IBM Library Server Print Preview

```
        DOCNUM = SC33-6641-00
   DATETIME = 09/25/97 11:10:12
          BLDVERS = 1.2
TITLE = REXX/VSE V6R1 User's Guide
             AUTHOR =
COPYR = © Copyright IBM Corp. 1988, 1995
 PATH = /home/webapps/epubs/htdocs/book
```

COVER Book Cover

IBM VSE/Enterprise Systems Architecture
VSE Central Functions

REXX/VSE User's Guide

Version 6 Release 1

Document Number SC33-6641-00

Program Number
5686-066

File Number S370/390-39

NOTICES Notices

```
 ___ Note! __________________________________________________________
|                                                                    |
| Before using this information and the product it supports, be sure |
| to read the general information under "Notices" in                 |
| topic NOTICES_1.                                                   |
|                                                                    |
|____________________________________________________________________|
```

EDITION Edition Notice

Second Edition (April 1995)

This edition applies to Version 6 Release 1 of IBM REXX/VSE, which is part of VSE/Central Functions, Program Number 5686-066, and to all subsequent releases and modifications until otherwise indicated in new editions.

Publications are *not* stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

Comments may be addressed to:

```
IBM Corporation      or to:        IBM Deutschland Entwicklung GmbH
Attn: Dept ECJ - BP/003D            Department 3248
6300 Diagonal Highway               Schoenaicher Strasse 220
Boulder, CO 80301,                 D-71032 Boeblingen
U.S.A.                             Federal Republic of Germany
```

CONTENTS Table of Contents

Summarize

FIGURES Figures

NOTICES_1 Notices

Subtopics:

NOTICES_1.1 Programming Interface Information

This book is intended to help the customer write programs in the REXX programming language and customize services that REXX/VSE 6.1.0 provides for REXX processing. This book documents General-use Programming Interface and Associated Guidance Information that REXX/VSE 6.1.0 provides.

General-use programming interfaces allow the customer to write programs that obtain the services of REXX/VSE 6.1.0.

NOTICES_1.2 Trademarks and Service Marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

Systems Application Architecture	SAA
IBM	CICS
BookManager	Library Reader
VSE/ESA	MVS/ESA
Operating System/400	OS/400
Operating System/2	OS/2

PREFACE About This Book

This book describes the REstructured eXtended eXecutor (REXX) language. The REXX language is implemented through

- The REXX/VSE Interpreter
- The Library for REXX/370 in REXX/VSE, which you can use to run compiled REXX programs.

The interpreter is also called the language processor. The Library for REXX/370 in REXX/VSE is also called a compiler's runtime processor. REXX/VSE is a partial implementation of Level 2 Systems Application Architecture (SAA) REXX on the VSE/ESA system.

Subtopics:

PREFACE.1 Who Should Use This Book

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. Because of the broad range of experience in readers, this book is divided into two parts.

"PART I -- Learning the REXX Language"

This part is for inexperienced REXX programmers who have at least some knowledge of JCL and know how to create PROC members and a sublibrary. Programmers unfamiliar with VSE/ESA should first read *VSE/ESA System Control Statements*, SC33-6613. Experienced programmers new to REXX can also read this section to learn the basics of the REXX language.

"PART II -- Using REXX"

This part is for programmers already familiar with the REXX language and experienced with the workings of VSE/ESA. It describes more complex aspects of the REXX language and how they work in VSE/ESA.

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

PREFACE.2 How This Book Is Organized

This book includes chapter previews, examples, and exercises.

Purpose of Each Chapter: At the beginning of each chapter is a statement about the purpose of the chapter.

Examples: Throughout the book, you will find examples that you can try as you read. Examples including REXX keyword instructions uppercase any REXX keywords. Similarly, examples showing VSE/ESA control statements or VSE/POWER commands uppercase keyword operands and command names. Information that can vary is in lowercase. This use of uppercase and lowercase is to help you distinguish between words that cannot change and words that can. It does **not** mean that you must type REXX instructions and VSE/ESA control statements or VSE/POWER commands with certain words in uppercase and others in lowercase.

Here are some examples. The following REXX keyword instruction contains the REXX keyword SAY, which cannot vary, and a phrase, which can vary.

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

The next example shows a system control statement. The system control statement name and keywords are in uppercase because they cannot vary. The library and sublibrary are in lowercase because they can vary.

```
  // LIBDEF *,SEARCH=(prd1.base,rexxlib.samples)
```

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.

Terminology: REXX/VSE is interactive only from the operator's console. Keep this reservation in mind regarding any terminology in this book suggesting interactive input and output. For example, *displaying* output refers to presenting it through the current output stream; *entering* information refers to providing it through the current input stream.

A REXX program can be called an *exec*.

PREFACE.3 Where to Find More Information

The following books contain information related to the topics covered in this book.

Subtopics:

PREFACE.3.1 REXX/VSE Publications

REXX/VSE Reference, SC33-6642.

PREFACE.3.2 VSE/ESA Publications

VSE/ESA System Control Statements, SC33-6613

VSE/POWER Application Programming, SC33-6636

PREFACE.3.3 SAA Publications

SAA Common Programming Interface REXX Level 2 Reference, SC24-5549

SAA Common Programming Interface Communications Reference, SC26-4399

PREFACE.3.4 IBM Compiler and Library Publications

IBM Compiler and Library for REXX/370; Introducing the Next Step in REXX Programming, G511-1430

IBM Compiler and Library for REXX/370; User's Guide and Reference, SH19-8160

PREFACE.4 Referenced Program Products

This book refers to the following product:

° Library for SAA REXX/370, Program Number 5695-014

° Compiler for SAA REXX/370, Program Number 5695-013.

All occurrences of VSE/ESA refer to VSE/ESA Version 2, Release 1, Program Number 5690-VSE.

1.0 PART I -- Learning the REXX Language

The REXX language is a versatile general-purpose programming language new and experienced programmers can use. 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" in topic 1.1 -- The REXX language has many features that make it a powerful programming tool.

° Chapter 2, "Writing and Running a REXX Program" in topic 1.2 -- Programs are easy to write and have few syntax rules.

° Chapter 3, "Using Variables and Expressions" in topic 1.3 -- Variables, expressions, and operators are essential when writing programs that do arithmetic and comparisons.

° Chapter 4, "Controlling the Flow within a Program" in topic 1.4 -- You can use instructions to branch, loop, or interrupt the flow of a program.

° Chapter 5, "Using Functions" in topic 1.5 -- A function is a sequence of instructions that can perform a specific task and must return a value.

° Chapter 6, "Writing Subroutines and Functions" in topic 1.6 -- You can write internal and external routines that a program calls.

° Chapter 7, "Manipulating Data" in topic 1.7 -- Compound variables and parsing are two ways to manipulate data.

Several REXX instructions either get information from the input stream or write information to the output stream. The INDD and OUTDD fields in the module name table identify the default input and output streams, respectively. If you have not changed the defaults, the current input stream is SYSIPT and the current output stream is SYSLST.

° SAY sends information to the output stream.
° PARSE PULL and PULL get information from the top of the data stack or, if the stack is empty, from the input stream.
° TRACE sends information to the output stream.
° PARSE EXTERNAL gets information from the input stream.
° EXECIO reads information from or writes it to the specified output stream or device.

The USERID built-in function returns the current user ID. (See topic 1.5.2.6 for more information.)

Subtopics:

1.1 Chapter 1. Introduction

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

Subtopics:

1.1.1 What is REXX?

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

REXX programs can do many tasks, including the automation of VSE/Operations. For example, if you use the JCL EXEC command to call a REXX program, you can leave JCL statements on the stack for VSE/ESA to process. This enables you to insert JCL statements or data into the current job stream. REXX programs can run in any partition. They can communicate with POWER through the SAS interface.

1.1.2 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. You need not predefine variables. You can type instructions in upper, lower, or mixed case. The few rules about REXX format are covered in "Syntax of REXX Instructions" in topic 1.2.3.

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 program running in REXX/VSE encounters an error, REXX writes messages describing the error to the current output stream. You can also use the REXX TRACE instruction and the interactive debug facility to locate errors in programs.

Interpreted language: The REXX/VSE product includes the REXX/VSE interpreter. When a REXX program runs, the interpreter directly processes each line. Languages that are not interpreted must be compiled into machine language and possibly link-edited before they are run.

The REXX/VSE product also includes the Library for REXX/370 in REXX/VSE. You can use this component to run compiled programs. (See topic 1.1.5 for information about the benefits of using a compiler.)

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

1.1.3 Components of REXX

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

- Clauses, which can be instructions, null clauses, or labels. Instructions can be:
  - Keyword instructions
  - Assignments
  - Commands (both REXX/VSE commands and host commands, such as ADDRESS POWER commands)

  The language processor processes keyword instructions and assignments.
- Built-in functions -- These functions are built into the language processor and provide convenient processing options.
- External functions -- REXX/VSE provides these functions that 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.

1.1.4 REXX and Systems Application Architecture (SAA)

Systems Application Architecture* (SAA*) REXX defines a common set of language elements you can use in several environments. REXX/VSE is a partial implementation of Level 2 SAA REXX on the VSE/ESA* system. The SAA environments are the following:

- MVS
  - Base system (TSO/E, APPC/MVS, batch)
  - CICS*
  - IMS
- VM CMS
- Operating System/400* (OS/400*)
- Operating System/2* (OS/2*).

1.1.5 Benefits of Using a Compiler

The IBM Compiler for REXX/370 (Program Number 5695-013) and the IBM Library for REXX/370 in REXX/VSE 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

Subtopics:

1.1.5.1 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 issues commands to the host shows minimal improvement because REXX cannot decrease the time the host takes to process the commands.

1.1.5.2 Reduced System Load

Compiled REXX programs run faster than interpreted programs. Because a program has to be compiled only once, running compiled programs reduces system load and improves response time for frequently run programs.

For example, a REXX program that performs many arithmetic operations might take 12 seconds to run on the interpreter. Running the program 60 times 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.

1.1.5.3 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.

1.1.5.4 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.

1.1.5.5 Portability of Compiled Programs

A compiled REXX program can run under other operating systems, such as MVS/ESA* or VM CMS. A REXX program compiled under VM CMS or MVS/ESA can run under REXX/VSE.

1.1.5.6 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.

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*.

1.2 Chapter 2. Writing and Running a REXX Program

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

Subtopics:

1.2.1 Before You Begin

The default location for all parts of REXX/VSE is the PRD1.BASE sublibrary. All descriptions and examples in this book refer to this sublibrary.

Before you can run a REXX program you need to put the program in a sublibrary in the active PROC chain.

1.2.2 What is a REXX Program?

A REXX program consists of REXX language instructions that the REXX interpreter interprets directly. (The Library for REXX/370 in REXX/VSE runs compiled programs.) A program can also contain commands that the host environment executes, such as ADDRESS POWER commands.

One advantage of the REXX language is its similarity to ordinary English. This similarity makes it easy to read and write a REXX program. For example, to write a line to the output stream, you use the REXX instruction SAY followed by the line. SAY writes output to the current output stream. If you have not changed the default, the output stream is SYSLST.

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /* Sample REXX Program                                          */   |
|    SAY 'This is a REXX program.'                                       |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```
Figure 1. Example of a Simple Program

This program starts with a comment line to identify it as a REXX program. A comment begins with /* and ends with */. More about comments and why you might need a REXX program identifier appears later (topic 1.2.3.3).

When you run the program, the SAY instruction sends to the output stream:

```
      This is a REXX program.
```

Even in a longer program, the instructions are similar to ordinary English and are easy to understand. For example, you could use the following to call the program ADDTWO, which adds two numbers:

```
  // LIBDEF *,SEARCH=(prd1.base,rexxlib.samples)
  // EXEC REXX=addtwo
  42
  21
  /&
```

Here is the ADDTWO program:

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /**************************** REXX *********************************/|
|   /*  This program adds two numbers and produces their sum.          */|
|   /*******************************************************************/|
|    PULL number1                      /* Assigns: number1=42 */         |
|    PULL number2                      /* Assigns: number2=21 */         |
|    sum = number1 + number2                                             |
|    SAY 'The sum of the two numbers is' sum'.'                          |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```
Figure 2. Example of a Longer Program

When you run the example program, PULL gets input from the current input stream. The default is SYSIPT.

The first PULL instruction assigns the variable number1 the value 42. The second PULL instruction assigns the variable number2 the value 21. The next line contains an assignment. The language processor adds the values in number1 and number2 and assigns the result, 63, to sum. Finally, the SAY instruction sends to the output stream the line:

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

Before you try any examples, please read the next two sections, "Syntax of REXX Instructions" in topic 1.2.3 and "Running a Program" in topic 1.2.5.

1.2.3 Syntax of REXX Instructions

Some programming languages have rigid rules about how and where you enter characters on each line. For example, assembler statements must begin in a certain column. REXX, on the other hand, has simple syntax rules. You can use upper or lower or mixed case. REXX has no restrictions about the columns in which you can type.

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

```
  SAY 'You can type in any column'
                      SAY 'You can type in any column'
                                        SAY 'You can type in any column'
```

These instructions are sent to the output stream:

```
     You can type in any column
     You can type in any column
     You can type in any column
```

Subtopics:

1.2.3.1 The Format of REXX Instructions

The REXX language has free format. This means you can insert extra spaces between words. For example, the following all mean the same:

```
    total=num1+num2
    total =num1+num2
    total = num1+num2
    total = num1 + num2
```

You can also insert blank lines throughout a program without causing an error.

1.2.3.2 The Letter Case of REXX Instructions

You can enter a REXX instruction in lowercase, uppercase, or mixed case. For example, SAY, Say, and say all have the same meaning. The language processor translates alphabetic characters to uppercase, unless you enclose them in single or double quotation marks.

Using Quotation Marks in an Instruction: A series of characters within matching quotation marks is a **literal string**. The following examples contain literal strings.

```
  SAY 'This is a REXX literal string.'  /* Using single quotation marks */

  SAY "This is a REXX literal string."  /* Using double quotation marks */
```

Do not enclose a literal string with one each of the two different types of quotation marks. For example, the following is **incorrect**:

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

If you omit the quotation marks around a literal string in a SAY instruction, the language processor usually translates the statement to uppercase. For example,

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

results in:

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

(This assumes none of the words is the name of a variable that you have already assigned a value. In REXX, the default value of a variable is its own name in uppercase.)

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

```
            SAY "This isn't difficult!"
```

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 difficult!'
```

Either way, the outcome is the same.

```
           This isn't difficult!
```

Ending an instruction: A line usually contains one instruction except when it contains a semicolon (;) or ends with a comma (,).

The end of the line or a semicolon indicates the end of an instruction. If you put one instruction on a line, the end of the line delineates the end of the instruction. If you put multiple instructions on one line, you must separate adjacent instructions with a semicolon.

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

This example would result in three lines.

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

Continuing an instruction: A comma is the continuation character. It 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. Here is how the comma continuation character works when a literal string is being continued on the next line.

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

The comma at the end of the first line adds a space (between extended and REXX when the two lines are concatenated for output. A single line results:

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

The following two instructions are identical and yield the same result:

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

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

The space between the two separate strings is preserved:

```
      This is a string.
```

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 in the line, 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 results in one line no space in 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:

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

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

These examples result in:

```
      This isa string.
```

In both examples, the concatenation operator deletes spaces between the two strings.

The following example demonstrates the free format of REXX.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /************************* 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.'                           |
|                                                                       |
|                                                                       |
|_______________________________________________________________________|
```

Figure 3. Example of Free Format

Running this example results in six lines of identical output, 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. The language processor ignores blank lines, and it ignores spaces that are greater than one. This flexibility of format lets you insert blank lines and spaces to make a program easier to read.

Blanks and spaces are significant only during parsing. "Parsing Data" in topic 1.7.2 describes parsing.

1.2.3.3 Types of REXX Clauses

REXX clauses can be: instructions, null clauses, and labels. Instructions can be keyword instructions, assignments, or commands. The following example shows a program with these types of clauses. A description of each type of clause follows the example.

```
 ________________________________________________________________________
|                                                                       |
|                                                                       |
|   /* QUOTA REXX program. Two car dealerships are competing to  */     |
|   /* sell the most cars in 30 days.  Who will win?             */     |
|                                                                       |
|   store_a=0; store_b=0                                                |
|   DO 30                                                               |
|      CALL sub                                                         |
|   END                                                                 |
|   IF store_a>store_b THEN SAY "Store_a wins!"                         |
|    ELSE IF store_b>store_a THEN SAY "Store_b wins!"                   |
|     ELSE SAY "It's a tie!"                                            |
|   EXIT                                                                |
|                                                                       |
|   sub:                                                                |
|   store_a=store_a+RANDOM(0,20) /* RANDOM returns a random number in */ |
|   store_b=store_b+RANDOM(0,20) /* in specified range, here 0 to 20  */ |
|   RETURN                                                              |
|                                                                       |
|                                                                       |
|_______________________________________________________________________|
```

Keyword Instructions: 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, DO can group instructions and execute them repetitively, and IF tests whether a condition is met. SAY writes to the current output stream.

IF, THEN and ELSE 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. (Note that a semicolon is needed before the ELSE if you are putting an ELSE clause on the same line with a THEN.) If you want to put more than one instruction after a THEN or ELSE, use a DO before the group of instructions and an END after them. More information about the IF instruction appears in "Using Conditional Instructions" in topic 1.4.2.

The EXIT keyword tells the language processor to end the program. Using EXIT in the preceding example is necessary because, otherwise, the language processor would execute the code in the subroutine after the label sub:. EXIT is not necessary in some programs (such as those without subroutines), but it is good programming practice to include it. More about EXIT appears in "EXIT Instruction" in topic 1.4.4.1.

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 the preceding program, a simple assignment instruction is: store_a=0. The left side of the assignment (before the equal sign) contains the name of the variable to receive a value from the right side (after the equal sign). The right side can be an actual value (such as 4) or an expression. An expression is something that needs to be evaluated, such as an arithmetic expression. The expression can contain numbers, variables, or both.

```
     number = 4 + 4


     number = number + 4
```

In the first example, the value of number is 8. If the second example directly followed the first in a program, the value of number would become 12. More about expressions is in "Using Expressions" in topic 1.3.3.

Label: A label, such as sub: 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 OPTIONS ETMODE is the first instruction in your program.) A label identifies a portion of the program and is commonly used in subroutines and functions, and with the SIGNAL instruction. (Note that you need to include a RETURN instruction at the end of a subroutine to transfer control back to the main program.) More about the use of labels appears in Chapter 6, "Writing Subroutines and Functions" in topic 1.6 and "SIGNAL Instruction" in topic 1.4.4.3.

Null Clause: A null clause consists of only blanks or comments or both. The language processor ignores null clauses, but they make a program easier to read.

Comments 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 of a program can describe the overall purpose of the program and perhaps list special considerations. A comment next to an individual instruction can clarify its purpose.

Note: For portability reasons, you are recommended to start each REXX program with a comment that includes the word REXX. The first comment in a program is the REXX program identifier. It immediately identifies the program to readers as a REXX program.

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

Commands: A command is a clause consisting of only an expression. Commands are sent to a previously defined environment for processing. (You should enclose in quotation marks any part of the expression not to be evaluated.) The example program did not include any commands. The following example includes a command in an ADDRESS instruction:

```
  /* REXX program including a command */
  job.1="* $$ JOB jnm=testjob"
  job.2="// JOB testjob"
  job.3="// EXEC testprog"
  job.0=3
  ADDRESS power "PUTQE RDR STEM job."
```

ADDRESS is a keyword instruction. When you specify an environment and a command on an ADDRESS instruction, a single command is sent to the environment you specify. In this case, the environment is power. The command is the expression that follows the environment:

```
  "PUTQE RDR STEM job."
```

This PUTQE command puts the job testjob on the POWER RDR queue. By default the job is class A. For more details about changing the host command environment, see "Changing the Host Command Environment" in topic 2.1.2.3.

More information about issuing commands appears in Chapter 8, "Using Commands from a Program" in topic 2.1.

1.2.4 Programs Using Double-Byte Character Set Names

You can use double-byte character set (DBCS) names in your REXX programs for literal strings, symbols, and comments. Such character strings can be single-byte, double-byte, or a combination of both. To use DBCS names, OPTIONS ETMODE must be the first instruction in the program. This specifies that the language processor should check strings containing DBCS characters for validity. You must enclose DBCS characters within shift-out (SO) and shift-in (SI) delimiters. (The SO character is X'0E', and the SI character is X'0F') The SO and SI characters are non-printable. In the following example, the less than (<) and greater than (>) symbols represent shift-out (SO) and shift-in (SI), respectively. 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 a program using a DBCS variable name and a DBCS subroutine label.

```
/*   REXX  */
OPTIONS 'ETMODE'         /* ETMODE to enable DBCS variable names  */
<.S.Y.M.D> = 10          /* Variable with DBCS characters between */
                         /* shift-out (<) and shift-in (>)        */
y.<.S.Y.M.D> = JUNK
CALL <.D.B.C.S.R.T.N>     /* Call subroutine with DBCS name       */
EXIT
<.D.B.C.S.R.T.N>:         /* Subroutine with DBCS name            */
DO i = 1 TO 10
  IF y.i = JUNK THEN      /* Does y.i match the DBCS variable's
                              value?                              */
    SAY 'Value of the DBCS variable is : ' <.S.Y.M.D>
END
RETURN
```

Example 2

The following example shows DBCS characters in an EXECIO command and some other uses of DBCS variable names. DBCS characters are passed to a program called through LINKPGM and are passed with the built-in function LENGTH.

