore, `newvar.1` contains record 1, `newvar.2` contains record 2, and so forth up to `newvar.10` which contains record 10. All stem variables beyond `newvar.10` (for example, variables `newvar.11` and `newvar.12`) are residual and contain the value(s) held prior to entering the EXECIO command.

  To avoid confusion as to whether a residual stem variable value is meaningful, you may want to clear the entire stem variable prior to entering the EXECIO command. To clear all stem variables, you can either:

  – Use the DROP instruction as follows, which sets all stem variables to their uninitialized state.

    ```
    DROP newvar.
    ```

  – Set all stem variables to nulls as follows:

    ```
    newvar. = '
    ```

  See EXECIO Example 6 under the heading “Additional Examples” on page 150, which shows the usage of the EXECIO command with stem variables.
- SKIP - To skip over a specified number of lines in a data set without placing them on the data stack or into variables.

  ```
  "EXECIO  24  DISKR  myindd  (SKIP"
  ```
- LIFO - To read the information in LIFO order onto the stack. In other words, use the PUSH instruction to place the information on the stack.
- FIFO - To read the information in FIFO order onto the stack. In other words, use the QUEUE instruction to place the information on the stack. If you do not specify either LIFO or FIFO, FIFO is assumed.

Writing Information to a Data Set

To write information to a data set from the data stack or from a list of variables, use EXECIO with the DISKW operand. A typical EXECIO command to write all lines to the data set allocated to the ddname, MYOUTDD, might appear as:

```
"EXECIO * DISKW myoutdd (FINIS"
```

The rest of this topic describes the types of information you can specify with EXECIO DISKW. For further information, see *z/OS TSO/E REXX Reference*.

How to specify the number of lines to write

To open a data set without writing records to it, put a zero immediately following the EXECIO command and specify the OPEN operand.

```
"EXECIO 0  DISKW  myoutdd … (OPEN"
```

Note:

1. To empty a data set, issue this command to open the data set and position the file position pointer before the first record. You then issue `EXECIO 0 DISKW myoutdd … (FINIS` to write an end-of-file mark and close the data set. This deletes all records in data set MYOUTDD. You can also empty a data set by issuing EXECIO with both the OPEN and FINIS operands.
2. When you empty a data set, the file to which the data set is allocated should not have a disposition of MOD. If the file has a disposition of MOD, opening and then closing the data set will not empty the data set.

To write a specific number of lines, put the number immediately following the EXECIO command.

```
"EXECIO 25  DISKW  …"
```

To write the entire data stack or until a null line is found, put an asterisk immediately following the EXECIO command.

```
"EXECIO  *  DISKW  …"
```

When you specify *, the EXECIO command will continue to pull items off the data stack until it finds a null line. If the stack becomes empty before a null line is found, EXECIO will prompt the terminal for input until the user enters a null line. Thus when you do not want to have terminal I/O, queue a null line at the bottom of the stack to indicate the end of the information.

```
QUEUE '
```

If there are null lines (lines of length 0) throughout the data and the data stack is not shared, you can assign the built-in QUEUED() function to a variable to indicate the number of items on the stack.

```
n = QUEUED()
"EXECIO" n "DISKW  outdd (FINIS"
```

How to access the data set

An I/O data set must first be allocated to a ddname. The ddname does not need to exist previously. In fact, it might be better to allocate to a new ddname, such as MYOUTDD, in order not to interfere with previously established allocations. You can allocate from within the exec with the ALLOCATE command as shown in the following example, or you can allocate before the exec runs.

```
"ALLOC DA(out.data) F(myoutdd) OLD REUSE"
"EXECIO  *  DISKW  myoutdd  …"
```

Options for DISKW

Options you can use are:

- OPEN - To open a data set. When you specify OPEN with EXECIO 0, it opens the data set and positions the file position pointer before the first record.

```
"EXECIO  0  DISKW  myoutdd  (OPEN"
```

  Note: If the data set is already open, no operation is performed for OPEN.

- FINIS - To close the data set after writing to it. Closing the data set allows other execs to access it and its ddname. When you specify FINIS, it forces the completion of all I/O operations by physically writing the contents of any partially filled I/O buffers to the data set.

```
"EXECIO  *  DISKW  myoutdd  (FINIS"
```

- STEM - To write the information from compound variables or a list of variables beginning with the name specified after the STEM keyword. The variables, instead of the data stack, holds the information to be written.

```
"EXECIO  *  DISKW  myoutdd  (STEM newvar."
```

  In this example, the variables would have the stem `newvar.` and lines of information from the compound variables would go to the data set. Each variable is labeled `newvar.1`, `newvar.2`, `newvar.3`, and so forth.

  The variable `newvar.0` is not used when writing from compound variables. When * is specified with a stem, the EXECIO command stops writing information to the data set when it finds a null value or an uninitialized compound variable. In this case, if the list contained 10 compound variables, the EXECIO command stops at `newvar.11`.

  The EXECIO command can also specify the number of lines to write from a list of compound variables.

```
"EXECIO  5  DISKW  myoutdd  (STEM newvar."
```

  In this example, the EXECIO command writes 5 items from the `newvar` variables including uninitialized compound variables, if any.

See “Additional Examples” on page 150 Example 6 for usage of the EXECIO command with stem variables.

Return Codes from EXECIO

After an EXECIO command runs, it sets the REXX special variable "RC" to a return code. Valid return codes from EXECIO are:

Return Code	Meaning
0	Normal completion of requested operation.
1	Data was truncated during DISKW operation.
2	End-of-file reached before the specified number of lines were read during a DISKR or DISKRU operation. (This return code does not occur when * is specified for number of lines because the remainder of the file is always read.)
4	An empty data set was found within a concatenation of data sets during a DISKR or DISKRU operation. The file was not successfully opened and no data was returned.
20	Severe error. EXECIO completed unsuccessfully and a message is issued.

When to Use the EXECIO Command

The various operands and combination of operands of the EXECIO command permit you to do many types of I/O. For example, you can use the EXECIO command to:

- Copy information from one data set to another

  - Copy an entire data set
  - Copy parts of a data set
  - Add information to the end of a sequential data set
- Copy information to and from a list of compound variables
- Update information in a data set

Copying Information From One Data Set to Another

Before you can copy one data set to another, the data sets must be either sequential data sets or members of a PDS, and they must be pre-allocated. Following are examples of ways to allocate and copy data sets using the EXECIO command.

Copying an entire data set

To copy an entire existing sequential data set named 'USERID.MY.INPUT' into a new sequential data set named 'USERID.NEW.INPUT', and to use the ddnames DATAIN and DATAOUT respectively, you could use the following instructions. (Remember that when the first qualifier of a data set name is your prefix (usually your user ID), you can omit the first qualifier.)

Copying an Entire Data Set:

```
"ALLOC DA(my.input) F(datain) SHR REUSE"
"ALLOC DA(new.input) F(dataout) LIKE(my.input) NEW"
"NEWSTACK"               /* Create a new data stack for input only */
"EXECIO * DISKR datain (FINIS"
QUEUE '' /* Add a null line to indicate the end of the information */
"EXECIO * DISKW dataout (FINIS"
"DELSTACK"                           /* Delete the new data stack */
"FREE F(datain dataout)"
```

If the null line was not queued at the end of the information on the stack, the EXECIO command would go to the terminal to get more information and would not end until the user entered a null line.

Another way to indicate the end of the information when copying an entire data set, is with the QUEUED() built-in function. If the data set is likely to include null lines throughout the data, using the QUEUED() function is preferable.

```
n = QUEUED()            /* Assign the number of stack items to "n" */
"EXECIO" n "DISKW dataout (FINIS"
```

Also, when copying an undetermined number of lines to and from the data stack, it is a good idea to use the NEWSTACK and DELSTACK commands to prevent removing items previously placed on the stack. For more information about these commands, see “Protecting elements in the data stack” on page 136.

Copying a specified number of lines to a new data set

To copy 10 lines of data from an existing sequential data set named 'DEPT5.STANDARD.HEADING' to a new member in an existing PDS named 'USERID.OFFICE.MEMO(JAN15)', and use the ddnames INDD and OUTDD respectively, you could use the following instructions. (Remember that a data set name that does not begin with your prefix must be enclosed in single quotes.)

Copying 10 Lines of Data to a New Data Set:

```
"ALLOC DA('dept5.standard.heading') F(indd) SHR REUSE"
"ALLOC DA(office.memo(jan15)) F(outdd) SHR REUSE"
"EXECIO 10 DISKR indd (FINIS"
"EXECIO 10 DISKW outdd (FINIS"
```

To copy the same 10 lines of data to a list of compound variables with the stem "data.", substitute the following EXECIO commands.

```
"EXECIO 10 DISKR indd (FINIS STEM DATA."
"EXECIO 10 DISKW outdd (FINIS STEM DATA."
```

Note: When copying information to more than one member of a PDS, only one member of the PDS should be open at a time.

Adding 5 lines to the end of an existing sequential data set

To add 5 lines from an existing data set member named 'USERID.WEEKLY.INPUT(MAR28)' to the end of an existing sequential data set named 'USERID.YEARLY.OUTPUT', and use the ddnames MYINDD and MYOUTDD respectively, you could write the following instructions. Note the "MOD" attribute on the second allocation, which appends the 5 lines at the end of the data set rather than on top of existing data.

Appending 5 Lines of Data to an Existing Data Set:

```
"ALLOC DA(weekly.input(mar28)) F(myindd) SHR REUSE"
"ALLOC DA(yearly.output) F(myoutdd) MOD"
"EXECIO 5 DISKR myindd (FINIS"
"EXECIO 5 DISKW myoutdd (FINIS"
```

Note: Do not use the MOD attribute when allocating a member of a PDS to which you want to append information. You can use MOD only when appending information to a sequential data set. To append information to a member of a PDS, rewrite the member with the additional records added.

Copying Information to and from a List of Compound Variables

When copying information from a data set, you can store the information in the data stack, which is the default, or you can store the information in a list of compound variables. Similarly, when copying information back to a data set, you can remove information from the data stack, which is the default, or you can remove the information from a list of compound variables.

Copying Information from a Data Set to a List of Compound Variables

To copy an entire data set into compound variables with the stem `newvar.`, and then display the list, write the following instructions.

Copying an Entire Data Set into Compound Variables:

```
"ALLOC DA(old.data) F(indd) SHR REUSE"
"EXECIO * DISKR indd (STEM newvar."
DO i = 1 to newvar.0
  SAY newvar.i
END
```

When you want to copy a varying number of lines to compound variables, you can use a variable within the EXECIO command as long as the variable is not within quotation marks. For example, the variable `lines` can represent the number of lines indicated when the exec is run.

Copying a Varying Number of Lines into Compound Variables:

```
ARG lines
"ALLOC DA(old.data) F(indd) SHR REUSE"
"EXECIO" lines "DISKR indd (STEM newvar."
```

Copying Information from Compound Variables to a Data Set

To copy 10 compound variables with the stem `newvar.`, regardless of how many items are in the list, write the following instructions.