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


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



"EXECIO * DISKR mylib.sublib.data.proc (STEM <.D.B.C.S__.S.T.E.M>. FINIS"



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


  /*****************************************************************/
  /*   Say each DBCS stem variable that EXECIO sets                */
  /*****************************************************************/


  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 */


/*******************************************************************/
/* The ADDRESS instruction specifies the LINKPGM host command     */
/* environment to call program PROCA29 to process a line.
/* This line includes 2 variable names to pass 2 parameters, one  */
/* of which is a DBCS variable name.  The LINKPGM host command    */
/* environment routine looks up the value of the two variables    */
/* and passes their values to the address LINKPGM command         */
/* "proca29".                                                     */
/*******************************************************************/
ADDRESS LINKPGM "proca29  line_len  line1_<.v.a.l.u.e>"
```

1.2.5 Running a Program

You need to store your REXX program in a sublibrary with a member type of PROC. You can run a program:

° From batch by using the JCL EXEC command

° From another program by calling ARXEXEC or ARXJCL. The recommended method is using the JCL EXEC command. Calling REXX in this way lets you leave JCL statements on the stack for VSE/ESA to process. This lets you insert JCL statements or data into the current job stream. Using ARXEXEC or ARXJCL does not allow you to leave JCL statements on the stack. However, this method is compatible with MVS/ESA.

Subtopics:

1.2.5.1 Using the JCL EXEC Command to Run a REXX Program

The program must be a member of a sublibrary in the active PROC chain. If the program myprog is in the library mainlib, sublibrary sublib1, its full name is in the format library.sublibrary.program_name.filetype, for example, mainlib.sublib1.myprog.proc. The library name can be up to 7 characters. (This is a VSE/ESA stipulation.) The sublibrary name, program_name, and file type are each up to 8 characters. To use the JCL EXEC command to run a REXX program in batch, specify

```
REXX=program_name
```

on the JCL EXEC statement. For example:

```
// LIBDEF PROC SEARCH=rexxlib.samples
// EXEC REXX=myprog
```

REXX treats the program_name as a member with a type of PROC. If you omit the name, specify blanks, or use a name longer than 8 characters, VSE JCL reports an error and stops processing. REXX calls the Librarian services to search the active PROC chain for the program. For example, if the program_name is myprog, the Librarian services search for myprog.PROC. REXX accesses the program through the Librarian services.

You can include optional parameters on the call, by specifying:

```
PARM=parameters
```

Here is an example:

```
*
// LIBDEF PROC,SEARCH=rexxlib.samples
// EXEC REXX=program1,PARM='m n o'
```

You can pass only one argument to the program you are calling, but the argument can consist of more than one token. In the example, the program receives the argument: m n o. If you are passing an argument to a program, you need to include a PARSE ARG (or ARG) instruction in the program to retrieve the arguments. For example, if you call a program by using:

```
// EXEC REXX=program1,PARM='1 2 3'
```

You could include the following instruction in the program:

```
PARSE ARG var1 var2 var3
```

This would give var1 the value 1, var2 the value 2, and var3 the value 3. For more information, see "Specifying Values When Calling a Program" in topic 1.2.8.2 and "Passing Arguments" in topic 1.2.8.4.

1.2.5.2 Using ARXEXEC or ARXJCL

To call a REXX program from another program, you can use ARXEXEC or ARXJCL. You can use ARXJCL to run a REXX program in two ways:

- Call ARXJCL from a non-REXX program
- Specify ARXJCL on the JCL EXEC statement.

To use ARXJCL on the JCL EXEC statement, specify the member name of the program and any arguments After PARM=. For example:

```
// LIBDEF PROC,SEARCH=rexxlib.samples
// EXEC ARXJCL,PARM='MYPROG arg1 arg2'
```

To use ARXEXEC or ARXJCL to call a REXX program from a non-REXX program, you need to specify parameters that define the program and supply other related information. For details, see the *REXX/VSE Reference*.

1.2.5.3 Defining Language Processor Environments

Before you can run a program, a language processor environment must exist. A language processor environment defines the way a REXX program is processed and how it accesses system services. REXX/VSE provides the default parameters module ARXPARMS to define language processor environments. REXX/VSE sets the defaults but a system programmer can modify them.

Subtopics:

1.2.5.3.1 What Is a Language Processor Environment?

A language processor environment defines characteristics, such as:

- The search order for locating commands and external routines
- The devices for reading and writing data
- 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 programs running in the environment can use the data stack
- The names of routines that handle system services, such as I/O operations, program loading, obtaining and freeing storage, and data stack requests.

Note: A language processor environment differs from a host command environment. The language processor environment is the environment in which a REXX program runs. The host command environment is the environment to which the language processor passes commands for execution. The language processor environment defines the host command environments.

For more information about defining language processor environments, see the *REXX/VSE Reference*.

1.2.5.4 Customizing a Language Processor Environment

An individual or an installation can customize a language processor environment in two ways:

- Change the values in the default parameters module ARXPARMS.
- Call the initialization routine ARXINIT and specify parameters to change default parameters.

For more information about customizing a language processor environment, see the *REXX/VSE Reference*.

1.2.6 Interpreting Error Messages

When you run a program that contains an error, an error message often includes 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 program has a syntax error.

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|    /************************** REXX **********************************/|
|    /* This REXX program contains a deliberate error of not closing   */|
|    /* a comment.  Without the error, it would pull input to produce  */|
|    /* a greeting.                                                    */|
|    /******************************************************************/|
|                                                                        |
|    PULL who                       /* Get the person's name.            |
|    IF who = '' THEN                                                    |
|       SAY 'Hello, stranger'                                            |
|    ELSE                                                                |
|       SAY 'Hello,' who                                                 |
|                                                                        |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 4. Example of a Program with a Syntax Error

When the program runs, the language processor sends the following lines to the output stream. (If you have not changed the default, the output stream is SYSLST.)

```
       7 +++ PULL who                    /* Get the person's name.IF who =
  '' THEN SAY 'Hello stranger'ELSE   SAY 'Hello' who
  ARX0006I Error running HELLO, line 7: Unmatched "/*" or quote
```

The program runs until the language processor detects the error, the missing */ at the end of the comment. The PULL instruction does not use the data from the data stack or input stream because this line contains the syntax error. The program ends, and the language processor sends the error messages.

The first error message begins with the line number of the statement where the language processor detected the error. Three pluses (+++) and the contents of the statement follow this.

```
       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. A message containing the program name, the line where the language processor found the error, and an explanation of the error follow this.

```
  ARX0006I Error running HELLO, line 7: Unmatched "/*" or quote
```

For more information about the error, you can go to the message explanations in the error messages section of the *VSE/ESA Messages and Codes*.

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

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

1.2.7 How to Prevent Translation to Uppercase

The language processor generally translates alphabetic characters to uppercase before processing them. The alphabetic characters can be within a program, such as words in a REXX instruction, or they can be external to a program and processed as input. You can prevent the translation to uppercase as follows:

Characters within a Program: To prevent translation of alphabetic characters in a program to uppercase, simply enclose the characters in single or double quotation marks. The language processor does not change numbers and special characters, regardless of whether they are in quotation marks. Suppose you use a SAY instruction with a phrase containing a mixture of alphabetic characters, numbers, and special characters; the language processor changes only the alphabetic characters.

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

results in:

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

(This example assumes none of the words are the names of variables that have been assigned other values.)

Quotation marks ensure that information in a program is processed exactly as typed. This is important in the following situations:

° For output that must be lowercase or a mixture of uppercase and lowercase.

° To ensure that commands are processed correctly. For example, if a variable name in a program is the same as a command name, the program can end in error when the command is issued. It is a good programming practice to avoid using variable names that are the same as commands and to enclose all commands in quotation marks.

Characters Input to a Program: When reading input or passing input from another program, the language processor also changes alphabetic characters to uppercase before processing them. To prevent translation to uppercase, use the PARSE instruction.

For example, the following program reads input from the input stream and sends this information to the output stream.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|    /************************** REXX ***********************************/ |
|    /* This REXX program gets the name of an animal from the input     */ |
|    /* stream and sends it to the output stream.                       */ |
|    /*******************************************************************/ |
|                                                                           |
|    PULL animal                     /* Get the animal name.*/              |
|    SAY animal                                                             |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```
Figure 5. Example of Reading Input and Writing Output

If the input is tyrannosaurus, the language processor produces the output:

```
    TYRANNOSAURUS
```

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

```
     PARSE PULL animal
```

Now if the input is TyRannOsauRus, the output is:

```
     TyRannOsauRus
```

Subtopics:

- 1.2.7.1 Exercises - Running and Modifying the Example Programs

---

1.2.7.1 Exercises - Running and Modifying the Example Programs

You can write and run the preceding example. Now change the PULL instruction to a PARSE PULL instruction and note the difference.

1.2.8 Passing Information to a Program

When a program runs, you can pass information to it in several ways:

° By using PULL to get information from the data stack or input stream
° By specifying input when calling the program.

Subtopics:

- 1.2.8.1 Getting Information from the Data Stack or Input Stream
- 1.2.8.2 Specifying Values When Calling a Program
- 1.2.8.3 Preventing Translation of Input to Uppercase
- 1.2.8.4 Passing Arguments

1.2.8.1 Getting Information from the Data Stack or Input Stream

The PULL instruction is one way for a program to receive input. Repeating an earlier example shows this. Here is how to call the ADDTWO program.

```
  // LIBDEF *,SEARCH=(prd1.base,rexxlib.samples)
  // EXEC REXX=addtwo
  42
  21
  /&
```

Here is the ADDTWO program.

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /**************************** REXX *****************************/   |
|   /*  This program adds two numbers and produces their sum.       */   |
|   /***************************************************************/   |
|    PULL number1                                                         |
|    PULL number2                                                         |
|    sum = number1 + number2                                              |
|    SAY 'The sum of the two numbers is' sum'.'                           |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 6. Example of a Program That Uses PULL

The PULL instruction can extract more than one value at a time from the input stream by separating a line of input. The following variation of the example shows this.

```
  // LIBDEF *,SEARCH=(prd1.base,rexxlib.samples)
  // EXEC REXX=addtwo
  42 21
  /&
```

```
 ________________________________________________________________________
|                                                                         |
|                                                                         |
|   /**************************** REXX ******************************/   |
|   /*  This program adds two numbers and says their sum             */   |
|   /*****************************************************************/   |
|    PULL number1 number2                                                 |
|    sum = number1 + number2                                              |
|    SAY 'The sum of the two numbers is' sum'.'                           |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 7. Variation of an Example that Uses PULL

The PULL instruction extracts the numbers 42 and 21 from the input stream.

Notes:

1. For the PULL instruction to extract information from the input stream, the data stack must be empty. More information about the data stack appears in Chapter 11, "Storing Information in the Data Stack" in topic 2.4.

2. If you are using SYSIPT for input and your program does not read all the lines, VSE JCL treats any remaining SYSIPT data as JCL statements. In this case, VSE JCL may issue the message

   ```
   1S01D   INVALID STATEMENT
   ```

   to the operator's console.

1.2.8.2 Specifying Values When Calling a Program

Another way for a program to receive input is through values you specify when you call the program. For example to pass the two numbers 42 and 21 to a program named ADD, you could use the JCL EXEC command:

```
  *
  // LIBDEF *, SEARCH=(prd1.base,rexxlib.samples)
  // EXEC REXX=add,PARM='42 21'
```

The program ADD uses the ARG instruction to assign the input to variables as shown in the following example.

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

Figure 8. Example of a Program That Uses the ARG Instruction

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 ARG or PULL, errors can occur, as the following sections describe.

Specifying Too Few Values: If you specify fewer values than the number of variables after PULL or ARG, the extra variables are set to the null string. Here is an example in which you pass only one number to the program:

```
  // EXEC REXX=add,PARM='42'
```

The language processor assigns the value 42 to number1, the first variable following ARG. It assigns the null string to number2, the second variable. In this situation, the program ends with an error when it tries to add the two variables. In other situations, the program might not end in error.

Specifying Too Many Values: When you specify more values than the number of variables following PULL or ARG, the last variable gets the remaining values. For example, you pass three numbers to the program ADD:

```
  // EXEC REXX=add,PARM='42 21 10'
```

The language processor assigns the value 42 to number1, the first variable following ARG. It assigns the value 21 10 to number2, the second variable. In this situation, the program ends with an error when it tries to add the two variables. In other situations, the program 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. (In this case, number1 receives 42, number2 receives 21, and the period ensures the 10 is discarded. If there is no extra information, the period is ignored. You can also use a period as a placeholder 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.

1.2.8.3 Preventing Translation of Input to Uppercase

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

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /**************************** REXX ********************************/ |
|   /*  This program receives the last name, first name, and score of */ |
|   /*  a student and reports the name and score.                     */ |
|   /******************************************************************/ |
|    PARSE ARG lastname firstname score                                  |
|    SAY firstname lastname 'received a score of' score'.'               |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```

Figure 9. Example of a Program That Uses PARSE ARG

Subtopics:

- 1.2.8.3.1 Exercises - Using the ARG Instruction

1.2.8.3.1 Exercises - Using the ARG Instruction

The left column shows the input values sent to a program. The right column is the ARG instruction within the program that receives the input. What value does each variable receive?

```
        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
```

1.2.8.4 Passing Arguments

Values passed to a program are usually called arguments. An argument can consist of one word or a string of words. Blanks separate words within an argument from each other. The number of arguments passed depends on how the program is called.

Subtopics:

1.2.8.4.1 Using the CALL Instruction or a REXX Function Call

When you call a REXX program using either the CALL instruction or a REXX function call, you can pass up to 20 arguments to the program. Separate each argument from the next with a comma.

1.2.8.4.2 Using the JCL EXEC Command

You can pass only one argument using PARM= on the JCL EXEC statement. However, keep in mind that one argument can consist of many words. An argument, if present, appears as a single string. Give special consideration to argument strings containing commas. For example, if you specify:

```
// EXEC REXX=myprog, PARM='1,2'
```

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

```
PARSE ARG A ',' B
SAY 'A is ' A    /* Produces: 'A is 1' */
SAY 'B is ' B    /* Produces: 'B is 2' */
```

For more information about functions and subroutines, see Chapter 6, "Writing Subroutines and Functions" in topic 1.6. For more information about arguments, see "Parsing Multiple Strings as Arguments" in topic 1.7.2.4.

1.3 Chapter 3. Using Variables and Expressions

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

Subtopics:

1.3.1 Program Variables

One of the most powerful aspects of computer programming is the ability to process variable data to achieve a result. Regardless of the complexity of a process, when data is unknown or varies, you substitute a symbol for the data. This is much like substituting 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**.

1.3.2 Using Variables

A variable is a character or group of characters representing a value. A variable can contain either single- or double-byte characters or both. (Double-byte characters are valid only if OPTIONS ETMODE is the first instruction of your program.) 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 program is written. In the following example, the user's name is unknown, so it is represented by the variable who.

```
                               /* Gets name from current input stream */
PARSE PULL who                 /* and puts it in variable "who"       */
```

Subtopics:

1.3.2.1 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 variable name is variable1.

```
variable1 = 5
SAY variable1
```

As a result of the preceding assignment statement, the language processor assigns variable1 the value 5, and the SAY produces:

```
 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. (OPTIONS ETMODE must be the first instruction in your program 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 2 bytes, and count the shift-out (SO) and shift-in (SI) as 1 byte each.

- SO (X'0E') and SI (X'0F') must delimit DBCS characters within a DBCS name. 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    A   Word3   number  the_ultimate_value
```

Also, if OPTIONS ETMODE is the first instruction in your program, the following are valid DBCS variable names, where < represents shift-out, > 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>
```

1.3.2.2 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 within quotation marks, such as:

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

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

  ```
  variable1 = variable2
  ```

  In the preceding 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, its value is the value of its own name translated to uppercase. For example, if the variable new has not been assigned a value, then

```
SAY new
```

produces

```
NEW
```

1.3.2.3 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. Incorrect, because the first character is a number.

2. Valid

3. Valid

4. Valid

5. Valid, but it is a special variable name that you should use only to receive results from a subroutine.

1.3.3 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 do on the numbers, variables, and strings. There are four types of operators: arithmetic, comparison, logical, and concatenation.

Subtopics:

1.3.3.1 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 this default by using the NUMERIC DIGITS instruction. For information about the NUMERIC DIGITS instruction, see the *REXX/VSE 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 said to be in 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.

You can use floating point numbers to represent very large or very small numbers. For more information about floating point numbers, see the *REXX/VSE Reference*.

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

The arithmetic operators you can use are:

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*	(Prefix -) Same as the subtraction 0 - number
+*number*	(Prefix +) Same as the addition 0 + number

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

```
     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 computes 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, the language processor evaluates the expression as follows:

° First it evaluates expressions within parentheses.

° Then it evaluates expressions with operators of higher priority before expressions with operators of lower priority.

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

Figure 10. 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
```

Subtopics:

- 1.3.3.1.1 Using Arithmetic Expressions
- 1.3.3.1.2 Exercises--Calculating Arithmetic Expressions

1.3.3.1.1 Using Arithmetic Expressions

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

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /****************************** REXX *********************************/|
|   /* This program 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 program     */|
|   /* when you run it.                                                  */|
|   /*********************************************************************/|
|                                                                          |
|    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'.'                                    |
|                                                                          |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 11. Example Using Arithmetic Expressions

1.3.3.1.2 Exercises--Calculating Arithmetic Expressions

1. What line of output does the following program produce?

```
/****************************** 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

1.3.3.2 Comparison Operators

Expressions that use comparison operators do not return a number value as do arithmetic expressions. Comparison expressions return either 1, which represents true, or 0, which represents false.

Comparison operators can compare numbers or strings and perform evaluations, such as:

Are the terms equal? (A = Z)

Is the first term greater than the second? (A > Z)

Is the first term less than the second? (A < Z)

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

```
(A = Z)  Does 4 = 3?        0 (False)
(A > Z)  Is 4 > 3?             1 (True)
(A < Z)  Is 4 < 3?             0 (False)
```

The more commonly used comparison operators are as follows:

Operator	**Meaning**
=	Equal
==	Strictly Equal
\ =	Not equal
\ ==	Not strictly 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 (\). You can use the two characters interchangeably according to availability and personal preference. This book uses the backslash (\) character.

Subtopics:

1.3.3.2.1 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 */
```

1.3.3.2.2 Using Comparison Expressions

You often use a comparison expression in an IF...THEN...ELSE instruction. The following example uses an IF...THEN...ELSE instruction to compare two values. For more information about this instruction, see "IF...THEN...ELSE Instructions" in topic 1.4.2.1.

```
|                                                                      |
|                                                                      |
|   /****************************** REXX ********************************/|
|   /* This program compares what you paid for lunch for two            */|
|   /* days in a row and then comments on the comparison.               */|
|   /********************************************************************/|
|                                                                      |
|   PARSE PULL yesterday    /* Gets yesterday's price from input stream */ |
|                                                                      |
|                                                                      |
|   PARSE PULL today            /*  Gets today's price  */             |
|                                                                      |
|                                                                      |
|   IF today > yesterday 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."     |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```

Figure 12. Example Using a Comparison Expression

1.3.3.2.3 Exercises - Using Comparison Expressions

1. Based on the preceding example of using a comparison expression, what result does the language processor produce from 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 language processor produces the following sentences:

   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

1.3.3.3 Logical (Boolean) Operators

Logical expressions, like comparison expressions, return 1 (true) or 0 (false) when processed. Logical operators combine two comparisons and return 1 or 0 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

```
          Negates--returning the opposite response.  For example:

             \ 0                 /* opposite of 0, so result is 1    */

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

Subtopics:

1.3.3.3.1 Using Logical Expressions

You can use logical expressions in complex conditional instructions and 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.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /****************************** REXX *******************************/ |
|   /* This program 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.'                                                    |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```
Figure 13. Example Using Logical Expressions

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 program, it produces the result:

```
      Go skiing.
```

1.3.3.3.2 Exercises - Using Logical Expressions

A student applying to colleges has decided to evaluate them 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 more than 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
```

1.3.3.4 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	Concatenates terms and places one blank between them. If more than one blank separates terms, this becomes a single blank. For example: `SAY true      blue          /* result is TRUE BLUE */`
||	Concatenates terms with no blanks between them. For example: `(8 / 2)||(3 * 3)            /* result is 49  */`
abuttal	Concatenates terms with no blanks between them. For example: `per_cent'%'                /* if per_cent = 50, result is 50%  */` You can use abuttal only with terms that are of different types, such as a literal string and a symbol, or when only a comment separates two terms.

Subtopics:

- 1.3.3.4.1 Using Concatenation Operators

1.3.3.4.1 Using Concatenation Operators

One way to format output is to use variables and concatenation operators as in the following example.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /****************************** REXX *********************************/|
|   /* This program formats data into columns for output.                 */|
|   /***********************************************************************/|
|     sport = 'base'                                                        |
|     equipment = 'ball'                                                    |
|     column = '          '                                                 |
|     cost = 5                                                              |
|                                                                           |
|     SAY sport||equipment column '$' cost                                  |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```
Figure 14. Example Using Concatenation Operators

The result of this example is:

```
           baseball         $ 5
```

A more sophisticated way to format information is with parsing and templates. Information about parsing appears in "Parsing Data" in topic 1.7.2.

1.3.3.5 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)
```

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

Figure 15. Overall Operator Priority

\ or ¬ - +	Prefix operators
**	Power (exponential)
* / % //	Multiply and divide
+ -	Add and subtract
blank || *abuttal*	Concatenation operators
== = >< and so on	Comparison operators
&	Logical AND
| &&	Inclusive OR and exclusive OR

Thus, given the following values

```
A = 8
B = 2
C = 10
```

the language processor would evaluate the previous example

```
IF (A > 7**B) & (B < 3)
```

as follows:

1. Evaluate what is inside the first set of parentheses.

   a. Evaluate A to 8.

   b. Evaluate B to 2.

   c. Evaluate 7**2.

   d. Evaluate 8 > 49 is false (0).

2. Evaluate what is inside the next set of parentheses.

   a. Evaluate B to 2.

   b. Evaluate 2 < 3 is true (1).

3. Evaluate 0 & 1 is 0

Subtopics:

- 1.3.3.5.1 Exercises - Priority of Operators

1.3.3.5.1 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 the previous exercise in topic 1.3.3.3.2, 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
```

1.3.4 Tracing Expressions with the TRACE Instruction

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

Tracing Operations: To trace operations within an expression, use the TRACE I (TRACE Intermediates) form of the TRACE instruction. The language processor breaks down all expressions that follow the instruction and analyzes them 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. (The line numbers are not part of the program. They facilitate the discussion of the example that follows it.)

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|    1 /************************ REXX ***************************/          |
|    2 /*  This program uses the TRACE instruction to show how   */          |
|    3 /*  an expression is evaluated, operation by operation.   */          |
|    4 /*********************************************************/          |
|    5  a = 9                                                               |
|    6  y = 2                                                               |
|    7  TRACE I                                                             |
|    8                                                                      |
|    9  IF a + 1 > 5 * y THEN                                               |
|   10     SAY 'a is big enough.'                                           |
|   11  ELSE NOP                /* No operation on the ELSE path */         |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```

Figure 16. TRACE Shows How REXX Evaluates an Expression

When you run the example, the SAY instruction produces:

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

The 9 is the line number. The *-* indicates that what follows is the data from the program, IF a + 1 < 5 * y. The remaining lines break down all the expressions.

Tracing Results: To trace only the final result of an expression, use the TRACE R (TRACE Results) form of the TRACE instruction. The language processor analyzes all expressions that follow the instruction as follows:

```
      >>>   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 a + 1 > 5 * y
      >>>   "0"
```

In addition to tracing operations and results, the TRACE instruction offers other types of tracing. The *REXX/VSE Reference* describes these.

Subtopics:

1.3.4.1 Exercises - Using the TRACE Instruction

Write a program with a complex expression, such as:

```
   IF (a > z) | (c < 2 * d) THEN ....
```

Define a, z, c, and d in the program and use the TRACE I instruction.

ANSWER

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /****************************** REXX ********************************/ |
|   /* This program uses the TRACE instruction to show how the language */ |
|   /* processor evaluates an expression, operation by operation.       */ |
|   /*********************************************************************/ |
|    a = 1                                                                  |
|    z = 2                                                                  |
|    c = 3                                                                  |
|    d = 4                                                                  |
|                                                                           |
|    TRACE I                                                                |
|                                                                           |
|    IF (a > z) | (c < 2 * d) THEN                                          |
|      SAY 'At least one expression was true.'                              |
|    ELSE                                                                   |
|      SAY 'Neither expression was true.'                                   |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```

Figure 17. Possible Solution

When you run this program, it produces:

```
      12 *-* IF (a > z) | (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.
```

1.4 Chapter 4. Controlling the Flow within a Program

Purpose: This chapter introduces instructions that alter the sequential execution of a program and demonstrates how to use those instructions.

Subtopics:

1.4.1 Conditional, Looping, and Interrupt Instructions

Generally, when a program 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 change the order of execution within a program by using REXX instructions that cause the language processor to skip some instructions, repeat others, or transfer control to another part of the program. These REXX instructions can be classified as follows:

- Conditional instructions 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 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 *repetitor*...END
  DO WHILE *expression*...END
  DO UNTIL *expression*...END

- Interrupt instructions tell the language processor to leave the program entirely or leave one part of the program and go to another part, either permanently or temporarily. The REXX interrupt instructions are:

  EXIT
  SIGNAL *label*
  CALL *label*...RETURN

1.4.2 Using Conditional Instructions

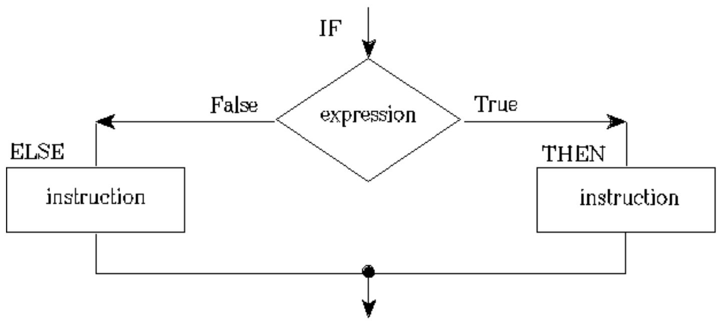
There are two types of conditional instructions:

- IF...THEN...ELSE can direct the execution of a program to one of two choices.
- SELECT WHEN...OTHERWISE...END can direct the execution to one of many choices.

Subtopics:

1.4.2.1 IF...THEN...ELSE Instructions

The examples of IF...THEN...ELSE instructions in previous chapters demonstrate 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 use a semicolon before the ELSE to separate the THEN clause from the ELSE clause.