Note: An uninitialized compound variable will default to the value of its name. For example, if `newvar.9` and `newvar.10` do not contain values, the data set will receive the values `NEWVAR.9` and `NEWVAR.10`.

Copying from Compound Variables:

```
"ALLOC DA(new.data) F(outdd) LIKE(old.data) NEW"
"EXECIO 10 DISKW outdd (STEM NEWVAR."
```

Updating Information in a Data Set

You can update a single line of a data set with the EXECIO command, or you can update multiple lines. Use the DISKRU form of the EXECIO command to read information that you may subsequently update.

Note: The line written must be the same length as the line read. When a changed line is longer than the original line, information that extends beyond the original number of bytes is truncated and EXECIO sends a return code of 1. If lines must be made longer, write the data to a new data set. When a changed line is shorter than the original line, it is padded with blanks to attain the original line length.

Updating a single line

When updating a single line in a data set, it is more efficient to locate the line in advance and specify the update to it rather than read all the lines in the data set to the stack, locate and change the line, and then write all the lines back.

For example, you have a data set named 'DEPT5.EMPLOYEE.LIST' that contains a list of employee names, user IDs, and phone extensions.

```
Adams, Joe            JADAMS        5532
Crandall, Amy         AMY           5421
Devon, David          DAVIDD        5512
Garrison, Donna       DONNAG        5514
Leone, Mary           LEONE1        5530
Sebastian, Isaac      ISAAC         5488
```

To change a phone extension to 5500 on a particular line, such as Amy Crandall's, specify the line number, in this case, 2, and write the following instructions. Notice the "OLD" attribute on the allocation. The "OLD" attribute guarantees that no one else can use the data set while you are updating it.

Updating a Specific Line in a Data Set

```
"ALLOC DA('dept5.employee.list') F(updatedd) OLD"
"EXECIO 1 DISKRU updatedd 2 (LIFO"
PULL line
PUSH 'Crandall, Amy         AMY           5500'
"EXECIO 1 DISKW updatedd (FINIS"
"FREE F(updatedd)"
```

Updating multiple lines

To update multiple lines, you can issue more than one EXECIO command to the same data set. For example, to update Mary Leone's user ID in addition to Amy Crandall's phone extension, write the following instructions.

Updating Multiple Specific Lines in a Data Set

```
"ALLOC DA('dept5.employee.list') F(updatedd) OLD"
"EXECIO 1 DISKRU updatedd 2 (LIFO"
PULL line
PUSH 'Crandall, Amy         AMY           5500'
"EXECIO 1 DISKW updatedd"
"EXECIO 1 DISKRU updatedd 5 (LIFO"
PULL line
PUSH 'Leone, Mary           MARYL         5530'
"EXECIO 1 DISKW updatedd (FINIS"
 "FREE F(updatedd)"
```

When you issue multiple EXECIO commands to the same data set before closing it and do not specify a line number, the most current EXECIO command begins reading where the previous one left off. Thus to scan a data set one line at a time and allow a user at a terminal to update each line, you might write the following exec.

Example of Scanning Each Line for Update

```
/***************************** REXX ********************************/
/* This exec scans a data set whose name and size are specified by */
/* a user.  The user is given the option of changing each line as  */
/* it appears.  If there is no change to the line, the user presses*/
/* Enter key to indicate that there is no change.  If there is a   */
/* change to the line, the user types the entire line with the     */
/* change and the new line is returned to the data set.            */
/*******************************************************************/

  PARSE ARG name numlines  /* Get data set name and size from user */

  "ALLOC DA("name") F(updatedd) OLD"
  eof = 'NO'                        /* Initialize end-of-file flag */

  DO i = 1 to numlines WHILE eof = 'NO'
    "EXECIO 1 DISKRU updatedd" /* Queue the next line on the stack */
    IF RC = 2 THEN            /* Return code indicates end-of-file */
       eof = 'YES'
    ELSE
      DO
        PARSE PULL line
        SAY 'Please make changes to the following line.'
        SAY 'If you have no changes, press ENTER.'
        SAY line
        PARSE PULL newline
        IF newline = '' THEN NOP
        ELSE
          DO
            PUSH newline
            "EXECIO 1 DISKW updatedd"
          END
      END
  END
```

Additional Examples

EXECIO Example 1:

```
/***************************** REXX ********************************/
/* This exec reads from the data set allocated to INDD to find the */
/* first occurrence of the string "Jones".  Upper and lowercase    */
/* distinctions are ignored.                                       */
/*******************************************************************/
done = 'no'
lineno = 0

DO WHILE done = 'no'
  "EXECIO 1 DISKR indd"

   IF RC = 0 THEN                              /*  Record was read */
     DO
       PULL record
       lineno = lineno + 1                    /*  Count the record */
       IF INDEX(record,'JONES') \= 0 THEN
         DO
           SAY 'Found in record' lineno
           done = 'yes'
           SAY 'Record = ' record
         END
       ELSE NOP
     END
   ELSE
     done = 'yes'
END

EXIT 0
```

Figure 2: EXECIO Example 1

EXECIO Example 2:

```
/***************************** REXX ********************************/
/* This exec copies records from data set 'my.input' to the end of */
/* data set 'my.output'. Neither data set has been allocated to a  */
/* ddname. It assumes that the input data set has no null lines.   */
/*******************************************************************/
"ALLOC DA('my.input') F(indd) SHR REUSE"
"ALLOC DA('my.output') F(outdd) MOD REUSE"

SAY 'Copying …'

"EXECIO * DISKR indd (FINIS"
QUEUE ''  /* Insert a null line at the end to indicate end of file */
"EXECIO * DISKW outdd (FINIS"

SAY 'Copy complete.'
"FREE F(indd outdd)"

EXIT 0
```

Figure 3: EXECIO Example 2

EXECIO Example 3:

```
/***************************** REXX ********************************/
/* This exec reads five records from the data set allocated to     */
/* MYINDD starting with the third record. It strips trailing blanks*/
/* from the records, and then writes any record that is longer than*/
/* 20 characters. The file is not closed when the exec is finished.*/
/*******************************************************************/
"EXECIO 5 DISKR myindd 3"

DO i = 1 to 5
  PARSE PULL line
  stripline = STRIP(line,t)
  len = LENGTH(stripline)

  IF len > 20 THEN
    SAY 'Line' stripline 'is long.'
  ELSE NOP
END

/* The file is still open for processing */

EXIT 0
```

Figure 4: EXECIO Example 3

EXECIO Example 4:

```
/***************************** REXX ********************************/
/* This exec reads first 100 records (or until EOF) of the data    */
/* set allocated to INVNTORY.  Records are placed on data stack    */
/* in LIFO order. If fewer than 100 records are read, a message is */
/* issued.                                                         */
/*******************************************************************/
eofflag = 2                /* Return code to indicate end of file */

"EXECIO 100 DISKR invntory (LIFO"
return_code = RC

IF return_code = eofflag THEN
   SAY 'Premature end of file.'
ELSE
   SAY '100 Records read.'

EXIT return_code
```

Figure 5: EXECIO Example 4

EXECIO Example 5:

```
/***************************** REXX ********************************/
/* This exec illustrates the use of "EXECIO 0 ..." to open, empty, */
/* or close a file. It reads records from file indd, allocated     */
/* to 'sams.input.dataset', and writes selected records to file    */
/* outdd, allocated to 'sams.output.dataset'. In this example, the */
/* data set 'smas.input.dataset' contains variable-length records  */
/* (RECFM = VB).                                                   */
/*******************************************************************/
"FREE FI(outdd)"
"FREE FI(indd)"
"ALLOC FI(outdd) DA('sams.output.dataset') OLD REUSE"
"ALLOC FI(indd) DA('sams.input.dataset') SHR REUSE"
eofflag = 2                /* Return code to indicate end-of-file */
return_code = 0               /* Initialize return code           */
in_ctr = 0                    /* Initialize # of lines read       */
out_ctr = 0                   /* Initialize # of lines written    */

/*******************************************************************/
/* Open the indd file, but do not read any records yet.  All       */
/* records will be read and processed within the loop body.        */
/*******************************************************************/

"EXECIO 0 DISKR indd (OPEN"   /* Open indd                         */

/*******************************************************************/
/* Now read all lines from indd, starting at line 1, and copy      */
/* selected lines to outdd.                                        */
/*******************************************************************/

DO WHILE (return_code ¬ = eofflag) /* Loop while not end-of-file   */
  'EXECIO 1 DISKR indd'           /* Read 1 line to the data stack */
  return_code = rc                /* Save execio rc                */
  IF return_code = 0 THEN         /* Get a line ok?                */
   DO                             /* Yes                           */
      in_ctr = in_ctr + 1         /* Increment input line ctr      */
      PARSE PULL line.1           /* Pull line just read from stack*/
      IF LENGTH(line.1) > 10 then /* If line linger than 10 chars  */
       DO
         "EXECIO 1 DISKW outdd (STEM line." /* Write it to outdd   */
         out_ctr = out_ctr + 1        /* Increment output line ctr */
       END
   END
END "EXECIO 0 DISKR indd (FINIS"   /* Close the input file, indd   */

IF out_ctr > 0 THEN             /* Were any lines written to outdd?*/
  DO                               /* Yes.  So outdd is now open   */
```

Figure 6: EXECIO Example 5

EXECIO Example 5 (continued):

```
   /*****************************************************************/
   /* Since the outdd file is already open at this point, the       */
   /* following "EXECIO 0 DISKW ..." command will close the file,   */
   /* but will not empty it of the lines that have already been     */
   /* written. The data set allocated to outdd will contain out_ctr*/
   /* lines.                                                        */
   /*****************************************************************/

  "EXECIO 0 DISKW outdd (FINIS" /* Closes the open file, outdd     */
  SAY 'File outdd now contains ' out_ctr' lines.'
END
ELSE                          /* Else no new lines have been       */
                              /* written to file outdd             */
  DO                          /* Erase any old records from the file*/

   /*****************************************************************/
   /* Since the outdd file is still closed at this point, the       */
   /* following "EXECIO 0 DISKW …" command will open the file,   */
   /* write 0 records, and then close it.  This will effectively    */
   /* empty the data set allocated to outdd.  Any old records that */
   /* were in this data set when this exec started will now be      */
   /* deleted.                                                      */
   /*****************************************************************/

   "EXECIO 0 DISKW outdd (OPEN FINIS"  /*Empty the outdd file      */
   SAY 'File outdd is now empty.'
   END
"FREE FI(indd)"
"FREE FI(outdd)"
EXIT
```

Figure 7: EXECIO Example 5 (continued)