```
     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
          PULL playgolf     /* Gets data from input stream */
          If playgolf='YES' THEN SAY 'Fore!'
        END
```

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

1.4.2.2 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 'Let''s play tennis!'
            ELSE NOP
         END
      ELSE
         SAY 'We should 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?
```

```
    _________________________________________________________________________
   |                                                                         |
   |                                                                         |
   |   /******************************** REXX *******************************/|
   |   /* This program 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 'Let''s play tennis!'                                    |
   |      ELSE                                                               |
   |         SAY 'We should take our raincoats!'                             |
   |                                                                         |
   |                                                                         |
   |_________________________________________________________________________|
```
Figure 18. Example of Missing Instructions

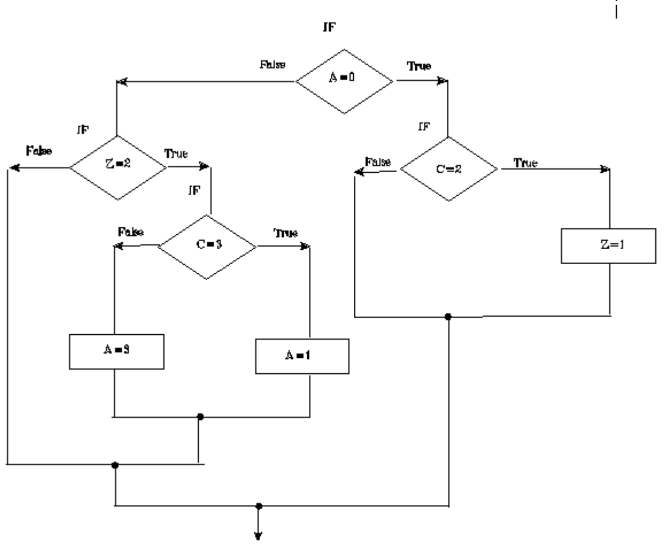
```
   Looking at the program you might assume the ELSE belongs to the first IF.
   However, the language processor associates an ELSE with the nearest
   unpaired IF.  The outcome is as follows:

         What a lovely day!
         We should take our raincoats!
```

Subtopics:

- 1.4.2.2.1 Exercise - Using the IF...THEN...ELSE Instruction

1.4.2.2.1 Exercise - Using the IF...THEN...ELSE Instruction

```
   Write the REXX instructions for the following flowchart:
```

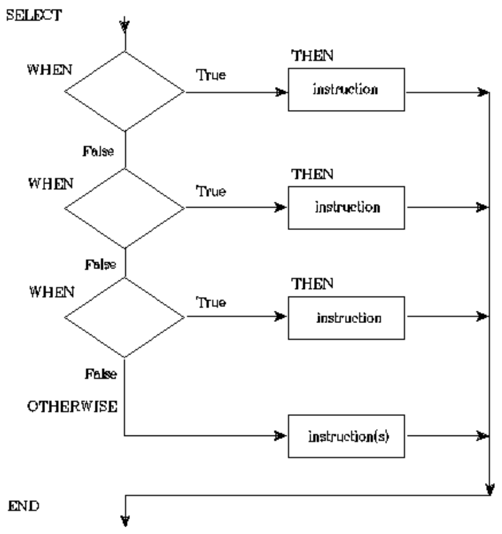


```
   ANSWER

        IF a = 0 THEN
           IF c = 2 THEN
              z = 1
           ELSE NOP
        ELSE
           IF z = 2 THEN
```

```
          IF c = 3 THEN
             a = 1
          ELSE
             a = 3
        ELSE NOP
```

1.4.2.3 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 IF...THEN...ELSE, 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.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /******************************** REXX *******************************/|
|   /* This program receives input with a person's age and sex.  In      */|
|   /* reply, it produces 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'.'                   |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 19. Example Using SELECT WHEN...OTHERWISE...END

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

Subtopics:

- 1.4.2.3.1 Exercises - Using SELECT WHEN...OTHERWISE...END

1.4.2.3.1 Exercises - Using SELECT WHEN...OTHERWISE...END

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

Write a program that uses the input of a number from 1 to 12, representing the month, and produces the number of days in that month. Assume the user specifies the month number as an argument when calling the program. (Include in the program an ARG instruction to assign the month number into the variable month). Then have the program produce the number of days. For month 2, this can be 28 or 29.

ANSWER

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX *******************************/|
|   /* This program uses the input of a whole number from 1 to 12 that   */|
|   /* represents a month.  It produces the number of days in that       */|
|   /* month.                                                            */|
|   /*********************************************************************/|
|                                                                         |
|    ARG 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'.'                        |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 20. Possible Solution

1.4.3 Using Looping Instructions

There are two types of looping instructions, **repetitive loops** and **conditional loops**. Repetitive loops let you repeat instructions a certain number of times. 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.

Subtopics:

- 1.4.3.1 Repetitive Loops
- 1.4.3.2 Conditional Loops
- 1.4.3.3 Combining Types of Loops
- 1.4.3.4 Nested DO Loops

1.4.3.1 Repetitive Loops

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

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

When you run this example, it produces 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
```

Subtopics:

1.4.3.1.1 Infinite Loops

What happens when the control variable of a loop cannot attain the last number? For example, in the following program 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, producing an endless number of lines saying Number 1.

If your program is in an infinite loop, contact the operator to cancel it.

1.4.3.1.2 DO FOREVER Loops

Sometimes you might want to write an infinite loop purposely; for instance, in a program that reads records from a file until it reaches the end of the file. 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" in topic 1.4.4.1.

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************* REXX ********************************/|
|   /* This program processes strings until the value of a string is     */|
|   /* a null string.                                                    */|
|   /*********************************************************************/|
|      DO FOREVER                                                      |
|        PULL string                     /* Gets string from input stream */|
|        IF string = '' THEN                                           |
|        PULL file_name                                                |
|        IF file_name = '' THEN                                        |
|          EXIT                                                        |
|        ELSE                                                          |
|          DO                                                          |
|           result = process(string)    /* Calls a user-written function */|
|                                       /* to do processing on string.   */|
|           IF result = 0 THEN SAY "Processing complete for string:" string|
|           ELSE SAY "Processing failed for string:" string            |
|          END                                                         |
|      END                                                             |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```
Figure 21. Example Using a DO FOREVER Loop

This example sends strings to a user-written function for processing and then issues a message that the processing completed successfully or failed. When the input string is a blank, the loop ends and so does the program. You can also end the loop without ending the program by using the LEAVE instruction. The following topic describes this.

1.4.3.1.3 LEAVE Instruction

The LEAVE 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:

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************** REXX *******************************/|
|   /* This program uses the LEAVE instruction to exit from a DO         */|
|   /* FOREVER loop.                                                     */|
|   /*********************************************************************/|
|      DO FOREVER                                                      |
|        PULL string                  /* Gets string from input stream    */|
|        IF string = 'QUIT' then                                       |
|           LEAVE                                                      |
|        ELSE                                                          |
|           DO                                                         |
|           result = process(string)    /* Calls a user-written function */|
|                                       /* to do processing on string.   */|
|           IF result = 0 THEN SAY "Processing complete for string:" string|
|           ELSE SAY "Processing failed for string:" string            |
|           END                                                        |
|      END                                                             |
|      SAY 'Program run complete.'                                     |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```
Figure 22. Example Using the LEAVE Instruction

1.4.3.1.4 ITERATE Instruction

The ITERATE instruction 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, the language processor increases and tests a control variable or tests a condition 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
```

1.4.3.1.5 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 does not match what is expected. For instance, in the example of using the LEAVE Instruction in topic 1.4.3.1.3, what happens when the input is Quit and a PARSE PULL instruction replaces the PULL instruction?

   ```
   PARSE PULL file_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.

   ```
    Index is now 10
   ```

2. The program 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 input is quit, QUIT, or Quit, the language processor translates the input to uppercase before comparing it to QUIT.

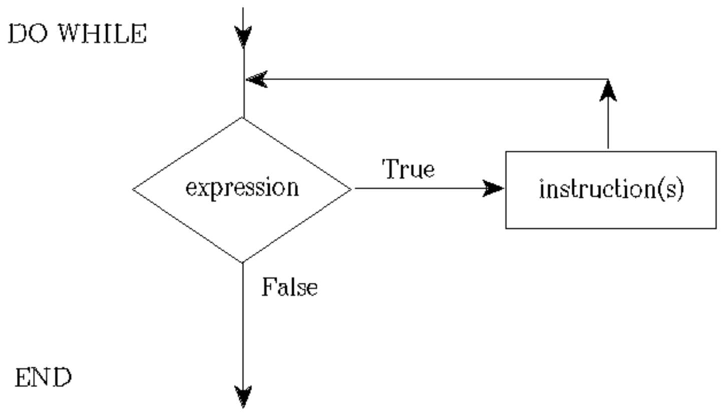
1.4.3.2 Conditional Loops

There are two types of conditional loops, DO WHILE and DO UNTIL. One or more expressions control both types of loops. 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.

Subtopics:

1.4.3.2.1 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 language processor never executes the loop.

You can use a DO WHILE loop instead of the DO FOREVER loop in the example of using the LEAVE instruction in topic 1.4.3.1.3. 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 first PULL of the following example.

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /******************************** REXX *******************************/|
|   /* This program uses a DO WHILE loop to send a string to a            */|
|   /* user-written function for processing.                              */|
|   /*********************************************************************/|
|     PULL string                       /* Gets string from input stream */|
|     DO WHILE string \= 'QUIT'                                             |
|           result = process(string)    /* Calls a user-written function */|
|                                       /* to do processing on string.   */|
|           IF result = 0 THEN SAY "Processing complete for string:" string|
|           ELSE SAY "Processing failed for string:" string                 |
|           PULL string                                                     |
|     END                                                                   |
|     SAY 'Program run complete.'                                           |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```

Figure 23. Example Using DO WHILE

1.4.3.2.2 Exercise - Using a DO WHILE Loop

Write a program with a DO WHILE loop that uses as input a list of responses about whether passengers on a commuter airline want a window seat. The flight has 8 passengers and 4 window seats. Discontinue the

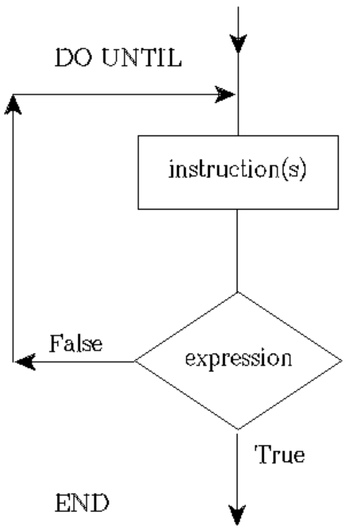
loop when all the window seats are taken. After the loop ends, produce
the number of window seats taken and the number of responses processed.

ANSWER

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************** REXX *******************************/|
|   /* This program 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 the program has not yet read the responses of   */|
|      /* all 8 passengers and while all the window seats are not taken. */|
|      /*******************************************************************/|
|                                                                      |
|       PULL window                /* Gets "Y" or "N" from input stream  */|
|       passenger = passenger + 1  /* Increase number of passengers by 1 */|
|       IF window = 'Y' THEN                                               |
|         window_seats = window_seats + 1 /* Increase window seats by 1  */|
|       ELSE NOP                                                           |
|     END                                                                  |
|                                                                      |
|     SAY window_seats 'window seats were assigned.'                       |
|     SAY passenger 'passengers were questioned.'                          |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```
Figure 24. Possible Solution

1.4.3.2.3 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:

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************** REXX ******************************/ |
|   /* This program 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)                            |
|        PULL answer                  /* Gets ANSWER from input stream  */ |
|        time = time + 1                                                   |
|      END                                                                 |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 25. Example Using DO UNTIL

1.4.3.2.4 Exercise - Using a DO UNTIL Loop

Change the program in the previous exercise in topic 1.4.3.2.2 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

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /********************************* REXX *******************************/|
|   /* This program 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 while the program has not yet read the responses of   */|
|      /* all 8 passengers and while all the window seats are not taken. */|
|      /*****************************************************************/|
|                                                                          |
|       PULL window                /* Gets "Y" or "N" from input stream  */|
|       passenger = passenger + 1  /* Increase number of passengers by 1 */|
|       IF window = 'Y' THEN                                               |
|          window_seats = window_seats + 1 /* Increase window seats by 1 */|
|       ELSE NOP                                                           |
|     END                                                                  |
|     SAY window_seats 'window seats were assigned.'                       |
|     SAY passenger 'passengers were questioned.'                          |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 26. Possible Solution

1.4.3.3 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 the quantity is less than 50, 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
```

1.4.3.4 Nested DO Loops

Like nested IF...THEN...ELSE instructions, DO loops can contain 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 name of the control variable of the loop. If the LEAVE instruction is for the inner loop, processing leaves the inner loop and goes to the outer loop. If the LEAVE instruction is for the outer loop, processing leaves 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
```

Subtopics:

- 1.4.3.4.1 Exercises - Combining Loops

1.4.3.4.1 Exercises - Combining Loops

1. What happens when the following program runs?

```
     DO outer = 1 TO 3
       SAY                    /* Produces 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                    /* Produces 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, it produces 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
```

1.4.4 Using Interrupt Instructions

Instructions that interrupt the flow of a program can cause the program to:

° End (EXIT)

° Skip to another part of the program marked by a label (SIGNAL)

° Go temporarily to a subroutine either within the program or outside the program (CALL or RETURN).

Subtopics:

1.4.4.1 EXIT Instruction

The EXIT instruction causes a REXX program to unconditionally end and return to where the program was called. If another program called the REXX program, EXIT returns to that calling program. More about calling external routines appears later in this chapter and in Chapter 6, "Writing Subroutines and Functions" in topic 1.6.

Besides ending a program, EXIT can also return a value to the caller of the program. If the program was called as a subroutine from another REXX program, the value is received in the REXX special variable RESULT. If the program was called as a function, the value is received in the original expression at the point where the function was called. 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.

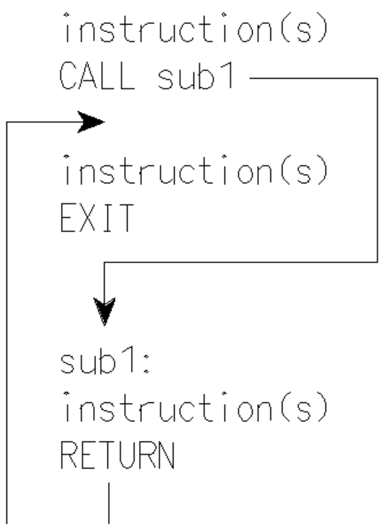
```
/********************************* REXX ****************************/
/* This program uses the EXIT instruction to end the program and  */
/* return a value indicating whether a job applicant gets the     */
/* job.  A value of 0 means the applicant does not qualify for    */
/* the job, but a value of 1 means the applicant gets the job.    */
/* The value is placed in the REXX special variable RESULT.       */
/******************************************************************/
PULL months_experience      /* Gets number from input stream      */
PULL references             /* Gets "Y" or "N" from input stream  */
PULL start_tomorrow         /* Gets "Y" or "N" from input stream  */

IF (months_experience > 24) & (references = 'Y') & (start_tomorrow= 'Y')
THEN job = 1                       /* person gets the job         */
ELSE job = 0                       /* person does not get the job */

EXIT job
```

Figure 27. Example Using the EXIT Instruction

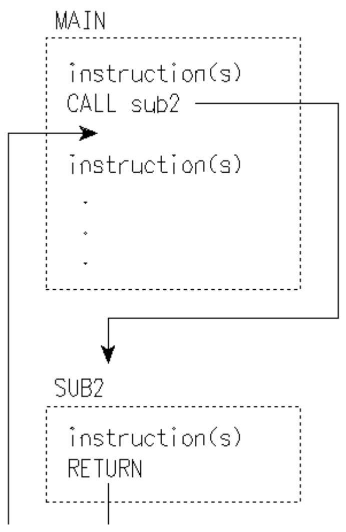
1.4.4.2 CALL and RETURN Instructions

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



When calling an external subroutine, CALL passes control to the program 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 program.

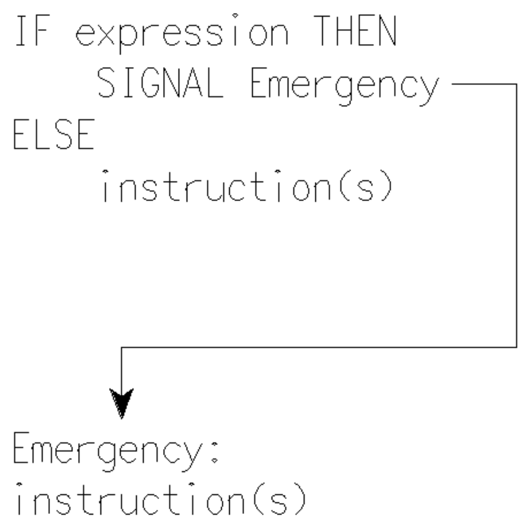


For more information about calling subroutines, see Chapter 6, "Writing Subroutines and Functions" in topic 1.6.

1.4.4.3 SIGNAL Instruction

The SIGNAL instruction, like CALL, interrupts the usual flow of a program and causes control to pass to a specified label. The label to which control passes can be 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. When you use SIGNAL from an internal routine, the internal routine does 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 programs or to provide an emergency course of action. It should not be used as a convenient way to move from one place in a program to another. SIGNAL does not provide a way to return as does the CALL instruction described in the previous topic.

For more information about the SIGNAL instruction, see topic 2.2.1.2, and the *REXX/VSE Reference*.

1.5 Chapter 5. Using Functions

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

Subtopics:

1.5.1 What is a Function?

A function 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 are built into the language processor. More about built-in functions appears later in this chapter.

- User-written functions are those an individual user writes or an installation supplies. These can be internal or external. An **internal function** is part of the current program that starts at a label. An **external function** is a self-contained program or program outside of the calling program. More information about user-written functions appears in "Writing Subroutines and Functions" in topic 1.6.2.

- Function packages are groups of functions and subroutines that an individual user writes or an installation supplies. They are link-edited into phases and categorized as user, local, and system. REXX/VSE external functions are provided in a system function package. More information about REXX/VSE external functions appears in "REXX/VSE External Functions" in topic 2.3.1.

Regardless of the kind of function, all functions return a value to the program that issued the function call. To call a function, type the function name, immediately followed by parentheses enclosing arguments to the function, if any. **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. Arguments can be:

- Constant

```
function(55)
```

- Symbol

```
function(symbol_name)
```

- Option that the function recognizes

```
function(option)
```

- Literal string

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

- Unspecified or omitted

```
function()
```

- Another function

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

- Combination of argument types

```
function('With literal string', 55, option)
function('With literal string',, option) /* Second argument omitted */
```

All functions must return values. When the function returns a value, the value replaces the function call. In the following example, the language processor adds the value the function returns to 7 and produces the sum.

```
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.

Subtopics:

- 1.5.1.1 Example of a Function

1.5.1.1 Example of a Function

Calculations that functions represent 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:

```
/****************************** REXX **********************************/
/*  This program receives three numbers as arguments 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
```

Figure 28. Finding a Maximum Number

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 you can use it as follows:

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

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

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

1.5.2 Built-In Functions

More than 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 or strings or both 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 the *REXX/VSE Reference*.

Subtopics:

1.5.2.1 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.

1.5.2.2 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 VAR, LIT, or BAD to indicate the state of the symbol (variable, literal, or bad).

1.5.2.3 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)
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)

1.5.2.4 Formatting Functions

Function	**Description**
CENTER or 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 (1)	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.

(1) Is a non-SAA built-in function REXX/VSE provides.

1.5.2.5 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 (2)	Returns the word number of the first word of a specified phrase found within the input string.
INDEX (2)	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 a specified number indicates.
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.

(2) Is a non-SAA built-in function REXX/VSE provides.

1.5.2.6 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.
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 (2)	This function always returns a 0.
LINESIZE (2)	Returns the width of the current output device. ASSGN(STDOUT) returns the name of the current output device.
QUEUED	Returns the number of lines remaining in the external data queue at the time when the function is called.
SOURCELINE	Returns either the line number of the last line in the source file or the source line a number specifies.
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 (2)	Returns the current user ID. This is the last user ID specified on the SETUID command, the user ID of the calling REXX program if one program calls another, the user ID under which the job is running, or the job name.
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.

1.5.2.7 Testing Input with Built-In Functions

Some of the built-in functions provide a convenient way to test input. When a program uses input, the user might provide input that is not valid. For instance, in the example of using comparison expressions in topic 1.3.3.2.2, the program uses a dollar amount in the following instruction.

```
    PARSE PULL yesterday  /* Gets yesterday's price from input stream */
```

If the program pulls only a number, the program processes that information correctly. However, if the program pulls a number preceded by a dollar sign or pulls a word, such as nothing, the program returns an error. To avoid getting an error, you can check the input with the DATATYPE function as follows.

```
    IF DATATYPE(yesterday) \= 'NUM'
    THEN DO
           SAY 'The input amount was in the wrong format.'
           EXIT
         END
```

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

Subtopics:

1.5.2.7.1 Exercise - Writing a Program with Built-In Functions

Write a program that checks a file name for a length of 8 characters. If the name is longer than 8 characters, the program truncates it to 8 and sends a message indicating the shortened name. Use the LENGTH and the SUBSTR built-in functions (the *REXX/VSE Reference* describes these).