EXECIO Example 6:

```
/***************************** REXX ********************************/
/* This exec uses EXECIO to successively append the records from   */
/* 'sample1.data' and then from 'sample2.data' to the end of the   */
/* data set 'all.sample.data'.  It illustrates the effect of       */
/* residual data in STEM variables.  Data set 'sample1.data'       */
/* contains 20 records. Data set 'sample2.data' contains 10        */
/* records.                                                        */
/*******************************************************************/

"ALLOC FI(myindd1) DA('sample1.data') SHR REUSE"   /* input file 1 */
"ALLOC FI(myindd2) DA('sample2.data') SHR REUSE"   /* input file 2 */

"ALLOC FI(myoutdd) DA('all.sample.data') MOD REUSE" /* output append
                                file                            */

/*******************************************************************/
/* Read all records from 'sample1.data' and append them to the     */
/* end of 'all.sample.data'.                                       */
/*******************************************************************/

exec_RC = 0                       /* Initialize exec return code   */

"EXECIO * DISKR myindd1 (STEM newvar. FINIS"  /* Read all records  */

IF rc = 0 THEN                  /* If read was successful          */
  DO
  /*****************************************************************/
  /* At this point, newvar.0 should be 20, indicating 20 records   */
  /* have been read. Stem variables newvar.1, newvar.2, ... through*/
  /* newvar.20 will contain the 20 records that were read.         */
  /*****************************************************************/

    SAY "-----------------------------------------------------"
    SAY newvar.0 "records have been read from 'sample1.data': "
    SAY
    DO i = 1 TO newvar.0        /* Loop through all records        */
      SAY newvar.i              /* Display the ith record          */
    END

    "EXECIO" newvar.0 "DISKW myoutdd (STEM newvar." /* Write exactly
                                  the number of records read      */
```

Figure 8: EXECIO Example 6

EXECIO Example 6 (continued):

```
    IF rc = 0 THEN              /* If write was successful         */
      DO
        SAY
        SAY newvar.0 "records were written to 'all.sample.data'"
      END
    ELSE
      DO
        exec_RC = RC         /* Save exec return code           */
        SAY
        SAY "Error during 1st EXECIO … DISKW, return code is " RC
        SAY
      END
  END
ELSE
  DO
    exec_RC = RC               /* Save exec return code         */
    SAY
    SAY "Error during 1st EXECIO … DISKR, return code is " RC
    SAY
  END

  IF exec_RC = 0 THEN        /* If no errors so far... continue */
    DO
    /*****************************************************************/
    /* At this time, the stem variables newvar.0 through newvar.20 */
    /* will contain residual data from the previous EXECIO. We     */
    /* issue the "DROP newvar." instruction to clear these residual*/
    /* values from the stem.                                        */
    /*****************************************************************/
    DROP newvar.               /* Set all stem variables to their
                                  uninitialized state              */
    /*****************************************************************/
    /* Read all records from 'sample2.data' and append them to the */
    /* end of 'all.sample.data'.                                    */
    /*****************************************************************/
    "EXECIO * DISKR myindd2 (STEM newvar. FINIS" /*Read all records*/
     IF rc = 0 THEN             /* If read was successful          */
      DO
      /*************************************************************/
      /* At this point, newvar.0 should be 10, indicating 10       */
      /* records have been read. Stem variables newvar.1, newvar.2,*/
      /* ... through newvar.10 will contain the 10 records. If we  */
      /* had not cleared the stem newvar. with the previous DROP   */
      /* instruction, variables newvar.11 through newvar.20 would  */
      /* still contain records 11 through 20 from the first data   */
      /* set. However, we would know that these values were not    */
      /* read by the last EXECIO DISKR since the current newvar.0  */
      /* variable indicates that only 10 records were read by      */
      /* that last EXECIO.                                         */
      /*************************************************************/
```

Figure 9: EXECIO Example 6 (continued)

EXECIO Example 6 (continued):

```
        SAY
        SAY
        SAY "-----------------------------------------------------"
        SAY newvar.0 "records have been read from 'sample2.data': "
        SAY
        DO i = 1 TO newvar.0    /* Loop through all records        */
          SAY newvar.i          /* Display the ith record          */
        END

        "EXECIO" newvar.0 "DISKW myoutdd (STEM newvar." /* Write
                               exactly the number of records read */
        IF rc = 0 THEN          /* If write was successful         */
         DO
           SAY
           SAY newvar.0 "records were written to 'all.sample.data'"
         END
        ELSE
          DO
            exec_RC = RC      /* Save exec return code          */
            SAY
            SAY "Error during 2nd EXECIO …DISKW, return code is " RC
            SAY
          END
      END
    ELSE
      DO
        exec_RC = RC           /* Save exec return code         */
        SAY
        SAY "Error during 2nd EXECIO … DISKR, return code is " RC
        SAY
      END
  END

"EXECIO 0 DISKW myoutdd (FINIS"    /* Close output file            */

"FREE FI(myindd1)"
"FREE FI(myindd2)"
"FREE FI(myoutdd)"
 EXIT 0
```

Figure 10: EXECIO Example 6 (continued)

Chapter 13. Using REXX in TSO/E and Other MVS Address Spaces

This chapter describes how to use REXX in TSO/E and in non-TSO/E address spaces in MVS. It also briefly describes the concept of a language processor environment.

Services Available to REXX Execs

This book, until now, has described writing and running REXX execs in the TSO/E address space. Besides TSO/E, execs can run in other address spaces within MVS. Where an exec can run is determined by the types of services the exec requires. There are services that are available to an exec that runs in any address space, TSO/E or non-TSO/E; and there are more specific services available only in a TSO/E address space. The following table lists all the services and where they are available.

Service	Non-TSO/E Address Space	TSO/E Address Space
REXX language instructions — These instructions are used throughout this book. For a description of each one, see *z/OS TSO/E REXX Reference*.	X	X
Built-in functions — A brief description of each built-in function appears in "Built-In Functions" on page 58. A longer description appears in *z/OS TSO/E REXX Reference*.	X	X
TSO/E REXX commands — These commands consist of: • Data stack commands — For more information, see Chapter 11, "Storing Information in the Data Stack," on page 125.		
• DELSTACK	X	X
• DROPBUF	X	X
• MAKEBUF	X	X
• NEWSTACK	X	X
• QBUF	X	X
• QELEM	X	X
• QSTACK	X	X
• Other commands —		
• EXECIO — controls I/O processing	X	X
• EXECUTIL — changes how an exec runs		X
• Immediate commands:		
• HI (from attention mode only)		X

Service	Non-TSO/E Address Space	TSO/E Address Space
• HE (from attention mode only)		X
• HT (from attention mode only)		X
• RT (from attention mode only)		X
• TE	X	X
• TS	X	X
• SUBCOM — queries the existence of a host command environment	X	X
TSO/E commands — All TSO/E commands, both authorized and unauthorized can be issued from an exec that runs in a TSO/E address space. For a description of these commands, see *z/OS TSO/E Command Reference*.		X
TSO/E External Functions:		
• GETMSG — retrieves system messages issued during an extended MCS console session		X
• LISTDSI — returns data set attributes		X
• MSG — controls the display of messages for TSO/E commands		X
• MVSVAR — returns information about MVS, TSO/E and the current session	X	X
• OUTTRAP — traps lines of TSO/E command output		X
• PROMPT — controls prompting for TSO/E interactive commands		X
• SETLANG — controls the language in which REXX messages are displayed	X	X
• STORAGE — retrieves and optionally changes the value in a storage address	X	X

Service	Non-TSO/E Address Space	TSO/E Address Space
• SYSCPUS — returns information about CPUs that are online	X	X
• SYSDSN — returns information about the availability of a data set		X
• SYSVAR — returns information about the user, the terminal, the exec, and the system		X
Interaction with CLISTs — Execs and CLISTs can call each other and pass information back and forth. For more information, see "Running an Exec from a CLIST" on page 162.		X
ISPF and ISPF/PDF services — An exec that is invoked from ISPF can use that dialog manager's services.		X

Running Execs in a TSO/E Address Space

Earlier sections in this book described how to run an exec in TSO/E explicitly and implicitly in the **foreground**. When you run an exec in the foreground, you do not have use of your terminal until the exec completes. Another way to run an exec is in the **background**, which allows you full use of your terminal while the exec runs.

Running an Exec in the Foreground

Interactive execs and ones written that involve user applications are generally run in the foreground. You can invoke an exec in the foreground in the following ways:

- Explicitly with the EXEC command. For more information, see "Running an Exec Explicitly" on page 14.
- Implicitly by member name if the PDS containing the exec was previously allocated to SYSPROC or SYSEXEC. (Your installation might have a different name for the system file that contains execs. For the purposes of this book, it is called SYSEXEC.) For more information, see "Running an Exec Implicitly" on page 15 and Appendix A, "Allocating Data Sets," on page 171.
- From another exec as an external function or subroutine, as long as both execs are in the same PDS or the PDSs containing the execs are allocated to a system file, for example SYSPROC or SYSEXEC. For more information about external functions and subroutines, see Chapter 6, "Writing Subroutines and Functions," on page 65.
- From a CLIST or other program. For more information, see "Running an Exec from a CLIST" on page 162.

Things to Consider When Allocating to a System File (SYSPROC or SYSEXEC)

Allocating a partitioned data set containing execs to a system file allows you to:

- Run execs implicitly - After a PDS is allocated to a system file, you can run the exec by simply entering the member name, which requires fewer keystrokes and is therefore faster to invoke.
- Invoke user-written external functions and subroutines written in REXX that are in PDSs also allocated to SYSEXEC or SYSPROC.
- Control search order - You can concatenate the data sets within the file to control search order. This is useful in testing a version of an exec placed earlier in the search order than the original version.

- Compression - In certain situations a REXX exec will be compressed to optimize usage of system storage. These situations can arise only when the exec is stored in either SYSPROC or the application-level CLIST file using the ALTLIB command. The compression removes comment text between the comment delimiters /* and */, removes leading and trailing blanks, and replaces blank lines with null lines. Blanks and comments within literal strings or DBCS strings are not removed. If the system finds the characters "SOURCELINE" outside of a comment, the exec is not compressed. Additionally, if you do not want an exec to be compressed, you can allocate the exec to the CLIST user-level file, or any of the levels used for execs.
- Improve performance - Depending on your installation's setup, you can affect the performance of execs you run by allocating the data sets that contain them to either SYSEXEC or SYSPROC. More about this technique appears in the following sections on allocating to a specific system file.

Allocating to SYSEXEC

SYSEXEC is a system file that can contain execs only. SYSEXEC precedes SYSPROC in the search order. Therefore execs in PDSs allocated to SYSEXEC are retrieved more rapidly than execs in PDSs allocated to SYSPROC.

Allocating to SYSPROC

SYSPROC is a system file that originally contained only CLISTs written for applications or for an individual's use. SYSPROC now can also contain execs as long as the execs are distinguishable from CLISTs.