ANSWER

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   /****************************** REXX *********************************/ |
|   /*  This program tests the length of a file name.                    */ |
|   /*  If the name is longer than 8 characters, the program truncates  */ |
|   /*  extra characters and sends a message indicating the shortened   */ |
|   /*  name.                                                            */ |
|   /*********************************************************************/ |
|   PULL name                     /* Gets name from input stream        */ |
|                                                                           |
|   IF LENGTH(name) > 8 THEN      /* Name is longer than 8 characters   */ |
|     DO                                                                    |
|       name = SUBSTR(name,1,8)   /* Shorten name to first 8 characters */ |
|       SAY 'The name you specified was too long.'                          |
|       SAY  name 'will be used.'                                           |
|     END                                                                   |
|   ELSE NOP                                                            |
|                                                                       |
|                                                                       |
|_______________________________________________________________________|
```

Figure 29. Possible Solution

1.6 Chapter 6. Writing Subroutines and Functions

Purpose: This chapter shows how to write subroutines and functions and discusses their differences and similarities.

Subtopics:

1.6.1 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 program, marked by a label, and only that program uses the routine.
External	A member of a sublibrary in the active PROC or PHASE chain or in the SVA. One or more programs can call an external routine.

In many aspects, subroutines and functions are the same. However, they are different in a few major aspects, such as how to call them and the way they return values.

- Calling a subroutine

  To call a subroutine, use the CALL instruction followed by the subroutine name (label or program member name). You can optionally follow this with up to 20 comma-separated arguments. The subroutine call is an entire instruction.

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

- Calling a function

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

  ```
  z = 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 program 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 program, the value is returned with either the RETURN or EXIT instruction.

  ```
  RETURN value
  ```

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

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

Subtopics:

1.6.1.1 When to Write Subroutines Rather Than 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 a program that turns them into either a subroutine or a function. For example, you can call the built-in function SUBSTR as either a function or a subroutine. This is how to call SUBSTR as a function to shorten a word to its first eight characters:

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

You get the same results if you call SUBSTR as a subroutine:

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

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.

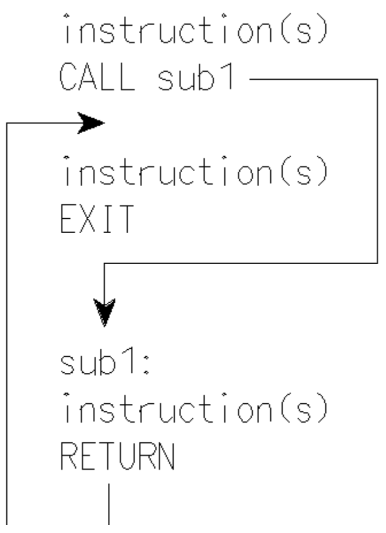
° Do I need a value returned as an expression within an instruction? If so, write a function.

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

1.6.2 Writing Subroutines and Functions

A subroutine is a series of instructions that a program calls to perform a specific task. The instruction that calls the subroutine is the CALL instruction. You can use the CALL instruction several times in a program to call 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.



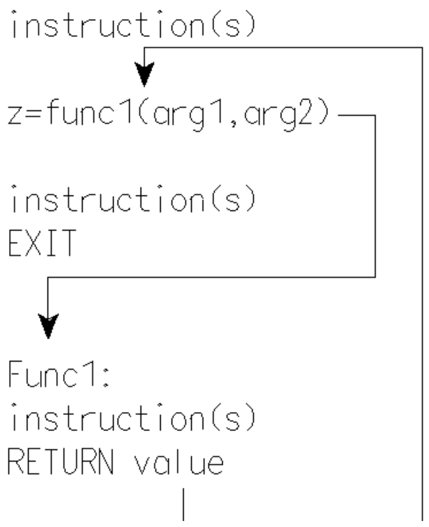
A function is a series of instructions that a program calls to perform a specific task and return a value. As Chapter 5, "Using Functions" in topic 1.5 describes, a function can be built-in or user-written. Call a user-written function the same way as a built-in function: specify the function name immediately followed by parentheses that can contain arguments. There can be no blanks between the function name and the left parenthesis. The parentheses can contain up to 20 arguments or no arguments at all.

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

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

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

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

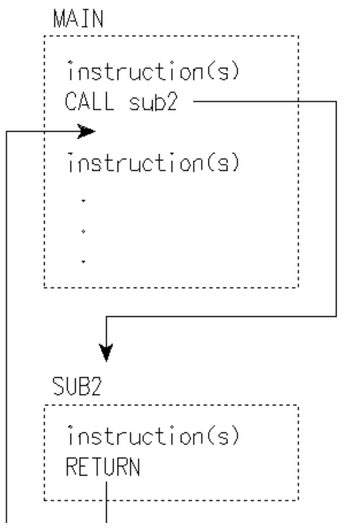


Both subroutines and functions can be **internal** (a label designates these) or **external** (a sublibrary member name that contains the subroutine or function designates these). The two preceding examples illustrate an internal subroutine named sub1 and an internal function named func1.

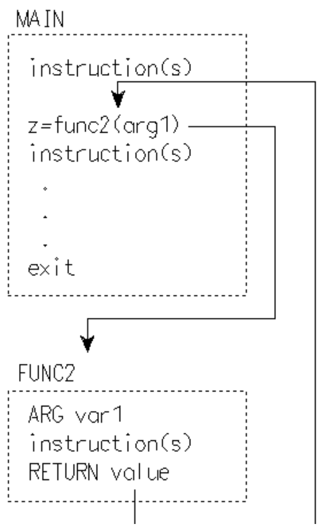
IMPORTANT NOTE

Because internal subroutines and functions generally appear after the main part of the program, when you have an internal subroutine or function, it is important to end the main part of the program with the EXIT instruction.

The following illustrates an external subroutine named sub2.



The following illustrates an external function named func2.



Subtopics:

1.6.2.1 When to Use Internal Versus External Subroutines or Functions

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

- Size of the subroutine or function. Very large subroutines and functions often are external, whereas small ones fit easily within the calling program.

- How you want to pass information. It is quicker to pass information through variables in an internal subroutine or function. The next topic describes passing information this way.

- Whether the subroutine or function might be of value to more than one program or user. If so, an external subroutine or function is preferable.

- Performance. For functions, 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" in topic 2.3.2.1.

1.6.2.2 Passing Information

A program and its internal subroutine or function can share the same variables. Therefore, you can use commonly shared variables to pass information between caller and internal subroutine or function. You can also use arguments to pass information to and from an internal subroutine or an internal function. External subroutines, however, cannot share variables with the caller. To pass information to them, you need to use arguments or some other external way, such as the data stack. (Remember: An internal 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.)

Subtopics:

1.6.2.2.1 Passing Information by Using Variables

When a program and its internal subroutine or function share the same variables, the value of a variable is what was last assigned. This is regardless of whether the assignment was in the main part of the program or in the subroutine or function.

The following example shows passing information to a subroutine. The variables number1, number2, and answer are shared. The value of answer is assigned in the subroutine and used in the main part of the program.

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************* REXX ********************************/|
|   /*  This program receives a calculated value from an internal        */|
|   /*  subroutine and uses that value in a SAY instruction.             */|
|   /*********************************************************************/|
|                                                                      |
|     number1 = 5                                                      |
|     number2 = 10                                                     |
|     CALL subroutine                                                  |
|     SAY answer                        /* Produces 15 */              |
|     EXIT                                                             |
|                                                                      |
|     subroutine:                                                      |
|     answer = number1 + number2                                       |
|     RETURN                                                           |
|                                                                      |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```

Figure 30. Example of Passing Information in a Variable Using a Subroutine

The next example is the same, except it passes information to a function rather than a subroutine. The subroutine includes the variable answer on the RETURN instruction. The language processor replaces the function call with the value in answer.

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************* REXX ********************************/|
|   /*  This program receives a calculated value from an internal        */|
|   /*  function and uses SAY to produce that value.                     */|
|   /*********************************************************************/|
|                                                                      |
|     number1 = 5                                                      |
|     number2 = 10                                                     |
|     SAY add()                           /* Produces 15 */            |
|     SAY answer                          /* Also produces 15 */       |
|     EXIT                                                             |
|                                                                      |
|     add:                                                             |
|     answer = number1 + number2                                       |
|     RETURN answer                                                    |
|                                                                      |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```

Figure 31. Example of Passing Information in a Variable Using a Function

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

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************* REXX ********************************/|
|   /*    NOTE: This program contains an error.                          */|
|   /* It uses a DO loop to call an internal subroutine, and the         */|
|   /* subroutine uses a DO loop with the same control variable as the   */|
|   /* main program.  The DO loop in the main program runs only once.    */|
|   /*********************************************************************/|
|                                                                      |
|     number1 = 5                                                      |
|     number2 = 10                                                     |
|     DO i = 1 TO 5                                                    |
|       CALL subroutine                                                |
|       SAY answer                                /* Produces 105 */   |
|     END                                                              |
|     EXIT                                                             |
|                                                                      |
|     subroutine:                                                      |
|     DO i = 1 TO 5                                                    |
|       answer = number1 + number2                                     |
|       number1 = number2                                              |
|       number2 = answer                                               |
|     END                                                              |
|     RETURN                                                           |
|                                                                      |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```

Figure 32. Example of a Problem Caused by Passing Information in a Variable Using a Subroutine

The next example is the same, except it passes information using a function instead of a subroutine:

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /******************************* REXX ********************************/|
|   /*    NOTE: This program contains an error.                          */|
|   /* It uses a DO loop to call an internal function, and the           */|
|   /* function uses a DO loop with the same control variable as the     */|
|   /* main program.  The DO loop in the main program runs only once.    */|
|   /*********************************************************************/|
|                                                                         |
|     number1 = 5                                                         |
|     number2 = 10                                                        |
|     DO i = 1 TO 5                                                       |
|       SAY add()                             /* Produces 105 */           |
|     END                                                                 |
|     EXIT                                                                |
|                                                                         |
|     add:                                                                |
|     DO i = 1 TO 5                                                       |
|       answer = number1 + number2                                        |
|       number1 = number2                                                 |
|       number2 = answer                                                  |
|     END                                                                 |
|     RETURN answer                                                       |
|                                                                         |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 33. Example of a Problem Caused by Passing Information in a Variable Using a Function

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

- The PROCEDURE instruction, as the next topic describes.

- Different variable names in a subroutine or function than in the main part of the program. For a subroutine, you can pass arguments on the CALL instruction; "Passing Information by Using Arguments" in topic 1.6.2.2.2 describes this.

Protecting Variables with the PROCEDURE Instruction: When you use the PROCEDURE instruction immediately after the subroutine or function label, all variables in the subroutine or function become local to the subroutine or function; they are shielded from the main part of the program. You can also use the PROCEDURE EXPOSE instruction to protect all but a few specified variables.

The following examples show how results differ when a subroutine or function uses or does not use PROCEDURE.

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /* This program uses a PROCEDURE instruction to protect the          */|
|   /* variables within its subroutine.                                  */|
|   /*********************************************************************/|
|    number1 = 10                                                         |
|    CALL subroutine                                                      |
|    SAY number1 number2          /* Produces  10  NUMBER2 */             |
|    EXIT                                                                 |
|                                                                         |
|    subroutine: PROCEDURE                                                |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN                                                               |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 34. Example of Subroutine Using the PROCEDURE Instruction

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /* This program does not use a PROCEDURE instruction to protect the  */|
|   /* variables within its subroutine.                                  */|
|   /*********************************************************************/|
|    number1 = 10                                                         |
|    CALL subroutine                                                      |
|    SAY number1 number2          /* Produces 7  5  */                    |
|    EXIT                                                                 |
|                                                                         |
|    subroutine:                                                          |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN                                                               |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```

Figure 35. Example of Subroutine without the PROCEDURE Instruction

The next two examples are the same, except they use functions rather than subroutines:

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /*  This program uses a PROCEDURE instruction to protect the         */|
|   /*  variables within its function.                                   */|
|   /*********************************************************************/|
|    number1 = 10                                                         |
|    SAY pass() number2               /* Produces  7  NUMBER2 */          |
|    EXIT                                                                 |
|                                                                         |
|    pass: PROCEDURE                                                      |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN number1                                                       |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```
Figure 36. Example of Function Using the PROCEDURE Instruction

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /*  This program does not use a PROCEDURE instruction to protect the */|
|   /*  variables within its function.                                   */|
|   /**********************************************************************/|
|    number1 = 10                                                         |
|    SAY pass() number2                   /* Produces 7  5  */            |
|    EXIT                                                                 |
|                                                                         |
|    pass:                                                                |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN number1                                                       |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```
Figure 37. Example of Function without the PROCEDURE Instruction

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 or function.

The next example uses PROCEDURE EXPOSE in a subroutine:

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /* This program 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 the SAY instructuion        */|
|   /* produces this name in uppercase.                                  */|
|   /**********************************************************************/|
|    number1 = 10                                                         |
|    CALL subroutine                                                      |
|    SAY number1 number2          /* produces 7  NUMBER2 */               |
|    EXIT                                                                 |
|                                                                         |
|    subroutine: PROCEDURE EXPOSE number1                                 |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN                                                               |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```
Figure 38. Example Using PROCEDURE EXPOSE in Subroutine

The next example is the same except PROCEDURE EXPOSE is in a function instead of a subroutine.

```
 _________________________________________________________________________
|                                                                         |
|                                                                         |
|   /******************************** REXX ********************************/|
|   /*  This program uses a PROCEDURE instruction with the EXPOSE option */|
|   /*  to expose one variable, number1, in its function.                */|
|   /**********************************************************************/|
|    number1 = 10                                                         |
|    SAY pass() number1                  /* Produces 5  7 */              |
|    EXIT                                                                 |
|                                                                         |
|    pass: PROCEDURE EXPOSE number1                                       |
|    number1 = 7                                                          |
|    number2 = 5                                                          |
|    RETURN number2                                                       |
|                                                                         |
|                                                                         |
|_________________________________________________________________________|
```
Figure 39. Example Using PROCEDURE EXPOSE in a Function

For more information about the PROCEDURE instruction, see the *REXX/VSE Reference*.

1.6.2.2.2 Passing Information by Using Arguments

A way to pass information to either internal or external subroutines or functions is through arguments. When calling a subroutine, you can pass up to 20 arguments separated by commas on the CALL instruction as follows:

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

In a function call, you can pass up to 20 arguments separated by commas.

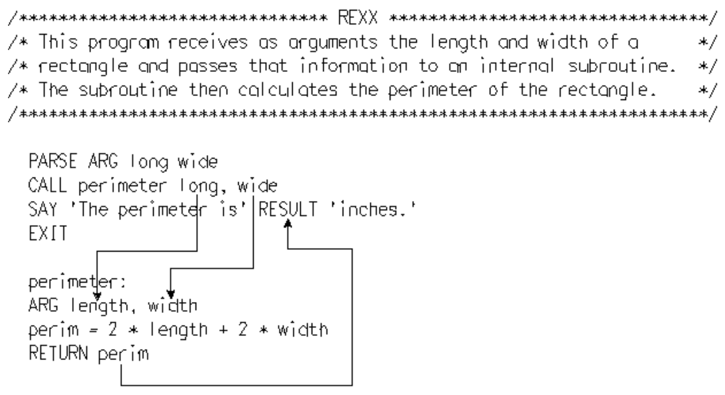
```
     function(argument1,argument2,argument3,...)
```

Using the ARG Instruction: A subroutine or function can receive the arguments with the ARG instruction. In the ARG instruction, commas also separate arguments.

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

The names of the arguments that are passed do not have to be the same as those on the ARG instruction because information is passed by position rather than by argument name. The first argument sent is the first argument received and so forth. You can also set up a template in the CALL instruction or function call. The language processor then uses this template in the corresponding ARG instruction. For information about parsing with templates, see "Parsing Data" in topic 1.7.2.

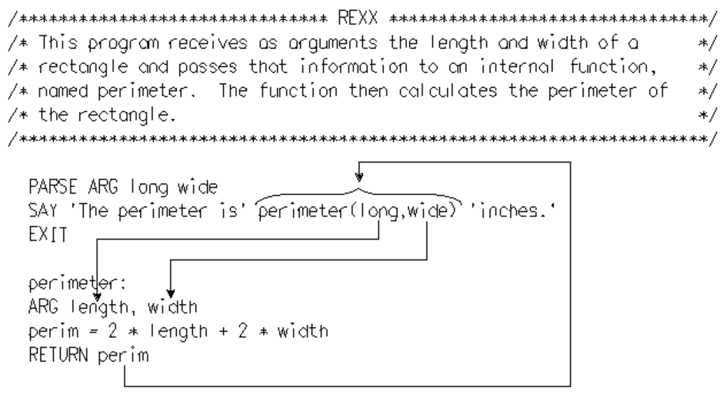
In the following example, the main routine sends information to a subroutine that computes the perimeter of a rectangle. The subroutine returns a value in the variable perim by specifying the value in the RETURN instruction. The main program receives the value in the special variable RESULT.



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

  PARSE ARG long wide
  CALL perimeter long, wide
  SAY 'The perimeter is' RESULT 'inches.'
  EXIT

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

Figure 40. Example of Passing Arguments on the CALL Instruction

The next example is the same except it uses ARG in a function instead of a subroutine.



```
/******************************** REXX *********************************/
/* This program 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
```

```
|                                                                        |
|                                                                        |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```

Figure 41. Example of Passing Arguments on the Call to an Internal Routine

In the two preceding examples, notice the positional relationships between long and length, and wide and width. Also notice how information is received from variable perim. Both programs include perim on a RETURN instruction. For the program with a subroutine, the language processor assigns the value in perim to the special variable RESULT. For the program using a function, the language processor replaces the function call perimeter(long,wide) with the value in perim.

Using the ARG Built-in Function: Another way for a subroutine or function to receive arguments is with the ARG built-in function. This function returns the value of a particular argument. A number 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 the *REXX/VSE Reference*.

1.6.2.3 Receiving Information from a Subroutine or Function

Although a subroutine or function 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 their names if no values have been assigned).

  ```
  RETURN value1 value2 value3
  ```

- A literal string

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

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

  ```
  RETURN 5 * number
  ```

Subtopics:

1.6.2.3.1 Example - Writing an Internal and an External Subroutine

Write a program that plays a simulated coin toss game and produces the accumulated scores.

There should be four possible inputs:

- 'HEADS'
- 'TAILS'
- '' (Null--to quit the game)
- None of these three (incorrect response).

Write an internal subroutine without arguments to check for valid input. Send valid input to an external subroutine that uses the RANDOM built-in function to generate random outcomes. Assume HEADS = 0 and TAILS = 1, and use RANDOM as follows:

```
  RANDOM(0,1)
```

Compare the valid input with the value from RANDOM. If they are the same, the user wins one point; if they are different, the computer wins one point. Return the result to the main program where results are tallied.

ANSWER

```
/***************************** REXX *******************************/
/* This program plays a simulated coin toss game.                 */
/* The input can be heads, tails, or null ("") to quit the game.  */
/* First an internal subroutine checks input for validity.        */
/* An external subroutine uses the RANDOM built-in function to    */
/* obtain a simulation of a throw of dice and compares the user   */
/* input to the random outcome.  The main program receives        */
/* notification of who won the round.  It maintains and produces  */
/* scores after each round.                                       */
/******************************************************************/
 PULL flip                     /* Gets "HEADS", "TAILS", or ""     */
                               /* from input stream.               */
 computer = 0; user = 0        /* Initializes scores to zero       */
 CALL check                    /* Calls internal subroutine, check */
 DO FOREVER
   CALL throw                  /* Calls 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
   PULL flip
   CALL check                  /* Call internal subroutine, check  */
 END
 EXIT
```

Figure 42. Possible Solution (Main Program)

```
/*************************** REXX ***********************************/
/*  This internal subroutine checks for valid input of "HEADS",      */
/*  "TAILS", or "" (to quit).  If the input is anything else, the    */
/*  subroutine says the input is not valid and gets the next input.  */
/*  The subroutine keeps repeating until the input is valid.         */
/*  Commonly used variables return information to the main program   */
/*******************************************************************/
check:
 DO UNTIL outcome = 'correct'
   SELECT
     WHEN flip = 'HEADS' THEN
       outcome = 'correct'
     WHEN flip = 'TAILS' THEN
       outcome = 'correct'
     WHEN flip = '' THEN
       EXIT
     OTHERWISE
       outcome = 'incorrect'
       PULL flip
   END
 END
 RETURN
```

Figure 43. Possible Solution (Internal Subroutine Named CHECK)

```
|                                                                           |
|   /******************************* REXX ********************************/|
|   /*  This external subroutine receives the valid input, analyzes it,  */|
|   /*  gets a random "flip" from the computer, and compares the two.    */|
|   /*  If they are the same, the user wins.  If they are different,     */|
|   /*  the computer wins.  The routine returns the outcome to the       */|
|   /*  calling program.                                                 */|
|   /*********************************************************************/|
|   throw:                                                                  |
|    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 chose incorrectly   */         |
|                                                                           |
|    RETURN outcome                                                         |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```

Figure 44. Possible Solution (External Subroutine named THROW)

1.6.2.3.2 Exercise - Writing a Function

Write a function named AVG that receives a list of numbers separated by blanks and computes their average. The final answer can be a decimal number. To call this function, you would use:

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

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

ANSWER

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

Figure 45. Possible Solution

1.6.3 Subroutines and Functions--Similarities and Differences

The following table highlights similarities and differences between subroutines and functions:

Similarities between Subroutines and Functions
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.

Differences between Subroutines and Functions

	Subroutines	**Functions**

Calling	Call by using the CALL instruction, followed by the subroutine name and, optionally, up to 20 arguments.	Call by specifying the function's name, immediately followed by parentheses that optionally contain up to 20 arguments.
Returning a Value	*Might* return a value to the caller. If you include a value on the RETURN instruction, the language processor assigns this value to the REXX special variable RESULT.	*Must* return a value. Specify a value on the RETURN instruction; the language processor replaces the function call with this value.

1.7 Chapter 7. Manipulating Data

Purpose: This chapter describes how to use compound variables and stems and explains parsing.

Subtopics:

1.7.1 Using Compound Variables and Stems

Sometimes it is useful to store groups of related data in a way that makes data retrieval easy. For example, you could store a list of employee names in an array and retrieve them by number. An array is an arrangement of elements in one or more dimensions, identified by a single name. An array called employee could contain 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 use the number of the element to access an element in an array. For example, employee(1) would retrieve Adams, Joe. In REXX, you use compound variables.

Subtopics:

1.7.1.1 What Is a Compound Variable?

You can use compound variables to create an array or a list of variables in REXX. A compound variable, for example: employee.1, consists of a stem and a tail. A stem is a symbol with a period at the end. Here are some examples of stems:

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

A tail is similar to a subscript. It follows the stem and consists of additional parts of the name that can be constant symbols (as in employee.1), simple symbols (as in employee.n), or null. Thus, in REXX, subscripts need not necessarily be numeric. A compound variable contains at least one period with characters on both sides of it. Here are some more examples of compound variables:

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

You cannot do any substitution for the name of the stem but you can use substitution for the tail. For example:

```
employee.7='Amy Martin'
new=7
employee.new='May Davis'
say employee.7              /* Produces: May Davis */
```

As with other REXX variables, if you have not previously assigned a value to a variable in a tail, it takes on the value of its own name in uppercase.

```
    first = 'Fred'
    last = 'Higgins'
    name = first.last          /* NAME is assigned FIRST.Higgins        */
                               /* The value FIRST appears because the   */
                               /* variable FIRST is a stem, which       */
                               /* cannot change.                        */
    SAY name.first.middle.last /* Produces NAME.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
      PARSE PULL employee.i
   END
```

If you use the same names used in the example of the employee array, you 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'
```

After 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      /* Produces 'Devon, David' */
```

For more information about compound variables, see the *REXX/VSE Reference*.