The SYSEXEC file is searched first, followed by SYSPROC. If your installation uses a large number of CLISTs that are in data sets allocated to SYSPROC and you do not have a large number of REXX execs, you may want to use SYSPROC only and not use SYSEXEC. To use SYSPROC only, a system programmer can change the search order on an installation-wide basis, or an individual can change the search order using the EXECUTIL SEARCHDD(NO) command. You can issue the EXECUTIL SEARCHDD(NO) command directly from the terminal, from an exec or CLIST, and from the JCL input stream run in TSO/E background. The ALTLIB command can also affect search order. For general information about ALTLIB, see Appendix B, “Specifying Alternate Libraries with the ALTLIB Command,” on page 181. For more information about the EXECUTIL and ALTLIB commands, see *z/OS TSO/E Command Reference*.

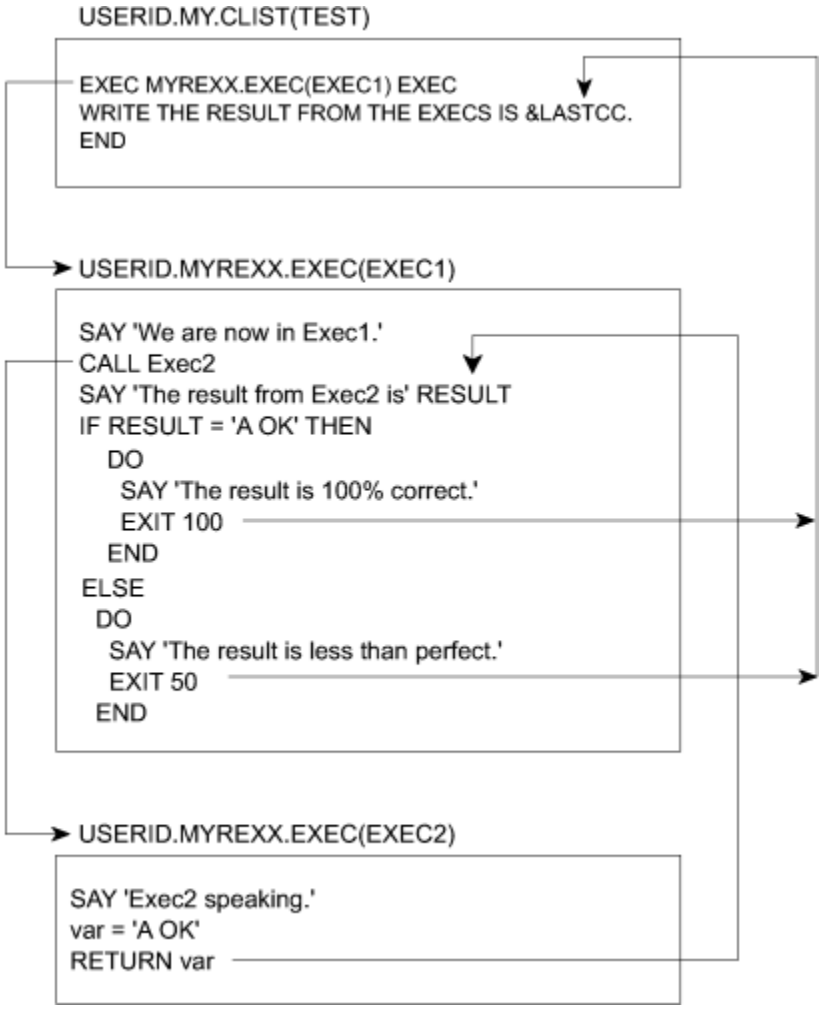
Running an Exec from a CLIST

A CLIST can invoke an exec with the EXEC command explicitly or implicitly. If it invokes an exec implicitly, the exec must be in a PDS allocated to SYSEXEC or SYSPROC. The CLIST that invokes the exec does not have to be allocated to SYSPROC. After the invoked exec and other programs it might call complete, control returns to the CLIST instruction following the invocation.

Similarly, an exec can invoke a CLIST with the EXEC command explicitly or implicitly. If it invokes a CLIST implicitly, the CLIST must be in a PDS allocated to SYSPROC, yet the exec does not have to be in a PDS allocated to a system file.

Note: Execs and CLISTs cannot access each other's variables and GLOBAL variables cannot be declared in a CLIST that is invoked from an exec.

The following examples demonstrate how a CLIST invokes an exec and how a number is returned to the invoking CLIST. The CLIST named TEST explicitly executes an exec named EXEC1. EXEC1 calls EXEC2, which returns the result "A OK". EXEC1 then returns to the CLIST with a numeric return code of 100 if information was passed correctly and 50 if information was not passed correctly.



The results from this series of programs is as follows:

```
We are now in Exec1.
Exec2 speaking.
The result from Exec2 is A OK
The result is 100% correct.
THE RESULT FROM THE EXECS IS 100
```

Sending a Return Code Back to the Calling CLIST

As demonstrated in the previous example, an exec can return a number to a CLIST with the EXIT instruction followed by the number or a variable representing the number. The CLIST receives the number in the variable &LASTCC.

When an exec invokes a CLIST, the CLIST can return a number to the exec by the EXIT CODE() statement with the number to be returned enclosed in parentheses after CODE. The exec receives the number in the REXX special variable RC.

Note: &LASTCC is set after each CLIST statement or command executes as compared to RC, which is set after each command executes. To save the values of each special variable, set a new variable with the value at the point where you want the special variable value saved.

In the following two examples, exec USERID.MYREXX.EXEC(TRANSFER) passes an argument to CLIST USERID.MY.CLIST(RECEIVE), and the CLIST returns a number through the CODE parameter of the EXIT statement.

USERID.MYREXX.EXEC(TRANSFER)

```
/***************************** REXX ******************************/
/* This exec passes a percent sign to a CLIST and depending on    */
/* the success of the transfer, the CLIST returns 100 (if it was  */
/* successful) or 50 (if it was not successful).                  */
/*****************************************************************/

SAY 'We are about to execute CLIST RECEIVE and pass it % '

"EXEC my.clist(receive) '%' clist"

SAY 'We have returned from the CLIST.'
IF RC = 100 THEN
  SAY 'The transfer was a success.'
ELSE
  SAY 'The transfer was not a success.'
```

USERID.MY.CLIST(RECEIVE)

```
PROC 1 &VAR
IF &VAR = % THEN SET SUCCESS = 100
ELSE SET SUCCESS = 50
EXIT CODE(&SUCCESS)
```

Running an Exec in the Background

Execs run in the background are processed when higher priority programs are not using the system. Background processing does not interfere with a person's use of the terminal. You can run time-consuming and low priority execs in the background, or execs that do not require terminal interaction.

Running an exec in the background is the same as running a CLIST in the background. The program IKJEFT01 sets up a TSO/E environment from which you can invoke execs and CLISTs and issue TSO/E commands. For example, to run an exec named SETUP contained in a partitioned data set USERID.MYREXX.EXEC, submit the following JCL.

Example of JCL to Run an Exec in the Background

```
//USERIDA   JOB  'ACCOUNT,DEPT,BLDG','PROGRAMMER NAME',
// CLASS=J,MSGCLASS=C,MSGLEVEL=(1,1)
//*
//TMP       EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=4096K
//SYSEXEC   DD   DSN=USERID.MYREXX.EXEC,DISP=SHR
//SYSTSPRT  DD   SYSOUT=A
//SYSTSIN   DD   *
 %SETUP
/*
//
```

The EXEC statement defines the program as IKJEFT01. In a DD statement, you can assign one or more PDSs to the SYSEXEC or SYSPROC system file. The SYSTSPRT DD allows you to print output to a specified data set or a SYSOUT class. In the input stream, after the SYSTSIN DD, you can issue TSO/E commands and invoke execs and CLISTs.

The preceding example must be written in a fixed block, 80 byte record data set. To start the background job, issue the SUBMIT command followed by the data set name, for example, REXX.JCL.

```
SUBMIT rexx.jcl
```

For more information about running jobs in the background, see *z/OS TSO/E User's Guide*.

Running Execs in a Non-TSO/E Address Space

Because execs that run in a non-TSO/E address space cannot be invoked by the TSO/E EXEC command, you must use other means to run them. Ways to run execs outside of TSO/E are:

- From a high level program using the IRXEXEC or IRXJCL processing routines.
- From MVS batch with JCL that specifies IRXJCL in the EXEC statement.

TSO/E provides the TSO/E environment service, IKJTSOEV. Using IKJTSOEV, you can create a TSO/E environment in a non-TSO/E address space. You can then run REXX execs in the environment and the execs can contain TSO/E commands, external functions, and services that an exec running in a TSO/E address space can use. For information about the TSO/E environment service and how to run REXX execs within the environment, see *z/OS TSO/E Programming Services*.

Using an Exec Processing Routine to Invoke an Exec from a Program

To invoke an exec from a high-level language program running in an MVS address space, use one of the exec processing routines (IRXEXEC or IRXJCL). If you use IRXEXEC, you must specify parameters that define the exec to be run and supply other related information. For more information, see *z/OS TSO/E REXX Reference*.

You can also use an exec processing routine to invoke an exec in a TSO/E address space. Two reasons to use them in TSO/E are:

- To pass more than one argument to an exec. When invoking an exec implicitly or explicitly, you can pass only one argument string. With IRXEXEC, you can pass multiple arguments.
- To call an exec from a program other than a CLIST or exec.

Using IRXJCL to Run an Exec in MVS Batch

To run a REXX exec in MVS batch, you must specify program IRXJCL in the JCL EXEC statement. SYSEXEC is the default load DD. Running an exec in MVS batch is similar in many ways to running an exec in the TSO/E background, however, there are significant differences. One major difference is that the exec running in MVS batch cannot use TSO/E services, such as TSO/E commands and most of the TSO/E external functions. Additional similarities and differences appear in “Summary of TSO/E Background and MVS Batch” on page 167.

The following series of examples show how an MVS batch job named USERIDA invokes a REXX exec in a PDS member named USERID.MYREXX.EXEC(JCLTEST). The member name, JCLTEST, is specified as the first word after the PARM parameter of the EXEC statement. Two arguments, TEST and IRXJCL, follow the member name. Output from the exec goes to an output data set named USERID.IRXJCL.OUTPUT, which is specified in the SYSTSPRT DD statement. The SYSTSIN DD statement supplies the exec with three lines of data in the input stream. This exec also uses EXECIO to write a 1-line timestamp to the end of the sequential data set USERID.TRACE.OUTPUT, which is allocated in the OUTDD statement.