1.7.1.2 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 by using an assignment that includes a stem. For example, number.=0 initializes all array elements in the array named number. to 0.

You can change the values of all compound variables in an array the same way. For example, to change all employee names to Nobody, use the following assignment instruction:

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

As a result, all compound variables beginning with the stem employee., previously assigned or not, have the value Nobody. After a stem assignment, you can assign individual compound variables new values.

```
      employee.='Nobody'
      SAY employee.5             /* Produces 'Nobody' */
      SAY employee.10            /* Produces 'Nobody' */
      SAY employee.oldest        /* Produces 'Nobody' */

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

You can use stems with the EXECIO command when reading to and writing from a file. See "Using EXECIO to Process Information to and from Files" in topic 2.5.3 for information about EXECIO. You can also use stems with the OUTTRAP external function when trapping command output from ADDRESS POWER commands. For information about OUTTRAP, see "Using the OUTTRAP Function" in topic 2.3.1.2.

Subtopics:

1.7.1.2.1 Exercises - Using Compound Variables and Stems

1. After these assignment instructions, what do the following SAY instructions produce?

```
        a = 3           /* assigns '3' to variable 'A'  */
        d = 4           /*         '4' to          'D'  */
        c = 'last'      /*      'last' to          'C'  */
        a.d = 2         /*         '2' to         'A.4' */
        a.c = 5         /*         '5' to      'A.last' */
```

```
    z.a.d = 'cv3d'  /*      'cv3d' to       'Z.3.4' */
```

a. SAY a

b. SAY D

c. SAY c

d. SAY a.a

e. SAY A.D

f. SAY d.c

g. SAY c.a

h. SAY a.first

i. SAY z.a.4

2. After these assignment instructions, what output do the SAY instructions produce?

```
    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.   D.last
g.   C.3
h.   A.FIRST
i.   cv3d
```

2.

```
a.   empty
b.   full
c.   empty
```

1.7.2 Parsing Data

Parsing is separating data and assigning parts of it into one or more variables. Parsing can assign each word in the data into a variable or can divide the data into smaller parts. Parsing is also useful to format data into columns.

The variables to receive data are named in a template. A template is a model telling how to split the data. It can be a simple as a list of variables to receive data. More complex templates can contain patterns; "Parsing with Patterns" in topic 1.7.2.3 explains patterns.

Subtopics:

1.7.2.1 Parsing Instructions

The REXX parsing instructions are PULL, ARG, and PARSE. (PARSE has several variants.)

PULL Instruction: Earlier chapters showed PULL as an instruction that reads input and assigns it to one or more variables. If the data stack contains information, the PULL instruction takes information from the data stack. When the data stack is empty, PULL takes information from the current input stream. If you have not changed the default, the current input stream is SYSIPT. See Chapter 11, "Storing Information in the Data Stack" in topic 2.4 for information about the data stack.

```
  /* This REXX program parses the string "Knowledge is power."       */
    PULL word1 word2 word3
          /* word1 contains 'KNOWLEDGE'  */
          /* word2 contains 'IS'         */
          /* word3 contains 'POWER.'     */
```

PULL uppercases character information before assigning it into variables. If you do not want uppercase translation, use the PARSE PULL instruction.

```
  /* This REXX program parses the string: "Knowledge is power."      */
    PARSE PULL word1 word2 word3
          /* word1 contains 'Knowledge'  */
          /* word2 contains 'is'         */
          /* word3 contains 'power.'     */
```

You can include the optional keyword UPPER on any variant of the PARSE instruction. This causes the language processor to uppercase character information before assigning it into variables. For example, using PARSE UPPER PULL... gives the same result as using PULL.

ARG Instruction: The ARG instruction takes information passed as arguments to a program, function, or subroutine, and puts it into one or more variables. To pass the three arguments Knowledge is power. to a REXX program named sample:

1. Call the program and pass the arguments by specifying on the JCL EXEC statement:

```
REXX=sample,PARM='Knowledge is power.'
```

2. Use the ARG instruction to receive the three arguments into variables.

```
/* SAMPLE -- A REXX program using ARG  */
   ARG word1 word2 word3
        /* word1 contains 'KNOWLEDGE'  */
        /* word2 contains 'IS'         */
        /* word3 contains 'POWER.'     */
```

ARG uppercases the character information before assigning the arguments into variables.

If you do not want uppercase translation, use the PARSE ARG instruction instead of ARG.

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

PARSE UPPER ARG has the same result as ARG. It uppercases character information before assigning it into variables.

PARSE VALUE ... WITH Instruction: The PARSE VALUE...WITH instruction parses a specified expression, such as a literal string, into one or more variables whose names follow the WITH subkeyword.

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

PARSE VALUE does not uppercase character information before assigning it into variables. If you want uppercase translation, use PARSE UPPER VALUE. You could use a variable instead of a string in PARSE VALUE (you would first assign the variable the value):

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

Or you can use PARSE VAR to parse a variable.

PARSE VAR Instruction: The PARSE VAR instruction parses a specified variable into one or more variables.

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

PARSE VAR does not uppercase character information before assigning it into variables. If you want uppercase translation, use PARSE UPPER VAR.

1.7.2.2 More about Parsing into Words

In the preceding examples, the number of words in the data to parse is always the same as the number of variables in the template. Parsing always assigns new values to all variables named in the template. If there are more variable names than words in the data to parse, the leftover variables receive null (empty) values. If there are more words in the data to parse than variable names in the template, each variable gets one word of data in sequence except the last variable, which gets the remainder of the data.

In the next example, there are more variable names in the template than words of data; the leftover variable receives a null value.

```
    PARSE VALUE 'Extra variables' WITH word1 word2 word3
          /* word1 contains 'Extra'          */
          /* word2 contains 'variables'      */
          /* word3 contains ''               */
```

In the next example there are more words in the data than variable names in the template; the last variable gets the remainder of the data. The last variable name can contain several words and possibly leading and trailing blanks.

```
    PARSE VALUE 'More  words    in data' WITH var1 var2 var3
          /* var1 contains 'More'         */
          /* var2 contains 'words'        */
          /* var3 contains '   in data'   */
```

Parsing into words generally removes leading and trailing blanks from each word before putting it into a variable. However, when putting data into the last variable, parsing removes one word-separator blank but retains any extra leading or trailing blanks. There are two leading blanks before words. Parsing removes both the word-separator blank and the extra leading blank before putting 'words' into var2. There are four leading blanks before in. Because var3 is the last variable, parsing removes the word-separator blank but keeps the extra leading blanks. Thus, var3 receives '   in data' (with three leading blanks).

A period in a template acts as a placeholder. It receives no data. 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.

```
    string='Example of using placeholders to discard junk'
    PARSE VAR string var1 . var2 var3 .
          /* var1 contains 'Example'      */
          /* var2 contains 'using'        */
          /* var3 contains 'placeholders' */
          /* The periods collect the words 'of' and 'to discard junk' */
```

For more information about parsing instructions, see the *REXX/VSE Reference*.

1.7.2.3 Parsing with Patterns

The simplest template is a group of blank-separated variable names. This parses data into blank-delimited words. The preceding examples all use this kind of template. Templates can also contain patterns. A pattern can be a string, a number, or a variable representing either of these.

String: If you use a string in a template, parsing checks the input data for a matching string. When assigning data into variables, parsing generally skips over the part of the input string that matches the string in the template.

```
    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. (Notice also that part2 contains a value that begins with a blank. Parsing splits the input string at the matching text. It puts data up to the start of the match in one variable and data starting after the match in the next variable.

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.

```
    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. A signed integer indicates a relative column position.

An unsigned integer or an integer with the prefix of an equal sign (=) 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. 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.'  */
```

The following code has the same result:

```
    quote = 'Ignorance is bliss.'
             ....+....1....+....2

    PARSE VAR quote 1 part1 =5 part2
           /* part1 contains 'Igno'             */
           /* part2 contains 'rance is bliss.'  */
```

Specifying the numeric pattern 1 is optional. If you do not use a numeric pattern to indicate a starting point for parsing, this defaults to 1. The example also shows that the numeric pattern 5 is the same as =5.

If a template has several numeric patterns and a later one is lower than a preceding one, parsing loops back to the column the lower number specifies.

```
   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.' */
```

Thus, you could use numeric patterns to skip over part of the data:

```
   quote = 'Ignorance is bliss.'
            ....+....1....+....2

   PARSE VAR quote 2 var1 3 5 var2 7 8 var3 var 4 var5
   SAY var1||var2||var3 var4 var5  /* || means concatenate  */
                                   /* Says: grace is bliss. */
```

A signed integer in a template separates the data according to *relative column position*. The plus or minus sign indicates movement right or left, respectively, from the starting position. In the next example, remember that part1 starts at column 1 (by default because there is no number to indicate a starting point).

```
    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.'           */
```

+5 part2 means parsing puts into part2 data starting in column 6 (1+5=6). +5 part3 means data put into part3 starts with column 11 (6+5=11), and so on. The use of the minus sign is similar to the use of the plus sign. It identifies a relative position in the data string. The minus sign "backs up" (moves to the left) in the data string.

```
    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.'      */
```

In this example, part1 receives characters starting at column 1 (by default). +10 part2 receives characters starting in column 11 (1+10=11). +3 part3 receives characters starting in column 14 (11+3=14). -3 part4 receives characters starting in column 11 (14-3=11).

To provide more flexibility, you can define and use *variable numeric patterns* in a parsing instruction. To do this, first define the variable as an unsigned integer before the parsing instruction. Then, in the parsing instruction, enclose the variable in parentheses and specify one of the following before the left parenthesis:

- A plus sign (+) to indicate column movement to the right
- A minus sign (-) to indicate column movement to the left
- An equal sign (=) to indicate an absolute column position.

(Without +, -, or = before the left parenthesis, the language processor would consider the variable to be a string pattern.) The following example uses the variable numeric pattern movex.

```
    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.'      */
```

For more information about parsing, see the *REXX/VSE Reference*.

1.7.2.4 Parsing Multiple Strings as Arguments

When passing arguments to a function or a subroutine, you can specify multiple strings to be parsed. The ARG, PARSE ARG, and PARSE UPPER ARG instructions parse arguments. These are the only parsing instructions that work on multiple strings.

To pass multiple strings, use commas to separate adjacent strings.

The next example passes three arguments to an internal subroutine.

```
    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' */
```

The first argument is two words "String One" to parse into three variable names, word1, word2, and word3. The third variable, word3, is set to null because there is no third word. The second and third arguments are parsed entirely into variable names string2 and string3.

For more information about passing multiple arguments, see the *REXX/VSE Reference*.

Subtopics:

1.7.2.4.1 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 (Note that *v2* contains '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
```

2.0 PART II -- Using REXX

Besides being a versatile general-purpose programming language, REXX can interact with POWER, which expands its capabilities. This part of the book is for programmers already familiar with the REXX language and experienced in VSE/ESA. The chapters in this part cover the following topics.

° Chapter 8, "Using Commands from a Program" in topic 2.1 -- A REXX program can issue REXX/VSE commands and ADDRESS POWER commands.

° Chapter 9, "Diagnosing Problems within a Program" in topic 2.2 -- Several debugging options are available in a program.

° Chapter 10, "Using REXX/VSE External Functions" in topic 2.3 -- External functions interact with the system to do specific tasks.

° Chapter 11, "Storing Information in the Data Stack" in topic 2.4 -- The data stack is useful in I/O and other types of special processing.

° Chapter 12, "Processing Data and Input/Output Processing" in topic 2.5 -- You can use the EXECIO command to process information to and from files.

Several REXX instructions send information to the current output stream or retrieve it from the current input stream. These instructions are:

° PARSE EXTERNAL--gets information from the current input stream.
° PARSE PULL and PULL--get information from the current input stream.
° SAY--sends information to the current output stream.
° TRACE--sends information to the current output stream.
° EXECIO--reads information from or writes it to the specified output stream or device.

If you have not changed the defaults, the current input stream is SYSIPT, and the current output stream is SYSLST. You can use the ASSGN external function to return the name of the current input or output stream.

Subtopics:

2.1 Chapter 8. Using Commands from a Program

Purpose: This chapter describes how to use commands in a REXX program.

Subtopics:

2.1.1 Types of Commands

A REXX program can issue several types of commands. The main categories of commands are:

REXX/VSE commands
These commands do REXX-related tasks in a program, such as:

° Control I/O processing of information to and from files (EXECIO)

° Perform data stack services (MAKEBUF, DROPBUF, QBUF, QELEM, NEWSTACK, DELSTACK, QSTACK)

- Check for the existence of a host command environment (SUBCOM)
- Run a REXX program in the active PROC chain (EXEC)
- Set the user ID (and password) that is associated with the REXX program (SETUID)

There are also REXX/VSE immediate commands: HI, HT, RT, TE, TQ, and TS. See the *REXX/VSE Reference*, SC33-6642, for details about the immediate commands.

More information about REXX/VSE commands appears throughout the book where the related task is discussed.

ADDRESS POWER commands
These commands operate only in the POWER environment. They permit you to:

- Put a job on a queue (PUTQE). See the *REXX/VSE Reference*, SC33-6642, for details.
- Retrieve a job from a queue (GETQE). See the *REXX/VSE Reference*, SC33-6642, for details.
- Return job completion messages into the stem specified by OUTTRAP (QUERYMSG).
- Send a CTL service request to POWER. (The *VSE/POWER Application Programming*, SC33-6636, lists the POWER commands you can send through a CTL service request. *VSE/POWER Administration and Operation*, SC33-6633, explains their syntax.)

ADDRESS LINK/LINKPGM commands
These are any commands with parameter lists invoked by LINK/LINKPGM.

ADDRESS JCL commands
JCL commands can be issued via a REXX program. The REXX program must have been invoked by // EXEC REXX. See the *REXX/VSE Reference*, SC33-6642, for detailed information about issuing JCL commands via a REXX program.

When a program issues a command, the REXX special variable RC is set to the return code. A program can use the return code to determine a course of action within the program. Every time a command is issued, RC is set. Thus, RC contains the return code from the most recently issued command.

Subtopics:

2.1.1.1 Using Quotations Marks in Commands

Generally, to differentiate commands from other types of instructions, you enclose the command within single or double quotation marks. If the command is not enclosed within quotation marks, it is processed as an expression and might end in error. For example, the language processor treats an asterisk (*) as a multiplication operator.

2.1.1.2 Using Variables in Commands

When a command contains a variable, the value of the variable is not used if the variable is within quotation marks. The language processor uses the value of a variable only for variables outside quotation marks. For example, suppose the variable queue contains RDR. For the following, the language processor would *not* substitute RDR for queue:

```
ADDRESS power "PUTQE queue STEM job."
```

But if you change the quotation marks so that queue is not within them, the language processor would substitute RDR for queue:

```
  ADDRESS power "PUTQE" queue "STEM job."
```

The preceding is the same as:

```
  ADDRESS power "PUTQE RDR STEM job."
```

2.1.1.3 Calling Another REXX Program as a Command

Previously, this book discussed how to call another program as an external routine (Chapter 6, "Writing Subroutines and Functions" in topic 1.6). You can also call a program from another program explicitly with the EXEC command or implicitly by member name. Like an external routine, a program called 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 program that is called passes a value to the REXX special variable RC.

Calling Another Program with the EXEC Command: To explicitly call another program from within a program, use the EXEC command as you would any other REXX/VSE command. The called program 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 call a program named CALC and pass it an argument of four numbers, you could include the following instructions:

```
     "EXEC 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
```

Calling Another Program Implicitly: To implicitly call another program from within a program, use the member name. Because it is treated as a command, enclose the member name and the argument, if any, within quotation marks. For example, to implicitly call a program named 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
```

2.1.2 Issuing Commands from a Program

The following sections explain what a host command environment is, how commands are passed to host command environments, and how to change the host command environment.

Subtopics:

2.1.2.1 What is a Host Command Environment?

An environment for executing commands is called a host command environment. Before a program runs, an active host command environment is defined to handle commands the program issues. When the language processor encounters a command, it passes the command to the host command environment for processing.

When a REXX program runs on a host system, there is at least one default environment available for executing commands.

The host command environments are as follows:

VSE This is the default host command environment. You can use the VSE host command environment to call REXX/VSE commands (such as MAKEBUF and NEWSTACK) and services. (You cannot use ADDRESS POWER commands in this environment.)

POWER This environment is for Spool Access Support (SAS) requests and services, GET, CTL, and PUT. The POWER host command environment lets you:

- Use the PUTQE command to put elements on a POWER queue and the GETQE command to retrieve POWER queue elements
- Send a CTL service request to POWER. See *VSE/POWER Application Programming*, SC33-6636, for a list of the POWER commands that you can issue through a CTL service request. See *VSE/POWER Administration and Operation*, SC33-6633, for the syntax of these commands.
- Execute REXX/VSE commands.

LINK and LINKPGM
Host command environments for loading and calling programs. They let you load and call a phase from the active PHASE search chain. They differ in:

- the format of the parameter list that the program receives
- the capability of passing multiple parameters
- variable substitution for the parameters
- the ability of the called program to update the parameters.

JCL An environment that lets you issue JCL commands via a REXX program. The REXX program must has been invoked by // EXEC REXX. You may issue JCL commands which do not require any input data, or issue JCL commands requiring SYSIPT data. Use the REXXIPT function to accomplish this.

2.1.2.2 How Is a Command Passed to the Host Environment?

The language processor evaluates each expression in a REXX program. This evaluation results in a character string (which may be the null string). The character string is then prepared as is appropriate and submitted to the host command environment. The environment processes the string as a command, and, after processing is complete, returns control to the language processor. If the string is not a valid command for the current host command environment, a failure occurs and the special variable RC contains the return code from the host command environment.

2.1.2.3 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.

- **Single command**

  When an ADDRESS instruction includes both the name of the host command environment and a command, only that command is sent to the specified environment. After the command is complete the former host command environment becomes active again. The following ADDRESS instruction sends a single PDISPLAY command to the POWER environment.

  ```
  ADDRESS power "PDISPLAY RDR,*MYJOB"
  ```

- **All commands**

  When an ADDRESS instruction includes only the name of the host command environment, all commands issued afterward within that program are processed as that environment's commands.

  For example, the default environment is VSE. To change the environment for all commands that follow, you can use:

  ```
  ADDRESS power
  ```

  This ADDRESS instruction affects only the host command environment of the program that uses the instruction. If a program calls an external routine, the host command environment is the default environment regardless of the host command environment of the calling program. Upon return to the original program, the host command environment that the ADDRESS instruction previously established is resumed.

Subtopics:

2.1.2.3.1 Determining the Active Host Command Environment

To find out which host command environment is currently active, use the ADDRESS built-in function.

```
curenv = ADDRESS()
```

In this example, curenv is set to the active host command environment, for example, VSE.

2.1.2.3.2 Checking if a Host Command Environment Is Available

To check if a particular host command environment is available before trying to send commands to that environment, use the REXX/VSE SUBCOM command followed by the name of the host command environment, such as POWER.

```
SUBCOM power
```

If the environment is present, the REXX special variable RC is set to 0. If the environment is not present, RC is set to 1. For example, you could find out if the POWER environment is available before trying to use a GETQE command:

```
SUBCOM power
IF rc=0 THEN
DO
  ADDRESS power "PUTQE RDR STEM myjob."
  ADDRESS power "GETQE RDR JOBNAME job1 CLASS a"
END
ELSE...
```

2.1.2.3.3 Examples Using the ADDRESS Instruction

1. The following example shows how to check if the current environment is POWER, send several commands the POWER environment, and change the environment back to its original value:

```
curenv=ADDRESS()
IF curenv='POWER' THEN NOP
ELSE ADDRESS 'POWER'
"PUTQE RDR STEM mystem."
"GETQE LST JOBNAME myjob CLASS A"
ADDRESS (curenv)
```

2. The following example shows how to send a single command to POWER. Suppose your LST queue contains the following entries:

Figure 46. LST Queue Contents

Job Name	**Class**	**Job Number**
MYJOB	A	1
MYJOB	A	2
MYJOB	Q	3
NEWJOB	A	4

The following command retrieves from the LST queue the job MYJOB Class A.

```
ADDRESS power 'GETQE LST JOBNAME myjob CLASS A'
```

3. The following examples show using the LINK and LINKPGM environments to call another program. In each pair of examples, the first example includes no arguments to pass to the program, while the second example includes arguments.

```
ADDRESS LINK 'PROG1'
ADDRESS LINK 'EXPONEN parm'
```

```
      ADDRESS LINKPGM 'PROG1'
      ADDRESS LINKPGM 'ADDNUMS n1 n2 n3 ... nn'
```

2.2 Chapter 9. Diagnosing Problems within a Program

Purpose: This chapter describes how to trace command output and other debugging techniques.

Subtopics:

2.2.1 Debugging Programs

When you encounter an error in a program, there are several ways to locate the error.

° The TRACE instruction shows how the language processor evaluates each operation. (TRACE writes to the output stream. If you have not changed the default, the output stream is SYSLST.) For information about using the TRACE instruction to evaluate expressions, see "Tracing Expressions with the TRACE Instruction" in topic 1.3.4. For information about using the TRACE instruction to evaluate host commands, see the next section, "Tracing Commands with the TRACE Instruction" in topic 2.2.1.1.

° REXX/VSE sets the special variables RC and SIGL as follows:

RC	Indicates the return code from a command.
SIGL	Indicates the line number from which there was a transfer of control because of a function call, a SIGNAL instruction, or a CALL instruction.

° The TS immediate command starts tracing. The TE immediate command ends tracing, including interactive debug. You can use TS and TE in a REXX program or specify TS or TE on a call to ARXIC from a non-REXX program. For more information about interactive debug, see "Tracing with the Interactive Debug Facility" in topic 2.2.1.3.

Subtopics:

2.2.1.1 Tracing Commands with the TRACE Instruction

The TRACE instruction has many options for various types of tracing, including C for commands and E for errors.

TRACE C: After TRACE C, the language processor traces each command before executing it and then executes it and sends the return code from the command to the current output stream.

The return code from the MAKEBUF command is the current number of buffers. If you use MAKEBUF without TRACE C, the following produces no output:

```
/* REXX program */
"MAKEBUF"
```

However, the following program:

```
/* REXX program with TRACE C*/
TRACE C
"MAKEBUF"
```

produces:

```
       3 *-* "MAKEBUF"
         >>>  "MAKEBUF"
```

```
        +++ RC(1) +++
```

Note: RC contains the current number of buffers. In the example, the 1 means this is the first MAKEBUF command you have used.

TRACE E: When you specify TRACE E in a program, the language processor traces any host command that results in a nonzero return code after it executes and sends the return code from the command to the output stream.

If a program includes TRACE E and issues an incorrect command, the program sends to the output stream error messages, the line number, the command, and the return code from the command. For example, the following code:

```
  /* REXX program with error--misspelled command */
  TRACE E
  MAKBUF
```

would return:

```
       3 *-* "MAKBUF"
         +++ RC(-3) +++
```

The line number is 3, the incorrect command is MAKBUF, and the return code is -3.