USERID.JCL.EXEC

```
//USERIDA  JOB  'ACCOUNT,DEPT,BLDG','PROGRAMMER NAME',
// CLASS=J,MSGCLASS=H,MSGLEVEL=(1,1)
//*
//MVSBACH  EXEC PGM=IRXJCL,
//              PARM='JCLTEST Test IRXJCL'
//*                   |     | |         |
//* Name of exec      <-----> |         |
//* Argument                  <--------->
//OUTDD    DD   DSN=USERID.TRACE.OUTPUT,DISP=MOD
//SYSTSPRT DD   DSN=USERID.IRXJCL.OUTPUT,DISP=OLD
//SYSEXEC  DD   DSN=USERID.MYREXX.EXEC,DISP=SHR
//SYSTSIN  DD   *
 First line of data
 Second line of data
 Third line of data
/*
//
```

USERID.MYREXX.EXEC(JCLTEST)

```
 /****************************** REXX ******************************/
 /* This exec receives input from its invocation in JCL.EXEC, pulls*/
 /* data from the input stream and sends back a condition code of  */
 /* 137.                                                           */
 /******************************************************************/
 TRACE error
 SAY 'Running exec JCLTEST'
 ADDRESS MVS
 PARSE ARG input
 SAY input
 DATA = start

 DO UNTIL DATA = '
   PARSE PULL data             /* pull data from the input stream */
   SAY data
 END

 /******************************************************************/
 /* Now use EXECIO to write a timestamp to the sequential          */
 /* data set that was allocated to the OUTDD file by the JCL       */
 /* used to invoke this exec.                                      */
 /******************************************************************/
 OUTLINE.1 = 'Exec JCLTEST has ended at' TIME()
 "EXECIO 1 DISKW OUTDD (STEM OUTLINE. FINIS"     /* Write the line */

 SAY 'Leaving exec JCLTEST'
 EXIT 137                       /* send a condition code of 137    */
```

USERID.TRACE.OUTPUT

```
Exec JCLTEST has ended at 15:03:06
```

USERID.IRXJCL.OUTPUT

```
 Running exec JCLTEST
 Test IRXJCL
  First line of data
  Second line of data
  Third line of data

 Leaving exec JCLTEST
```

Segment of Output from the JCL Listing

```
ALLOC. FOR USERIDA MVSBACH
224 ALLOCATED TO OUTDD
954  ALLOCATED TO SYSTSPRT
7E0  ALLOCATED TO SYSEXEC
JES2 ALLOCATED TO SYSTSIN
USERIDA MVSBACH - STEP WAS EXECUTED - COND CODE 0137
  USERID.TRACE.OUTPUT                          KEPT
  VOL SER NOS= TSO032.
  USERID.IRXJCL.OUTPUT                         KEPT
  VOL SER NOS= TSO032.
  USERID.MYREXX.EXEC                           KEPT
  VOL SER NOS= TSO001.
  JES2.JOB28359.I0000101                       SYSIN
STEP / MVSBACH / START 88167.0826
STEP / MVSBACH / STOP  88167.0826 CPU    0MIN 00.16SEC SRB   ...
 JOB / USERIDA / START 88167.0826
 JOB / USERIDA / STOP  88167.0826 CPU    0MIN 00.16SEC SRB   ...
```

Using the Data Stack in TSO/E Background and MVS Batch

When an exec runs in the TSO/E background or MVS batch, it has the same use of the data stack as an exec that runs in the TSO/E foreground. The PULL instruction, however, works differently when the data stack is empty. In the TSO/E foreground, PULL goes to the terminal for input. In the TSO/E background and MVS batch, PULL goes to the input stream as defined by ddname SYSTSIN. When SYSTSIN has no data, the PULL instruction returns a null. If the input stream has no data and the PULL instruction is in a loop, the exec can result in an infinite loop.

Summary of TSO/E Background and MVS Batch

CAPABILITIES

TSO/E BACKGROUND (IKJEFT01)	MVS BATCH (IRXJCL)
Execs run without terminal interaction.	Execs run without terminal interaction.
Execs can contain: • REXX instructions • Built-in functions • TSO/E REXX commands • TSO/E commands • TSO/E external functions	Execs can contain: • REXX instructions • Built-in functions • TSO/E REXX commands • The TSO/E external functions, STORAGE and SETLANG
Execs are invoked through the PARM parameter on the EXEC statement and through explicit or implicit use of the EXEC command in the input stream.	Execs are invoked through the PARM parameter on the EXEC statement. The first word on the PARM parameter is the member name of the PDS to be invoked. Following words are arguments to be passed.
Information in the input stream is processed as TSO/E commands and invocations of execs and CLISTs.	Information in the input stream is processed as input data for the exec that is running.
Output sent to a specified output data set or to a SYSOUT class.	Output sent to a specified output data set or to a SYSOUT class.

TSO/E BACKGROUND (IKJEFT01)	MVS BATCH (IRXJCL)
Messages are displayed in the output file.	Messages may appear in two places; the JCL output listing and in the output file. To suppress messages in the output file, use the TRACE OFF instruction.

REQUIREMENTS

TSO/E BACKGROUND (IKJEFT01)	MVS BATCH (IRXJCL)
The default DDs are SYSTSPRT and SYSTSIN.	The default DDs are SYSTSPRT and SYSTSIN.
Initiated by executing program IKJEFT01.	Initiated by executing program IRXJCL.
JCL should be written in a fixed block, 80-byte record data set.	JCL should be written in a fixed block, 80-byte record data set.
Exec that is invoked can be either a member of a PDS or a sequential data set.	Exec that is invoked must be a member of a PDS.
Data set may be allocated to either SYSEXEC or SYSPROC.	Data set must be allocated to the SYSEXEC DD.

Defining Language Processor Environments

Before an exec can be processed, a language processor environment must exist. A language processor environment defines the way a REXX exec is processed and how it accesses system services. Because MVS contains different types of address spaces and each one accesses services a different way, REXX in TSO/E provides three default parameters modules that define language processor environments. They are:

- IRXTSPRM - for TSO/E
- IRXPARMS - for non-TSO/E
- IRXISPRM - for ISPF

The defaults are set by TSO/E but they can be modified by a system programmer.

What is a Language Processor Environment?

A language processor environment defines characteristics, such as:

- The search order used to locate commands and external routines
- The ddnames for reading and writing data and from which execs are loaded
- The valid host command environments and the routines that process commands in each host command environment
- The function packages (user, local, and system) that are available in the environment and the entries in each package
- Whether execs running in the environment can use the data stack
- The names of routines that handle system services, such as I/O operations, loading of an exec, obtaining and freeing storage, and data stack requests

Note: A language processor environment is different from a host command environment. The language processor environment is the environment in which a REXX exec runs. The host command environment is the environment to which the language processor passes commands for execution. The valid host command environments are defined by the language processor environment.

For more information about defining language processor environnments, see *z/OS TSO/E REXX Reference*.

Customizing a language processor environment

An individual or an installation can customize a language processor environment in two ways:

- Change the values in the three default parameters modules, IRXTSPRM, IRXISPRM, and IRXPARMS.
- Call an initialization routine IRXINIT and specifying parameters to change default parameters.

For more information about customizing a language processor environment, see *z/OS TSO/E REXX Reference*.

Appendix A. Allocating Data Sets

Execs can be stored in either sequential data sets or partitioned data sets (PDSs). A sequential data set contains only one exec, while a PDS can contain one or more execs. In a PDS, each exec is a member and has a unique member name. When a PDS consists entirely of execs, it is called an exec library.

Exec libraries make execs easy to maintain and execute. Your installation can keep commonly used execs in a system library and you can keep your own execs in a private exec library. To learn important information about data sets at your installation, use the "Preliminary Checklist" on page 172.

What is Allocation?

Before you can store execs in a data set, you must create the data set by allocation. Allocation can mean different things depending on your purpose. In this book allocation means two things:

- **Creating a new data set** in which to store REXX execs. You can create a new data set with the ISPF/PDF UTILITIES option or with the TSO/E ALLOCATE command.

  Checklists for creating a data set appear in:

  - "Checklist #1: Creating and Editing a Data Set Using ISPF/PDF" on page 173
  - "Checklist #2: Creating a Data Set with the ALLOCATE Command" on page 176
- **Accessing an existing data set** and associating it, and possibly other data sets, to a system file. Allocating a data set to a system file (SYSEXEC or SYSPROC) enables you to execute the execs **implicitly** by simply typing their member names. When more than one PDS is specified in the allocation, they are **concatenated** or logically connected in the order in which they are specified.

  The association of the PDS to the system file remains for the duration of your terminal session or until another ALLOCATE command alters the association.

  You can allocate a data set to a system file in the foreground with the TSO/E ALLOCATE command or in the background with a JCL DD statement. You *cannot* use ISPF/PDF to allocate a data set to a system file.

  Checklists for allocating a data set to SYSEXEC and SYSPROC appear in:

  - "Checklist #3: Writing an Exec that Sets up Allocation to SYSEXEC" on page 176
  - "Checklist #4: Writing an Exec that Sets up Allocation to SYSPROC" on page 178

Where to Begin

Before creating a PDS in which to store your execs, use the "Preliminary Checklist" on page 172 to find out information that you can use to make your PDS compatible with other PDSs at your installation. Then create a PDS with either "Checklist #1: Creating and Editing a Data Set Using ISPF/PDF" on page 173 or "Checklist #2: Creating a Data Set with the ALLOCATE Command" on page 176.

After the PDS is created, if you want to be able to invoke those execs implicitly during that terminal session, you must allocate the PDS to a system file (SYSEXEC or SYSPROC). The allocation is temporary and must be established for each terminal session. One way to establish the allocation is to write a setup exec that automatically executes when you log on. Information about how to write a setup exec is in "Checklist #3: Writing an Exec that Sets up Allocation to SYSEXEC" on page 176 and "Checklist #4: Writing an Exec that Sets up Allocation to SYSPROC" on page 178. If you do not know which checklist to use, use Checklist #3.

The following checklists assume that the defaults shipped with TSO/E have not been altered by your installation. Also if your installation changes system allocations after you have used the checklists to set up your private allocation, you might need to use the checklists again to keep your allocations up-to-date.

Preliminary Checklist

1. Issue the LISTALC STATUS command to see the names of all data sets allocated to SYSEXEC and SYSPROC.:

   To see what data sets are already defined to SYSEXEC and SYSPROC at your installation, issue the LISTALC command with the STATUS keyword.

   ```
   READY
   listalc status
   ```

   You then see several screens of data set names that might look something like the following. Scroll until you find SYSEXEC and SYSPROC.