For more information about the TRACE instruction, see the *REXX/VSE Reference*, SC33-6642.

2.2.1.2 Using REXX Special Variables RC and SIGL

As mentioned earlier, the REXX language has three special variables: RC, SIGL, and RESULT. REXX/VSE sets these variables during particular situations and you can use them in an expression at any time. If REXX/VSE did not set a value, a special variable has the value of its own name in uppercase, as do other variables in REXX. You can use two special variables, RC and SIGL, to help diagnose problems within programs.

RC: RC stands for return code. The language processor sets RC every time a program issues a command. When a command ends without error, RC is usually 0. When a command ends in error, RC is whatever return code is assigned to that error.

In the following example, a SAY instruction showing the RC follows an incorrect command:

```
  /* REXX program with error--'READER' should be 'RDR'  */
  ADDRESS POWER "PDELETE READER, ALL"
  SAY 'The return code from the command is' RC
```

After the incorrect SAS interface command, the return code from the SAS interface is 0.

The RC variable can be especially useful in an IF instruction to determine which path a program should take.

```
    ADDRESS POWER "PDELETE RDR,ALL"
    IF rc ¬= 0 THEN
      CALL error1
    ELSE NOP
```

Note: Every command sets a value for RC, so it does not remain the same for the duration of a program. When using RC, make sure it contains the return code of the command you want to test.

SIGL: The language processor sets the SIGL special variable in connection with a transfer of control within a program because of a function or a SIGNAL or CALL instruction. When the language processor transfers control to another routine or another part of the program, it sets the SIGL special variable to the line number from which the transfer occurred. (The line numbers in the following example are to aid in discussion after the example. They are not part of the program.)

```
  1  /* REXX */
  2  :
  3  CALL routine
  4  :
  5
  6  routine:
  7  SAY 'We came here from line' SIGL      /* SIGL is set to 3 */
  8  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 in a program, 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 usually take place, such as the transmission of error messages. (The line numbers are to aid the discussion and are not part of the program.)

```
01 /* REXX program with error -- 'READER' should be 'RDR'  */
02 SIGNAL ON ERROR
03 ADDRESS POWER "PCANCEL"
04
05 ADDRESS POWER "PDELETE READER,ALL"    /* line containing error */
06 .
07 .
08 .
09 EXIT
10
11 ERROR:
12 SAY 'The return code from the command on line' SIGL 'is' RC
```

This produces:

```
The return code from the command on line 5 is -3
```

For more information about the SIGNAL instruction, see the *REXX/VSE Reference*, SC33-6642.

2.2.1.3 Tracing with the Interactive Debug Facility

The interactive debug facility lets a user control the execution of a program. (In a batch environment, the interaction is between the input stream and the program.) The language processor reads from the input stream, and writes output to the output stream. If you have not changed the defaults, the input stream is SYSIPT, and the output stream is SYSLST.

If running from the operator's console, interactive debug pauses for input after most instructions. If you are using files for input and output, instead of pausing, interactive debug reads the next line from the input stream at each pause point.

Subtopics:

2.2.1.3.1 Starting Interactive Debug

To start interactive debug, specify ? before the option of a TRACE instruction, for example: TRACE ?A. There can be no blank(s) between the question mark and the option. Interactive debug is not carried over into external routines that are called but is resumed when the routines return to the traced program.

2.2.1.3.2 Options within Interactive Debug

After interactive debug starts, you can provide one of the following during each pause or each time the language processor reads from the input stream.

° A null line, which continues tracing. The language processor continues execution until the next pause or read from the input stream. Repeatedly input of a null line, therefore, steps from pause point to pause point until the program ends.

° An equal sign (=), which re-executes the last instruction traced. The language processor re-executes the previously traced instruction with values possibly modified by instructions read from the input stream. (The input can also be an assignment, which changes the value of a variable.)

° Additional instructions. This input can be any REXX instruction, including a command or call to another program. This input is processed before the next instruction in the program is traced. For example, the input could be a TRACE instruction that alters the type of tracing:

```
TRACE L    /* Makes the language processor pause at labels only */
```

The input could be an assignment instruction. This could change the flow of a program, by changing the value of 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 the command ends with a nonzero return code, the ELSE path is taken. To force taking the first path, the input during interactive debug could be:

```
RC = 0
```

2.2.1.3.3 Ending Interactive Debug

You can end interactive debug in one of the following ways:

- Use the TRACE OFF instruction as input. To end tracing, you can include TRACE OFF to be read as input from the current input stream. If you have not changed the default, the input stream is SYSIPT. The TRACE OFF instruction ends tracing, as stated in the message at the beginning of interactive debug:

  ```
  +++ Interactive trace.    TRACE OFF to end debug, ENTER to continue. +++
  ```

- Use the TRACE ? instruction as input

  The question mark prefix before a TRACE option can end interactive debug as well as beginning it. The question mark reverses the previous setting (on or off) for interactive debug. Thus you can use TRACE ?R within a program to start interactive debug, and provide input of another TRACE instruction with ? before the option to end interactive debug but continue tracing with the specified option.
- Use TRACE with no options as input. If you specify TRACE with no options in the input stream, this turns off interactive debug but continues tracing with TRACE Normal in effect. (TRACE Normal traces only failing commands after execution.)
- Let the program run until it ends. Interactive debug automatically ends when the program that started tracing ends. You can end the program prematurely using as input an EXIT instruction. The EXIT instruction ends both the program and interactive debug.
- Use the TE immediate command. The TE immediate command ends tracing of REXX programs. The program continues processing, but tracing is off. In interactive debug, you can provide TE as input to end tracing.

2.3 Chapter 10. Using REXX/VSE External Functions

Purpose: This chapter shows how to use external functions and describes function packages.

Subtopics:

2.3.1 REXX/VSE External Functions

In addition to the built-in functions, REXX/VSE provides external functions that you can use to do specific tasks.

The REXX/VSE external functions are:

- ASSGN - Returns the name of the current input or output stream, and, optionally, changes it.

- OUTTRAP - Returns the name of the variable in which trapped output is stored or traps lines of POWER command output.

- REXXIPT - Lets a program read data stored in compound variables. (The program can read the data as if it were SYSIPT data.)

- REXXMSG - Specifies the output destination where REXX/VSE messages are routed to. Offers the possibility to suppress all REXX error messages.

- SETLANG - Retrieves and optionally changes the language of REXX messages. The function returns the previous language setting.
- SLEEP - Specifies the number of seconds a REXX program is requested to wait until it continues processing.
- STORAGE - Returns a specified number of bytes of data from a specified address in storage. You can optionally overwrite the storage.
- SYSVAR - Returns VSE system information.

Following are brief explanations about how to use these external functions. For complete information, see *REXX/VSE Reference*, SC33-6642.

Subtopics:

2.3.1.1 Using the ASSGN Function

ASSGN returns the name of the current input or output stream, or, optionally, changes it.

ASSGN(STDIN) returns the name of the current input stream; ASSGN(STDOUT) returns the name of the current output stream. To return the name of the current stream and change it to the specified value, you can use one of the following:

```
ASSGN(STDIN,sysipt)             /* Changes input to SYSIPT          */
ASSGN(STDIN,syslog)             /* Changes input to SYSLOG          */
ASSGN(STDIN,filename)           /* Changes input to specified file  */
ASSGN(STDOUT,syslst)            /* Changes outout to SYSLST         */
ASSGN(STDOUT,syslog)            /* Changes output to SYSLOG         */
ASSGN(STDOUT,filename)          /* Changes output to specified file */
```

Note: Using SYSLST with STDIN or SYSIPT with STDOUT results in REXX error 40.

2.3.1.2 Using the OUTTRAP Function

OUTTRAP returns the name of the variable where trapped output is stored (with no arguments) or traps:

- error information from PUTQE and GETQE
- command output from POWER commands (CTL requests) sent to the SAS interface or error information if the command fails.

(See *VSE/POWER Application Programming*, SC33-6636, for a list of POWER commands you can send through a CTL service request. See *VSE/POWER Administration and Operation*, SC33-6633 for the syntax of these commands.) OUTTRAP may not trap all of the output from the SAS interface; it traps only the output routed back through the interface.

Specify OUTTRAP with no arguments to return the name of the variable in which trapped output is stored. If you use OUTTRAP with no arguments and no trapping is in effect, then it returns OFF.

```
y = OUTTRAP()
SAY y         /* Produces the variable name being used to store   */
              /* output or "OFF" if trapping is off.              */
```

Specify a stem (the part of a compound variable up to and including the first period) after the function call to trap lines of command output (or error information). See "Using Compound Variables and Stems" in topic 1.7.1 for details about compound variables.)

```
OUTTRAP("trapvar.")
```

This starts trapping, using the stem trapvar. for the numbered series of compound variables. The compound variables trapvar.1, trapvar.2, trapvar.3, and so on, each receive a line of output. If you do not set a limit to the number of output lines, the numbering of variables continues to a maximum of 999999999 lines. The total number of lines stored is in trapvar.0.

With the CONCAT option, OUTTRAP stores output from successive commands in consecutive order until reaching the maximum number of lines. For example, if a command produces 3 lines of output and the next command produces 2 lines, OUTTRAP stores output in trapvar.1 through trapvar.5. CONCAT is the default. With NOCONCAT, OUTTRAP overwrites stored lines. For example, if OUTTRAP stores 3 lines from a command (in trapvar.1 through trapvar.3) and the next command produces 2 lines, OUTTRAP stores them in trapvar.1 and trapvar.2. (Also, trapvar.3 would no longer contain the third line from the first command's output. Before OUTTRAP stores output, trapvar. is dropped, as if the REXX instruction DROP trapvar. is used.)

Here is an example using NOCONCAT:

```
  y = OUTTRAP('var.','*',NOCONCAT)
  ADDRESS power "PDISPLAY LST,ALL"
     SAY 'The number of lines stored is' var.0
```

To limit the number of lines of output saved, specify a number after the variable name.

```
     y = OUTTRAP('trapvar.',5)
```

Specifying 5 after the variable name trapvar means that up to 5 lines of command output are stored. Trapvar.1 through trapvar.5 contain the output. The number of lines of stored output, in this case 5, is in trapvar.0. Subsequent lines of command output are not saved.

For more information, see *REXX/VSE Reference*, SC33-6642.

2.3.1.3 Using the REXXIPT Function

You can use the REXXIPT function to read data stored in compound variables as if it were SYSIPT data. To have access to SYSIPT data, you must use the JCL card // EXEC REXX= to call the program containing the REXXIPT function. (Otherwise, you receive an error.)

First, store the lines of data into compound variables. For example:

```
  line.1="Now is the time"
  line.2="for all good men"
  line.3="to come to the aid of their country."
  line.0=3   /* number of lines of data */
```

Call the REXXIPT function before using the ADDRESS instruction:

```
  oldstem = REXXIPT(line.)
  ADDRESS LINK "myphase"
```

The REXXIPT function call specifies the name of a stem, in this case, line.. The variables line.1 through *line.n* contain the lines of data to read. (In this case, line.1 contains Now is the time, line.2 contains for all good men, and line.3 contains
to come to the aid of their country.). The variable line.0 contains the number of lines available to read; in this case, line.0 contains 3.

The called program uses the VSE/ESA OPEN, GET, and CLOSE macros using DTFDI to read records from SYSIPT. It reads the contents of the compound variables in order. (It reads line.1, then line.2, then line.3.) A record of fewer than 80 bytes is padded with blanks. A record of more than 80 bytes is truncated. Reading past the the last record acts as the end of file condition. If you call the same program a second time and it reads the records again, reading starts at the first record.

You can use the REXXIPT function for the following environments:

- ADDRESS JCL
- ADDRESS LINK
- ADDRESS LINKPGM.

2.3.1.4 Using the REXXMSG Function

You can use the REXXMSG function to specify the output destination where REXX/VSE messages are routed to. Use REXXMSG() to find out the current message destination without setting anything.
You can also use REXXMSG to suppress all REXX/VSE messages. Initially REXXMSG is set to "ON". See *REXX/VSE Reference*, SC33-6642, for REXXMSG examples.

2.3.1.5 Using the SETLANG Function

You can use the SETLANG function to determine the language in which REXX messages are currently being produced and optionally to 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 produced. Figure 47 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 this case, SETLANG returns the language code in effect before the call to SETLANG and sets the language to the code you have specified. The language codes you can specify depend on the language features that are installed on your system.

Figure 47. Language Codes for SETLANG Function That Replace the Function Call

Language Code	**Language**
ENP	US English--all uppercase
ENU	US English-mixed case (uppercase and lowercase) (This is the default.)

To find out the language in which REXX messages are currently being presented, use the SETLANG function with no argument:

```
      curlang=SETLANG()     /* curlang is set to the 3-character     */
                            /* code of the current language setting. */
```

To set the language to uppercase US English for subsequent REXX messages, use the SETLANG function with the 3-character code, ENP, enclosed in parentheses:

```
      oldlang=SETLANG("ENP")  /* oldlang is set to the previous          */
                              /* language setting.                       */
                              /* The current setting is ENP.             */
```

2.3.1.6 Using the SLEEP Function

You can use the SLEEP function to specify the time in seconds a REXX program is requested to wait. See *REXX/VSE Reference*, SC33-6642, for an example of SLEEP.

2.3.1.7 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. See *REXX/VSE Reference*, SC33-6642, for examples of STORAGE.

2.3.1.8 Using the SYSVAR Function

You can use the SYSVAR function to get, depending on the *arg_name*, VSE system information:

- the highest return code from VSE JCL
- the VSE JCL jobname
- the Librarian return and reason code of an EXECIO command for Libr members
- the VSE/POWER jobname
- the jobnumber of the VSE/POWER job calling the REXX program
- the partition ID
- the VSE/ESA supervisor version.

See *REXX/VSE Reference*, SC33-6642, for examples of SYSVAR.

2.3.2 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. You can write functions in REXX or in any language that VSE/ESA supports and that can follow REXX parameter passing conventions. Routines in a function package must be written in a programming language that produces object code, which can be link-edited into a phase. The routine must also support the system interface for function packages. Some programming languages that meet these qualifications are assembler, COBOL, PL/I, and REXX that is compiled.

There are three types of function packages.

- User packages -- User-written external functions that are available to an individual. These packages 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 REXX/VSE external functions. System packages are searched after user and local packages.

The default name for the user packages is ARXFUSER, and the default name for the local package is ARXFLOC. The *REXX/VSE Reference*, SC33-6642, contains information about providing your own system function package or more than one local or user function package.

Subtopics:

2.3.2.1 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 program 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--REXX/VSE searches user, local, and system function packages, in that order.

- External function--A member of a sublibrary in the active PROC or PHASE chain.

See *REXX/VSE Reference*, SC33-6642, for details about the search order.

2.4 Chapter 11. Storing Information in the Data Stack

Purpose: 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.

Subtopics:

2.4.1 What is a Data Stack?

REXX/VSE 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 that 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 that elements are added to the top of a stack and to the bottom of a queue. The figure that follows shows this.

With 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).

With 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).

```
               |      "   |                |     "     |
               |___________|                |___________|
               |   elem6   |                |  elem1    |
               |___________|                |___________|
               |   elem5   |                |  elem2    |
               |___________|                |___________|
               |   elem4   |                |  elem3    |
               |___________|                |___________|
    Stack      |   elem3   |                |  elem4    |       Queue
               |___________|                |___________|
               |   elem2   |                |  elem5    |
               |___________|                |___________|
               |   elem1   |                |  elem6    |
               |___________|                |___________|
                                            |    "      |
```

As the following figure shows, the data stack that REXX uses combines the techniques used in adding elements to stacks and queues. You can add elements on the top or the bottom of a data stack. Removal of elements from the data stack, however, occurs only from the top of the stack.

```
               |      "   |
               |___________|
    Data       |   elem1   |
               |___________|
    Stack      |   elemA   |
               |___________|
               |     "     |
```

2.4.2 Manipulating the Data Stack

Several REXX instructions manipulate the data stack. PUSH and QUEUE add elements to the data stack. PULL and PARSE PULL remove elements from the data stack.

Adding Elements to the Data Stack: PUSH and QUEUE store information on the data stack.

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 a program, the data stack would appear as follows:

```
                   PUSH
               |          |
               |___________|
               |   elem1   |
               |___________|
               |   elemA   |
               |___________|
               |     "     |
                  QUEUE
```

Note: When adding elements in a particular order to the data stack, some people find it less confusing to use the same instruction consistently, 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. These instructions appear earlier in this book extracting information from the input stream. (When the data stack is empty, PULL removes information from the input stream. If you have not changed the default, the input stream is SYSIPT.)

PULL and PARSE PULL - remove one element from the top of the data stack.

```
     PULL stackitem
```

Based on the examples from "Adding Elements to the Data Stack," the variable stackitem contains the value of elem1 with the characters translated to uppercase.

```
     SAY stackitem           /* Produces: STRING 1 FOR THE DATA STACK */
```

If you use PARSE PULL rather than PULL, the language processor does not translate the value to uppercase.

```
     PARSE PULL stackitem
     SAY stackitem           /* Produces: String 1 for the data stack */
```

After either of the preceding examples, the data stack appears as follows:

```
             PULL
       |      "     |
       |____________|
       |   elemA    |
       |____________|
       |            |
```

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, use the QUEUED function with no arguments:

```
      SAY QUEUED()         /* Produces a decimal number */
```

To remove all elements from a data stack and list them, you can use the QUEUED function as follows:

```
        number = QUEUED()
        DO number
          PULL element
          SAY element
        END
```

Subtopics:

- 2.4.2.1 Exercise - Using the Data Stack

2.4.2.1 Exercise - Using the Data Stack

Write a program that puts the letters V, S, E on the data stack in such a way that they spell "VSE" when removed. Use the QUEUED built-in function and the PULL and SAY instructions to help remove the letters and list them. To put the letters on the stack, you can use the REXX instructions PUSH, QUEUE, or a combination of the two.

ANSWER

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /******************************** REXX *****************************/  |
|   /* This program uses the PUSH instruction to put the letters V,S,E */  |
|   /* on the data stack in reverse order.                             */  |
|   /*******************************************************************/  |
|                                                                          |
|    PUSH 'E'                      /***************************/           |
|    PUSH 'S'                      /* Data in stack is:       */           |
|    PUSH 'V'                      /*    (third push)     V   */           |
|                                  /*    (second push)    S   */           |
|    number = QUEUED()             /*    (first push)     E   */           |
|    DO number                     /***************************/           |
|      PULL stackitem                                                      |
|      SAY stackitem                                                       |
|    END                                                                   |
|                                                                          |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 48. Possible Solution 1

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /******************************** REXX ******************************/ |
|   /* This program uses the QUEUE instruction to put the letters V,S,E */ |
|   /* on the data stack in that order.                                 */ |
|   /********************************************************************/ |
|                                                                          |
|    QUEUE 'V'                     /***************************/           |
|    QUEUE 'S'                     /*  Data in stack is:      */           |
|    QUEUE 'E'                      /*     (first queue)    V  */          |
|                                   /*     (second queue)   S  */          |
|                                   /*     (third queue)    E  */          |
|    DO QUEUED()                    /***************************/          |
|     PULL stackitem                                                       |
|     SAY stackitem                                                        |
|    END                                                                   |
|                                                                          |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 49. Possible Solution 2

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /******************************** REXX *****************************/  |
|   /* This program uses the PUSH and QUEUE instructions to put V,S,E  */  |
|   /* on the data stack.                                              */  |
|   /*******************************************************************/  |
|                                                                          |
|    PUSH 'S'                       /***************************/          |
|    QUEUE 'E'                      /*  Data in stack is:      */          |
|    PUSH 'V'                       /*     (second push)    V  */          |
|                                   /*     (first push)     S  */          |
|                                   /*     (first queue)    E  */          |
|    DO QUEUED()                    /***************************/          |
|      PULL stackitem                                                      |
|      SAY stackitem                                                       |
|    END                                                                   |
|                                                                          |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 50. Possible Solution 3

2.4.3 Processing of the Data Stack

You can think of a data stack as a temporary holding place for information. Every REXX/VSE program has a separate data stack available for each REXX environment that is initialized. (There is one data stack for an environment unless you create additional ones with NEWSTACK.)

When a program issues a PULL instruction, and when it issues a command, the data stack is searched first for information and, if that is empty, information is retrieved from the input stream.

```
                          |



                    ___________
                   |           |
                   |___________|
                   |           |
                   |___________|
             Data  |           |
                   |___________|
            Stack  |           |
                   |___________|
                   |           |
                   |___________|



                          |



                    ___________
                   |           |
                   |           |
     Input Stream  |           |
                   |___________|
```

Some types of input that can be stored on the data stack are:

° Data for the PULL and PARSE PULL instructions

  When a program 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 input stream for input.

  Note: To prevent the language processor from searching the data stack, you can use the PARSE EXTERNAL instruction instead of PULL. PARSE EXTERNAL gets input directly from the input stream and bypasses the data stack.

° Responses to commands

  A program can put information on the data stack for a command's use.

° Similarly, a program can put data from the input stream on the data stack for a command's use.

- Commands to be issued after the program ends

  When a program ends, the language processor considers all elements remaining on the data stack to be JCL and submits them to Job Control. (See "Leaving Data on the Stack" in topic 2.4.5 for more information.)

- Information the EXECIO command reads from and writes to files 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 Files" in topic 2.5.3.

2.4.4 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 program ends.
- It can pass information between REXX programs and other types of programs.

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 program's use.
- Pass a large number of arguments or an unknown number of arguments between a routine (subroutine or function) and the main program.
- Store data items from an input file that the EXECIO command has read. For information about the EXECIO command, see "Using EXECIO to Process Information to and from Files" in topic 2.5.3.
- Share information between a REXX program and any other program.

Passing Information between a Routine and the Main Program: You can use the data stack to pass information from a program to an external routine without using arguments. The program pushes or queues the information on the stack and the routine pulls it off and uses it. The figure that follows shows a program that puts information on the stack and calls an external routine. The second figure shows the external routine.

```
 ______________________________________________________________________
|                                                                      |
|                                                                      |
|   /*************************** REXX ********************************/    |
|   /* This program places the letters 'V', 'S', 'E' on the data     */    |
|   /* stack. It then calls an external routine that changes the     */    |
|   /* data stack, pulls a line from the stack without uppercasing   */    |
|   /* it, and sends it to the output stream.                        */    |
|   /*****************************************************************/    |
|                                                                      |
|   QUEUE 'V'                                                          |
|   QUEUE 'S'                                                          |
|   QUEUE 'E'                                                          |
|                                                                      |
|   CALL external                                                      |
|                                                                      |
|   PARSE PULL stackitem                                               |
|   SAY  stackitem                                                     |
|                                                                      |
|                                                                      |
|______________________________________________________________________|
```

Figure 51. Using the Data Stack to Pass Information from the Main Program

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|   /*************************** REXX ********************************/    |
|   /* This program reads the name of the operating system from the  */    |
|   /* stack and puts an item on the stack.                          */    |
|   /*****************************************************************/    |
|                                                                          |
|   EXTERNAL:                                                              |
|   new_stack = ''                                                         |
|   number = QUEUED()                                                      |
|                                                                          |
|   DO number                                                              |
|     PULL stackitem                                                       |
|     new_stack = new_stack||stackitem                                     |
|   End                                                                    |
|                                                                            |
|   PUSH 'You are working on a' new_stack 'system.' /* Puts item on stack */ |
|                                                                            |
|                                                                            |
|____________________________________________________________________________|
```
Figure 52. External Routine

2.4.5 Leaving Data on the Stack

If you call REXX by using the JCL EXEC command, you can leave JCL statements on the stack. VSE/ESA can then process the JCL statements left on the stack. This means you can insert JCL statements or data into the current job stream.

JCL statements must be 80 characters. If a stack entry has fewer than 80 characters, it is padded with trailing blanks. If it has more than 80 characters, only the first 80 are used; the rest are ignored. After program processing is done, the 80-character entries left on the stack are passed to VSE/ESA. VSE/ESA treats these as a JCL procedure. See the *VSE/ESA System Control Statements*, SC33-6613, for rules about the contents of a JCL procedure.

2.4.6 Creating a Buffer on the Data Stack

When a program calls a routine (subroutine or function) and both the program and the routine use the data stack, the stack becomes a way to share information. However, programs and routines that do not purposely share information from the data stack might unintentionally do so and end in error. To help prevent this, you can use the MAKEBUF and DROPBUF commands. MAKEBUF creates a buffer, which you can think of as an extension to the stack. DROPBUF 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 a program to create a temporary extension to the stack. The buffer allows a program 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.

A program 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 a program issues three MAKEBUF commands, RC is set to 3 after the third MAKEBUF command.

Note: To protect elements on the stack, a program can create a new stack with the NEWSTACK command. For information about the NEWSTACK command, see "Protecting Elements in the Data Stack" in topic 2.4.7.

Subtopics:

2.4.6.1 Creating a Buffer with the MAKEBUF Command

To create a buffer on the data stack before adding more elements to the stack, use the 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.

```
                   |      "   |
                   |__________|
                   |   newX   |
                   |__________|
                   |   newY   |  _  QUEUE
               ____|__________|___
  MAKEBUF          |          |
               ____|__________|___
                   |   old1   |
                   |__________|
                   |   oldA   |
                   |__________|
                   |     "    |
```

To create this buffer, you could use the following instructions:

```
      'MAKEBUF'
      PUSH 'newX'
      QUEUE 'newY'
```

Removing Elements from a Stack with a Buffer: The buffer MAKEBUF created does not prevent a program from accessing elements below it. After a program removes the elements added after the MAKEBUF command, then it removes elements added before the MAKEBUF command was issued.

Given the previous illustration, the program can issue three PULL instructions to remove the following elements 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'
```

2.4.6.2 Dropping a Buffer with the DROPBUF Command

When a program has no more need for a buffer on the data stack, it can use the DROPBUF command to remove the buffer (and its contents). DROPBUF removes the most recently created buffer.

```
       DROPBUF |      "   |
               |___________|
               |   old1    |
               |___________|
               |   oldA    |
               |___________|
               |     "     |
```

To drop a specific buffer on the data stack and all buffers created after it, use the REXX/VSE DROPBUF command with the number of the buffer. The first MAKEBUF creates buffer 1, the second creates buffer 2, and so on. For example, suppose a program issues three MAKEBUF commands that create three buffers. Issuing DROPBUF 2 removes the second and third buffers and all elements within them.

To remove all elements from the entire data stack including elements placed on the data stack before buffers were added, use DROPBUF 0. This creates an empty data stack. (You should use this with caution.)

Note: When an element is removed below a buffer, the buffer disappears. Thus, when you are removing elements below a buffer, the DROPBUF command you use might remove the incorrect buffer and its elements.

To prevent a program from removing elements below a buffer, use the QUEUED built-in function or the REXX/VSE NEWSTACK command, as "Protecting Elements in the Data Stack" in topic 2.4.7 describes.

2.4.6.3 Finding the Number of Buffers with the QBUF Command

To find out how many buffers the MAKEBUF command created, use the REXX/VSE QBUF command. QBUF returns the number of buffers created in the REXX special variable RC.

```
     'MAKEBUF'

  .
  .
  .
     'MAKEBUF'

  .
  .
  .
     'QBUF'
     SAY 'The number of buffers is' RC         /* RC = 2 */
```

QBUF returns the total number of buffers created, not just the ones a single program created. Thus, if a program issues two MAKEBUF commands and calls a routine that issues two more, when the routine issues a QBUF command, RC returns the total number of buffers created, which is four.

2.4.6.4 Finding the Number of Elements in a Buffer

To find out how many elements are in the most recently created buffer, use the REXX/VSE QELEM command. QELEM returns the number of elements in the most recently created buffer in the REXX special variable RC.

```
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 that the MAKEBUF command created. 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 the *REXX/VSE Reference*.

Subtopics:

- 2.4.6.4.1 Exercises - Creating a Buffer on the Data Stack

2.4.6.4.1 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. The following instructions:

```
'MAKEBUF'
QUEUE 'prompt'
'MAKEBUF'
QUEUE 'data'
QUEUE 'info'
QUEUE 'item'
'MAKEBUF'
```

created this data stack:

```
          _____ ____________ _____
 MAKEBUF 3 |            |
      _____|____________|_____
           |    data    |
           |____________|
           |    info    |
           |____________|
           |    item    |
      _____|____________|_____
 MAKEBUF 2 |            |
      _____|____________|_____
           |   prompt   |
      _____|____________|_____
 MAKEBUF 1 |            |
      _____|____________|_____
```

Answer the following questions based on this information.

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 (The language processor uppercases b because it was queued without quotation marks and pulled without PARSE.)

d. 2

e. 1

f. 0

2.

a. 4

b. 3

c. 0

d. 1, 1, 1

2.4.7 Protecting Elements in the Data Stack

In certain environments, it is often important for a program to isolate stack elements from other programs. A program might want to protect stack elements from a routine (subroutine or function) that it calls.

To protect elements on the data stack, you can create a new data stack with the REXX/VSE NEWSTACK command. To delete the new data stack and all elements in it, use the REXX/VSE DELSTACK command. Programs can create multiple stacks before deleting them.

Note: Before a program returns to its caller, the called program should issue a DELSTACK command for each NEWSTACK command it issued, unless the called program intends for the caller to also use the new data stack.

Subtopics:

2.4.7.1 Creating a New Data Stack with the NEWSTACK Command

The NEWSTACK command creates a private data stack that is completely isolated from the original data stack. A program and the routines that it calls cannot access the elements on the original data stack until it (or its routines) issues a DELSTACK command. When there are no more elements in the new data stack, information is taken from the input stream.

Note: If you use NEWSTACK, you also need to use 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.

```
                                         |       "  |
               ___________               |___________|
 Original     |   old1    |              |   newX    |    New
              |___________|              |___________|
 Stack        |   oldA    |              |   newY    |    Stack
              |___________|              |___________|
                                         |     "     |
```

To create this new stack, you could use the following instructions:

```
PUSH 'oldA'
PUSH 'old1'
'NEWSTACK'
QUEUE 'newY'
PUSH 'newX'
```

2.4.7.2 Deleting a Private Stack with the DELSTACK Command

When a program wants to delete the new stack and remove all elements placed on the new stack, it can issue the REXX/VSE DELSTACK command. DELSTACK 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.**

2.4.7.3 Finding the Number of Stacks

To find out how many stacks exist, use the REXX/VSE QSTACK command. QSTACK returns the total number of stacks, including the original data stack, in the REXX special variable RC.

```
'NEWSTACK'
 :
'NEWSTACK'
 :
'QSTACK'
SAY 'The number of stacks is' RC           /* RC contains 3 */
```

QSTACK returns the total number of stacks, not only the ones created for a single program. Suppose a program issues two NEWSTACK commands and calls a routine that issues two more. When the routine issues a QSTACK command, RC contains the total number of stacks, which is five.

For more information about these commands, see the *REXX/VSE Reference*.

Subtopics:

2.4.7.3.1 Additional Example

```
/********************************* REXX *****************************/
/* This program 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 the */
/* 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     /* Produces 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 the stack.  If stack is     */
                 /* empty, pull an element from the input stream.    */

 'DELSTACK' /* Remove the new stack and return to the original stack.*/

 PULL anyitem
 SAY anyitem     /* Produces: ATTENTION PLEASE!  THIS IS A TEST.     */
```

Figure 53. Data Stack Example

2.5 Chapter 12. Processing Data and Input/Output Processing

Purpose: This chapter describes dynamic modification of a single REXX expression and I/O processing of files.

Subtopics:

2.5.1 Types of Processing

The word *processing* here means the performance of operations and calculations on data. Ordinary 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 files

The REXX/VSE EXECIO command in a program reads information from a file to the data stack (or a list of variables) and writes information from the data stack (or list of variables) back to a file.

2.5.2 Dynamic Modification of a Single REXX Expression

Typically the language processor evaluates REXX expressions and the result replaces the expression. For example, the arithmetic expression 5 + 5 evaluates to 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.

Subtopics:

2.5.2.1 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".

```
number=2
message='SAY "The square of 2 is"'
square='** 2'
INTERPRET message number square   /* "The square of 2 is 4" */
```

You can also group a number of instructions within a string, assign the string to a variable, and use INTERPRET to execute the instructions:

```
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 the *REXX/VSE Reference*.

2.5.3 Using EXECIO to Process Information to and from Files

A program uses the EXECIO command to perform the input and output (I/O) of information to and from a file. The information can be stored in the data stack for serialized processing or in a list of variables for random processing.

Subtopics:

2.5.3.1 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 file

- Write information to a file
- Open a file without reading or writing any records
- Empty a file
- Copy information from one file to another
- Copy information to and from a list of compound variables
- Add information to the end of a file
- Update information in a file one line at a time.

2.5.3.2 Using the EXECIO Command

EXECIO reads information from or writes information to a file. You can also use EXECIO to open a file without reading or writing any records or to empty a file. EXECIO reads information from a file with either the DISKR or DISKRU operands. Using these operands, you can also open a file without reading its records. See "Reading Information from a File" for more information about the DISKR and DISKRU operands. EXECIO writes information to a file with the DISKW operand. Using this operand, you can also open a file without writing records or empty an existing file. See "Writing Information to a File" in topic 2.5.3.2.2 for more information on the DISKW operand.

EXECIO operates on I/O files of the following types:

- Sublibrary members of any type. The REXX program must specify the full name of the member on the EXECIO command. (The full name consists of a library name, sublibrary name, member name, and member type, for example: mylib.mysublib.myfile.typea.) See the description of NODATA and DATA that follows.

- SYSIPT and SYSLST. These names are reserved words on the EXECIO command. You can use only DISKR (not DISKRU) with SYSIPT. You can use only DISKW with SYSLST.

  This option is only for members of a sublibrary.

  NODATA
  DATA
  : indicates whether the member contains SYSIPT data. NODATA specifies no SYSIPT data in the member. DATA specifies the member contains SYSIPT data.

    This option is valid only for DISKW and is required only for opening a member of a sublibrary. This option is ignored for other types of files.

    The default is NODATA for a new member. For a member that already exists, the default is its value from when it was created.

- SAM files. Only SAM files on disk are supported.

Before EXECIO can perform I/O to or from a SAM file, you need to use DLBL to associate the file with a file name. The following example associates userid.my.input with the file name myinp:

```
  // DLBL myinp,'userid.my.input'
```

On EXECIO for SAM files, you also need to specify additional operands that you do not specify for other types of files. See topic 2.5.3.2.2 for details.

If you use EXECIO to read information from a file 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 file to a list of variables, the first file line is stored in variable1, the second file line is stored in variable2, and so on. You can randomly access data read into a list of variables. After the information is in the data stack or in a list of variables, the program can test it, copy it to another file, or update it before returning it to the original file.

Subtopics:

2.5.3.2.1 Reading Information from a File

To read information from a file to the data stack or to a list of variables, use EXECIO with either the DISKR or DISKRU operand. To read all lines from the sublibrary member mylib.mysub.myfile.text, you could use:

```
    "EXECIO * DISKR  mylib.mysub.myfile.typea  (FINIS"
```

The asterisk immediately after EXECIO specifies reading the entire file rather than only a certain number of lines.

To read all lines from a sequential file named my.data.set you would first use DLBL to associate the file with a file name, such as myfile, as follows:

```
  // DLBL myfile,'my.data.set'
```

You could then use the following EXECIO command:

```
  "EXECIO * DISKR  myfile (FINIS RECFORM fixblk RECSIZE 80 BLKSIZE 400"
```

This EXECIO command includes the additional operands that SAM files require. See topic 2.5.3.2.2 for descriptions.

To read all lines from the default input stream, you could use:

```
    "EXECIO * DISKR sysipt (FINIS"
```

(Remember: use only DISKR for SYSIPT.)

The rest of the examples in this chapter primarily use files that are sublibrary members. Remember that for SAM files you need to use DLBL to associate the file with a file name before using EXECIO and that you need to include the additional operands as options (see topic 2.5.3.2.2) on the EXECIO command. For further information, see the *REXX/VSE Reference*.

How to specify the number of lines to read: In the preceding examples, the asterisk immediately after EXECIO specifies reading the entire file. To read a specific number of lines, put the number immediately after EXECIO:

```
     "EXECIO 25  ..."
```

When all the information is on the data stack, you can queue a null line to indicate the end of the information. If there are null lines throughout the data, you can use the built-in function QUEUED to determine the number of items on the stack. (Examples using QUEUE and QUEUED for writing output are in topic 2.5.3.2.2.)

To open a file without reading any records, specify 0 immediately after EXECIO and specify the OPEN operand.

```
     "EXECIO 0  DISKR mylib.mysub.myfile.typea (OPEN"
```

How to specify the number of bytes to read: Library members with logical record format "string" consist of only one line. For dumps, for example, this line can be large. Using option BYTES, you can read such a file in portions.

```
     "EXECIO 1 DISKR mylib.mysub.myfile.typeb (BYTES 5000"
```

Breaking large string library members in parts requires less storage for command execution.

Using DISKR or DISKRU: Depending on the purpose you have for the input file, use either the DISKR or DISKRU operand.

° DISKR - Reading Only

  To start I/O from a file that you want only to read, use the DISKR operand with the FINIS option. The FINIS option closes the file after the information is read. Closing the file allows other programs to access the file.

```
       "EXECIO  *  DISKR ... (FINIS"
```

  Note: Do not use the FINIS option if you want the next EXECIO in your program to continue reading at the line immediately following the last line read.

° DISKRU - Reading and Updating

  To start I/O to a file that you want to read and update, use the DISKRU operand without the FINIS option. (Remember: you cannot use DISKRU for SYSIPT.) Because you can update only the last line that was read, you usually read and update a file one line at a time, or go immediately to the single line that needs updating. The file 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 File" in topic 2.5.3.2.5.

Option of specifying a starting line number: If you want to start reading at a line other than the beginning of the file, specify the line number at which to begin. For example, to read all the lines starting at line 100 to the data stack, you could use:

```
     "EXECIO  *  DISKR  mylib.mysub.myfile.typea  100  (FINIS"
```

To start at line 100 and read only 5 lines to the data stack, use:

```
     "EXECIO 5  DISKR  mylib.mysub.myfile.typea  100  (FINIS"
```

To open a file at line 100 without reading lines to the data stack, use:

```
     "EXECIO  0  DISKR  mylib.mysub.myfile.typea  100  (OPEN"
```

See "Options" in topic 2.5.3.2.2 for information about DISKR, DISKRU, and DISKW options.

Option of specifying a starting byte number: If you read library members with logical record format "string" in parts, you can specify the start byte where the reading is to begin.

```
     "EXECIO 1 DISKR mylib.mysub.myfile.typeb (BYTES 5000 STRBYTE 800"
```

2.5.3.2.2 Writing Information to a File

To write information to a file from the data stack or from a list of variables, use EXECIO with the DISKW operand. To write all lines on the stack to the sublibrary member mylib.mysub.myfile.typea, you could use:

```
     "EXECIO * DISKW mylib.mysub.myfile.typea (FINIS"
```

The asterisk immediately after EXECIO specifies writing all the lines on the stack to the file rather than only a certain number of lines.

To write all lines from the stack to a sequential file named my.data.set you would first use DLBL to associate the file with a file name: // DLBL myfile,'my.data.set'. You could then use an EXECIO command (which must include the additional operands SAM files require):

```
  "EXECIO * DISKW  myfile (FINIS RECFORM fixblk RECSIZE 80 BLKSIZE 400"
```

To write all lines from the input stream SYSIPT to the operator's console, you could use:

```
    "EXECIO * DISKR SYSIPT"
    QUEUE ''
    "EXECIO * DISKW SYSLST"
```

How to specify the number of lines to write: In the preceding examples, the asterisk immediately after EXECIO specifies writing all the lines. To write a specific number of lines, put the number immediately after EXECIO:

```
     "EXECIO 25  DISKW  ..."
```

To write the entire data stack or to write until a null line is found, you can use EXECIO * in conjunction with the QUEUE instruction or QUEUED built-in function. Using EXECIO * causes EXECIO to 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 looks for input in the input stream. ASSGN(STDIN) returns the name of the current input stream. If you do not want EXECIO to check the input stream, 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 result of the QUEUED built-in function to a variable to indicate the number of items on the stack.

```
      n = QUEUED()
      "EXECIO" n "DISKW  joeslib.joessub.joesfile.typea (FINIS"
```

Note: The stack can contain a null line but the language processor converts this to a blank line when writing to the file.

To open a file without writing records to it, specify 0 after EXECIO and specify the OPEN operand.

```
     "EXECIO 0  DISKW  mylib.mysub.myfile.typea (OPEN"
```

Note: To empty a file, you can use two EXECIO commands:

```
     "EXECIO  0  DISKR  mylib.mysub.myfile.typea  (OPEN"
     "EXECIO  0  DISKW  mylib.mysub.myfile.typea  (FINIS"
```

The first command opens the file and positions the file position pointer before the first record. The second command writes an end-of-file mark and closes the file. This deletes all records in mylib.mysub.myfile.typea. You can also empty a file by using EXECIO with both the OPEN and FINIS operands.

How to specify the number of bytes to write: You can specify the BYTES option to write a library member with logical record format "string".

```
     "EXECIO 1 DISKW mylib.mysub.myfile.typeb (BYTES 500"
```

Options: Options you can use are:

° OPEN - Opens a file. When you specify OPEN with EXECIO 0, this opens the data set and positions the file position pointer before the first record.

```
     "EXECIO 0 DISKR mylib.mysub.myfile.typea (OPEN"
     "EXECIO 0 DISKW mylib.mysub.myfile.typea (OPEN"
```

Note: If the file is already open, no operation is performed for OPEN.

° FINIS - Closes the file after reading it or writing to it. Closing the file lets other programs access it. For reading, FINIS also resets the current positional pointer to the beginning of the file. For writing, FINIS forces the completion of all I/O operations by physically writing the contents of any partially filled I/O buffers to the file.

```
     "EXECIO  *  DISKR  mylib.mysub.myfile.typea (FINIS"
     "EXECIO  *  DISKW  mylib.mysub.myfile.typea (FINIS"
```

° STEM - Specifies reading the information from or writing it to variables (instead of the data stack). If you specify a simple variable after STEM (rather than a stem, which ends in a period), the variable names are simply appended with numbers. In this case, you cannot easily access the variables by using an index in a loop. If you specify a compound variable after STEM, you can access the variables by using an index in a loop.

```
     "EXECIO  *  DISKR  mylib.mysub.myfile.typea  (STEM newvar."
     "EXECIO  *  DISKW  mylib.mysub.myfile.typea  (STEM newvar."
```

Both examples use the stem newvar..

The DISKR command places lines of information or records from the file in variables. If 10 lines of information are read, newvar.1 contains record 1, newvar.2 contains record 2, and so forth, up to newvar.10, which contains record 10. The number of items in the list of compound variables is in the special variable newvar.0. Thus, if 10 lines of information are read into the newvar. variables, newvar.0 contains the number 10. Each stem variable beyond newvar.10 (for example, variable newvar.11) is residual; it contains the value that it held before the EXECIO command.

To avoid confusion about whether a residual stem variable value is meaningful, you may want to clear the entire stem variable before entering the EXECIO command. To clear all stem variables, you can either:

- Use the DROP instruction as follows to set all stem variables to their uninitialized state:

```
       DROP newvar.
```

- Set all stem variables to nulls as follows:

```
      newvar. = ''
```

The DISKW command writes lines of information from the compound variables newvar.1, newvar.2, newvar.3, and so on, to the file. The variable newvar.0 is not used.

When writing from variables, if you use * with a stem, the EXECIO command stops writing information to the file when it finds a null value or an uninitialized compound variable. For example, if the list contains 10 compound variables, the EXECIO command stops at newvar.11.

You can specify the number of lines EXECIO reads to or writes from a list of compound variables.