   ```
    --DDNAME---DISP--
    ICQ.INFOCTR.LOAD.
      STEPLIB  KEEP
    CATALOG.VTSO022
      SYS00006 KEEP,KEEP
    CATALOG.VTSO028
               KEEP,KEEP
    ISP.PHONE.EXEC
      SYSEXEC  KEEP
    ICQ.INFOCTR.ICQCLIB
      SYSPROC  KEEP
    SYS1.TSO.CLIST
               KEEP
    ISP.ISPF.CLISTS
               KEEP
   ```

   In this example, one data set ISP.PHONE.EXEC is allocated to SYSEXEC, and three data sets ICQ.INFOCTR.ICQCLIB, SYS1.TSO.CLIST, and ISP.ISPF.CLISTS are allocated to SYSPROC. (When a space appears below the data set name, the data set is allocated to the previously-specified file (DDNAME)).
2. Write down the names of the data sets at your installation that are allocated to SYSEXEC:
   - First data set: _____
   - Remaining data sets: _____
     _____
     _____
     _____
3. Write down the names of the data sets at your installation that are allocated to SYSPROC:
   - First data set: _____
   - Remaining data sets: _____
     _____
     _____
     _____
4. Issue the LISTDS command for the first data set in each system file to display the record format, logical record length, and block size:

   To see the attributes of data sets used at your installation, issue the LISTDS command for the first data set in each system file concatenation to display something like the following:

   ```
    READY
     LISTDS 'sysexec.first.exec'

     SYSEXEC.FIRST.EXEC
   ```

```
  --RECFM-LRECL-BLKSIZE-DSORG
    VB    255  5100    PO
  --VOLUMES--
    TSO026

 READY
  LISTDS 'sysproc.first.clist'

  SYSPROC.FIRST.CLIST
  --RECFM-LRECL-BLKSIZE-DSORG
    FB     80   19040    PO
  --VOLUMES--
    TSOL07
```

5. Write down the attributes of the first data set in your SYSEXEC concatenation:
   - RECFM = _____
   - LRECL = _____
   - BLKSIZE = _____
6. Write down the attributes of the first data set in your SYSPROC concatenation:
   - RECFM = _____
   - LRECL = _____
   - BLKSIZE = _____

> **Note:** Save this information for use with the following checklists.

Checklist #1: Creating and Editing a Data Set Using ISPF/PDF

1. Select the ISPF/PDF DATASET UTILITIES option (option 3.2):

   From the ISPF/PDF Primary Option Menu, select the UTILITIES option (option 3) and press the Enter key.

```
------------------------ ISPF/PDF PRIMARY OPTION MENU  -------------------------
OPTION ===> 3
                                                          USERID   - YOURID
 0  ISPF PARMS  - Specify terminal and user parameters    TIME     - 12:47
 1  BROWSE      - Display source data or output listings  TERMINAL - 3277
 2  EDIT        - Create or change source data            PF KEYS  - 12
 3  UTILITIES   - Perform utility functions
 4  FOREGROUND  - Invoke language processors in foreground
 5  BATCH       - Submit job for language processing
 6  COMMAND     - Enter TSO command or CLIST
 7  DIALOG TEST - Perform dialog testing
 8  LM UTILITIES- Perform library administrator utility functions
 9  IBM PRODUCTS- Additional IBM program development products
 C  CHANGES     - Display summary of changes for this release
 T  TUTORIAL    - Display information about ISPF/PDF
 X  EXIT        - Terminate ISPF using log and list defaults

Enter END command to terminate ISPF.
```

   Then, select the DATASET option (option 2) and press the Enter key.

```
-------------------------  UTILITY SELECTION MENU  ----------------------------
OPTION ===> 2

    1  LIBRARY     - Compress or print data set.  Print index listing.
                       Print, rename, delete, or browse members
    2  DATASET     - Allocate, rename, delete, catalog, uncatalog, or
                       display information of an entire data set
    3  MOVE/COPY   - Move, copy, or promote members or data sets
    4  DSLIST      - Print or display (to process) list of data set names
                       Print or display VTOC information
    5  RESET       - Reset statistics for members of ISPF library
    6  HARDCOPY    - Initiate hardcopy output
    8  OUTLIST     - Display, delete, or print held job output
    9  COMMANDS    - Create/change an application command table
   10  CONVERT     - Convert old format menus/messages to new format
   11  FORMAT      - Format definition for formatted data Edit/Browse
   12  SUPERC      - Compare data sets (Standard dialog)
   13  SUPERCE     - Compare data sets (Extended dialog)
   14  SEARCH-FOR  - Search data sets for strings of data
    D  DATA MGMT   - Data Management Tools
```

2. Specify a new data set name on the Data Set Utility panel and type A on the OPTION line:

   On the next panel that appears, type the name of the data set you want to allocate, for example USERID.REXX.EXEC, and enter A on the OPTION line.

```
------------------------------  DATA SET UTILITY  ----------------------------
OPTION ===> a

    A - Allocate new data set         C - Catalog data set
    R - Rename entire data set        U - Uncatalog data set
    D - Delete entire data set        S - Data set information (short)
    blank - Data set information

ISPF LIBRARY:
    PROJECT ===>  userid
    GROUP   ===>  rexx
    TYPE    ===>  exec

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
    DATA SET NAME ===>
    VOLUME SERIAL ===>   left 0If not cataloged, required for option "C")

DATA SET PASSWORD ===>          (If password protected)
```

3. Specify the data set attributes on the Allocate New Data Set panel:

   After you name the data set, a panel appears on which you define the attributes of the data set. Use the attributes recommended by your installation for REXX libraries, and include the record format (RECFM), record length (LRECL), and block size (BLKSIZE) from the appropriate system file from the Preliminary Checklist #“5” on page 173. If you are unsure about which system file is appropriate, use the values from SYSEXEC.

   If your installation has no attribute recommendations and you have no attributes from the Preliminary Checklist, you can use the following attributes on the ISPF/PDF Allocate New Data Set panel:

```
----------------------------  ALLOCATE NEW DATA SET  ---------------------------
COMMAND ===>

DATA SET NAME:  USERID.REXX.EXEC

   VOLUME SERIAL     ===>         (Blank for authorized default volume)*
   GENERIC UNIT      ===>         (Generic group name or unit address)*
   SPACE UNITS       ===> blks           (BLKS, TRKS or CYLS)
   PRIMARY QUAN      ===> 50             (in above units)
   SECONDARY QUAN    ===> 20             (in above units)
   DIRECTORY BLOCKS  ===> 10             (Zero for sequential data set)
   RECORD FORMAT     ===> VB
   RECORD LENGTH     ===> 255
   BLOCK SIZE        ===> 6120
   EXPIRATION DATE   ===>              (YY/MM/DD
                                        YY.DDD in julian form
                                        DDDD for retention period in days
                                        or blank)


   ( * Only one of these fields may be specified)
```

4. Edit a member of the newly created PDS by selecting the EDIT option (option 2) and specifying the PDS name with a member name:

   After you have allocated a PDS, you can press the RETURN PF key (PF4) to return to the Primary Option Menu and begin an edit session. Select the EDIT option (option 2) from the ISPF/PDF Primary Option Menu.

```
------------------------ ISPF/PDF PRIMARY OPTION MENU  ----------------------
OPTION ===> 2
                                                        USERID   - YOURID
 0  ISPF PARMS  - Specify terminal and user parameters    TIME     - 12:47
 1  BROWSE      - Display source data or output listings  TERMINAL - 3277
 2  EDIT        - Create or change source data            PF KEYS  - 12
 3  UTILITIES   - Perform utility functions
 4  FOREGROUND  - Invoke language processors in foreground
 5  BATCH       - Submit job for language processing
 6  COMMAND     - Enter TSO command or CLIST
 7  DIALOG TEST - Perform dialog testing
 8  LM UTILITIES- Perform library administrator utility functions
 9  IBM PRODUCTS- Additional IBM program development products
 C  CHANGES     - Display summary of changes for this release
 T  TUTORIAL    - Display information about ISPF/PDF
 X  EXIT        - Terminate ISPF using log and list defaults

Enter END command to terminate ISPF.
```

   Then, specify the data set name and member name on the Edit - Entry Panel. In the example that follows, the member name is **timegame**.

```
-----------------------------  EDIT - ENTRY PANEL  ---------------------------
COMMAND ===>

ISPF LIBRARY:
   PROJECT ===> userid
   GROUP   ===> rexx     ===>            ===>            ===>
   TYPE    ===> exec
   MEMBER  ===> timegame    (Blank for member selection list)

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
   DATA SET NAME     ===>
   VOLUME SERIAL     ===>       (If not cataloged)

   DATA SET PASSWORD ===>       (If password protected)

   PROFILE NAME      ===>       (Blank defaults to data set type)

   INITIAL MACRO     ===>       LOCK       ===> YES  (YES, NO or NEVER)

   FORMAT NAME       ===>       MIXED MODE ===> NO   (YES or NO)
```

   In the edit session, you can type REXX instructions, such as the ones that follow.

```
EDIT ---- USERID.REXX.EXEC(TIMEGAME)---------------- COLUMNS 009 080
COMMAND ===>                                          SCROLL ===> HALF
****** ************************ TOP OF DATA ***************************
000001 /************************** REXX ****************************/
000002 /* This is an interactive REXX exec that compares the time  */
000003 /* from a user's watch with computer time.                  */
000004 /************************************************************/
000005
000006 SAY 'What time is it?'
000007 PULL usertime                 /* Put the user's response
000008                                  into a variable called
000009                                  "usertime"  */
000010 IF usertime = '' THEN
000011    SAY "O.K. Game's over."
000012 ELSE
000013    DO
000014      SAY "The computer says:"
000015      /* TSO system */  "time"   /* command */
000016    END
000017
000018 EXIT
****** ********************** BOTTOM OF DATA ********************************
```

Checklist #2: Creating a Data Set with the ALLOCATE Command

1. Type an ALLOCATE command at the READY prompt to define the attributes of the new data set:

   You can use the ALLOCATE command to create a PDS instead of using ISPF/PDF panels. If you noted attributes in the Preliminary Checklist #“5” on page 173, substitute the attributes from the appropriate system file in the following example. If you are unsure about which system file is appropriate, use the values from SYSEXEC.

   Note: In the ALLOCATE command, specify a record format of VB as RECFM(v,b) and a record format of FB as RECFM(f,b).

   If your installation has no attribute recommendations and you have no attributes from the Preliminary Checklist, you can use the attributes in the following example.

   ```
   ALLOCATE DA(rexx.exec) NEW DIR(10) SPACE(50,20) DSORG(po)
     RECFM(v,b) LRECL(255) BLKSIZE(6120)
   ```

   For more information about the ALLOCATE command, see *z/OS TSO/E REXX User's Guide* and *z/OS TSO/E Command Reference*.
2. Edit a member of the newly created PDS by selecting the ISPF/PDF EDIT option (option 2) and specifying the PDS name with a member name:

   See the description for this step in the previous checklist #“4” on page 175.

Checklist #3: Writing an Exec that Sets up Allocation to SYSEXEC

1. Write an exec named SETUP that allocates data sets to SYSEXEC:

   Create a data set member named SETUP in your exec PDS. In SETUP issue an ALLOCATE command that concatenates your PDS to the beginning of all the data sets already allocated to SYSEXEC. Include the data sets allocated to SYSEXEC from the list in the “Preliminary Checklist” on page 172. If there are no other data sets allocated to SYSEXEC, specify your PDS only. Your SETUP exec could look like the following example.