```
    "EXECIO 5 DISKR  mylib.mysub.myfile.typea (STEM newvar."
    "EXECIO 5 DISKW  mylib.mysub.myfile.typea (STEM newvar."
```

In these examples, EXECIO reads 5 items to the newvar. variables or writes 5 items from them.

See Figure 60 in topic 2.5.3.2.6 for an example of EXECIO with stem variables.

Accessing SAM files requires additional operands that are not needed for other files.

BLKSIZE *n*
: *n* specifies the block size of the file. The maximum is 32700. See *VSE/ESA System Macros User's Guide*, SC33-6615, for details about the block size.

RECFORM FIXBLK
RECFORM FIXUNB
RECFORM VARBLK
RECFORM VARUNB
: Specifies the record format is fixed blocked, fixed unblocked, variable blocked, or variable unblocked.

RECSIZE *n*
: Specifies the record size. This is required if you specify RECFORM FIXUNB or RECFORM FIXBLK. Do not specify RECSIZE if you specify RECFORM VARBLK or RECFORM VARUNB. Records are blank-extended if they are too short. If the records are too long, EXECIO ends with an error.

See the *REXX/VSE Reference*, SC33-6642, for information about return codes from EXECIO.

2.5.3.2.3 Copying Information from One File to Another

Before you can copy one file to another, the files must be sublibrary members, SAM files, or SYSIPT or SYSLST. (For SAM files, you must use a DLBL before using EXECIO.)

Copying an entire file: To copy the entire sublibrary member mylib.mysub.myfile.typea to joeslib.joessub.joesfile.typea, you could use the following instructions:

```
    "NEWSTACK" /* Create a new data stack for input only */
    "EXECIO * DISKR mylib.mysub.myfile.typea (FINIS"
    QUEUE '' /* Add a null line to indicate the end of information */
    "EXECIO * DISKW joeslib.joessub.joesfile.typea (FINIS"
    "DELSTACK" /* Delete the new data stack */
```

If the program does not queue a null line at the end of the information on the stack, the EXECIO command goes to the input stream to get more information and does not end until it encounters a null line.

You can also use the QUEUED built-in function to indicate the end of the information when copying an entire file. If the file 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 mylib.mysub.myfile.typea (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" in topic 2.4.7.

To copy the entire sublibrary member mylib.mysub.myfile.typeb with logical record format "string" to joeslib.joessub.joesfile.typeb you could use:

```
      "NEWSTACK" /* Create a new data stack for input only */
      "EXECIO * DISKR mylib.mysub.myfile.typeb (  FINIS"
      "PARSE PULL item" /* Determine number of bytes */
      "QUEUE item"
      "EXECIO 1 DISKW joeslib.joessub.joesfile.typeb ( BYTES "length(item),
         " FINIS"
      "DELSTACK" /* Delete the new data stack */
```

To copy the entire sublibrary member mylib.mysub.myfile.typeb with logical record format "string" to joeslib.joessub.joesfile.typeb using multiple copy steps you could use:

```
      "NEWSTACK" /* Create a new data stack for input only */
      "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 100 OPEN"
      DO UNTIL RC = 2 /* Loop till end of file */
        "EXECIO 1 DISKW joeslib.joessub.joesfile.typeb ( BYTES 100"
        "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 100"
      END
      "EXECIO 0 DISKW joeslib.joessub.joesfile.typeb ( BYTES 100 FINIS"/* Close file */
      "EXECIO 0 DISKR mylib.mysub.myfile.typeb ( FINIS" /* Close file */
      "DELSTACK" /* Delete the new data stack */
```

To copy the entire sublibrary member mylib.mysub.myfile.typeb with logical record format "string" to joeslib.joessub.joesfile.typea with logical record format "fixed", you could use:

```
      "NEWSTACK" /* Create a new data stack for input only */
      "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 100 OPEN"
      DO UNTIL RC = 2 /* Loop till end of file */
        "EXECIO 1 DISKW joeslib.joessub.joesfile.typea "
        "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 100"
      END
      "EXECIO 0 DISKW joeslib.joessub.joesfile.typeb ( FINIS"/* Close file */
      "EXECIO 0 DISKR mylib.mysub.myfile.typea ( FINIS" /* Close file */
      "DELSTACK" /* Delete the new data stack */
```

Copying a specified number of lines to a new file: To copy 10 lines of data from the sublibrary member mylib.mysub.myfile.typea to the sublibrary member joeslib.joessub.joesfile.typea, you could use:

```
    "EXECIO 10 DISKR mylib.mysub.myfile.typea (FINIS"
    "EXECIO 10 DISKW joeslib.joessub.joesfile.typea (FINIS"
```

To copy the same 10 lines of data to a list of compound variables with the stem data., use:

```
    "EXECIO 10 DISKR mylib.mysub.myfile.typea (FINIS STEM data."
    "EXECIO 10 DISKW joeslib.joessub.joesfile.typea (FINIS STEM data."
```

Copying a specified number of bytes to a new file: To copy 800 bytes from sublibrary member "mylib.mysub.myfile.typeb" with logical record format "string" starting with byte 4001 to the sublibrary member "joeslib.joessub.joesfile.typeb", you could use:

```
      "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 800 STRTBYTE 4001 STEM var.",
         " FINIS"
      "EXECIO 1 DISKW joeslib.joessub.joesfile.typeb ( BYTES "LENGTH(var.1),
         " STEM var. FINIS"
```

Adding lines to the end of a file: To add 5 lines from a sublibrary member named my.input.file.text to the end of a sublibrary member named new.output.file.text, you could use:

```
   "EXECIO 5 DISKR my.input.file.text (OPEN STEM var. FINIS"    /* Read input file     */
   "EXECIO * DISKR new.output.file.text (OPEN STEM var2. FINIS" /* Read output file    */
   "EXECIO * DISKW new.output.file.text (OPEN STEM var2."       /* Go to end of output */
   "EXECIO 5 DISKW new.output.file.text (STEM var. FINIS"       /* Add new records     */
```

Adding bytes to the end of a file: To add 500 bytes from sublibrary member named mylib.mysub.myfile.typeb with logical record format "string" starting with byte 2000 to the end of a sublibrary member joeslib.joessub.joesfile.typeb, you could use:

```
      "EXECIO 1 DISKR mylib.mysub.myfile.typeb ( BYTES 500 STRTBYTE 2000 STEM var.",
         " FINIS"
      "EXECIO 1 DISKR joeslib.joessub.joesfile.typeb ( OPEN STEM var2. ",
         " FINIS"
      "EXECIO 1 DISKW joeslib.joessub.joesfile.typeb ( OPEN STEM var2. ",
  " BYTES "length(var2.1)
      "EXECIO 1 DISKW joeslib.joessub.joesfile.typeb ( BYTES "LENGTH(var.1),
         " STEM var. FINIS"
```

2.5.3.2.4 Copying Information to and from Compound Variables

When copying information from a file, 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 file, 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 File to a List of Compound Variables: To copy an entire file into compound variables with the stem newvar., and then send the list to the output stream, use:

```
    "EXECIO * DISKR mylib.mysub.myfile.typea (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 program is run.

```
    ARG lines
    "EXECIO" lines "DISKR mylib.mysub.myfile.typea (STEM newvar."
```

Copying Information from Compound Variables to a File: To copy 10 compound variables with the stem newvar., regardless of how many items are in the list, you could use the following:

```
    "EXECIO 10 DISKW mylib.mysub.myfile.typea (STEM NEWVAR."
```

Note: An uninitialized compound variable has the value of its own name in uppercase. For example, if newvar.9 and newvar.10 do not contain values, the file receives the values NEWVAR.9 and NEWVAR.10.

2.5.3.2.5 Updating Information in a File

You can use EXECIO to update a single line of a file or multiple lines. Use the DISKRU form of the EXECIO command to read information that you may subsequently update.

Notes on files with fixed record length:

° 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 file. When a changed line is shorter than the original line, it is padded with blanks to attain the original line length.

° When using DISKRU, the value for the *lines* operand following EXECIO must be 1. If you use a value greater than 1, you receive an error message and a return code of 20, and the program ends. After a line is written, trying to rewrite the line causes an error.

° You cannot use DISKRU with SYSIPT.

Notes on files with logical record format "string":

° The length of a "string"-type file can change.

° After one part of the record i swritten, trying to write another part without a DISKRU operation in between causes an error.

Updating a single line: When updating a single line in a file, it is more efficient to locate the line in advance and specify the update to it than to read all the lines in the file to the stack, locate and change the line, and then write all the lines back.

Suppose you have a sublibrary member named dept5.employee.list.text 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
```

You can change the information on a particular line. For example, to change the phone extension on line 2 to 5500, you could use:

```
    "EXECIO 1 DISKRU dept5.employee.list.text 2 (LIFO"
    PULL line
    PUSH 'Crandall, Amy         AMY           5500'
    "EXECIO 1 DISKW dept5.employee.list.text (FINIS"
```

Updating a single string in a file: Suppose you have a sublibrary member named dept5.employee.list.string with logical record format "string" consisting of 1 record with string "Confidential" at position 110. To replace this string by a blank, you could use:

```
      "EXECIO 1 DISKRU dept5.employee.list.string (BYTES 12 STRTBYTE 110 LIFO"
      PULL line
      newdata.1 = ' '
      "EXECIO 1 DISKW dept5.employee.list.string (BYTES 1 STEM newdata."
```

If you want to replace the following 5 bytes by string "Unconfidential", you could continue:

```
      "EXECIO 1 DISKRU dept5.employee.list.string (BYTES 5 LIFO"
      PULL line
      newdata.1 = 'Unconfidential'
      "EXECIO 1 DISKW dept5.employee.list.string (BYTES 14 STEM newdata."
```

Updating multiple lines: To update multiple lines, you can use more than one EXECIO command for the same file. For example, to update Mary Leone's user ID in addition to Amy Crandall's phone extension, use the following instructions.

```
    "EXECIO 1 DISKRU dept5.employee.list.text 2 (LIFO"
    PULL line
    PUSH 'Crandall, Amy         AMY           5500'
    "EXEXIO 1 DISKW  dept5.employee.list.text"
    "EXECIO 1 DISKRU dept5.employee.list.text 5 (LIFO"
    PULL line
    PUSH 'Leone, Mary           MARYL         5530'
    "EXECIO 1 DISKW dept5.employee.list.text (FINIS"
```

2.5.3.2.6 Additional Examples

```
 ____________________________________________________________________________
|                                                                            |
|                                                                            |
|   /***************************** REXX *********************************/ |
|   /* This program reads from a file to find the first occurrence      */ |
|   /* of the string "Jones". It ignores upper and lowercase            */ |
|   /* distinctions (by using PULL, which uppercases data it reads).    */ |
|   /*                                                                  */ |
|   /********************************************************************/ |
|   done = 'no'                                                              |
|   lineno=0                                                                 |
|   DO WHILE done = 'no'                                                     |
|     "EXECIO 1 DISKR store.employee.list.text"                              |
|                                                                            |
|      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 54. EXECIO Example 1

```
 ____________________________________________________________________________
|                                                                            |
|                                                                            |
|   /***************************** REXX *********************************/ |
|   /* This program copies records from the sublibrary member           */ |
|   /* STORE.INPUT.JAN20.TEXT to the end STORE.OUTPUT.JAN20.TEXT.       */ |
|   /* The program assumes that the input file has no null lines.       */ |
|   /*******************************************************************/ |
|                                                                        |
|   SAY 'Copying ...'                                                    |
|                                                                        |
|   "EXECIO * DISKR store.input.jan20.text (FINIS"                       |
|   QUEUE ''   /* Insert a null line at the end to indicate end of file */ |
|   "EXECIO * DISKW store.output.jan20.text (FINIS"                      |
|                                                                        |
|   SAY 'Copy complete.'                                                 |
|                                                                        |
|   EXIT 0                                                               |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```
Figure 55. EXECIO Example 2

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /***************************** REXX *********************************/|
|   /* This program starts at the third record and reads 5 records from  */|
|   /* the sublibrary member STORE.SALES.MAR3.TEXT.  It strips trailing  */|
|   /* blanks from the records and writes any record that is longer      */|
|   /* than 20 characters.  It does not close the file when finished.    */|
|   /*******************************************************************/|
|   "EXECIO 5 DISKR store.sales.mar3.text 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 56. EXECIO Example 3

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /***************************** REXX *********************************/|
|   /* This program reads the first 100 records (or until EOF) of the    */|
|   /* sublibrary member STORE.STOCK.FEB13.TEXT.  (It issues a message   */|
|   /* if it reads fewer than 100 records.)  It puts records on the data */|
|   /* stack in LIFO order.                                              */|
|   /*******************************************************************/|
|   eofflag = 2            /* Return code to indicate end of file */      |
|                                                                        |
|   "EXECIO 100 DISKR store.stock.feb13.text (LIFO"                      |
|   return_code = RC                                                     |
|                                                                        |
|   IF return_code = eofflag THEN                                        |
|      SAY 'Premature end of file.'                                      |
|   ELSE                                                                 |
|      SAY '100 Records read.'                                           |
|   DROPBUF 0                                                            |
|   EXIT return_code                                                     |
|                                                                        |
|                                                                        |
|________________________________________________________________________|
```
Figure 57. EXECIO Example 4

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /***************************** REXX ********************************/ |
|   /* This program uses "EXECIO 0 ..." to open, empty, or close a      */ |
|   /* sequential file.  It reads records from DANS.IN.DATA             */ |
|   /* and writes selected records to DANS.OUT.DATA.                    */ |
|   /* DANS.IN.DATA has variable-length records (RECFORM = VARBLK).     */ |
|   /* (Before using EXECIO, use DLBL to associate DANS.IN.DATA with    */ |
|   /* INPUT and DANS.OUT.DATA with OUTPUT.)                            */ |
|   /******************************************************************/ |
|   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 file INPUT, but do not read any records yet.            */ |
|   /* All records are read and processed within the loop body.         */ |
|   /******************************************************************/ |
|                                                                        |
|   /* Open INPUT */                                                     |
|   "EXECIO 0 DISKR input (OPEN RECFORM VARBLK BLKSIZE 400"              |
|                                                                        |
|   /******************************************************************/ |
|   /* Now read all lines from INPUT, starting at line 1, and copy      */ |
|   /* selected lines to OUTPUT.                                        */ |
|   /******************************************************************/ |
|                                                                        |
|   DO WHILE (return_code \ = eofflag) /* Loop while not end-of-file    */ |
|     'EXECIO 1 DISKR input'                  /* Read 1 line            */ |
|                                             /* to 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 longer than 10 chars  */ |
|          DO                                                              |
|            /* Write to output     */                                     |
|            "EXECIO 1 DISKW output (STEM line. RECFORM VARBLK"            |
|             out_ctr = out_ctr + 1        /* Increment output line ctr */ |
|          END                                                             |
|      END                                                                 |
|   END                                                                    |
|   /* Close the input file, INPUT  */                                     |
|   "EXECIO 0 DISKR input (FINIS "                                         |
|                                                                          |
|   IF out_ctr > 0 THEN             /* Were any lines written to output?*/ |
|     DO                            /* Yes.  So output is now open      */ |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 58. EXECIO Example 5

```
 __________________________________________________________________________
|                                                                          |
|                                                                          |
|      /*****************************************************************/ |
|      /* Because OUTPUT is already open at this point, the following   */ |
|      /* "EXECIO 0 DISKW..." command closes the file                   */ |
|      /* but does not empty it of the lines that have already been     */ |
|      /* written.  OUTPUT will contain out_ctr lines.                  */ |
|      /*****************************************************************/ |
|                                                                          |
|     /* Close the open file */                                            |
|     "EXECIO 0 DISKW output (Finis RECFORM VARBLK"                        |
|     SAY 'OUTPUT now contains ' out_ctr' lines.'                          |
|   END                                                                    |
|   ELSE                         /* Else no new lines have been         */ |
|                                /* written to OUTPUT.                  */ |
|     DO                         /* Erase any old records from it.      */ |
|                                                                          |
|      /*****************************************************************/ |
|      /* Because OUTPUT is still closed at this point, the             */ |
|      /* following "EXECIO 0 DISKW..." command opens the file,         */ |
|      /* writes 0 records, and then closes it.  This effectively       */ |
|      /* empties OUTPUT.  This deletes any old records that            */ |
|      /* were in the file when the program started.                    */ |
|      /*****************************************************************/ |
|                                                                          |
|      /* Empty DANS.OUT.DATA */                                           |
|      "EXECIO 0 DISKW output (OPEN FINIS RECFORM VARBLK BLKSIZE 400"      |
|      SAY 'Output is now empty.'                                          |
|      END                                                                 |
|   EXIT                                                                   |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```

Figure 59. EXECIO Example 5 (continued)

```
 ________________________________________________________________________
|                                                                        |
|                                                                        |
|   /****************************** REXX ********************************/|
|   /* This program uses EXECIO to successively append records from    */|
|   /* STORE.DATA.ONE.TEXT and then STORE.DATA.TWO.TEXT to the end     */|
|   /* of STORE.DATA.ALL.TEXT.  It shows the effect of residual data   */|
|   /* in STEM variables.  STORE.DATA.ONE.TEXT contains 20 records.    */|
|   /* STORE.DATA.TWO.TEXT contains 10 records.                        */|
|   /*******************************************************************/|
|                                                                        |
|                                                                        |
|   /*******************************************************************/|
|   /* Read all records from STORE.DATA.ONE.TEXT and append them to    */|
|   /* the end of STORE.DATA.ALL.TEXT.                                 */|
|   /*******************************************************************/|
|                                                                        |
|   prog_rc = 0                 /* Initialize program return code  */   |
|                                                                        |
|   /* Read all records    */                                            |
|   "EXECIO * DISKR store.data.one.text (STEM newvar. FINIS"             |
|                                                                        |
|   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, and so on  */|
|     /* through newvar.20 contain the 20 records that were read.      */|
|     /*****************************************************************/|
|                                                                        |
|       SAY "-----------------------------------------------------"      |
|       SAY newvar.0 "records have been read from store.data.one.text."  |
|       SAY                                                              |
|       DO i = 1 TO newvar.0       /* Loop through all records         */|
|         SAY newvar.i             /* Produces the ith record          */|
|       END                                                              |
|                                                                        |
|       /* Write exactly the number of records read.      */             |
|       "EXECIO" newvar.0 "DISKW store.data.all.text (STEM newvar."      |
|       IF rc = 0 THEN             /* If write was successful          */|
|         DO                                                             |
|           SAY                                                          |
|           SAY newvar.0 "records were written to store.data.all.text."  |
|         END                                                            |
|       ELSE                                                             |
|         DO                                                             |
|           prog_rc = RC           /* Save program return code      */   |
|           SAY                                                          |
|           SAY "Error during 1st EXECIO ... DISKW; return code is " RC  |
|           SAY                                                          |
|         END                                                            |
|     END                                                                |
|                                                                          |
|                                                                          |
|__________________________________________________________________________|
```
Figure 60. EXECIO Example 6

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|   ELSE                                                                    |
|     DO                                                                    |
|       prog_rc = RC               /* Save program return code        */   |
|       SAY                                                                 |
|       SAY "Error during 1st EXECIO ... DISKR, return code is " RC         |
|       SAY                                                                 |
|     END                                                                   |
|                                                                           |
|     IF prog_rc = 0 THEN          /* If no errors so far... continue  */  |
|       DO                                                                  |
|       /**************************************************************/  |
|       /* At this time, the stem variables newvar.0 through newvar.20 */  |
|       /* contain residual data from the previous EXECIO.             */  |
|       /* "DROP newvar." clears these residual values from the stem.  */  |
|       /**************************************************************/  |
|       DROP newvar.               /* Set all stem variables to their      |
|                                     uninitialized state             */  |
|       /**************************************************************/  |
|       /* Read all records from STORE.DATA.TWO.TEXT and append them   */  |
|       /* to the end of STORE.DATA.ALL.TEXT.                          */  |
|       /**************************************************************/  |
|                                                                           |
|       /* Read all records*/                                               |
|       "EXECIO * DISKR store.data.two.text (STEM newvar. FINIS"            |
|        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, and so on through newvar.10 contain 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            */  |
|         /* STORE.DATA.ONE.TEXT.                                      */  |
|         /* However, we would know that the last EXECIO DISKR did not */  |
|         /* read these values because the current newvar.0 variable   */  |
|         /* indicates that the last EXECIO read only 10 records.      */  |
|         /************************************************************/  |
|           SAY                                                             |
|           SAY                                                             |
|           SAY "-----------------------------------------------------"     |
|           SAY newvar.0 "records have been read from store.data.two.text."|
|           SAY                                                             |
|           DO i = 1 TO newvar.0   /* Loop through all records         */  |
|             SAY newvar.i         /* Produces the ith record          */  |
|           END                                                             |
|                                                                           |
|           "EXECIO" newvar.0 "DISKW store.data.all.text (STEM newvar."     |
|                         /* Writes exactly the number of records read  */  |
|           IF RC = 0 THEN         /* If write was successful          */  |
|            DO                                                             |
|              SAY                                                          |
|              SAY newvar.0 "records were written to 'store.data.all.text'"|
|            END                                                            |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```
Figure 61. EXECIO Example 6 (continued)

```
 ___________________________________________________________________________
|                                                                           |
|                                                                           |
|           ELSE                                                            |
|             DO                                                            |
|               prog_rc = RC       /* Save program return code         */  |
|               SAY                                                         |
|               SAY "Error during 2nd EXECIO ...DISKW, return code is " RC |
|               SAY                                                         |
|             END                                                           |
|         END                                                               |
|       ELSE                                                                |
|         DO                                                                |
|           prog_rc = RC           /* Save program return code         */  |
|           SAY                                                             |
|           SAY "Error during 2nd EXECIO ... DISKR, return code is " RC    |
|           SAY                                                             |
|         END                                                               |
|     END                                                                   |
|                                                                           |
|   "EXECIO 0 DISKW store.data.all.text (FINIS"  /* Close output file  */  |
|                                                                           |
|    EXIT 0                                                                 |
|                                                                           |
|                                                                           |
|___________________________________________________________________________|
```
Figure 62. EXECIO Example 6 (continued)

BIBLIOGRAPHY Bibliography

This bibliography lists some publications that provide additional information about REXX or the VSE/ESA system.

- *REXX/VSE Reference*, SC33-6642

- *VSE/ESA REXX/VSE Diagnosis Reference*, LY33-9189
- *VSE/ESA System Control Statements*, SC33-6613
- *VSE/ESA Guide to System Functions*, SC33-6611
- *VSE/POWER Application Programming*, SC33-6636
- *VSE/POWER Administration and Operation*, SC33-6633
- *VSE/ESA Messages and Codes*, SC33-6607
- *SAA Common Programming Interface REXX Level 2 Reference*, SC24-5549
- *IBM Compiler and Library for SAA REXX/370 Release 2: Introducing the Next Step in REXX Programming*, G511-1430-01
- *IBM Compiler and Library for SAA REXX/370 Release 2 User's Guide and Reference*, SH19-8160
- *IBM Compiler and Library for SAA REXX 370 Release 2 Diagnosis Guide*, SH19-8179.

INDEX Index

BACK_1 Communicating Your Comments to IBM

```
  IBM VSE/Enterprise Systems Architecture
  VSE Central Functions
  REXX/VSE User's Guide
  Version 6 Release 1


  Publication No. SC33-6641-00



  If you especially like or dislike anything about this book, please use one
  of the methods listed below to send your comments to IBM.  Whichever
```

method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of the book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

° If you prefer to send comments by mail, use the RCF form and either send it postage-paid in the United States, or directly to:

  IBM Deutschland Entwicklung GmbH
  Department 3248
  Schoenaicher Strasse 220
  D-71032 Boeblingen
  Federal Republic of Germany

° If you prefer to send comments by FAX, use this number:

  - (Germany): 07031+16-3456
  - (Other countries): (+49)+7031-16-3456

° If you prefer to send comments electronically, use this network ID:

  IBM Mail Exchange: DEIBMBM9 at IBMMAIL

  Internet: VSEPUBS at VNET.IBM.COM

Make sure to include the following in your note:

° Title and publication number of this book
° Page number or topic to which your comment applies.

COMMENTS Readers' Comments -- We'd Like to Hear from You

IBM VSE/Enterprise Systems Architecture
VSE Central Functions
REXX/VSE User's Guide
Version 6 Release 1

Publication No. SC33-6641-00

Overall, how satisfied are you with the information in this book?

Legend:

1 Very satisfied
2 Satisfied
3 Neutral
4 Dissatisfied
5 Very dissatisfied

	1	2	3	4	5
Overall satisfaction					

How satisfied are you that the information in this book is:

	1	2	3	4	5
Accurate					
Complete					
Easy to find					
Easy to understand					
Well organized					

```
|__________________________________________|_____|_____|_____|_____|_____|
| Applicable to your tasks                 |     |     |     |     |     |
|__________________________________________|_____|_____|_____|_____|_____|
```

Please tell us how we can improve this book:

```
    International Business Machines Corporation
    Attn: Dept ECJ - BP/003D
    6300 Diagonal Highway
    Boulder, CO  80301
```

```
Name   . . . . . . . . . _______________________________________________
Company or Organization  _______________________________________________
Address  . . . . . . . . _______________________________________________
                         _______________________________________________
                         _______________________________________________
Phone No.  . . . . . . . _______________________________________________
```

IBM Library Server Print Preview

DOCNUM = SC33-6641-00
DATETIME = 09/25/97 11:10:12
BLDVERS = 1.2
TITLE = REXX/VSE V6R1 User's Guide
AUTHOR =
COPYR = © Copyright IBM Corp. 1988, 1995
PATH = /home/webapps/epubs/htdocs/book