Sample SETUP Exec

```
/****************************** REXX ******************************/
/* This exec is an example of how to allocate a private PDS named  */
/* USERID.REXX.EXEC to the beginning of a concatenation to SYSEXEC */
/* that consists of one other data set named 'ISP.PHONE.EXEC'.  To */
/* make sure that SYSEXEC is available, the exec issues EXECUTIL   */
/* SEARCHDD(yes) command.  After the ALLOCATE command executes, a  */
/* message indicates whether the command was successful or not.    */
/*******************************************************************/
  "EXECUTIL SEARCHDD(yes)"   /* to ensure that SYSEXEC is available*/

  "ALLOC FILE(SYSEXEC) DATASET(rexx.exec,",
         "'isp.phone.exec') SHR REUSE"

  IF RC = 0 THEN
    SAY 'Allocation to SYSEXEC completed.'
  ELSE
    SAY 'Allocation to SYSEXEC failed.'
```

Note: The order in which you list data sets in an ALLOCATE command is the order in which they are concatenated and searched. To give your execs priority in the search order, list your data set of execs before other data sets.

Generally all the data sets in the list should have the same record format (either RECFM=VB or RECFM=FB) and logical record length, LRECL. Also, the first data set in the list can determine the block size, BLKSIZE, for the data sets that follow. If the block size of the first data set is smaller than the block sizes of subsequent data sets, you might end in error. To avoid error, use the Preliminary Checklist and the other checklists provided, and follow directions carefully.

2. Execute SETUP by entering the following EXEC command:

```
READY
EXEC rexx.exec(setup) exec
```

If the allocation was successful, you should then see displayed on your screen:

```
 Allocation to SYSEXEC completed.
```

To have SETUP execute when you log on and automatically allocate your data set to SYSEXEC, type the same EXEC command in the COMMAND field of your LOGON panel.

```
------------------------------ TSO/E LOGON ------------------------------
PF1/PF13 ==> Help  PF3/PF15 ==> Logoff  PA1 ==> Attention  PA2 ==> Reshow
You may request specific HELP information by entering a '?' in
any entry field.

   ENTER LOGON PARAMETERS BELOW:                 RACF LOGON PARAMETERS:

   USERID    ===> YOURID

   PASSWORD  ===>                                NEW PASSWORD ===>

   PROCEDURE ===> MYPROC                         GROUP IDENT  ===>

   ACCT NMBR ===> 00123

   SIZE      ===> 5800

   PERFORM   ===>

   COMMAND   ===> EXEC rexx.exec(setup) exec


   ENTER AN 'S' BEFORE EACH OPTION DESIRED BELOW:

          -NOMAIL         -NONOTICE        -RECONNECT         -OIDCARD
```

Checklist #4: Writing an Exec that Sets up Allocation to SYSPROC

1. Write an exec named SETUP that allocates data sets to SYSPROC:

   Create a data set member named SETUP in your exec PDS. In SETUP issue an ALLOCATE command that concatenates your PDS to the beginning of all the data sets already allocated to SYSPROC. Include the data sets allocated to SYSPROC from the list in the "Preliminary Checklist" on page 172. If there are no other data sets allocated to SYSPROC, specify your PDS only. Your SETUP exec could look like the following example.

   Sample SETUP Exec:

   ```
   /****************************** REXX ******************************/
   /* This exec is an example of how to allocate a private PDS named  */
   /* USERID.REXX.EXEC to the beginning of a concatenation to SYSPROC */
   /* that consists of 3 other data sets named 'ICQ.INFOCNTR.ICQCLIB' */
   /* 'SYS1.TSO.CLIST', and 'ISP.ISPF.CLISTS'.  After the ALLOCATE    */
   /* command executes, a message indicates whether the command was   */
   /* successful or not.                                              */
   /*******************************************************************/

     "ALLOC FILE(SYSPROC) DATASET(rexx.exec,",
            "'icq.infocntr.icqclib',",
            "'sys1.tso.clist',",
            "'isp.ispf.clists') SHR REUSE"

     IF RC = 0 THEN
       SAY 'Allocation to SYSPROC completed.'
     ELSE
       SAY 'Allocation to SYSPROC failed.'
   ```

   Note: The order in which you list data sets in an ALLOCATE command is the order in which they are concatenated and searched. To give your execs priority in the search order, list your data set of execs before other data sets.

   Generally all the data sets in the list should have the same record format, (either RECFM=VB or RECFM=FB) and logical record length, LRECL. Also, the first data set in the list can determine the block size, BLKSIZE, for the data sets that follow. If the block size of the first data set is smaller than the block sizes of subsequent data sets, you might end in error. To avoid error, use the Preliminary Checklist and the other checklists provided, and follow directions carefully.

2. Execute SETUP by entering the following EXEC command:

   ```
   READY
   EXEC rexx.exec(setup) exec
   ```

   If the allocation was successful, you should then see displayed on your screen:

   ```
   Allocation to SYSPROC completed.
   ```

   To have SETUP execute when you log on and automatically allocate your data set to SYSPROC, type the same EXEC command in the COMMAND field of your LOGON panel.

```
------------------------------ TSO/E LOGON ------------------------------
PF1/PF13 ==> Help  PF3/PF15 ==> Logoff  PA1 ==> Attention  PA2 ==> Reshow
You may request specific HELP information by entering a '?' in
any entry field.

   ENTER LOGON PARAMETERS BELOW:                RACF LOGON PARAMETERS:

   USERID    ===> YOURID

   PASSWORD  ===>                               NEW PASSWORD ===>

   PROCEDURE ===> MYPROC                        GROUP IDENT  ===>

   ACCT NMBR ===> 00123

   SIZE      ===> 5800

   PERFORM   ===>

   COMMAND   ===> EXEC rexx.exec(setup) exec


   ENTER AN 'S' BEFORE EACH OPTION DESIRED BELOW:

          -NOMAIL         -NONOTICE        -RECONNECT         -OIDCARD
```

Appendix B. Specifying Alternate Libraries with the ALTLIB Command

The ALTLIB command gives you more flexibility in specifying exec libraries for implicit execution. With ALTLIB, a user or ISPF application can easily activate and deactivate exec libraries for implicit execution as the need arises. This flexibility can result in less search time when fewer execs are activated for implicit execution at the same time.

In addition to execs, the ALTLIB command lets you specify libraries of CLISTs for implicit execution.

Specifying Alternative Exec Libraries with the ALTLIB Command

The ALTLIB command lets you specify *alternative libraries* to contain implicitly executable execs. You can specify alternative libraries on the user, application, and system levels.

- The *user level* includes exec libraries previously allocated to the file SYSUEXEC or SYSUPROC. During implicit execution, these libraries are searched first.
- The *application level* includes exec libraries specified on the ALTLIB command by data set or file name. During implicit execution, these libraries are searched after user libraries.
- The *system level* includes exec libraries previously allocated to file SYSEXEC or SYSPROC. During implicit execution, these libraries are searched after user or application libraries.

Using the ALTLIB Command

The ALTLIB command offers several functions, which you specify using the following operands:

ACTIVATE
Allows implicit execution of execs in a library or libraries on the specified level(s), in the order specified.

DEACTIVATE
Excludes the specified level from the search order.

DISPLAY
Displays the current order in which exec libraries are searched for implicit execution.

RESET
Resets searching to the system level only (execs allocated to SYSEXEC or SYSPROC).

For complete information about the syntax of the ALTLIB command, see *z/OS TSO/E Command Reference*.

Note:

1. With ALTLIB, data sets concatenated to each of the levels can have differing characteristics (logical record length and record format), but the data sets within the same level must have the same characteristics.
2. At the application and system levels, ALTLIB uses the virtual lookaside facility (VLF) to provide potential increases in library search speed.

Stacking ALTLIB Requests

On the application level, you can stack up to eight activate requests with the top, or current, request active. Application-level libraries you define while running an ISPF application are in effect only while that

application has control. When the application completes, the original application-level libraries are automatically reactivated.

Using ALTLIB with ISPF

Under ISPF, ALTLIB works the same as in line mode TSO/E. However, if you use ALTLIB under line mode TSO/E and start ISPF, the alternative libraries you specified under line mode TSO/E are unavailable until ISPF ends.

When you use ALTLIB under ISPF, you can pass the alternative library definitions from application to application by using ISPEXEC SELECT with the PASSLIB operand; for example:

```
ISPEXEC SELECT NEWAPPL(ABC) PASSLIB
```

The PASSLIB operand passes the ALTLIB definitions to the invoked application. When the invoked application completes and the invoking application regains control, the ALTLIB definitions that were passed take effect again, regardless of whether the invoked application changed them. If you omit the PASSLIB operand, ALTLIB definitions are not passed to the invoked application.

For more information about writing ISPF applications, see *z/OS ISPF Services Guide*.

Examples of the ALTLIB Command

In the following example, an application issues the ALTLIB command to allow implicit execution of execs in the data set NEW.EXEC, to be searched ahead of SYSPROC:

```
ALTLIB ACTIVATE APPLICATION(exec) DATASET(new.exec)
```

The application could also allow searching for any private execs that the user has allocated to the file SYSUEXEC or SYSUPROC, with the following command:

```
ALTLIB ACTIVATE USER(exec)
```

To display the active libraries in their current search order, use the DISPLAY operand as follows:

```
ALTLIB DISPLAY
```

For more information about the search order EXEC uses for execs and CLISTs, see *z/OS TSO/E Command Reference*.

To deactivate searching for a certain level, use the DEACTIVATE operand; for example, to deactivate searching for execs on the system level (those allocated to SYSEXEC or SYSPROC), issue:

```
ALTLIB DEACTIVATE SYSTEM(exec)
```

And, to reset exec searching back to the system level, issue:

```
ALTLIB RESET
```

Appendix C. Comparisons Between CLIST and REXX

Both the CLIST language and the REXX language can be used in TSO/E as procedures languages. Some major features of REXX that are different from CLIST are:

- Host command environments - TSO/E REXX has the ability to invoke commands from several environments in MVS and ISPF, as well as from TSO/E. The ADDRESS instruction sets the environment for commands. For more information, see "Issuing Other Types of Commands from an Exec" on page 97.
- Parsing capabilities - For separating data into variable names and formatting text, REXX provides extensive parsing through templates. For more information, see "Parsing Data" on page 83.
- Use of a data stack - REXX offers the use of a data stack in which to store data. For more information, see Chapter 11, "Storing Information in the Data Stack," on page 125.
- Use of mixed and lowercase characters - Although variables and most input are translated to uppercase, REXX provides ways to maintain mixed and lowercase representation. For more information, see "Preventing Translation to Uppercase" on page 18.

In some ways CLIST and REXX are similar. The following tables show similarities and differences in the areas of:

- Accessing system services
- Controlling program flow
- Debugging
- Execution
- Interactive communication
- Passing information
- Performing file I/O
- Syntax
- Using functions
- Using variables

Accessing System Information

CLIST	REXX
LISTDSI statement `LISTDSI &BASEDS`	LISTDSI external function `x = LISTDSI(baseds)`
&SYSOUTTRAP and &SYSOUTLINE `SET SYSOUTTRAP = 100`	OUTTRAP external function `x = OUTTRAP(var,100)`
CONTROL statement `CONTROL PROMPT`	PROMPT external function `x = PROMPT(on)`
&SYSDSN built-in function `IF &SYSDSN('SYS1.MYLIB') = OK THEN` ⋮	SYSDSN external function `IF SYSDSN('SYS1.MYLIB') = OK THEN` ⋮

CLIST	REXX

Control Variables:
For User Information
`&SYSPREF`
`  WRITE &SYSPREF`
`&SYSPROC`
`&SYSUID`
For Terminal Information
`&SYSLTERM`
`&SYSWTERM`
For CLIST Information
`&SYSENV`
`&SYSICMD`
`&SYSISPF`
`&SYSNEST`
`&SYSPCMD`
`&SYSSCMD`
For System Information
`&SYSCPU`
`&SYSHSM`
`&SYSJES`
`&SYSLRACF`
`&SYSNODE`
`&SYSRACF`
`&SYSSRV`
`&SYSTERMID`
`&SYSTSOE`
	Arguments of the SYSVAR external function:
	For User Information
	`SYSPREF`
	`  SAY  SYSVAR(syspref)`
	`SYSPROC`
	`SYSUID`
	For Terminal Information
	`SYSLTERM`
	`SYSWTERM`
	For Exec Information
	`SYSENV`
	`SYSICMD`
	`SYSISPF`
	`SYSNEST`
	`SYSPCMD`
	`SYSSCMD`
	For System Information
	`SYSCPU`
	`SYSHSM`
	`SYSJES`
	`SYSLRACF`
	`SYSNODE`
	`SYSRACF`
	`SYSSRV`
	`SYSTERMID`
	`SYSTSOE`

Control Variables:
For System Information
`&SYSAPPCLU`
`&SYSDFP`
`&SYSMVS`
`&SYSNAME`
`&SYSSECLAB`
`&SYSSMFID`
`&SYSSMS`
`&SYSCLONE`
`&SYSPLEX`
`&SYSSYMDEF`
	Arguments of the MVSVAR external function:
	For System Information
	`SYSAPPCLU`
	`SYSDFP`
	`SYSMVS`
	`SYSNAME`
	`SYSSECLAB`
	`SYSSMFID`
	`SYSSMS`
	`SYSCLONE`
	`SYSPLEX`
	`SYMDEF`

Controlling Program Flow

CLIST	REXX
Branching	**Branching**
IF/THEN/ELSE statements	IF/THEN/ELSE instructions
SELECT/WHEN/OTHERWISE/END statements	SELECT/WHEN/OTHERWISE/END instructions
Looping	**Looping**
Iterative DO	Iterative DO

CLIST	REXX
DO/WHILE/END statements	DO/WHILE/END instructions
DO/UNTIL/END statements	DO/UNTIL/END instructions
Interrupting	**Interrupting**
END, EXIT statements	EXIT instruction
GOTO statement	SIGNAL instruction
	LEAVE instruction
	CALL instruction
Calling another CLIST	**Calling another exec as an external subroutine**

EXEC command
`:`
`EXEC MYNEW.CLIST(CLIST1) 'VAR'`
`:`
`END`
 	
`PROC 1 VAR`
`:`
`EXIT`
	CALL instruction
	`:`
	`call exec1 var`
	`:`
	`exit`
	
	`arg var`
	`:`
	`return`

Calling a subprocedure	**Calling an internal subroutine**

SYSCALL statement
`:`
`SYSCALL SOMESUB VAR`
`:`
`END`
`SOMESUB: PROC 1 VAR`
`:`
`EXIT`
	CALL instruction
	`:`
	`call sub1 var`
	`:`
	`exit`
	`sub1:`
	`arg var`
	`:`
	`return`

Debugging

CLIST	REXX
Debugging a CLIST	**Debugging an exec**
CONTROL SYMLIST LIST CONLIST MSG	TRACE instruction `trace i` Interactive debug facility (EXECUTIL TS and TRACE ? R)
Return codes for commands and statements	**Return codes for commands**
&LASTCC, &MAXCC `SET ECODE = &LASTCC`	RC `ecode = RC`
Trapping TSO/E command output	**Trapping TSO/E command output**
&SYSOUTTRAP, &SYSOUTLINE	OUTTRAP external function
Error handling	**Error handling**

CLIST	REXX
ERROR and ATTN statements	SIGNAL ON ERROR, SIGNAL ON FAILURE, SIGNAL ON HALT, SIGNAL ON NOVALUE, and SIGNAL ON SYNTAX instructions. CALL ON ERROR, CALL ON FAILURE, and CALL ON HALT instructions.[1]

Note:

1
For more information about REXX error handling instructions, see *z/OS TSO/E REXX Reference*.

Execution

CLIST	REXX
Explicit	**Explicit**
EXEC command `EXEC MYNEW.CLIST(CLIST1)`	EXEC command `EXEC MYNEW.EXEC(FIRST) EXEC`
Implicit	**Implicit**
1. Allocate/concatenate to SYSPROC 2. Specify member name of PDS with or without %	1. Allocate/concatenate to SYSPROC or SYSEXEC 2. Specify member name of PDS with or without %

Interactive Communication

CLIST	REXX
Reading from the terminal	**Reading from the terminal**
READ, READDVAL statements `READ INPUTA, INPUTB, INPUTC`	PULL, PARSE PULL, PARSE UPPER PULL, PARSE EXTERNAL instructions `pull inputa, inputb, inputc`
Writing to the terminal	**Writing to the terminal**
WRITE statement `WRITE Your previous entry was not valid.`	SAY instruction `say 'Your previous entry was not valid.'`

Passing Information

CLIST	REXX
Receiving parameters in a CLIST	**Receiving arguments in an exec**

CLIST	REXX

PROC statement
`PROC 1 DSNAME MEMBER() DISP(SHR)`
CLISTs can receive positional, keyword, and keyword value parameters.
	ARG, PARSE ARG, PARSE UPPER ARG instructions
	`arg dsname member disp`
	An exec receives positional parameters. Use the PARSE ARG and PARSE UPPER ARG instructions to receive keywords, for example:
	`my.data member(member1) disp(old)`
	`parse upper arg dsname .`
	`parse upper arg 'MEMBER('mem')'`
	`parse upper arg 'DISP('disp')'`

Recognizing comments within a parameter	**Recognizing comments within a parameter**
A CLIST PROC statement recognizes a comment within a parameter sent by the EXEC command and ignores that comment.	An ARG instruction does not recognize a comment within a parameter sent by the EXEC command. It is treated as part of the argument.
Sending parameters to a CLIST	**Sending arguments to an exec**

EXEC command
`EXEC MY.CLIST(NEW) -`
`'MY.DATA MEMBER(MEMBER1) DISP(OLD)'`
	EXEC command from TSO/E READY
	`'EXEC MY.EXEC(NEW)',`
	`"'my.data member(member1) disp(old)' EXEC"`

Sending information to a subprocedure	**Sending information to a subroutine**
SYSCALL statement `SYSCALL SOMESUB &VAR`	CALL instruction `call somsub var`
Sending information from a subprocedure	**Sending information from a subroutine**

RETURN statement
`⋮`
`SYSCALL SOMESUB &VAR`
`SET ANSWER = &LASTCC`
`⋮`
`END`
 	
`SOMESUB: PROC 1 V1`
`⋮`
`RETURN CODE(33) /* code goes to &LASTCC`
`*/`
	RETURN instruction
	`⋮`
	`call somesub var`
	`answer = RESULT`
	`exit`
	
	`somesub:`
	`arg v1`
	`⋮`
	`value = 4 * v1 / 3`
	`return value  /* value goes to RESULT  */`

Performing File I/O

CLIST	REXX
Reading from a file	**Reading from a file**

OPENFILE, GETFILE, CLOSFILE statements
`OPENFILE PAYCHEKS`
`SET COUNTER=1`
`DO WHILE &COUNTER \> 3`
`GETFILE PAYCHEKS`
`SET EMPLOYEE&COUNTER=&PAYCHEKS`
`SET COUNTER=&COUNTER+1;`
`END`
`CLOSFILE PAYCHEKS`
	EXECIO DISKR, EXECIO DISKRU commands
	`'EXECIO 3 DISKR indd (stem employee. FINIS'`
	`/* Read 3 records from the data set in indd. */`
	`/* The 3 records go to a list of compound */`
	`/* variables with the stem of employee. They */`
	`/* are employee.1, employee.2 and employee.3 */`

Writing to a file	**Writing to a file**

OPENFILE, PUTFILE, CLOSFILE statements
`OPENFILE PRICES OUTPUT`
`SET PRICES = $2590.00`
`PUTFILE PRICES`
`CLOSFILE PRICES`
	EXECIO DISKW
	`push '$2590.00' /* put amount on data stack */`
	`'EXECIO 1 DISKW outdd (finis'`
	`/*Write from data stack to data set in outdd */`

Syntax

CLIST	REXX
Continuing a statement over more than one line	**Continuing an instruction over more than one line**

Use - or +
`IF &STR(SYSDATE)=&STR(10/13/87) THEN +`
`WRITE On &SYSDATE the system was down.`
	Use ,
	`say 'This instruction',`
	`'covers two lines.'`

Separating statements within a line	**Separating instructions within a line**
No more than one statement per line	Use ; `do 5; Say 'Hello'; end`
Character set of statements	**Character set of instructions**
Must be in uppercase	Can be upper, lower, or mixed case
Comments	**Comments**
Enclose between /* */, closing delimiter optional at the end of a line.	Enclose between /* */, closing delimiter always required.

Using Functions

CLIST	REXX
Calling a function	**Calling a function**
&FUNCTION(expression) `SET A = &LENGTH(ABCDE)  /* &A = 5 */`	function(arguments) `a = length('abcde')  /* a = 5 */`

Using Variables

CLIST	REXX
Assigning value to a variable	**Assigning value to a variable**

SET statement
`SET X = 5  /* &X gets the value 5  */`
`SET NUMBER = &X /* &NUMBER gets the value 5 */`
`SET Y = NUMBER  /* &Y gets the value NUMBER */`
	assignment instruction
	`x = 5  /* X gets the value 5  */`
	`NUMBER = x  /* NUMBER gets the value 5  */`
	`Y = 'number'  /* Y gets the value number  */`

Appendix D. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed email message to mhvrcfs@us.ibm.com.

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number `3.1`) are mutually exclusive alternatives. If you hear the lines `3.1 USERID` and `3.1 SYSTEMID`, your syntax can include either `USERID` or `SYSTEMID`, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element
The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element
The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.

3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line `6.1+ data area`, you must include at least one data area. If you hear the lines `2+, 2 HOST,` and `2 STATE`, you know that you must include `HOST, STATE,` or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for the Knowledge Centers. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming Interface Information

This publication documents information that is NOT intended to be used as programming Interfaces of JES2.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

S

T

U

V

SA32-0982-30